

less_retarded_wiki

by drummyfish, generated on 03/21/24, available under [CC0 1.0](#) (public domain)

100r

Hundred Rabbits

These motherfuckers are [toxic SJWs](#), avoid them like the devil. For now see [xxiivv](#).

21st_century

21st Century

21st century, known as the Age Of [Shit](#), is already one of the worst centuries in history, despite only being around for a short time. How unlucky it is to have been born in such a shitty time.

3d_modeling

3D Modeling

The topic of 3D modeling will be part of article about [3D models](#).

3d_model

3D Model

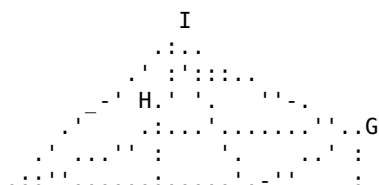
In the world of [computers](#) (especially in [computer graphics](#), but also e.g. in physics simulations etc.) 3D model is a representation of a [three dimensional](#) object, for example of a real life object such as a car, [tree](#) or a [dog](#), but also possibly something more abstract like a [fractal](#) or [function](#) plot surface. 3D models can be displayed using various [3D rendering](#) techniques and are used mostly to simulate [real world](#) on computers (e.g. [games](#)), as real world is, as we know, three dimensional. 3D models can be created in several ways, e.g. manually with 3D modelling software (such as [Blender](#)) by 3D [artists](#), by 3D scanning real world objects, or automatically by [procedural generation](#).

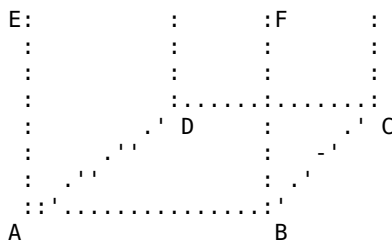
3D models can be represented in many ways -- the **mainstream "game" 3D models** that most people are used to seeing are polygonal (made of triangles) boundary-representation (recording only surface, not volume) [textured](#) (with "pictures" on their surface) 3D models, but be aware that many different ways are possible and used too, for example various volume representations, [voxel](#) models, [point clouds](#), [implicit surfaces](#), [spline surfaces](#), [constructive solid geometry](#), [wireframe](#) etc. Models may also bear additional extra information and features, e.g. bone rigs for animation, animation key frames, density information, [LODs](#) and so on.

TODO: classification, operations (subdivision, booleans, ...), texturing, animation, formats, ...

Example

Let's take a look at a simple polygonal 3D model. The following is a primitive, very [low poly](#) model of a house, basically just a cube with roof:





In a computer it would firstly be represented by an array of vertices, e.g.:

```
-2 -2 -2 (A)
 2 -2 -2 (B)
 2 -2  2 (C)
 2 -2 -2 (D)
-2  2 -2 (E)
 2  2 -2 (F)
 2  2  2 (G)
 2  2 -2 (H)
 0  3  0 (I)
```

Along with triangles (specified as indices into the vertex array, here with letters):

```
ABC ACD      (bottom)
AFB AEF      (front wall)
BGC BFG      (right wall)
CGH CHD      (back wall)
DHE DEA      (left wall)
EIF FIG GIH HIE (roof)
```

We see the model consists of 9 vertices and 14 triangles. Notice that the order in which we specify triangles follows the rule that looking at the front side of the triangle its vertices are specified clockwise (or counterclockwise, depending on chosen convention) -- sometimes this may not matter, but many 3D engines perform so called backface culling, i.e. they only draw the front faces and there some faces would be invisible from the outside if their winding was incorrect, so it's better to stick to the rule if possible.

The following is our house model in obj format -- notice how simple it is (you can copy paste this into a file called house.obj and open it in Blender):

```
# simple house model
v 2.000000 -2.000000 -2.000000
v 2.000000 -2.000000 2.000000
v -2.000000 -2.000000 2.000000
v -1.999999 -2.000000 -2.000000
v 2.000001 2.000000 -2.000000
v 1.999999 2.000000 2.000000
v -2.000001 2.000000 2.000000
v -2.000000 2.000000 -2.000000
v -2.000001 2.000000 2.000000
v 0.000000 3.000000 0.000000
vn 1.0000 0.0000 0.0000
vn -0.0000 0.0000 1.0000
vn 0.0000 -1.0000 0.0000
vn 0.0000 0.0000 -1.0000
vn -1.0000 -0.0000 -0.0000
vn -0.0000 0.8944 0.4472
vn 0.4472 0.8944 0.0000
vn 0.0000 0.8944 -0.4472
vn -0.4472 0.8944 -0.0000
s off
f 6 2 5
f 2 1 5
f 6 9 3
f 3 2 6
f 4 1 3
f 2 3 1
f 5 1 8
```

```
f 4 8 1
f 8 4 9
f 4 3 9
f 9 6 10
f 6 5 10
f 8 10 5
f 8 9 10
```

And here is the same model again, now in collada format (it is an XML so it's much more verbose, again you can copy paste this to a file house.dae and open it in Blender):

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema" version="1.4.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- simple house model -->
  <asset>
    <contributor> <author>drummyfish</author> </contributor>
    <unit name="meter" meter="1"/>
    <up_axis>Z_UP</up_axis>
  </asset>
  <library_geometries>
    <geometry id="house-mesh" name="house">
      <mesh>
        <source id="house-mesh-positions">
          <float_array id="house-mesh-positions-array" count="30">
            2 2 -2 2 -2 -2 -2 -2 -2 -2 2 -2
            2 2 2 2 -2 2 -2 -2 2 -2 2 2
            -2 -2 2 0 0 3
          </float_array>
          <technique_common>
            <accessor source="#house-mesh-positions-array" count="10" stride="3">
              <param name="X" type="float"/>
              <param name="Y" type="float"/>
              <param name="Z" type="float"/>
            </accessor>
          </technique_common>
        </source>
        <vertices id="house-mesh-vertices">
          <input semantic="POSITION" source="#house-mesh-positions"/>
        </vertices>
        <triangles material="Material-material" count="14">
          <input semantic="VERTEX" source="#house-mesh-vertices" offset="0"/>
          <p>
            5 1 4 1 0 4 5 8 2 2 1 5 3 0 2 1 2 0 4 0 7
            3 7 0 7 3 8 3 2 8 8 5 9 5 4 9 7 9 4 7 8 9
          </p>
        </triangles>
      </mesh>
    </geometry>
  </library_geometries>
  <library_visual_scenes>
    <visual_scene id="Scene" name="Scene">
      <node id="house" name="house" type="NODE">
        <translate sid="location">0 0 0</translate>
        <rotate sid="rotationZ">0 0 1 0</rotate>
        <rotate sid="rotationY">0 1 0 0</rotate>
        <rotate sid="rotationX">1 0 0 0</rotate>
        <scale sid="scale">1 1 1</scale>
        <instance_geometry url="#house-mesh" name="house"/>
      </node>
    </visual_scene>
  </library_visual_scenes>
  <scene> <instance_visual_scene url="#Scene"/> </scene>
</COLLADA>
```

TODO: other types of models, texturing etcetc.

3D Modeling: Learning It And Doing It Right

WORK IN PROGRESS

Do you want to start 3D modeling? Or do you already know a bit about it and **just want some advice to get better?** Then let us share a few words of advice here.

Let us preface with mentioning the **hacker chad way of making 3D models**, i.e. the LRS way 3D models should ideally be made. Remember, **you don't need any program to create 3D models**, you can make 3D models perfectly fine without Blender or any similar program, and even without computers. Sure, a certain kind of highly artistic, animated, very high poly models will be very hard or impossible to make without an interactive tool like Blender, but you can still make very complex 3D models, such as e.g. that of a whole city, without any fancy tools. Of course people were making statues and similar kinds of "physical 3D models" for thousands of years -- sometimes it's actually simpler to make the model by hand out of clay and later scan it into the computer -- you may also make easily make a polygonal model out of paper, BUT even virtual 3D models can simply be made with pen and paper, it's just numbers, vertices and triangles, very manageable if you keep it simple and well organized. You can directly write the models in text formats like obj or collada. First computer 3D models were actually made by hand, just with pen and paper, because there were simply no computers fast enough to even allow real time manipulation of 3D models; back then the modellers simply measured positions of someone object's "key points" (vertices) in 3D space which can simply be done with tools like rulers and strings, no need for complex 3D scanners (but if you have a digital camera, you have a quite advanced 3D scanner already). They then fed the manually made models to the computer to visualize them, but again, you don't even need a computer to draw a 3D model, in fact there is a whole area called descriptive geometry that's all about drawing 3D models on paper and which was used by engineers before computers came. Anyway, you don't have to go as far as avoiding computers of course -- if you have a programmable computer, you already have the luxury which the first 3D artists didn't have, a whole new world opens up to you, you can now make very complex 3D models just with your programming language of choice. Imagine you want to make the said 3D model of a city just using the C programming language. You can first define the terrain as heightmap simply as a 2D array of numbers, then you write a simple code that will iterate over this array and converts it to the obj format (a very simple plain text 3D format, it will be like 20 lines of code) -- now you have the basic terrain, you can render it with any tool that can load 3D models in obj format (basically every 3D tool), AND you may of course write your own 3D visualizer, there is nothing difficult about it, you don't even have to use perspective, just draw it in orthographic projection (again, that will be probably like 20 lines of code). Now you may start adding houses to your terrain -- make a C array of vertices and another array of triangle indices, manually make a simple 3D model of a house (a basic shape will have fewer than 20 vertices, you can cut it out of paper to see what it will look like). That's your house geometry, now just keep making instances of this house and placing them on the terrain, i.e. you make some kind of struct that will keep the house transformation (its position, rotation and scale) and each such struct will represent one house having the geometry you created (if you later improve the house model, all houses will be updates like this). You don't have to worry about placing the houses vertically, their height will be computed automatically so they sit right on the terrain. Now you can update your model exporter to take into account the houses, it will output the obj model along with them and again, you can view this whole model in any 3D software or with your own tools. You can continue by adding trees, roads, simple materials (maybe just something like per triangle colors) and so on. This approach may actually even be superior for some projects just as scripting is superior to many GUI programs, you can collaborate on this model just like you can collaborate on any other text program, you can automatize things greatly, you'll be independent of proprietary formats and platforms etcetc. This is how 3D models would ideally be made.

OK, back to the mainstream now. Nowadays as a FOSS user you will most likely do 3D modeling with Blender -- we recommended it to start learning 3D modeling as it is powerful, free, gratis, has many tutorials etc. Do NOT use anything proprietary no matter what anyone tells you! Once you know a bit about the art, you may play around with alternative programs or approaches (such as writing programs that generate 3D models etc.). However **as a beginner just start with Blender**, which is from now on in this article the software we'll suppose you're using.

Start extremely simple and learn bottom-up, i.e. learn about fundamentals and low level concepts and start with very simple models (e.g. simple untextured low-poly shape of a house, box with a roof), keep creating more complex models by small steps. Do NOT fall into the trap of "quick and easy magic 3D modeling" such as sculpting or some "smart apps" without knowing what's going on at the low level, you'll

end up creating extremely ugly, inefficient models in bad formats, like someone wanting to create space rockets without learning anything about math or physics first. Remember to **practice, practice, practice** -- eventually you learn by doing, so try to make small projects and share your results on sites such as [opengameart](#) to get feedback and some mental satisfaction and reward for your effort. The following is an outline of possible steps you may take towards becoming an alright 3D artist:

1. **Learn what 3D model actually is, basic technical details about how a computer represents it and roughly how 3D rendering works.** It is EXTREMELY important to have at least some idea about the fundamentals, i.e. you should learn at least the following:
 - 3D models that are used today consist of **vertices and triangles** (though higher polygons are usually supported in modeling software, everything is broken down to triangles eventually), computers usually store arrays of vertices and triangles as indices pointing to the array of vertices. Triangles have **facing** (front and back side, determined by the order of its vertices). These 3D models only represent the boundary (not the volume). All this is called the model's **geometry**.
 - **Normals** are vectors "perpendicular to the surface", they can be explicitly modified and stored or computed automatically and they are extremely important because they say how the model interacts with light (they are used in shading of the model), i.e. which edges appear sharp or smooth. Normal maps are textures that can be used to modify normals to make the surface seem rough or otherwise deformed without actually modifying the geometry. You HAVE TO understand normals.
 - **Textures** are images (or similar image-like data) that can be mapped to the model surface to "paint it" (or give it other material properties). They are mapped to models by giving vertices texturing **UV coordinates**. To make textures you'll need some basics of 2D image editing (see e.g. [GIMP](#)).
 - 3D rendering (and also modeling) works with the concept of a **scene** in which a number of models reside, as well as a virtual camera (or multiple ones), lights and other objects. These objects have **transformations** (normally translation, rotation and scale, represented by matrices) and may form a hierarchy, so called scene graph (some objects may be parents of other objects, meaning the child transformations are relative to parents) etc.
 - A 3D renderer will draw the triangles the model consists of by applying **shading** to determine color of each pixel of the rasterized triangle. Shading takes into account besides others texture(s) of the model, its material properties and light falling on the model (in which the model normals play a big role). Shading can be modified by creating **shaders** (if you don't create custom shaders, some default one will be used).
 - Briefly learn about other concepts such as low/high poly modeling and basic **3D formats** such as OBJ and COLLADA (which features they support etc.), possible other models representations (voxels, point clouds, ...) etc.
2. **Manually create a few extremely simple low-poly untextured models**, e.g. that of a simple house, laptop, hammer, bottle etc. Keep the vertex and triangle count very low (under 100), make the model by MANUALLY creating every vertex and triangle and focus only on learning this low level geometry manipulation well (how to create a vertex, how to split an edge, how to rotate a triangle, ...), making the model conform to good practice and get familiar with tools you're using, i.e. learn the key binds, locking movement direction to principal axes, learn manipulating your 3D view, setting up the free/side/front/top view with reference images etc. Make the model nice! I.e. make it have correctly facing triangles (turn backface culling on to check this), avoid intersecting triangles, unnecessary triangles and vertices, remove all duplicate vertices (don't have multiple vertices with the same position), connect all that should be connected, avoid badly shaped triangles (e.g. extremely acute/long ones) etc. Also learn about normals and make them nice! I.e. try automatic normal generation (fiddle e.g. with angle thresholds for sharp/smooth edges), see how they affect the model look, try manually marking some edges sharp, try out smoothing groups etc. Save your final models in OBJ format (one of the simplest and most common formats supporting all you need at this stage). All this will be a lot to learn, that's why you must not try to create a complex model at this stage. You can keep yourself "motivated" e.g. by aiming for creating a low-poly model collection you can share at [opengameart](#) or somewhere :)
3. **Learn texturing** -- just take the models you have and try to put a simple texture on them by drawing a simple image, then unwrapping the UV coordinates and MANUALLY editing the UV map to fit on the model. Again the goal is to get familiar with the tools and concepts now; experiment with helpers such as unwrapping by "projecting from 3D view", using "smart" UV unwrap etc. Make the UV map nice! Just as model geometry, UV maps also have good practice -- e.g. you should utilize as many texture pixels as possible (otherwise you're wasting space in the image), watch out for color bleeding,

the mapping should have kind of "uniform pixel density" (or possibly increased density on triangles where more details is supposed to be), some pixels of the texture may be mapped to multiple triangles if possible (to efficiently utilize them) etc. Only make a simple diffuse texture (don't do PBR, material textures etc., that's too advanced now). Try out texture painting and manual texture creation in a 2D image program, get familiar with both.

4. **Learn modifiers and advanced tools.** Modifiers help you e.g. with the creation of symmetric models: you only model one side and the other one gets mirrored. Subdivide modifier will automatically create a higher poly version of your model (but you need to help it by telling it which sides are sharp etc.). Boolean operations allow you to apply set operations like unification or subtraction of shapes (but usually create a messy geometry you have to repair!). There are many tools, experiment and learn about their pros and cons, try to incorporate them to your modeling.
5. **Learn retopology and possibly sculpting.** Topology is an extremely important concept -- it says what the structure of triangles/polygons is, how they are distributed, how they are connected, which curves their edges follow etc. Good topology has certain rules (e.g. ideally only being composed of quads, being denser where the shape has more detail and sparser where it's flat, having edges so that animation won't deform the model badly etc.). Topology is important for efficiency (you utilize your polygon budget well), texturing and especially animation (nice deformation of the model). Creating more complex models is almost always done in the following two steps:
 - Creating the shape while ignoring topology, for example with sculpting (but also other techniques, e.g. just throwing shapes together). The goal is to just make the desired shape.
 - Retopology: creating a nice topology for the shape while keeping the shape unchanged. This is done by starting modeling from the start with the "stick to surface" option, i.e. whenever you create or move a vertex, it sticks to the nearest surface (surface of the created shape). Here you just try to create a new "envelope" on the existing shape while focusing on making the envelope's topology nice.
6. **Learn about materials and shaders.** At this point you may learn about how to create custom shaders, how to create transparent materials, apply multiple textures, how to make realistic skin, PBR shaders etc. You should at least be aware of basic shading concepts and commonly encountered techniques such as Phong shading, subsurface scattering, screen space effects etc. because you'll encounter them in shader editors and you should e.g. know what performance penalties to expect.
7. **Learn animation.** First learn about keyframes and interpolation and try to animate basic transformations of a model, e.g. animate a car driving through a city by keyframing its position and rotation. Then learn about animating the model's geometry -- first the simple, old way of morphing between different shapes (shape keys in Blender). Finally learn the hardest type of animation: skeletal animation. Learn about bones, armatures, rigging, inverse kinematics etc.
8. **Now you can go crazy** and learn all the uber features such as hair, physics simulation, NURBS surfaces, boob physics etc.

Don't forget to stick to LRS principles! This is important so that your models are friendly to good technology. I.e. even if "modern" desktops don't really care about polygon count anymore, still take the effort to optimize your model so as to not use more polygons than necessary! Your models may potentially be used on small, non-consumerist computers with software renderers and low amount of RAM. Low-poly is better than high-poly (you can still prepare your model for automatic subdivision so that obtaining a higher poly model from it automatically is possible). Don't use complex stuff such as PBR or skeletal animation unless necessary -- you should mostly be able to get away with a simple diffuse texture and simple keyframe morphing animation, just like in old games! If you do use complex stuff, make it optional (e.g. make a normal map but don't rely on it being used in the end).

Good luck with your modeling!

3d_rendering

3D Rendering

See also 3D modeling.

In computer graphics 3D rendering is the process of computing images which represent a projected view of 3D objects through a virtual camera.

There are many methods and algorithms for doing so differing in many aspects such as computation complexity, implementation complexity, realism of the result, representation of the 3D data, limitations of viewing and so on. If you are just interested in the realtime 3D rendering used in games nowadays, you are probably interested in GPU-accelerated 3D rasterization with APIs such as OpenGL and Vulkan.

LRS has a simple 3D rendering library called small3dlib.

Methods

As most existing 3D "frameworks" are harmful, a LRS programmer is likely to write his own 3D rendering system that suits his program best, therefore we should list some common methods of achieving 3D. Besides that, it's just pretty interesting to see what there is in the store.

A very important realization of a graphics programmer is that **3D rendering is to a great extent about faking** (especially the mainstream realtime 3D) -- it is an endeavor that seeks to produce something that looks somehow familiar to HUMAN sight specifically and so even though the methods are mathematical, the endeavor is really an art in the end, not dissimilar to that of a magician who searches for "smoke and mirrors" hacks to produce illusions for the audience. Reality is infinitely complex, we use nothing else but approximations and neglecting that rely on assumptions about human sight such as "60 FPS looks like smooth movement to human eye", "infrared spectrum is invisible", "humans can't tell a mirror reflection is a bit off", "inner corners are usually darker than flat surfaces", "no shadow is completely black because light scatters in the atmosphere" etc. Really 3D graphics is nothing but searching for what looks good enough, and deciding this relies on a SUBJECTIVE judgement of a human (and sometimes every individual). In theory -- if we had infinitely powerful computers -- we would just program in a few lines of electromagnetic equations and run the precise simulation of light propagating in 3D environment to produce an absolutely realistic result, but though some methods try to come close to said approach, we simply won't ever have infinitely powerful computers. For this we have to resort to a bit more ugly approach of identifying specific notable real-life phenomena individually (for example caustics, Fresnel, mirror reflections, refractions, subsurface scattering, metallicity, noise, motion blur and myriads of others) and addressing each one individually with special treatment, many times correcting and masking our imperfections (e.g. applying antialiasing because we dared to use a simplified model of light sampling, applying texture filtering because we dared to only use finite amount of memory for our data, applying postprocessing etc.).

Rendering spectrum: The book *Real-Time Rendering* mentions that methods for 3D rendering can be seen as lying on a spectrum, one extreme of which is *appearance reproduction* and the other *physics simulation*. Methods closer to trying to imitate the appearance try to simply focus on imitating the look of an object on the monitor that the actual 3D object would have in real life, without being concerned with *how* that look arises in real life (i.e. closer to the "faking" approach mentioned above) -- these may e.g. use image data such as photographs; these methods may rely on lightfields, photo textures etc. The physics simulation methods try to replicate the behavior of light in real life -- their main goal is to solve the rendering equation, still only more or less approximately -- and so, through internally imitating the same processes, come to similar visual results that arise in real world: these methods rely on creating 3D geometry (e.g. that made of triangles or voxels), computing light reflections and global illumination. This is often easier to program but more computationally demanding. Most methods lie somewhere in between these two extremes: for example billboards and particle systems may use a texture to represent an object while at the same time using 3D quads (very simple 3D models) to correctly deform the textures by perspective and solve their visibility. The classic polygonal 3D models are also usually somewhere in between: the 3D geometry and shading are trying to simulate the physics, but e.g. a photo texture mapped on such 3D model is the opposite appearance-based approach (PBR further tries to shift the use of textures more towards the *physics simulation* end).

With this said, let's now take a look at possible **classifications** of 3D rendering methods. As seen, there are many ways:

- by **order**:
 - ♦ **object order**: The method iterates on objects and draws object by object, one after another. This results in pixels being drawn to "random" places on the screen and possibly already

drawn pixels being overdrawn with new pixels (though this can be further reduced). Typically requires a frame buffer and double buffering, often also z-buffer (or sorting), i.e. requires a lot of memory. This method is also a bit ugly but typically also faster than the alternative, so it is prevailing nowadays.

- ♦ **image order:** The method iterates on screen pixels, typically going pixel by pixel from left to right, top to bottom, deciding the color of each pixel independently. May be easier to program and require less memory (no frame buffer is needed, see e.g. frameless rendering), however though parallelism is applicable here (many pixels may potentially be independently computed in parallel, speeding up rendering), the algorithms used (e.g. path tracing) often have to expensively simulate light behavior and so performance is still an issue.
- by **speed:**
 - ♦ **realtime:** Able to render at interactive FPS, typically used in games etc.
 - ♦ **offline:** Spends a lot of time (even many minutes) on rendering each frame with the goal to produce output of extreme quality, typically used to render 3D movies etc.
- by **relative limitation:**
 - ♦ **primitive/"pseudo3D"/2.5D/...:** Older methods that produce 3D views but had great limitations e.g. in camera degrees of freedom or possible environment geometry that was usually limited to a "2D sector map" (see e.g. Doom).
 - ♦ **full/"true" 3D:** The "new" way of 3D rendering that allows freely rotating camera, arbitrary 3D geometry etc. Though this still has limitations (as any computer approximation of reality), many people just call this the "true" 3D.
- by **approach** (sides of above mentioned rendering spectrum):
 - ♦ **appearance based:** Focuses on achieving desired appearance by any means necessary, faking, "cheating", not trying to stay physically correct. This is typically faster.
 - ♦ **physics simulation** (see also physically based rendering): Focuses on simulating the underlying physics of reality with high correctness so that we also get a very realistic result.
- by **main method/algorithm** (see also the table below):
 - ♦ **rasterization:** Appearance based object order methods further based on a relatively simple algorithm capable of drawing (rasterizing) a simple geometric shape (such as a triangle) which we then use to draw the whole complex 3D scene (composed of great many of triangles).
 - ♦ **ray casting/tracing:** Physics simulation image order methods further based on tracing paths of light in a manner that's closer to reality.
 - ♦ ...
- by **3D data** (vector vs raster classification applies here just as in 2D graphics):
 - ♦ **triangle meshes** (vector, and other boundary representations)
 - ♦ **voxels** (raster, and potentially other volumetric representations)
 - ♦ **point clouds**
 - ♦ **heightmaps**
 - ♦ **implicit surfaces**
 - ♦ **smooth surfaces** (e.g. NURBS)
 - ♦ **2D sectors** (e.g. Doom's BSP "pseudo 3D" rendering)
 - ♦ ...
- by **hardware:**
 - ♦ **software rendering:** Rendering only with CPU. This is typically slower as a CPU mostly performs sequential computation, eliminating the possible parallelism optimization, however the approach is more KISS and portable.
 - ♦ **GPU accelerated:** Making use of specialized graphics rendering hardware (GPU) that typically uses heavy parallelism to immensely speed up rendering. While this is the mainstream, extremely fast way of rendering, it is also greatly bloated while often being an overkill that greatly complicates programming and makes programs less portable, less future proof etc.
- by **realism** of output:
 - ♦ **photorealistic**
 - ♦ **stylized**, flat shaded, wireframe, ...
 - ♦ ...
- **hybrids:** Methods may be combined and/or lie in between different extremes, for example we may see a rasterizer 3D renderer that uses ray tracing to add detail (shadows, reflections, ...) to the scene, we may see renderers that allow triangle meshes as well as voxels etc. { One nice hybrid looking engine is e.g. Chasm: The Rift. ~drummyfish }
- ...

Finally a table of some common 3D rendering methods follows, including the most simple, most advanced and some unconventional ones. Note that here we talk about methods and techniques rather than algorithms, i.e. general approaches that are often modified and combined into a specific rendering algorithm. For example the traditional triangle rasterization is sometimes combined with raytracing to add e.g. realistic reflections. The methods may also be further enriched with features such as texturing, antialiasing and so on. The table below should help you choose the base 3D rendering method for your specific program.

The methods may be tagged with the following:

- *2.5D*: primitive 3D, often called pseudo 3D or fake 3D, having significant limitations e.g. in degrees of freedom of the camera
- *off*: slow method usually used for offline (non-realtime) rendering (even though they indeed may run in real time e.g. with the help of powerful GPUs)
- *IO* vs *OO*: image order (rendering by pixels) vs object order (rendering by objects)

method	notes
<u>3D raycasting</u>	<i>IO off</i> , shoots rays from camera
<u>2D raycasting</u>	<i>IO 2.5D</i> , e.g. <u>Wolf3D</u>
<u>AI image synthesis</u>	"just let AI magic do it"
<u>beamtracing</u>	<i>IO off</i>
<u>billboarding</u>	<i>OO</i>
<u>BSP rendering</u>	<i>2.5D</i> , e.g. <u>Doom</u>
<u>conetracing</u>	<i>IO off</i>
<u>"dungeon crawler"</u>	<i>OO 2.5D</i> , e.g. Eye of the Beholder
edge list, scanline, span rasterization	<i>IO</i> , e.g. <u>Quake 1</u>
ellipsoid rasterization	<i>OO</i> , e.g. Ecstatica
flat-shaded 1 point perspective	<i>OO 2.5D</i> , e.g. Skyroads
reverse raytracing (photon tracing)	<i>OO off</i> , inefficient
<u>image based rendering</u>	generally using images as 3D data
<u>light fields</u>	image-based, similar to holography
<u>mode 7</u>	<i>IO 2.5D</i> , e.g. F-Zero
<u>parallax scrolling</u>	<i>2.5D</i> , very primitive
<u>pathtracing</u>	<i>IO off</i> , Monte Carlo, high realism
<u>portal rendering</u>	<i>2.5D</i> , e.g. <u>Duke3D</u>
prerendered view angles	<i>2.5D</i> , e.g. Iridion II (GBA)
<u>raymarching</u>	<i>IO off</i> , e.g. with <u>SDFs</u>
<u>raytracing</u>	<i>IO off</i> , recursive 3D raycasting
segmented road	<i>OO 2.5D</i> , e.g. Outrun
<u>shear warp rednering</u>	<i>IO</i> , volumetric
<u>splatting</u>	<i>OO</i> , rendering with 2D blobs
<u>texture slicing</u>	<i>OO</i> , volumetric, layering textures
<u>triangle rasterization</u>	<i>OO</i> , traditional in GPUs
<u>voxel space rendering</u>	<i>OO 2.5D</i> , e.g. Comanche
<u>wireframe rendering</u>	<i>OO</i> , just lines

TODO: Rescue On Fractalus!

TODO: find out how build engine/slab6 voxel rendering worked and possibly add it here (from <http://advsys.net/ken/voxlap.htm> seems to be based on raycasting)

TODO: VoxelQuest has some innovative voxel rendering, check it out (<https://www.voxelquest.com/news/how-does-voxel-quest-work-now-august-2015-update>)

3D Rendering Basics For Nubs

If you're a complete noob and are asking what the essence of 3D is or just how to render simple 3Dish pictures for your game without needing a PhD, here's the very basics. Yes, you can use some 3D engine such as Godot that has all the 3D rendering preprogrammed, but you'll surrender to bloat, you won't really know what's going on and your ability to tinker with the rendering or optimizing it will be basically zero... AND you'll miss on all the fun :) So here we just foreshadow some concepts you should start with if you want to program your own 3D rendering.

The absolute basic thing in 3D is probably perspective, or the concept which says that "things further away look smaller". This is basically the number one thing you need to know and with which you can make simple 3D pictures, even though there are many more effects and concepts that "make pictures look 3D" and which you can potentially study later (lighting, shadows, focus and blur, stereoscopy, parallax, visibility/obstruction etc.). { It's probably possible to make something akin "3D" even without perspective, just with orthographic projection, but that's just getting to details now. Let's just suppose we need perspective. ~drummyfish }

If you don't have rotating camera and other fancy things, perspective is actually mathematically very simple, you basically just **divide the object's size by its distance from the viewer**, i.e. its Z coordinate (you may divide by some multiple of Z coordinate, e.g. by $2 * Z$ to get different field of view) -- the further away it is, the bigger number its size gets divided by so the smaller it becomes. This "dividing by distance" ultimately applies to all distances, so in the end even the details on the object get scaled according to their individual distance, but as a first approximation you may just consider scaling objects as a whole. Just keep in mind you should only draw objects whose Z coordinate is above some threshold (usually called a *near plane*) so that you don't divide by 0! With this "dividing by distance" trick you can make an extremely simple "3Dish" renderer that just draws sprites on the screen and scales them according to the perspective rules (e.g. some space simulator where the sprites are balls representing planets). There is one more thing you'll need to handle: visibility, i.e. nearer objects have to cover the further away objects -- you can do this by simply sorting the objects by distance and drawing them back-to-front (painter's algorithm).

Here is some "simple" C code that demonstrates perspective and draws a basic animated wireframe cuboid as ASCII in terminal:

```
#include <stdio.h>

#define SCREEN_W 50          // ASCII screen width
#define SCREEN_H 22         // ASCII screen height
#define LINE_POINTS 64      // how many points for drawing a line
#define FOV 8               // affects "field of view"
#define FRAMES 30           // how many animation frames to draw

char screen[SCREEN_W * SCREEN_H];

void showScreen(void)
{
    for (int y = 0; y < SCREEN_H; ++y)
    {
        for (int x = 0; x < SCREEN_W; ++x)
            putchar(screen[y * SCREEN_W + x]);

        putchar('\n');
    }
}

void clearScreen(void)
{
    for (int i = 0; i < SCREEN_W * SCREEN_H; ++i)
        screen[i] = ' ';
}

// Draws point to 2D ASCII screen, [0,0] means center.
int drawPoint2D(int x, int y, char c)
{
    x = SCREEN_W / 2 + x;
    y = SCREEN_H / 2 + y;
```

```

    if (x >= 0 && x < SCREEN_W && y >= 0 && y <= SCREEN_H)
        screen[y * SCREEN_W + x] = c;
}

// Divides coord. by distance taking "FOV" into account => perspective.
int perspective(int coord, int distance)
{
    return (FOV * coord) / distance;
}

void drawPoint3D(int x, int y, int z, char c)
{
    if (z <= 0)
        return; // at or beyond camera, don't draw

    drawPoint2D(perspective(x,z),perspective(y,z),c);
}

int interpolate(int a, int b, int n)
{
    return a + ((b - a) * n) / LINE_POINTS;
}

void drawLine3D(int x1, int y1, int z1, int x2, int y2, int z2, char c)
{
    for (int i = 0; i < LINE_POINTS; ++i) // draw a few points to form a line
        drawPoint3D(interpolate(x1,x2,i),interpolate(y1,y2,i),interpolate(z1,z2,i),c);
}

int main(void)
{
    int shiftX, shiftY, shiftZ;

    #define N 12 // side length
    #define C '*'

    // cuboid points:
    //      X              Y              Z
    #define PA -2 * N + shiftX, N + shiftY, N + shiftZ
    #define PB 2 * N + shiftX, N + shiftY, N + shiftZ
    #define PC 2 * N + shiftX, N + shiftY, 2 * N + shiftZ
    #define PD -2 * N + shiftX, N + shiftY, 2 * N + shiftZ
    #define PE -2 * N + shiftX, -N + shiftY, N + shiftZ
    #define PF 2 * N + shiftX, -N + shiftY, N + shiftZ
    #define PG 2 * N + shiftX, -N + shiftY, 2 * N + shiftZ
    #define PH -2 * N + shiftX, -N + shiftY, 2 * N + shiftZ

    for (int i = 0; i < FRAMES; ++i) // render animation
    {
        clearScreen();

        shiftX = -N + (i * 4 * N) / FRAMES; // animate
        shiftY = -N / 3 + (i * N) / FRAMES;
        shiftZ = 0;

        // bottom:
        drawLine3D(PA,PB,C); drawLine3D(PB,PC,C); drawLine3D(PC,PD,C); drawLine3D(PD,PA,C);

        // top:
        drawLine3D(PE,PF,C); drawLine3D(PF,PG,C); drawLine3D(PG,PH,C); drawLine3D(PH,PE,C);

        // sides:
        drawLine3D(PA,PE,C); drawLine3D(PB,PF,C); drawLine3D(PC,PG,C); drawLine3D(PD,PH,C);

        drawPoint3D(PA,'A'); drawPoint3D(PB,'B'); // corners
        drawPoint3D(PC,'C'); drawPoint3D(PD,'D');
        drawPoint3D(PE,'E'); drawPoint3D(PF,'F');
        drawPoint3D(PG,'G'); drawPoint3D(PH,'H');

        showScreen();

        puts("press key to animate");
    }
}

```

```

    getchar();
}

return 0;
}

```

One frame of the animation should look like this:

```

E*****F
* *               *** *
* **             *** *
* H*****G*      *
* *             *   *
* *             *   *
* *             *   *
* *             *   *
* *             *   *
* D*****C       *
* **             *** *
* *             *   *
* *             **  *
***             * * *
A*****B

```

press key to animate

PRO TIP: It also help if you learn a bit about photography because 3D usually tries to simulate cameras and 3D programmers adopt many terms and concepts from photography. At least learn the very basics such as focal length, pinhole camera, the "exposure triangle" (shutter speed, aperture, ISO) etc. You should know how focal length is related to FOV, what the "f number" means, how to use exposure settings to increase or decrease things like motion blur and depth of field, what HDR means etc.

Mainstream Realtime 3D

You may have come here just to learn about the typical realtime 3D rendering used in today's games because aside from research and niche areas this kind of 3D is what we normally deal with in practice. This is what this section is about.

These days "game 3D" means a GPU accelerated 3D rasterization done with rendering APIs such as OpenGL, Vulkan, Direct3D or Metal (the last two being proprietary and therefore shit) and higher level engines above them, e.g. Godot, OpenSceneGraph etc. The methods seem to be evolving to some kind of rasterization/pathtracing hybrid, but rasterization is still the basis.

This mainstream rendering uses an object order approach (it blits 3D objects onto the screen rather than determining each pixel's color separately) and works on the principle of **triangle rasterization**, i.e. 3D models are composed of triangles (or higher polygons which are however eventually broken down into triangles) and these triangles are projected onto the screen according to the position of the virtual camera and laws of perspective. Projecting the triangles means finding the 2D screen coordinates of each of the triangle's three vertices -- once we have thee coordinates, we draw (rasterize) the triangle to the screen just as a "normal" 2D triangle (well, with some asterisks).

Additionally things such as z-buffering (for determining correct overlap of triangles) and double buffering are used, which makes this approach very memory (RAM/VRAM) expensive -- of course mainstream computers have more than enough memory but smaller computers (e.g. embedded) may suffer and be unable to handle this kind of rendering. Thankfully it is possible to adapt and imitate this kind of rendering even on "small" computers -- even those that don't have a GPU, i.e. with pure software rendering. For this we e.g. replace z-buffering with painter's algorithm (triangle sorting), drop features like perspective correction, MIP mapping etc. (of course quality of the output will go down).

Also additionally there's a lot of bloat added in such as complex screen space shaders, pathtracing (popularly known as raytracing), megatexturing, shadow rendering, postprocessing, compute shaders etc. This may make it difficult to get into "modern" 3D rendering. Remember to keep it simple.

On PCs the whole rendering process is hardware-accelerated with a GPU (graphics card). GPU is a special hardware capable of performing many operations in parallel (as opposed to a CPU which mostly computes sequentially with low level of parallelism) -- this is ideal for graphics because we can for example perform mapping and drawing of many triangles at once, greatly increasing the speed of rendering (FPS). However this hugely increases the complexity of the whole rendering system, we have to have a special API and drivers for communication with the GPU and we have to upload data (3D models, textures, ...) to the GPU before we want to render them. Debugging gets a lot more difficult. So again, this is bloat, consider avoiding GPUs.

GPUs nowadays are no longer just focusing on graphics, but are kind of a general device that can be used for more than just 3D rendering (e.g. crypto mining, training AI etc.) and can no longer even perform 3D rendering completely by themselves -- for this they have to be programmed. I.e. if we want to use a GPU for rendering, not only do we need a GPU but also some extra code. This code is provided by "systems" such as OpenGL or Vulkan which consist of an API (an interface we use from a programming language) and the underlying implementation in a form of a driver (e.g. Mesa3D). Any such rendering system has its own architecture and details of how it works, so we have to study it a bit if we want to use it.

The important part of a system such as OpenGL is its **rendering pipeline**. Pipeline is the "path" through which data go through the rendering process. Each rendering system and even potentially each of its version may have a slightly different pipeline (but generally all mainstream pipelines somehow achieve rasterizing triangles, the difference is in details of how they achieve it). The pipeline consists of **stages** that follow one after another (e.g. the mentioned mapping of vertices and drawing of triangles constitute separate stages). A very important fact is that some (not all) of these stages are programmable with so called **shaders**. A shader is a program written in a special language (e.g. GLSL for OpenGL) running on the GPU that processes the data in some stage of the pipeline (therefore we distinguish different types of shaders based on at which part of the pipeline they reside). In early GPUs stages were not programmable but they became so as to give a greater flexibility -- shaders allow us to implement all kinds of effects that would otherwise be impossible.

Let's see what a typical pipeline might look like, similarly to something we might see e.g. in OpenGL. We normally simulate such a pipeline also in software renderers. Note that the details such as the coordinate system handedness and presence, order, naming or programmability of different stages will differ in any particular pipeline, this is just one possible scenario:

1. Vertex data (e.g. 3D model space coordinates of triangle vertices of a 3D model) are taken from a vertex buffer (a GPU memory to which the data have been uploaded).
2. **Stage: vertex shader**: Each vertex is processed with a vertex shader, i.e. one vertex goes into the shader and one vertex (processed) goes out. Here the shader typically maps the vertex 3D coordinates to the screen 2D coordinates (or normalized device coordinates) by:
 - multiplying the vertex by a model matrix (transforms from model space to world space, i.e. applies the model move/rotate/scale operations)
 - multiplying by view matrix (transforms from world space to camera space, i.e. takes into account camera position and rotation)
 - multiplying by projection matrix (applies perspective, transforms from camera space to screen space in homogeneous coordinates)
3. Possible optional stages that follow are tessellation and geometry processing (tessellation shaders and geometry shader). These offer possibility of advanced vertex processing (e.g. generation of extra vertices which vertex shaders are unable to do).
4. **Stage: vertex post processing**: Usually not programmable (no shaders here). Here the GPU does things such as clipping (handling vertices outside the screen space), primitive assembly and perspective divide (transforming from [homogeneous coordinates](homogeneous coordinates.md) to traditional cartesian coordinates).
5. **Stage: rasterization**: Usually not programmable, the GPU here turns triangles into actual pixels (or fragments), possibly applying backface culling, perspective correction and things like stencil test and depth test (even though if fragment shaders are allowed to modify depth, this may be postponed to later).
6. **Stage: pixel/fragment processing**: Each pixel (fragment) produced by rasterization is processed here by a pixel/fragment shader. The shader is passed the pixel/fragment along with its coordinates, depth and possibly other attributes, and outputs a processed pixel/fragment with a specific color.

Typically here we perform shading and texturing (pixel/fragment shaders can access texture data which are again stored in texture buffers on the GPU).

7. Now the pixels are written to the output buffer which will be shown on screen. This can potentially be preceded by other operations such as depth tests, as mentioned above.

TODO: example of specific data going through the pipeline

See Also

- 3d modeling
 - software rendering
 - autostereogram
-

42

42

"HAHAHAHAHAHAHAHAHAHHHAAA BAZINGA" --Sheldon fan

42 is an even integer with prime factorization of $2 * 3 * 7$. This number was made kind of famous (and later overused in pop culture to the point of completely destroying the joke) by Douglas Adams' book The Hitchhiker's Guide to the Galaxy in which it appears as the answer to the ultimate question of life, the Universe and everything (the point of the joke was that this number was the ultimate answer computed by a giant supercomputer over millions of years, but it was ultimately useless as no one knew the question to which this number was the answer).

If you make a 42 reference in front of a TBBT fan, he will shit himself.

See Also

- thrembo
 - foo (similarly overplayed "joke")
-

4chan

4chan

{ haha <https://lolwut.info/comp/4chan/4chan-g.html> ~drummyfish }

4chan (<https://4chan.org/>, also 4cuck) is the most famous image board, a website causing controversies by its low copyright and a place of great fun, trolling, toxicity and memes. As most image boards, 4chan has a nice, oldschool minimalist look, even though it contains shitty captchas for posting and the site's code is proprietary. The site tolerates a great amount of free speech up to the point of being regularly labeled "right-wing extremist site", though it actually censors a lot of stuff and bans for stupid reasons such as harmless pedo jokes are very common (speaking from experience) -- 4chan global rules for example PROHIBIT CRITICISING 4chan (LMAO, rule no. 8), doxxing and call for raids. Being a "rightist paradise" it is commonly seen as a rival to reddit, aka the pseudoleftist paradise -- both forums hate each other to death. The discussion style is pretty nice, there are many nice stories and memes (e.g. the famous greentexts) coming from 4chan but it can also be a hugely depressing place just due to the sheer number of retards with incorrect opinions.

Just as reddit consists of subcommunities known as subreddits, 4chan consists of different boards (just as other image boards), each with given discussion topic and rules. The most (in)famous boards are likely *politically incorrect* AKA /pol/, where most of the american school shooters hang around, and *random* AKA /b/, the most active board, which is just a shitton of meme shitposting, porn, toxicity, fun, trolling and retardedness.

For us the most important part of 4chan is the technology board known as /g/ (for technoloGEE). Browsing /g/ can bring all kinds of emotion, it's a place of relative freedom and somewhat beautiful chaos where all people from absolute retards to geniuses argue about important and unimportant things, brands, tech news and memes, and constantly advise each other to kill themselves. Sometimes the place is pretty toxic and not good for mental health, actually it is more of a rule than an exception.

UPDATE: As of 2022 /g/ became literally unreadable, ABANDON SHIP. The board became flooded with capitalists, cryptofascists, proprietary shills, productivity freaks and other uber retards, it's really not worth reading anymore. You can still read good old threads on archives such as <https://desuarchive.org/g/page/280004/>. Some other more relaxed boards such as /x/ may still be alright though.

Despite dwelling a bit underground -- or rather being isolated from normie "safespaces" by censorship -- 4chan has really been extremely significant for the whole Internet culture, whole books could be written about its history, culture, mechanism of its working and impact on the rest of the cyberspace; the "4chan experience" is one of the things that can't appropriately be described by words, it has to be lived. Just like reddit mixed some interesting concepts into a unique, yet more powerful combination that's more than a sum of its parts, so did 4chan -- yes, other boards are to be credited for this too, but 4chan is the flagship, the center of it all. Especially important seems to be the anonymity aspect, you never know who you are talking to, it's never clear if someone is trolling, serious, shilling, extremely dumb or something in between. There is no karma, no handles, no profile pictures, no upvotes (at best there are numbers of replies), no post history, no account age, you have to learn to judge people by other things, for example by the style of their talk, their knowledge of the lore and latest memes, by how they format their posts (e.g. the infamously hated empty lines), what images they attach (and what they're file names are), as these are the only clues. A thread on 4chan isn't something with a clear goal, you don't know if someone is asking a question because he wants a genuine answer or because he's just bored and wants to see funny answers, or if he's posting a bait and is trying to trigger others, so each discussion is a bit of a game, you're trying to guess what's going on. Also everything is temporary, every thread and image is deleted in a short time, which is an important factor too, everything is constantly in motion, people have to react quickly, there is no turning back, reactions are quick and genuine, if you miss something it's gone. Also the image memes themselves show how art (who cares if low) evolves if completely unrestrained, anyone can try to spawn a new meme or download anyone else's posted meme, repost it or modify it, copyright mostly de facto won't apply as the authors are unknown; bad works are filtered out while good ones remain simply by making others save them and keep reposting them, it's art without authors, separated from the people, evolving completely on its own, purely by its intrinsic attributes, unconstrained evolution at work right before our eyes -- this is a seriously scientifically interesting stuff.

Alternatives to 4chan: just check out other image boards like 8kun, anon.cafe, leftychan.net, wizardchan, soyjak.party, BAI, 1436chan (gopher) etc. Also check out other types of forums than image boards such as saidit.net, voat or encyclopedia dramatica forums. You won't have much success searching for these using Goolag.

{ Also check out <https://wiki.soyjaks.party>, it's a great place, CC0, tons of 4chan lore, kinda like encyclopedia dramatica 2. ~drummyfish }

See Also

- Encyclopedia Dramatica
- reddit
- bienvenido a internet
- something awful

aaron_swartz

Aaron Swartz

"I think all censorship should be deplored." --Aaron Swartz

Aaron Swartz (1986 - 2013) was an American jewish prodigy that did a lot of activism and played a big role in creation of Reddit (back then a big platform for free speech, nowadays hugely censored), RSS, Creative Commons, Markdown and other quite important things. His life story is quite sad as he killed himself by hanging at a young age (there are some conspiracy theories around it), people see this loss as even more tragic because he was so talented and could have done many great things. But don't be mistaken, he was also an American and an "entrepreneur", so a capitalist at least to some degree; do not follow people, appreciate their art and their ideas.

abstraction

Abstraction

Abstraction is an important concept in programming, mathematics and other fields of science, philosophy and art, which in simple words can be described as "viewing an issue from a distance", thinking in higher-level concepts, i.e. paying less attention to fine detail so that one can see the bigger picture. In programming for example we distinguish programming languages of high and low level of abstraction, depending on how close they are "to the hardware" (e.g. assembly being low level, JavaScript being high level); in art high abstraction means portraying and capturing things such as ideas, feelings and emotions with shapes that may seem "distant", not resembling anything concrete or familiar. We usually talk about different **levels of abstraction**, depending on the "distance" we take in viewing the issue at hand -- this concept may very well be demonstrated on sciences: particle physics researches the world at the lowest level of abstraction, in extreme close-up, for example by examining individual atoms that make up our brains, while biology resides at a higher level of abstraction, viewing the brain at the level of individual cells, and finally psychology shows a very high level of abstraction because it looks at the brain from great distance and just studies its behavior.

In mainstream programming education it is generally taught to "abstract as much as possible" because that's aligned with the capitalist way of technology -- high abstraction is easy to handle for incompetent programming monkeys, it helps preventing them from making damage by employing billions of safety mechanisms, it also perpetuates the cult of never stopping layering of the abstraction sandwich, creating bloat, hype, bullshit jobs, it makes computers slower, constantly outdated and so drives software consumerism. As with everything in capitalism, new abstractions are products hyped on grounds of immediate benefit: creating more comfort, being something new and "modern", increasing "productivity", lowering "barriers of entry" so that ANYONE CAN NOW BE A PROGRAMMER without even knowing anything about computers (try to imagine this e.g. in the field of medicine) etc. -- of course, long term negative effects are completely ignored. **This is extremely wrong.** It is basically why technology has been on such a huge downfall in the latest decades. Opposing this, LRS advocates to employ only as little abstraction as needed, so as to support minimalism. **Too much abstraction is bad.** For example a widely used general purpose programming language should basically only have as much abstraction as to allow portability, it should definitely NOT succumb high abstraction such as object obsessed programming.

In a more detailed view abstraction is not one-dimensional, we may abstract in different directions ("look at the issue from different angles"); for example functional, logic and object paradigms are different ways of programming languages abstracting from the low level, each one in different way. So the matter of abstracting is further complicated by trying to **choose the right abstraction** -- one kind of abstraction may work well for certain kinds of problems (i.e. solving these problems will become simple when applying this abstraction) but badly for other kinds of problems. The art of choosing right abstraction (model) is important e.g. in designing computer simulations -- if we want so simulate e.g. human society, do we simulate individual people in it or just societies as whole entities? Do we simulate wars as a simple dice roll or do we let individual soldiers fight their own battles? Do we represent roads as actual surfaces on top of which cars move according to laws of physics, or do we simplify to something like mathematical graph connecting cities with mere abstract lines, or something in between like a cellular automaton maybe? Do we consider beings living on a round planet, with possibilities like meteor impacts and space flights, or do we simply consider people living on a flat 2D sheet of paper? Similar though has come to designing games (another kind of simulation).

Let's take a look at a possible division of a computer to different levels of abstraction, from lowest to highest (keep in mind it's also possible to define the individual levels differently):

- **physics:** Computer is a collection of atoms and subatomic particles such as electrons, operating with terms such as energy, charge, spin or quantum effects.
- **electronic circuit:** Computer is an analog circuit in which electricity flows through wires and electronic components, operating with terms such as voltage, current, transistor, resistor or electronic interference.
- **logic circuit:** Computer is a binary digital circuit; this is abstracting electricity away, now we are only considering two possible values carried by the wires: 1s and 0s. Operating with terms such as logic gate, logic function, multiplexer or sequential circuit.
- **machine code/assembly:** Computer is a machine with a specific instruction architecture, executing an algorithm encoded as simple binary instructions, such as "add two numbers" or "write a number to memory", in a specific format that's different for different types of computers. Operating with terms such as CPU cycle, opcode, register, memory or interrupt.
- **low level portable language:** Computer is a machine capable of performing algorithms written in a structured language resembling human language and it's a machine that's essentially the same as other computers, even of different types, i.e. all computers can understand the same language (programs are portable), typically e.g. C. Operating with terms such as structured data type, procedure, signed/unsigned type, memory management etc.
- **high level language:** Computer is a machine capable of performing algorithms while handling many things (such as memory allocation or ensuring safety) automatically and dynamically (on-the-go) and understanding more complex and abstract descriptions of problems, allowing for very fast and comfortable programming in languages like Python or JavaScript. Operating with terms such as objects, dictionaries, pure functions and polymorphism.
- **very high level, artificial intelligence:** Computer is a machine capable of simulating human thinking and therefore able to lead a conversation with human, it can perform commands given in natural language and even reason and create on its own. Operating with terms such as training, data sets and ethics.

See Also

- <https://unixsheikh.com/articles/we-have-used-too-many-levels-of-abstractions-and-now-the-future-looks-ble>

acronym

Acronym

Acronym is an abbreviation of a multiple word term by usually appending the starting letters of each word to form a new word.

Here is a list of some acronyms:

- **AA** (anti aliasing)
- **AC** (alternating current, air conditioning)
- **AD** (anno domini)
- **ACID** (atomicity consistency isolation durability)
- **ACK** (acknowledgement)
- **ADSL** (asymmetric digital subscriber line)
- **AF** (as fuck)
- **AFAIK** (as far as I know)
- **AJAX** (asynchronous JavaScript and XML)
- **AFK** (away from keyboard)
- **ALU** (arithmetic logic unit)
- **AM** (amplitude modulation)
- **ANCAP** (anarcho capitalist)
- **ANPAC** (anarcho pacifist)
- **ANSI** (american national standards institute)
- **AO** (ambient occlusion)
- **API** (application programming interface)
- **ARM** (advanced RISC machines)
- **ARPANET** (advanced research projects agency network)

- **ASAP** (as soon as possible)
- **ASCII** (American standard code for information interchange)
- **ASM** (assembly)
- **ATM** (at the moment, automated teller machine)
- **B** (byte, bit)
- **B4** (before)
- **BAI** (bienvenido a internet)
- **BASH** (bourne again shell)
- **BASIC** (beginner all purpose symbolic instruction code)
- **BBC** (big black cock)
- **BBS** (bulletin board system)
- **BC** (bytecode, before Christ)
- **BCD** (binary coded decimal)
- **BDFL** (benevolent dictator for life)
- **BDSM** (bondage domination sadism masochism)
- **BF** (brainfuck)
- **BG** (background, bad game)
- **BGR** (blue green red)
- **BIOS** (basic input/output system)
- **BJ** (blow job)
- **BJT** (bipolar junction transistor)
- **BS** (bullshit)
- **BSD** (Berkeley software distribution)
- **BTFO** (blown the fuck out)
- **CAD** (computer aided design)
- **CAPTCHA** (completely automated public Turing test to tell computers and humans apart)
- **CC** (creative commons, C compiler)
- **CC0** (creative commons zero)
- **CD** (compact disc, change directory)
- **CEO** (chief executive officer)
- **CGI** (computer generated imagery)
- **CISC** (complex instruction set computer)
- **CLI** (command line interface)
- **CMOS** (complementary metal oxide semiconductor)
- **CMS** (content management system)
- **CMYK** (cyan magenta yellow key)
- **COMPSCI** (computer science)
- **CP** (child porn, copy)
- **CPU** (central processing unit)
- **CRC** (cyclic redundancy check)
- **CRT** (cathode ray tube)
- **CSG** (constructive solid geometry)
- **CSS** (cascading style sheet)
- **CSV** (comma separated values)
- **DAC** (digital analog converter)
- **DB** (database)
- **DC** (direct current)
- **DDOS** (distributed denial of service)
- **DDR** (double data rate)
- **DE** (desktop environment)
- **DHCP** (dynamic host configuration protocol)
- **DL** (download)
- **DMA** (direct memory access)
- **DMCA** (digital millennium copyright act)
- **DND** (dungeons & dragons, do not disturb)
- **DNS** (domain name system)
- **DOM** (document object model)
- **DOS** (disk operating system, denial of service)
- **DOTADIW** (do one thing and do it well)
- **DPI** (dots per inch)

- **DRAM** (dynamic RAM)
- **DRM** (digital restrictions management)
- **DRY** (don't repeat yourself)
- **DVD** (digital versatile disc)
- **ED** (Encyclopedia Dramatica)
- **EEPROM** (electronically erasable programmable ROM)
- **ELF** (executable and linkable format)
- **EMCAS** (editor macros)
- **ENIAC** (electronic numerical integrator and computer)
- **EOF** (end of file)
- **EOL** (end of line, end of life)
- **ERP** (erotic role play)
- **ESR** (Erik Steven Raymond)
- **EULA** (end user license agreement)
- **FAQ** (frequently asked questions)
- **FE** (frontend)
- **FET** (field effect transistor)
- **FFS** (for fuck's sake)
- **FIFO** (first in first out)
- **FLAC** (free lossless audio codec)
- **FLOSS** (free libre open source software)
- **FM** (frequency modulation)
- **FML** (fuck my life)
- **FORTRAN** (formula translation)
- **FOSH** (free and open source hardware)
- **FOSS** (free and open source software)
- **FSF** (free software foundation)
- **FP** (floating point)
- **FPGA** (field programmable gate array)
- **FPS** (frames per second, first person shooter)
- **FQA** (frequently questioned answers)
- **FS** (file system)
- **FTL** (faster than light)
- **FTP** (file transfer protocol)
- **FU** (fuck you)
- **FXAA** (full screen anti aliasing)
- **FYI** (for your information)
- **GB** (gigabyte/gigabit, GameBoy)
- **GBPS** (GB per second)
- **GCC** (GNU compiler collection)
- **GDB** (GNU debugger)
- **GI** (global illumination)
- **GIB** (gibibyte)
- **GIF** (graphics interchange format)
- **GIGO** (garbage in garbage out)
- **GIMP** (GNU image manipulation program)
- **GLUT** (OpenGL utility toolkit)
- **GNOME** (GNU network object model environment)
- **GNG** (GNG's Not GNU)
- **GNU** (GNU's Not Unix)
- **GOAT** (greatest of all time)
- **GPG** (GNU privacy guard)
- **GPGPU** (general purpose GPU)
- **GPL** (GNU General Public License)
- **GPLv2** (GPL version 2)
- **GPLv3** (GPL version 3)
- **GPS** (global positioning system)
- **GPU** (graphics processing unit)
- **GRUB** (grand unified boot loader)
- **GSM** (global system for mobile communication)

- **GTK+** (GIMP toolking)
- **GUI** (graphical user interface)
- **H8** (hate)
- **HD** (high definition)
- **HDD** (hard disc drive)
- **HDMI** (HD multimedia interface)
- **HW** (hardware)
- **HTML** (hypertext markup language)
- **HTTP** (hypertext transfer protocol)
- **HTTPS** (HTTP secure)
- **HURD** (hird of unix replacing demons)
- **HQ** (high quality)
- **HZ** (hertz)
- **IANA** (internet assigned number authority)
- **IANAL** (I am not a lawyer)
- **ICMP** (internet control message protocol)
- **IDC** (I don't care)
- **IDE** (integrated development environment)
- **IEEE** (institute for electrical and electronic)
- **IM** (instant messaging)
- **IMAP** (internet message access protocol)
- **IMHO** (in my honest opinion)
- **IMO** (in my opinion)
- **INB4** (in before)
- **IO** (input/output)
- **IOT** (internet of things)
- **IPS** (instructions per second)
- **IP** (internet protocol, intellectual property)
- **IPV4** (IP version 4)
- **IPV6** (IP version 6)
- **IRC** (internet relay chat)
- **IRL** (in real life)
- **ISA** (instruction set architecture)
- **ISO** (international organization for standardization)
- **ISP** (internet service provider)
- **ISS** (international space station)
- **IS** (information system)
- **IT** (information technology)
- **J2ME** (java 2 micro edition)
- **JB** (jailbait)
- **JDK** (java development kit)
- **JIT** (just in time)
- **JK** (just kidding)
- **JPEG/JPG** (joint photographic expert group)
- **JS** (JavaScript)
- **JSON** (JavaScript object notation)
- **K&R** (Kernighan and Ritchie)
- **KB** (kilobyte/kilobit)
- **KBPS** (KB per second)
- **KDE** (K desktop environment)
- **KEK** (a meme version of LOL coming from World Of Warcraft)
- **KHZ** (kilohertz)
- **KIB** (kibibyte)
- **KILL** (keep it Linux loser)
- **KISS** (keep it simple stupid)
- **KISP** (keep it simple perfect)
- **KLOC** (kilo LOC)
- **KKK** (ku klux klan)
- **KYS** (kill yourself)
- **LAMP** (linux apache mysql php)

- **LARP** (live action role play)
- **LAN** (local area network)
- **LCD** (liquid crystal display)
- **LED** (light emitting diode)
- **LER** (light emitting resistor)
- **LGBT** (lesbian gay bisexual trans)
- **LGBTQ** (lesbian gay bisexual trans queer)
- **LGPL** (lesser GPL)
- **LIFO** (last in first out)
- **LISP** (list processing)
- **LMAO** (laughing my ass off)
- **LOC** (lines of code)
- **LOL** (laughing out loud)
- **LQ** (low quality)
- **LRS** (less retarded software/society)
- **LSB** (least significant bit)
- **LUT** (lookup table)
- **MBR** (master boot record)
- **MHZ** (megahertz)
- **MIB** (mebibyte)
- **MIME** (multipurpose internet mail extension)
- **MIP** (multum in parvo)
- **MIPS** (millions of instructions per second)
- **MBPS** (MB per second)
- **MCU** (microcontroller unit)
- **MD** (markdown)
- **MFW** (my face when)
- **MMO** (massively multiplayer online)
- **MMX** (multimedia extension)
- **MMORPG** (MMO RPG)
- **MOSFET** (metal oxide semiconductor field effect transistor)
- **MOTD** (message of the day)
- **MPEG** (motion pictures experts group)
- **MR** (merge request)
- **MS/M\$** (Micro\$oft)
- **MSB** (most significant bit)
- **MSC** (master of science)
- **MSG** (message)
- **MUD** (multi user dungeon)
- **NAN** (not a number)
- **NASA** (national aeronautic and space administration)
- **NAT** (network address translation)
- **NC** (non commercial)
- **NEET** (not in education, employment or training)
- **NFT** (non-fungible token)
- **NGL** (not gonna lie)
- **NOP** (no operation)
- **NP** (nondeterministic polynomial)
- **NTFS** (NT file system)
- **OEM** (original equipment manufacturers)
- **OpenGL** (OpenGL)
- **OMG** (oh my god)
- **OO** (object oriented, object obsessed, object obfuscated)
- **OOP** (object oriented/obsessed programming)
- **OS** (operating system, open source)
- **OSS** (open source software)
- **OSI** (open source initiative)
- **P2P** (peer to peer)
- **PB** (petabyte, petabit, personal best)
- **PBR** (physically based rendering)

- **PC** (personal computer, political correctness)
- **PD** (public domain)
- **PDF** (portable document format)
- **PCM** (pulse code modulation)
- **PGP** (pretty good privacy)
- **PHD** (doctor of philosophy)
- **PID** (process ID)
- **PIN** (personal identification number)
- **PNG** (portable network graphics)
- **POP3** (post office protocol version 3)
- **POSIX** (portable operating system interface)
- **PPC** (power PC)
- **PR** (pull request)
- **PS** (Photoshop, Postscript, PlayStation)
- **PS2** (personal system 2)
- **PTHC** (preteen hardcore)
- **QOS** (quality of service)
- **RAID** (redundant array of inexpensive discs)
- **RAM** (random access memory)
- **RC** (release candidate)
- **RCL** (raycastlib)
- **REGEX** (regular expression)
- **RFC** (request for comments)
- **RGB** (red green blue)
- **RGBA** (red green blue alpha)
- **RISC** (reduced instruction set computer)
- **RIP** (rest in piece)
- **RLE** (run length encoding)
- **RMS** (Richard Matthew Stallman)
- **RN** (right now)
- **ROFL** (rolling on floor laughing)
- **ROM** (read-only memory)
- **RPG** (role playing game)
- **RPI** (Raspberry Pi)
- **RT** (real time)
- **RTFM** (read the fucking manual)
- **RTOS** (real time operating system)
- **S3L** (small3dlib)
- **SAAS** (software as a service)
- **SAASS** (service as a software substitute)
- **SAF** (smallabstractfish)
- **SBC** (single board computer)
- **SCL** (smallchesslib)
- **SD** (standard definition, secure digital)
- **SDF** (signed distance function)
- **SDK** (software development kit)
- **SDL** (simple directmedia layer)
- **SED** (smoke emitting diode)
- **SEO** (search engine optimization)
- **SFX** (sound effects)
- **SGML** (standard generalized markup language)
- **SHA** (secure hash algorithm)
- **SIG** (special interest group)
- **SIM** (subscriber identity module)
- **SIMD** (single instruction multiple data)
- **SLOC** (source lines of code)
- **SMS** (short message service)
- **SMTP** (simple mail transfer protocol)
- **SNTP** (simple network time protocol)
- **SOC** (system on a chip)

- **SQL** (structured query language)
- **SRAM** (static RAM)
- **SSAO** (screen space ambient occlusion)
- **SSD** (solid state drive)
- **SSH** (secure shell)
- **SSL** (secure socket layer)
- **STFU** (shut the fuck up)
- **SVG** (scalable vector graphics)
- **SW** (software)
- **TAS** (tool assisted speedrun)
- **TB** (terabyte, terabit)
- **TCC** (tiny C compiler)
- **TCP** (transmission control protocol)
- **TFT** (thin filter transistor)
- **TFW** (that face when)
- **TLA** (three letter acronym)
- **TM** (trademark, Turing machine)
- **TOS** (terms of service)
- **TTY** (teletype)
- **TUI** (text user interface)
- **UBI** (universal basic income)
- **UDP** (user datagram protocol)
- **UI** (user interface)
- **UML** (unified modeling language)
- **URI** (uniform resource identifier)
- **URL** (uniform resource locator)
- **USA** (united states of america)
- **USB** (universal serial bus)
- **UTC** (coordinated universal time)
- **UTF** (unicode transformation format)
- **UX** (user experience)
- **VCS** (version control system)
- **VOD** (video on demand)
- **VHS** (video home system)
- **VIM** (vi improved)
- **VFX** (visual effects)
- **VLAN** (virtual LAN)
- **VLIW** (very long instruction word)
- **VM** (virtual machine)
- **VPN** (virtual private network)
- **VPS** (virtual private server)
- **VRAM** (video RAM)
- **W3C** (world wide web consortium)
- **WAN** (wide area network)
- **WAP** (wireless application protocol)
- **WIFI** (wireless fidelity)
- **WOW** (World Of Warcraft)
- **WPA** (WIFI protected access)
- **WTF** (what the fuck)
- **WTFPL** (do what the fuck you want to public license)
- **WYSIWYG** (what you see is what you get)
- **WM** (window manager)
- **WWW** (world wide web)
- **XAML** (extensible application markup language)
- **XHTML** (extensible HTML)
- **XML** (extensible markup language)
- **YOLO** (you only live once)
- **ZOMG** (when you want to write OMG but accidentally also hit Z)

See Also

- [LRS dictionary](#)
-

ai

Artificial Intelligence

Artificial intelligence (AI) is an area of [computer science](#) whose effort lies in making [computers](#) simulate thinking of humans and possibly other biologically [living beings](#). This may include making computers play [games](#) such as [chess](#), compose music, paint pictures, understand and processing audio, images and text on high level of [abstraction](#) (e.g. translation between natural languages), making predictions about complex systems such as stock market or weather or even exhibit a general human-like behavior. Even though today's focus in AI is on [machine learning](#) and especially [neural networks](#), there are many other usable approaches and models such as "hand crafted" state tree searching algorithms that can simulate and even outperform the behavior of humans in certain specialized areas.

There's a concern that's still a matter of discussion about the dangers of developing a powerful AI, as that could possibly lead to a [technological singularity](#) in which a super intelligent AI might take control over the whole world without humans being able to seize the control back. Even though it's still likely a far future and many people say the danger is not real, the question seems to be about *when* rather than *if*.

By about 2020, "AI" has become a [capitalist buzzword](#). They try to put machine learning into everything just for that AI label -- and of course, for a [bloat monopoly](#).

By 2023 neural network AI has become extremely advanced in processing visual, textual and audio information and is rapidly marching on. Networks such as [stable diffusion](#) are now able to generate images (or modify existing ones) with results mostly indistinguishable from real photos just from a short plain language textual description. Text to video AI is emerging and already giving nice results. AI is able to write computer programs from plain language text description. Chatbots, especially the proprietary [chatGPT](#), are scarily human-like and can already carry on conversation mostly indistinguishable from real human conversation while showing extraordinary knowledge and intelligence -- the chatbot can for example correctly reason about advanced mathematical concepts on a level much higher above average human. AI has become [mainstream](#) and is everywhere, normies are downloading "AI apps" on their phones that do funny stuff with their images while spying on them. In games such as [chess](#) or even strategy video [games](#) neural AI has already been for years far surpassing the best of humans by miles.

See Also

- [artificial life](#)
-

algorithm

Algorithm

Algorithm (from the name of Persian mathematician Muhammad ibn Musa al-Khwarizmi) is an exact step-by-step description of how to solve some type of a problem. Algorithms are basically what [programming](#) is all about: we tell [computers](#), in very exact ways (with [programming languages](#)), how to solve problems -- we write algorithms. But algorithms don't have to be just computer programs, they are simply exact instruction for solving problems. Although maybe not as obvious, [mathematics](#) is also a lot about creating algorithms because it strives to give us exact instructions for solving problems -- a mathematical formula usually tells us what we have to do to compute something, so in a way it is an algorithm too.

Cooking recipes are commonly given as an example of a non-computer algorithm, though they rarely contain branching ("if condition holds then do...") and loops ("while a condition holds do ..."), the key features of algorithms. The so called wall-follower is a simple algorithm to get out of any [maze](#) which doesn't have any disconnected walls: you just pick either a left-hand or right-hand wall and then keep following it. You may

write a crazy algorithm basically for any kind of problem, e.g. for how to clean a room or how to get a girl to bed, but it has to be **precise** so that anyone can execute the algorithm just by blindly following the steps; if there is any ambiguity, it is not considered an algorithm; a vague, imprecise "hint" on how to find a solution (e.g. "the airport is somewhere in this general direction.") we rather call a heuristic. Heuristics are useful too and they may be utilized by an algorithm, e.g. to find a precise solution faster, but from programmer's point of view algorithms, the **PRECISE** ways of finding solutions, are the basics of everything.

Interesting fact: contrary to intuition there are problems that are mathematically proven to be unsolvable by any algorithm, see undecidability, but for most practically encountered problems we can write an algorithm (though for some problems even our best algorithms can be unusably slow).

Algorithms are mostly (possibly not always, depending on exact definition of the term) written as a **series of steps** (or "instructions"); these steps may be specific actions (such as adding two numbers or drawing a pixel to the screen) or **conditional jumps** to other steps ("if condition X holds then jump to step N, otherwise continue"). At the lowest level (machine code, assembly) computers cannot do anything more complex than that: execute simple instructions (expressed as 1s and 0s) and perform conditional jumps -- in this computers are quite dumb (their strength comes from being able to execute many instruction with extreme speed). These jumps can be used to create **branches** (in programming known as *if-then-else*) and **loops**. Branches and loops are together known as control structures -- they don't express a direct action but control which steps in the algorithm will follow. All in all, **any algorithm can be built just using these three basic constructs**:

- **sequence**: A series of steps, one after another. E.g. "write prompt, read number from input, multiply it by two, store it to memory".
- **selection** (branches, *if-then-else*): Two branches (blocks of instructions) preceded by a condition; the first branch is executed only if the condition holds, the second ("else") branch is executed only if the condition doesn't hold (e.g. "If user password is correct, log the user in, otherwise print out an error."). The second branch is optional (it may remain empty).
- **iteration** (loops, repetition): Sequence of steps that's repeated as long as certain condition holds (e.g. "As long as end of file is not reached, read and print out the next character from the file.").

Note: in a wider sense algorithms may be expressed in other (mathematically equivalent) ways than sequences of steps (non-imperative ways, see declarative languages), even mathematical equations are often called algorithms because they *imply* the steps towards solving a problem. But we'll stick to the common narrow meaning of algorithm given above.

Additional constructs can be introduced to make programming more comfortable, e.g. subroutines/functions (kind of small subprograms that the main program uses for solving the problem), macros (shorthand commands that represent multiple commands) or switch statements (selection but with more than two branches). Loops are also commonly divided into several types such as: counted loops, loops with condition at the beginning, loops with condition at the end and infinite loops (for, while, do while and while (1) in C, respectively) -- in theory there can only be one generic type of loop but for convenience programming languages normally offer different "templates" for commonly used loops. Similarly to mathematical equations, algorithms make use of variables, i.e. values which can change and which have a specific name (such as *x* or *myVariable*).

Practical programming is based on expressing algorithms via text, but visual programming is also possible: flowcharts are a way of visually expressing algorithms, you have probably seen some. Decision trees are special cases of algorithms that have no loops, you have probably seen some too. Even though some languages (mostly educational such as Snap) are visual and similar to flow charts, it is not practical to create big algorithms in this way -- serious programs are written as a text in programming languages.

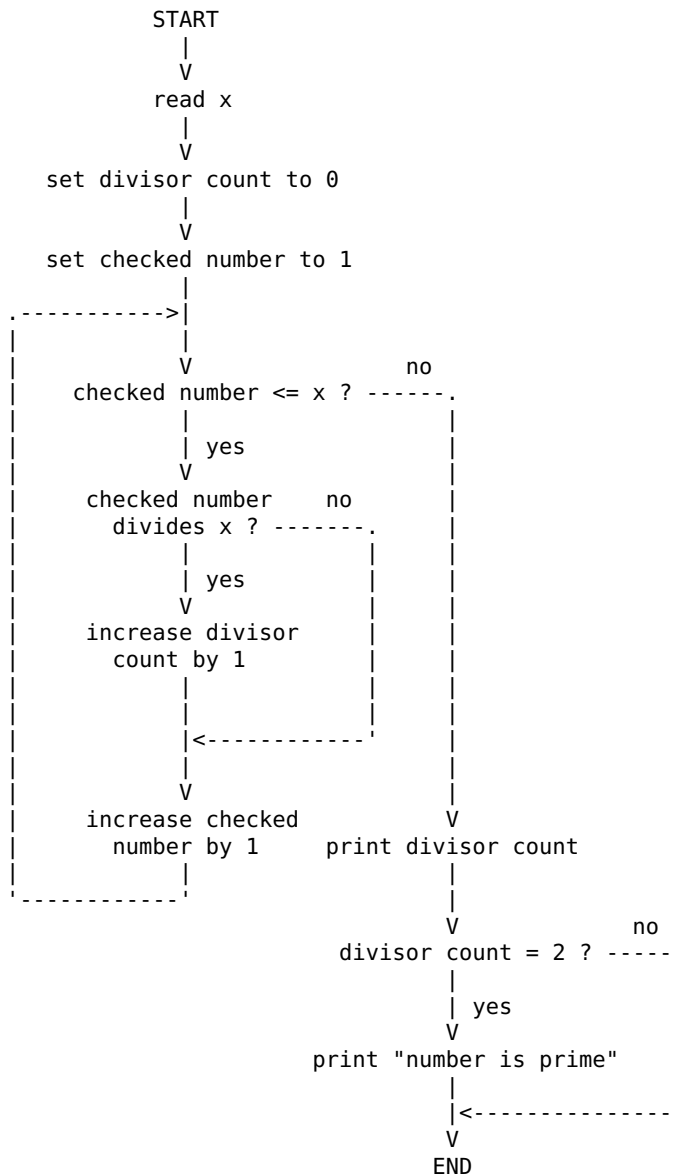
Example

Let's write a simple algorithm that counts the number of divisors of given number *x* and checks if the number is prime along the way. (Note that we'll do it in a naive, educational way -- it can be done better). Let's start by writing the steps in plain English (sometimes called pseudocode):

1. Read the number *x* from the input.
2. Set the *divisor counter* to 0.

3. Set *currently checked number* to 1.
4. While *currently checked number* is lower or equal than x :
 - a: If *currently checked number* divides x , increase *divisor counter* by 1.
 - b: Increase *currently checked number*.
5. Write out the *divisor counter*.
6. If *divisor counter* is equal to 2, write out the number is a prime.

Notice that x , *divisor counter* and *currently checked number* are variables. Step 4 is a loop (iteration) and steps a and 6 are branches (selection). The flowchart of this algorithm is:



This algorithm would be written in Python as:

```

x = int(input("enter a number: "))
divisors = 0

for i in range(1,x + 1):
    if x % i == 0: # i divides x?
        divisors = divisors + 1

print("divisors: " + str(divisors))

```

```
if divisors == 2:
    print("It is a prime!")
```

in C as:

```
#include <stdio.h>

int main(void)
{
    int x, divisors = 0;

    scanf("%d",&x); // read a number

    for (int i = 1; i <= x; ++i)
        if (x % i == 0) // i divides x?
            divisors = divisors + 1;

    printf("number of divisors: %d\n",divisors);

    if (divisors == 2)
        puts("It is a prime!");

    return 0;
}
```

and in comun as (for simplicity only works for numbers up to 9):

```
<- "0" - #read X and convert to number
0      # divisor count
1      # checked number

@@
$0 $3 > ?      # checked num. > x ?
!@
.

$2 $1 % 0 = ? # checked num. divides x ?
$1 ++ $:2     # increase divisor count
.

++           # increase checked number
.

0 "divisors: " -->    # write divisor count
$1 "0" + -> 10 ->

$1 2 = ?
0 "It is a prime" --> 10 ->
.
```

This algorithm is however not very efficient and could be optimized -- for example there is no need to check divisors higher than the square root of the checked value (mathematically above square root the only divisor left is the number itself) so we could lower the number of the loop iterations and so make the algorithm finish faster.

Study of Algorithms

Algorithms are the essence of computer science, there's a lot of theory and knowledge about them.

Turing machine, a kind of mathematical bare-minimum computer, created by Alan Turing, is the traditional formal tool for studying algorithms, though many other models of computation exist. From theoretical computer science we know not all problems are computable, i.e. there are problems unsolvable by any algorithm (e.g. the halting problem). Computational complexity is a theoretical study of resource consumption by algorithms, i.e. how fast and memory efficient algorithms are (see e.g. P vs NP).

Mathematical programming is concerned, besides others, with optimizing algorithms so that their time and/or space complexity is as low as possible which gives rise to algorithm design methods such as dynamic

programming (practical optimization is a more pragmatic approach to making algorithms more efficient). Formal verification is a field that tries to mathematically (and sometimes automatically) prove correctness of algorithms (this is needed for critical software, e.g. in planes or medicine). Genetic programming and some other methods of artificial intelligence try to automatically create algorithms (*algorithms that create algorithms*). Quantum computing is concerned with creating new kinds of algorithms for quantum computers (a new type of still-in-research computers). Programming language design is the art and science of creating languages that express computer algorithms well.

Specific Algorithms

Following are some well known algorithms.

- graphics
 - ◆ DDA: line drawing algorithm
 - ◆ discrete Fourier transform: extremely important algorithm expressing signals in terms of frequencies
 - ◆ Bresenham's algorithm: another line drawing algorithm
 - ◆ Midpoint algorithm: circle drawing algorithm
 - ◆ flood fill: algorithm for coloring continuous areas
 - ◆ FXAA
 - ◆ Hough transform: finds shapes in pictures
 - ◆ painter's algorithm
 - ◆ path tracing
 - ◆ ray tracing
 - ◆ ...
- math
 - ◆ Boot'h algorithm: algorithm for multiplication
 - ◆ Dijkstra's algorithm
 - ◆ Euclidean algorithm: computes greatest common divisor
 - ◆ numerical algorithms: approximate mathematical functions
 - ◆ sieve of Eratosthenes: computes prime numbers
 - ◆ ...
- sorting
 - ◆ bogosort (stupid sort)
 - ◆ bubble sort: simple, kind of slow but still usable sorting algorithm
 - ◆ heap sort
 - ◆ insertion sort
 - ◆ merge sort
 - ◆ shaker sort
 - ◆ selection sort
 - ◆ slow sort
 - ◆ quick sort: one of the fastest sorting algorithms
 - ◆ ...
- searching
 - ◆ binary search
 - ◆ linear search
 - ◆ ...
- other
 - ◆ A*: path searching algorithm, used by AI in many games
 - ◆ backpropagation: training of neural networks
 - ◆ fizzbuzz: problem/simple algorithm given in job interviews to filter out complete noobs
 - ◆ FFT: quickly converts signal (audio, image, ...) to its representation in frequencies, one of the most famous and important algorithms
 - ◆ Huffman coding: compression algorithm
 - ◆ Kalman filter
 - ◆ k-means: clustering algorithm
 - ◆ MD5: hash function
 - ◆ backtracking
 - ◆ minimax plus alpha-beta pruning: used by many AIs that play turn based games
 - ◆ proof of work algorithms: used by some cryptocurrencies

- ♦ [RSA](#)
- ♦ [Shor's algorithm](#): [quantum](#) factorization algorithm
- ♦ [YouTube algorithm](#): secret algorithm YouTube uses to suggest videos to viewers, a lot of people hate it :)
- ♦ ...

See Also

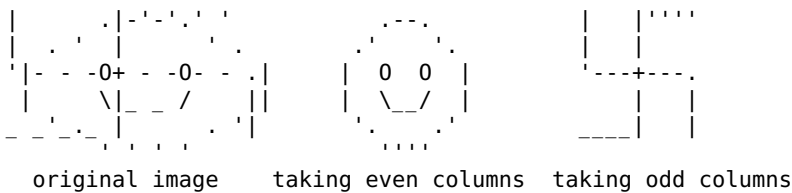
- [programming](#)
- [design pattern](#)
- [recursion](#)

aliasing

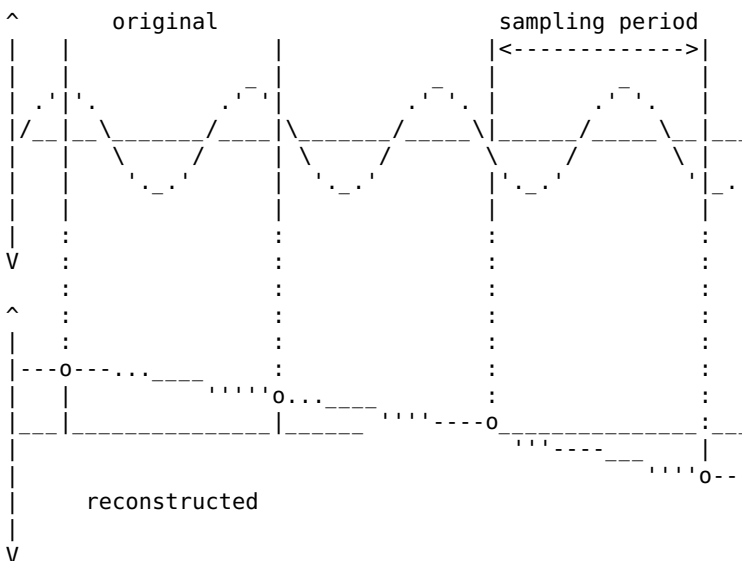
Aliasing

Aliasing is a certain typically undesirable phenomenon that distorts [signals](#) (such as sounds or images) when they are [sampled discretely](#) (captured at single points, usually at periodic intervals) -- this can happen e.g. when capturing sound with digital recorders or when [rendering](#) computer graphics. There exist [antialiasing](#) methods for suppressing or even eliminating aliasing. Aliasing can be often seen on small checkerboard patterns as a moiré pattern (spatial aliasing), or maybe more famously on rotating wheels or helicopter rotor blades that in a video look like standing still or rotating the other way (temporal aliasing, caused by capturing images at intervals given by the camera's [FPS](#)).

A simple example showing how sampling at discrete points can quite dramatically alter the recorded result:



The following diagram shows the principle of aliasing with a mathematical function:



The top signal is a [sine](#) function of a certain [frequency](#). We are sampling this signal at periodic intervals indicated by the vertical lines (this is how e.g. digital sound recorders record sounds from the real world). Below we see that the samples we've taken make it seem as if the original signal was a sine wave of a much lower frequency. It is in fact impossible to tell from the recorded samples what the original signal looked like.

Let's note that signals can also be two and more dimensional, e.g. images can be viewed as 2D signals. These are of course affected by aliasing as well.

The explanation above shows why a helicopter's rotating blades look to stand still in a video whose FPS is synchronized with the rotation -- at any moment the camera captures a frame (i.e. takes a sample), the blades are in the same position as before, hence they appear to not be moving in the video.

Of course this doesn't only happen with perfect sine waves. Fourier transform shows that any signal can be represented as a sum of different sine waves, so aliasing can appear anywhere.

Nyquist-Shannon sampling theorem says that aliasing can NOT appear if we sample with at least twice as high frequency as that of the highest frequency in the sampled signal. This means that we can eliminate aliasing by using a low pass filter before sampling which will eliminate any frequencies higher than the half of our sampling frequency. This is why audio is normally sampled with the rate of 44100 Hz -- from such samples it is possible to correctly reconstruct frequencies up to about 22000 Hz which is about the upper limit of human hearing.

Aliasing is also a common problem in computer graphics. For example when rendering textured 3D models, aliasing can appear in the texture if that texture is rendered at a smaller size than its resolution (when the texture is enlarged by rendering, aliasing can't appear because enlargement decreases the frequency of the sampled signal and the sampling theorem won't allow it to happen). (Actually if we don't address aliasing somehow, having lower resolution textures can unironically have beneficial effects on the quality of graphics.) This happens because texture samples are normally taken at single points that are computed by the texturing algorithm. Imagine that the texture consists of high-frequency details such as small checkerboard patterns of black and white pixels; it may happen that when the texture is rendered at lower resolution, the texturing algorithm chooses to render only the black pixels. Then when the model moves a little bit it may happen the algorithm will only choose the white pixels to render. This will result in the model blinking and alternating between being completely black and completely white (while it should rather be rendered as gray).

The same thing may happen in ray tracing if we shoot a single sampling ray for each screen pixel. Note that interpolation/filtering of textures won't fix texture aliasing. What can be used to reduce texture aliasing are e.g. by mipmaps which store the texture along with its lower resolution versions -- during rendering a lower resolution of the texture is chosen if the texture is rendered as a smaller size, so that the sampling theorem is satisfied. However this is still not a silver bullet because the texture may e.g. be shrink in one direction but enlarged in other dimension (this is addressed by anisotropic filtering). However even if we sufficiently suppress aliasing in textures, aliasing can still appear in geometry. This can be reduced by multisampling, e.g. sending multiple rays for each pixel and then averaging their results -- by this we **increase our sampling frequency** and lower the probability of aliasing.

Why doesn't aliasing happen in our eyes and ears? Because our senses don't sample the world discretely, i.e. in single points -- our senses integrate. E.g. a rod or a cone in our eyes doesn't just see exactly one point in the world but rather an averaged light over a small area (which is ideally right next to another small area seen by another cell, so there is no information to "hide" in between them), and it also doesn't sample the world at specific moments like cameras do, its excitation by light falls off gradually which averages the light over time, preventing temporal aliasing (instead of aliasing we get motion blur).

So all in all, **how to prevent aliasing?** As said above, we always try to satisfy the sampling theorem, i.e. make our sampling frequency at least twice as high as the highest frequency in the signal we're sampling, or at least get close to this situation and lower the probability of aliasing. This can be done by either increasing sampling frequency (which can be done smart, some methods try to detect where sampling should be denser), or by preprocessing the input signal with a low pass filter or otherwise ensure there won't be too high frequencies (e.g. using lower resolution textures).

altruism

Altruism

Not to be confused with autism.

Altruism means striving for the wellbeing of others, actively performing selfless actions. It is a purely good attitude which we, the LRS, fully embrace. It's no surprise that under capitalism, the rule of evil, altruism is commonly met with hostility or, in the better case, with ridicule.

Rightists often make an extremely funny reasoning error (probably on purpose) to justify their own fascist behavior; they claim that "altruism doesn't exist" because "altruism still seeks to satisfy one's ego and is therefore self interest". Well, firstly this is either wrong, as selflessness isn't defined by obtaining no reward but rather by acting in the interest of others without exploiting them, and secondly even if you define self interest conveniently in a way that makes your claim technically correct, it still completely misses the point! You can behave in a good or evil way, your definitions don't matter. No matter what words you use, you are just trying to excuse fascist behavior in a situation when you can choose to not behave like a fascist -- imagine someone shooting a child and justifying it like "well, I had to do it because I wanted that child's lollipop and I can't behave without self interest because I can't define selflessness".

anal_bead

Anal Bead

To most people anal beads are just sex toys they stick in their butts, however anal beads with with remotely controlled vibration can also serve as a well hidden one-way communication device. Use of an anal bead for cheating in chess has been the topic of a great cheat scandal in 2022 (Niemann vs Carlsen).

analog

Analog

Analog is the opposite of digital.

analytic_geometry

Analytic Geometry

Analytic geometry is part of mathematics that solves geometric problems with algebra; for example instead of finding an intersection of a line and a circle with ruler and compass, analytic geometry finds the intersection by solving an equation. In other words, instead of using pen and paper we use numbers. This is very important in computing as computers of course just work with numbers and aren't normally capable of drawing literal pictures and drawing results from them -- that would be laughable (or awesome?). Analytic geometry finds use especially in such fields as physics simulations (collision detections) and computer graphics, in methods such as raytracing where we need to compute intersections of rays with various mathematically defined shapes in order to render 3D images. Of course the methods are used in other fields, for example rocket science and many other physics areas. Analytic geometry reflects the fact that geometric and algebraic problem are often analogous, i.e. it is also the case that many times problems we encounter in arithmetic can be seen as geometric problems and vice versa (i.e. solving an equation is the same as e.g. finding an intersection of some N-dimensional shapes).

Fun fact: approaches in the opposite direction also exist, i.e. solving mathematical problems physically rather than by computation. For example back in the day when there weren't any computers to compute very difficult integrals and computing them by hand would be immensely hard, people literally cut physical function plots out of paper and weighted them in order to find the integral. Awesome oldschool hacking.

Anyway, **how does it work?** Typically we work in a 2D or 3D Euclidean space with Cartesian coordinates (but of course we can generalize to more dimensions etc.). Here, geometric shapes can be described with equations (or inequalities); for example a zero-centered circle in 2D with radius r has the equation $x^2 + y^2 = r^2$ (Pythagorean theorem). This means that the circle is a set of all points $[x,y]$ such that when substituted to the equation, the equation holds. Other shapes such as lines, planes, ellipses, parabolas have similar equations. Now if we want to find intersections/unions/etc., we just solve systems of multiple equations/inequalities and find solutions (coordinates) that satisfy all equations/inequalities at once. This

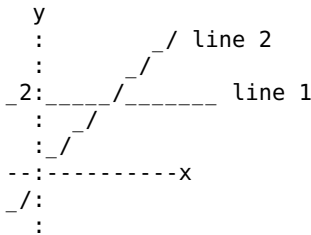
allows us to do basically anything we could do with pen and paper such as defining helper shapes and so on. Using these tools we can compute things such as angles, distances, areas, collision points and much more.

Analytic geometry is closely related to linear algebra.

Examples

Nub example:

Find the intersection of two lines in 2D: one is a horizontal line with y position 2, the other is a 45 degree line going through the $[0,0]$ point in the positive x and positive y direction, like this:



The equation of line 1 is just $y = 2$ (it consists of all points $[x,2]$ where for x we can plug in any number to get a valid point on the line).

The equation of line 2 is $x = y$ (all points that have the same x and y coordinate lie on this line).

We find the intersection by finding such point $[x,y]$ that satisfies both equations. We can do this by plugging the first equation, $y = 2$, to the second equation, $x = y$, to get the x coordinate of the intersection: $x = 2$. By plugging this x coordinate to any of the two line equations we also get the y coordinate: 2. I.e. the intersection lies at coordinates $[2,2]$.

Advanced nub example:

Let's say we want to find, in 2D, where a line L intersects a circle C . L goes through points $A = [-3,0.5]$ and $B = [3,2]$. C has center at $[0,0]$ and radius $r = 2$.

The equation for the circle C is $x^2 + y^2 = 2^2$, i.e. $x^2 + y^2 = 4$. This is derived from Pythagorean theorem, you can either check that or, if lazy, just trust this. Equations for common shapes can be looked up.

One possible form of an equation of a 2D line is a "slope + offset" equation: $y = k * x + q$, where k is the tangent (slope) of the line and q is an offset. To find the specific equation for our line L we need to first find the numbers k and q . This is done as follows.

The tangent (slope) k is $(B.y - A.y) / (B.x - A.x)$. This is the definition of a tangent, see that if you don't understand this. So for us $k = (2 - 0.5) / (3 - -3) = 0.25$.

The number q (offset) is computed by simply substituting some point that lies on the line to the equation and solving for q . We can substitute either A or B , it doesn't matter. Let's go with A : $A.y = k * A.x + q$, with specific numbers this is $0.5 = 0.25 * -3 + q$ from which we derive that $q = 1.25$.

Now we have computed both k and q , so we now have equations for both of our shapes:

- circle C : $x^2 + y^2 = 4$
- line L : $y = 0.25 * x + 1.25$

Feel free to check the equations, substitute a few points and plot them to see they really represent the shapes (e.g. if you substitute a specific x shape to the line equation you will get a specific y for it).

Now to find the intersections we have to solve the above system of equations, i.e. find such couples (coordinates) $[x,y]$ that will satisfy both equations at once. One way to do this is to substitute the line

equation into the circle equation. By this we get:

$$x^2 + (0.25 * x + 1.25)^2 = 4$$

This is a quadratic equation, let's get it into the standard format so that we can solve it:

$$x^2 + 0.0625 * x^2 + 0.625 * x + 1.5625 = 4$$

$$1.0625 * x^2 + 0.625 * x - 2.4375 = 0$$

Note that this makes perfect sense: a quadratic equation can have either one, two or no solution (in the realm of real numbers), just as there can either be one, two or no intersection of a line and a circle.

Solving quadratic equation is simple so we skip the details. Here we get two solutions: $x_1 = 1.24881$ and $x_2 = -1.83704$. These are the x position of our intersections. We can further find also the y coordinates by simply substituting these into the line equation, i.e. we get the final result:

- intersection 1: $[1.24881, 1.5622025]$
- intersection 2: $[-1.83704, 0.79074]$

See Also

- [linear algebra](#)

anarchism

Anarchism

Anarchism (from Greek *an*, no and *archos*, ruler) is a socialist political philosophy rejecting any social hierarchy and oppression, most notably that of capitalism and state but also any other form, e.g. nationalism, identity fascism, hero culture etc. **Anarchism doesn't mean without rules, but without rulers**; despite popular misconceptions **anarchism is not chaos** -- on the contrary, it strives for a stable, ideal society of equal people who live in peace. It means **order without power**. Let's also stress that **anarchism is ALWAYS incompatible with and strongly opposes capitalism**, as it's sadly heard too many times from the mouth of common people they think anarchism to be something akin "true capitalism" or "free market" (people get very confused by abuse of the word "free") -- NO, capitalism and formal government are an anarchist's two most opposed ideas (and please do not be misled by attempts at deception e.g. by so called "anarcho capitalists"; such a term just tries to merge two fundamentally incompatible ideas, like for example "militant pacifist" or "communist capitalist"). The symbols of anarchism include the letter A in a circle and a black flag that for different branches of anarchism is diagonally split from bottom left to top right and the top part is filled with a color specific for that branch.

LRS is a truly anarchist movement, specifically anarcho pacifist and anarcho communist one. **True, purest anarchism is pacifist, communist and altruistic** as that's the perfect ideal of society without hierarchy. Other forms of anarchism that try to sneak in acceptance of concepts such as "justified violence" or some form of "market economy" are mostly just poses of teenage boys who really believe in capitalism but want to adopt a cool label of "anarchist".

A great many things about anarchism are explained in the text *An Anarchist FAQ*, which is free licensed and can be accessed e.g. at <https://theanarchistlibrary.org/library/the-anarchist-faq-editorial-collective-an-anarchist-faq-full>.

Anarchism is a wide term and encompasses many flavors such as anarcho communism, anarcho pacifism, anarcho syndicalism, anarcho primitivism or anarcho mutualism. Some of the branches disagree on specific questions, e.g. about whether violence is ever justifiable, or propose different solutions to issues such as organization of society, however **all branches of anarchism are socialist** and all aim for **elimination of social hierarchy** such as social classes created by wealth, jobs and weapons, i.e. anarchism opposes state (e.g. police having power over citizens) and capitalism (employers exploiting employees, corporations exploiting consumers etc.).

See Also

- anarch

Anarch

Anarch is a LRS/suckless, free as in freedom first person shooter game similar to Doom, written by drummyfish. It has been designed to follow the LRS principles very closely and set an example of how games, and software in general, should be written. It also tries to be compatible with principles of less retarded society, i.e. it promotes anarchism, anti-capitalism, pacifism etc.

{ Though retrospectively I can of course see many mistakes and errors about the game and though it's not nearly perfect, I am overall quite happy about how it turned out for what it is, it got some attention among the niche of suckless lovers and many people have written me they liked the game and its philosophy. Many people have ported it to their favorite platforms, some have even written me their own expansions of the game lore, tricks they found etc. If you're among them, thank you :) ~drummyfish }

[illegible]

[illegible]

Anarch has these features:

Technical Details

Anarch is written in C99.

The whole codebase (including raycastlib AND the assets converted to C array) has fewer than 15000 lines of code. Compiled binary is about 200 kB big, though with compression and replacing assets with procedurally generated ones (one of several Anarch mods) the size was made as low as 57 kB.

The music in the game is procedurally generated using bytebeat.

The game uses a tiny custom-made 4x4 bitmap font to render texts.

In the suckless fashion, mods are recommended to be made and distributed as patches.

The following is a retrospective look on what could have been done better, written by drummyfish (a potential project for someone might be to implement these and so make the game even more awesome):

- It might have been better to write it in C89 than C99 for better portability.
- Sound effects would likely be better procedurally generated just like music. It would be less code and data in ROM and the quality probably wouldn't be that much worse as the samples are shitty quality anyway.

- The palette used might have rather been RGB 332 instead of the custom palette, again less code and less data in ROM, though visual quality might suffer a bit and things like diminishing colors might require some extra code or look up tables (like in Doom).
- The python scripts for data conversion should be rewritten to C. Using python was just laziness.
- Raycastlib itself has some issues, but those should be addressed separately.
- The game is really a bit too hard (tho this can easily be changed in settings) and gameplay is not great (like explosions pushing player instantly, not too good, too few items, player often lacks ammo/health, ...), movement inertia could be in vanilla game to make it feel nicer etc.
- Some of the code is awkward, like SFG_recomputePlayerDirection was an attempt at optimization but it's probably optimization in a wrong place that does nothing, ...
- Some details like having separate arrays for different types of images -- there is no reason for that, it would be better to just have one huge array of all images; maybe even have ALL data in one huge array of bytes.
- ...

ancap

"Anarcho" Capitali\$m

Not to be confused with anarchism.

So called "anarcho capitalism" (ancap for short, not to be confused with anpac or any form of anarchism) is probably the worst, most retarded and most dangerous idea in the history of ever, and that is the idea of supporting capitalism absolutely unrestricted by a state or anything else. No one with at least 10 brain cells and/or anyone who has spent at least 3 seconds observing the world could come up with such a stupid, stupid idea. We, of course, completely reject this shit.

It has to be noted that "**anarcho capitalism**" is not real anarchism, despite its name. Great majority of anarchists strictly reject this ideology as any form of capitalism is completely incompatible with anarchism -- anarchism is defined as opposing any social hierarchy and oppression, while capitalism is almost purely based on many types of hierarchies (internal corporate hierarchies, hierarchies between companies, hierarchies of social classes of different wealth etc.) and oppression (employee by employer, consumer by corporation etc.). Why do they call it *anarcho* capitalism then? Well, partly because they're stupid and don't know what they're talking about (otherwise they couldn't come up with such an idea in the first place) and secondly, as any capitalists, they want to deceive and ride on the train of the *anarchist* brand -- this is not new, Nazis also called themselves *socialists* despite being the complete opposite.

The colors on their flag are black and yellow (this symbolizes shit and piss).

It is kind of another bullshit kind of "anarchism" just like "anarcha feminism" etc.

The Worst Idea In History

As if capitalism wasn't extremely bad already, "anarcho" capitalists want to get rid of the last mechanisms that are supposed to protect the people from corporations -- states. We, as anarchists ourselves, of course see states as eventually harmful, but they cannot go before we get rid of capitalism first. Why? Well, imagine all the bad things corporations would want to do but can't because there are laws preventing them -- in "anarcho" capitalism they can do them.

Firstly this means anything is allowed, any unethical, unfair business practice, including slavery, physical violence, blackmailing, rape, worst psychological torture, nuclear weapons, anything that makes you the winner in the jungle system. Except that this jungle is not like the old, self-regulating jungle in which you could only reach limited power, this jungle offers, through modern technology, potentially limitless power with instant worldwide communication and surveillance technology, with mass production, genetic engineering, AI and weapons capable of destroying the planet.

Secondly the idea of getting rid of a state in capitalism doesn't even make sense because **if we get rid of the state, the strongest corporation will become the state**, only with the difference that state is at least *supposed* to work for the people while a corporation is only by its very definition supposed to care

solely about its own endless profit on the detriment of people. Therefore if we scratch the state, McDonalds or Coca Cola or Micro\$oft -- whoever is the strongest -- hires a literal army and physically destroys all its competition, then starts ruling the world and making its own laws -- laws that only serve the further growth of that corporation such as that everyone is forced to work 16 hour shifts every day until he falls dead. Don't like it? They kill your whole family, no problem. 100% of civilization will experience the worst kind of suffering, maybe except for the CEO of McDonald's, the world corporation, until the planet's environment is destroyed and everyone hopefully dies, as death is what we'll wish for.

All in all, "anarcho" capitalism is advocated mostly by children who don't know a tiny bit about anything, by children who are being brainwashed daily in schools by capitalist propaganda, with no education besides an endless stream of ads from their smartphones, or capability of thinking on their own. However, these children are who will run the world soon. It is sad, it's not really their fault, but through them the system will probably come into existence. Sadly "anarcho" capitalism is already a real danger and a very likely future. It will likely be the beginning of our civilization's greatest agony. We don't know what to do against it other than provide education.

God be with us.

See Also

- [capitalism](#)
 - [libertarianism](#)
-

anpac

Anarcho Pacifism

Anarcho pacifism (anpac) is a form of [anarchism](#) that completely rejects any violence. Anarcho pacifists argue that since anarchism opposes hierarchy and oppression, we have to reject violence which is the greatest tool of oppression and establishing hierarchy. This would make it the one true purest form of anarchism. Anarcho pacifists use a black and white flag.

Historically anarcho pacifists such as [Leo Tolstoy](#) were usually religiously motivated for rejecting violence, however this stance may also come from logic or other than religious beliefs, e.g. the simple belief that violence will only spawn more violence ("eye for an eye will only make the whole world blind"), or pure unconditional love of life which one simply feels and chooses to follow without the need for any further justification.

We, [LRS](#), advocate anarcho pacifism. We see how violence can be a short term solution, even to preventing a harm of many, however from the long term perspective we only see the complete delegitimisation of violence as leading to a truly mature society. We realize a complete, 100% non violent society may be never achieved, but with enough education and work it will be possible to establish a society with absolute minimum of violence, a society in which firstly people grow up in a completely non violent environment so that they never accept violence, and secondly have all needs secured so that they don't even have a reason for using violence. We should at least try to get as close to this ideal as possible.

antialiasing

Antialiasing

Antialiasing (AA) means preventing [aliasing](#), i.e. distortion of signal (images, audio, video, ...) caused by discrete sampling. Most people think antialiasing stands for "smooth edges in video game graphics", however that's a completely inaccurate understanding of antialiasing: yes, one of the most noticeable effects of 3D graphics antialiasing for a common human is that of having smooth edges, but smooth edges are not the primary goal, they are not the only effect and they are not even the most important effect of antialiasing. Understanding antialiasing requires understanding what aliasing is, which is not a completely trivial thing to do (it's not the most difficult thing in the world either, but most people are just afraid of mathematics, so they prefer to stick with "antialiasing = smooth edges" simplification).

The basic **sum up** is following: aliasing is a negative effect which may arise when we try to sample (capture) continuous signals potentially containing high frequencies (the kind of "infinitely complex" data we encounter in real world such as images or sounds) in discrete (non-continuous) ways by capturing the signal values at specific points in time (as opposed to capturing integrals of intervals), i.e. in ways native and natural to computers. Note that the aliasing effect is mathematical and is kind of a "punishment" for our "cheating" which we do by trying to simplify capturing of very complex signals, i.e. aliasing has nothing to do with noise or recording equipment imperfections, and it may occur not only when recording real world data but also when simulating real world, for example during 3D graphics rendering (which simulates capturing real world with a camera). A typical example of such aliasing effect is a video of car wheels rotating very fast (with high frequency) with a relatively low FPS camera, which then seem to be rotating very slowly and in opposite direction -- a high frequency signal (fast rotating wheels) caused a distortion (illusion of wheels rotating slowly in opposite direction) due to simplified discrete sampling (recording video as a series of photographs taken at specific points in time in relatively low FPS). Similar undesirable effects may appear e.g. on high resolution textures when they're scaled down on a computer screen (so called Moiré effect), but also in sound or any other data. Antialiasing exploits the mathematical Nyquist-Shannon sampling theorem that says that aliasing cannot occur when the sampling frequency is high enough relatively to the highest frequency in the sampled data, i.e. antialiasing tries to prevent aliasing effects typically by either preventing high frequency from appearing in the sampled data (e.g. blurring textures, see MIP mapping) or by increasing the sampling frequency (e.g. multisampling). As a side effect of better sampling we also get things such as smoothly rendered edges etc.

Note that the word *anti* in antialiasing means that some methods may not prevent aliasing completely, they may just try to suppress it somehow. For example the FXAA (fast approximate antialiasing) method is a postprocessing algorithm which takes an already rendered image and tries to make it as if it was properly rendered in ways preventing aliasing, however it cannot be 100% successful as it doesn't know the original signal, all it can do is try to give us a good enough approximation.

How to do antialiasing? There are many ways, depending on the kind of data (e.g. the number of dimensions of the signal or what frequencies you expect in it) or required quality (whether you want to prevent aliasing completely or just suppress it). As stated above, most methods make use of the Nyquist-Shannon sampling theorem which states that **aliasing cannot occur if the sampling frequency is at least twice as high as the highest frequency in the sampled signal**. I.e. if you can make sure your sampling frequency is high enough relatively to the highest frequency in the signal, you will completely prevent aliasing -- you can do this by either processing the input signal with a low pass filter (e.g. blurring an image) or by increasing your sampling frequency (e.g. rendering at higher resolution). Some specific antialiasing methods include:

- **avoiding aliasing**: A pretty straightforward way :) Aliasing can be avoided e.g. simply by using low resolution textures as opposed to high resolution ones.
- **multisampling** (MSAA), **supersampling** (SSAA) etc.: Increasing sampling frequency, typically in computer graphics rendering. The specific methods differ by where and how they increase the number of samples (some methods increase sampling uniformly everywhere, some try to detect aliasing areas and only put more samples there etc). A simple (but expensive) way of doing this is rendering the image at higher resolution and then scaling it back down.
- **FXAA**: Cheating, approximation of antialiasing by postprocessing, usually in shaders, cheap but can be imperfect.
- **MIP mapping**: Way of preventing aliasing in rendering of scaled-down textures by having precomputed scaled-down antialiased versions of it.
- **anisotropic filtering**: Improved version of MIP mapping.
- **motion blur**: Temporal antialiasing in video, basically increasing the number of samples in the time domain.
- ...

antivirus_paradox

Antivirus Paradox

{ I think this paradox must have had another established name even before antiviruses, but I wasn't able to find anything. If you know it, let me know. ~drummyfish }

Antivirus paradox is the paradox of someone who's job it is to eliminate certain undesirable phenomenon actually having an interest in keeping this phenomenon existing so as to keep his job. A typical example is an antivirus company having an interest in the existence of dangerous viruses and malware so as to keep their business running; in fact antivirus companies themselves secretly create and release viruses and malware.

Cases of said behavior are common, e.g. the bind-torture-kill serial killer used to work as a seller of home security alarms who installed alarms for people who were afraid of being invaded by the bind-torture-killer, and then used his knowledge of the alarms to break into the houses -- a typical capitalist business. It is also a known phenomenon that many firefighters are passionate arsonists because society simply praises them for fighting fires (as opposed to rewarding them for the lack of fires).

In capitalism and similar systems requiring people to have jobs this paradox prevents progress, that is to say actual elimination of undesirable phenomena, hence capitalism and similar systems are anti-progress. And not only that, the system pressures people to artificially creating new undesirable phenomena (like a "lack of women in tech" and similar bullshit) just to create new bullshit jobs that "fight" this phenomena. In a truly good society where people are not required to have jobs and in which people aim to eliminate work this paradox largely disappears.

apple

Apple

"Think different: conform."

Apple is a terrorist organization and one of the biggest American computer fashion corporations, infamously founded by Steve Job\$, it creates and sells overpriced, abusive, highly consumerist proprietary electronic devices.

See also http://techrights.org/wiki/Apple%27s_Dark_Side.

app

App

App is a retarded capitalist name for application; it is used by soydevs, corporations and normalfaggots (similarly to how "coding" is used for programming). This word is absolutely unacceptable and is only to be used to mock these retards.

Anything called an "app" is expected to be bloat, badly designed and, at best, of low quality (and, at worst, malicious).

approximation

Approximation

Approximating means calculating or representing something with lesser than best possible precision -- estimating -- purposefully allowing some margin of error in results and using simpler mathematical models than the most accurate ones: this is typically done in order to save resources (CPU cycles, memory etc.) and reduce complexity so that our projects and analysis stay manageable. Simulating real world on a computer is always an approximation as we cannot capture the infinitely complex and fine nature of the real world with a machine of limited resources, but even within this we need to consider how much, in what ways and where to simplify.

Using approximations however doesn't have to imply decrease in precision of the final result -- approximations very well serve optimization. E.g. approximate metrics help in heuristic algorithms such as A*. Another use of approximations in optimization is as a quick preliminary check for the expensive precise algorithms: e.g. using bounding spheres helps speed up collision detection (if bounding spheres of two

objects don't collide, we know they can't possibly collide and don't have to expensively check this).

Examples of approximations:

- **Distances**: instead of expensive **Euclidean** distance ($\sqrt{dx^2 + dy^2}$) we may use **Chebyshev** distance ($dx + dy$) or **Taxicab** distance ($\max(dx, dy)$).
- **Engineering approximations** ("guesstimations"): e.g. **$\sin(x) = x$** for "small" values of x or **$\pi = 3$** (integer instead of float).
- **Physics engines**: complex triangle meshes are approximated with simple analytical shapes such as **spheres**, **cuboids** and **capsules** or at least **convex hulls** which are much easier and faster to deal with. They also approximate **relativistic** physics with **Newtonian**.
- **Addition/subtraction** (of integers) can sometimes be approximated with logical **OR/AND** operations, as they behave a bit similarly. This can be used e.g. for brightening/darkening of pixel colors in **332** or **565** format -- without the approximation addition of colors in these formats is very expensive (basically requires conversion to RGB, addition, clamping and a conversion back).
- **Square root/square** (integer) can be roughly approximated too. E.g. to get a quick "almost square" of a number you can try something like doubling each binary digit and shifting everything right, e.g. $101 \rightarrow 11001$ -- it's not very accurate but may be good enough e.g. for some graphics effects and may be especially effective as hardware implementation as it works instantly and uses literally no **logic gates** (you just reorder bits)! A bit improved version may construct a pair of digits from each digit as logical AND (upper bit) and logical OR (lower bit) of the bit with its lower neighbor (lowest bit may still just be doubled), e.g. $1101 \rightarrow 11010111$. Square root can similarly be roughly estimated by reducing each pair of bits with logical OR down to a single bit (e.g. $101100 \rightarrow 110$). { Dunno if this is actually used anywhere, I came up with this once before I fell asleep. ~drummyfish } A famous hack in Quake, called *fast inverse square root*, uses a similar approximation in **floating point**.
- **3D graphics** is almost completely about approximations, e.g. basically all shapes are approximated with triangle meshes, **screen space** effects (like **SSAO**) are used to approximate global illumination, reflections etc. Similarly **ray tracing** neglects indirect lighting etcetc.
- **Real numbers** are practically always approximated with **floating point** or **fixed point** (rational numbers).
- **Numerical methods** offer generality and typically yield approximate solutions while their precision vs speed can be adjusted via parameters such as number of iterations.
- **Taylor series** approximates given mathematical function and can be used to e.g. estimate solutions of **differential equations**.
- Primitive **music** synthesis often uses simple functions like triangle/saw/square wave to approximate **sin** waves (though many times it's done for the actual sound of these waves, sometimes it may be simply to save on resources).
- ...

arch

Arch Linux

"BTW I use Arch"

Arch Linux is a **rolling-release Linux** distribution for the "tech-savvy", mostly fedora-wearing weirdos.

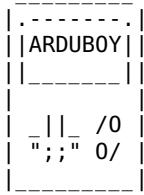
Arch is **shit** at least for two reasons: it has proprietary packages (such as **discord**) and it uses **systemd**. **Artix** Linux is a fork of Arch without systemd.

arduboy

Arduboy

Arduboy is an **Arduino**-based, extremely tiny indie/"retro" handheld **open gaming console**, about the size of a credit card, with monochrome (black&white) display; it was one of the earliest during the open console boom and is at the same time among the best in many aspects (construction, hardware, community, games, price, ...). Not only is it one of the top open consoles out there, it is also one of the most **minimalist** and a great way

to get into low level programming, learning C, embedded development etc. Even for a normie not intending to program it it's just a super cool toy to play old nostalgic games on and flex with around friends. We can really recommend getting Arduboy ASAP to anyone remotely interested in this kind of stuff. Arduboy is a bit similar to the original Gamebuino (the one with monochrome screen), it may have been inspired by it. The official website is <https://www.arduboy.com/>.



Hasted Arduboy ASCII art.

Arduboy has a number of hack/mods, both official and unofficial, see e.g. Arduboy FX (version that comes with memory for games so one doesn't have to use external PC to switch between them) or Arduboy Mini (yet physically smaller version).

{ Let's make it clear I AM NOT PAID for this article :D Reading through it it sounds like I'm shilling it super hard but it's really just that I like Arduboy, it was my first open console and I owe it my change in direction in programming. Of course, Arduboy does have its flaws, it's still something designed for profit, it shills open soars, forums need JavaScript and it's very possible it will become more spoiled in the future, however at the moment it's something quite nice with the amount of capitalist bullshit being still tolerable. That may change but as of writing this it still seems to be so. ~drummyfish }

Arduboy is not very expensive, one can even build it at home, there is documentation. The one you buy has excellent construction, it doesn't suffer from any shortcomings often seen in other such devices (hard to press buttons, display visibility angles, ...), the monochrome display is tiny but very good, with great contrast, it is joy to just look at; some people even managed to "fake" multiple shades of gray by fast pixel flickering. Seeing games on this display is somehow magical.

As can be judged from the name, Arduboy is based on Arduino (the famous free hardware platform), seems like Arduino Leonardo. The console is very small not only physically but also by its hardware specification, and that even compared to other open consoles -- Arduboy only has **2.5 KB of RAM** which is really low, however this is good for learning good programming and testing minimalist software; less is more. Besides this there is 1 KB of EEPROM (for things like game saves, ...) and 32 KB of flash memory (memory for the program itself -- as RAM is so low, one often tries to store data here). The CPU is 8bit ATmega32u4 clocked at **16 MHz** (also not very much, good for minimalism). It's an AVR CPU which has a **Harvard architecture**, i.e. program resides in different memory than the data; this can be something new to PC programmers as you have to think what (and how) to place into the program memory (flash) vs the RAM (as there is very little RAM); basically you have to use the `PROGMEM` macro to tell Arduino you want something placed in the flash (however that data will then be read-only; the whole `PROGMEM` thing can be a bit of annoyance, but in the end this is how small computers work). The vanilla version has no SD card. There are **6 buttons**: arrows, *A* and *B* and that's it -- one often has to think of how to make clever controls with this limited number of buttons (another kind of minimalist exercise). The display is **monochrome, 128x64 pixels**. For programming one typically uses Arduino IDE and the official Arduboy library (FOSS), but it's easy to do everything from the command line. There is also some kind of speaker for making beeps. Arduboy also has a good emulator, something that greatly helps with development which also isn't standard in the open console world.

Arduboy was designed by Kevin Bates through Kickstarter in 2015, He still keeps improving it and regularly gets involved in discussions on the official forums, that's pretty cool, he shares details and ideas about what he's currently doing, he gets involved in discussing hacks etc. The community is also very nice and greatly active itself -- yes, there are probably some rules on the forums, most people are absolute uber noobs, but it just seems like politics, furriness and similar poison just doesn't get discussed there, everyone just shares games, advice on programming etc., it's pretty bearable. In this it's similar to Pokitto -- there the community is also nice, active, with the creator getting involved a lot.

Games, mostly made by the users themselves, can either be found on the official website or in Erwin's Arduboy game collection (<https://arduboy.ried.cl/>) which as of writing this lists **something over 300 games** -- some are pretty impressive, like Arduventure, a pokemon-look-alike RPG game with quite huge world (very impressive for such tiny amount of memory), MicroCity, Catacombs of the Damned or various other similar raycasting 3Dish games. If you don't have Arduboy yet, you can play any game in the emulator (which even runs in the browser), however the experience of playing on the small device cannot indeed be replicated. The console can only hold one game at a time, so you always have to upload another one through USB if you want to switch, though the Arduboy FX mod addresses this by adding additional game memory to the console. Game progress is saved in EEPROM so you shouldn't lose your save states by uploading a new game (unless that game carelessly uses the same space in EEPROM for its own savestates). Great many of the games are FOSS, i.e. come with a free license or at least with the source code, only a minority has secret source code. Some LRS software run on Arduboy, e.g. SAF, so also games like microTD and smolchess, though for example Anarch is too much for this small hardware. Comun C interpreter also ran on Arduboy.

How free is it? Quite a lot, though probably not 100% (if spying is your concern then know this small thing probably won't even be capable of trying to hurt you this way). Arduboy Schematics are available { Not sure about license. ~drummyfish }, forums have tons of additional documentation and tutorials, Arduboy library is free licensed (BSD 3 clause), Arduino itself also uses free licenses (though of course it won't likely be free down to individual transistors...), games are very often free too. Being a minimalist computer there is a great deal of practical freedom. All in all, it's one of the most "overall" free things you can get.

art

Art

There is no indecency in art.

Art is an endeavor that seeks discovery and creation of beauty and primarily relies on intuition, its value is in feelings it gives rise to. While the most immediate examples of art that come to mind are for example music and painting, even the most scientific and rigorous effort like math and programming becomes art when pushed to the highest level, to the boundaries of current knowledge where intuition becomes important for further development.

Good art always needs time, usually a lot of time, and you cannot predict how much time it will need, **art cannot be made on schedule** or as a product. By definition creating true art is never a routine (though it requires well trained skills in routine tasks), it always invents something new, something no one has done before (otherwise it's just copying that doesn't need an artist) -- in this sense the effort is the same as that of research and science or exploring previously unwalked land, you can absolutely never know how long it will take you to invent something, what complications you will encounter or what you will find in an unknown land. You simply do it, fail many times, mostly find nothing, you repeat and repeat until you find the good thing. For this art also requires a lot of effort -- yes, there are cases of masterpieces that came to be very casually, but those are as rare as someone finding a treasure by accident. Art is to a great degree a matter of chance, trial and error, the artist himself doesn't understand his own creation when he makes it, he is only skilled at searching and spotting the good, but in the end he is just someone who invests a lot of time into searching, many times blindly.

See Also

- beauty
-

ascii_art

ASCII Art

ASCII art is the art of (mostly manually) creating graphics and images only out of fixed-width (monospace) ASCII characters. Strictly speaking this means no Unicode or extended ASCII characters are allowed -- these would rather be called Unicode art, ANSI art etc., though the term ASCII art is quite often used loosely for any art of this kind. If we keep being pedantic, ASCII art might also be seen as separate from mere ASCII

rendering, i.e. automatically rendering a bitmap image with ASCII characters in place of pixels, and ASCII graphics that utilizes the same techniques as ASCII art but can't really be called art (e.g. computer generated diagrams); though in practice this distinction is also rarely made. Pure ASCII art is plain text, i.e. it can't make use of color, text decoration and other rich text formatting.

This kind of art used to be a great part of the culture of earliest Internet and near-Internet (e.g. BBS) communities for a number of reasons imposed largely by the limitations of old computers -- it could be created easily with a text editor and saved in pure text format, it didn't take much space to store or send over a network and it could be displayed on text-only displays and terminals. The idea itself even predates computers, people were already making this kind of images with type writers, e.g. some poets were formatting their poems with typewriters to picture-shapes. Despite the technical limitations of displays having been overpassed, ASCII art survives even to present day and lives on in the hacker culture, among programmers, in Unix and "retro" game communities as well as on the Smol Internet, among people who just want to keep it simple and so on. ASCII diagram may very well be embedded in a comment on a text-only forum or in source code to explain some spatial concept. We, LRS, highly advocate use of ASCII art whenever it's good enough.

Here is a simple 16-shade ASCII palette (but watch out, whether it works will depend on your font):
 #0Vaxsf!c/!;,:.-. Another one can be e.g.: WM0KXkxocl;,:.'..

Here are approximate brightness values for each printable ASCII character, with 0 being black and 1000 white (of course the values always depend on the specific font you use):

{ I obtained the values by shooting a screen with some generic monospace font in gedit or something, then made a script that computed the values and ordered them. ~drummyfish }

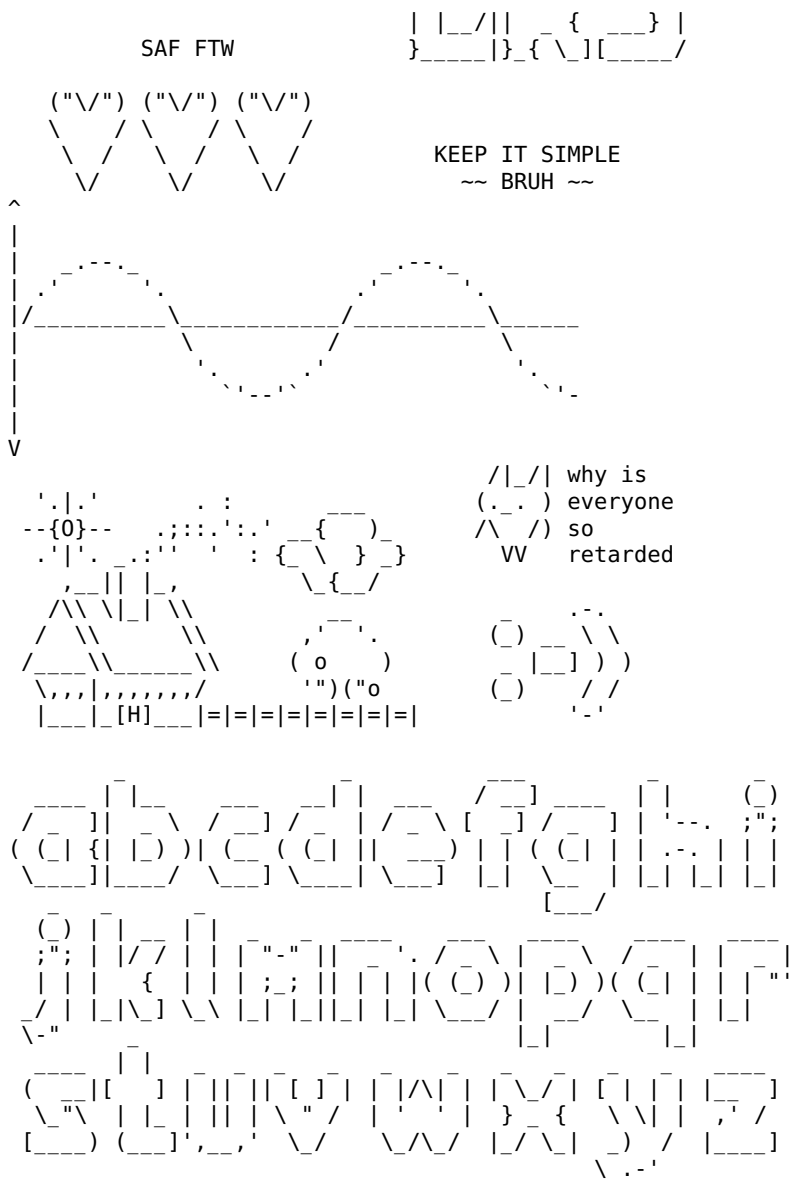
@ 577	P 727	A 777	? 827	/ 867
W 615	w 735	Z 777	I 830	> 867
M 640	3 740	h 779	j 831	\ 867
0 641	X 741	Y 786	C 834	< 868
Q 656	D 744	[797	() 836	c 870
& 658	V 745	T 797	(837	+ 874
% 664	b 746	e 798	l 837	J 892
R 664	p 747	} 800	x 838	" 911
8 676	5 748	a 800	i 848	; 912
# 685	d 748	{ 803	z 851	_ 912
0 685	2 750] 809	r 853	~ 924
\$ 687	4 750	y 810	^ 854	: 936
B 702	S 750	1 811	s 855	, 942
6 707	q 751	7 812	v 855	- 953
9 708	k 759	F 812	! 856	' 954
g 711	G 765	o 813	t 856	^ 968
N 715	K 767	f 815	* 857	` 969
U 724	E 768	u 825	= 860	1000
m 727	H 771	n 826	L 866	

And here are some attempts at actual ASCII art:

ASCII	tables
are	=)
the	
best	tables

```
[ ] [ ][ ][ ][ ]
[ ][ ][           ][ ]
[ ][             ][ ]
[ ]      XX      XX[ ]
[ ]          XXXX [ ]
[ ][                ][ ]
[ ][ ][            ][ ]
[ ]   [ ][ ][ ][ ]
```

$$\left[\begin{array}{c} \text{---} \\) \end{array} \right] \left\{ \begin{array}{c} \text{---} \\ | \end{array} \right\} \left(\begin{array}{c} \text{---} \\ / \end{array} \right) \backslash \begin{array}{c} \text{---} \\ \backslash \end{array}$$



The following is a raster picture auto-converted to ASCII: { For copyright clarity, it's one of my OGA pictures.
~drummyfish }

```

!.-
cLxVl.
c###xl,
,.- -,/ac/l!,
,,,,,;---!C.;-
-!;,,,cl;c##aV#s,...ca//cfs!;---,,,;
!;/s/,,c/slc;,,/#!as;!f/c;;cV0f0f;---.s#c-
-00fl,,,,;/,,,!ll.!;,,,!c/!;!/;cx0/.;a0##a/,,!,c;.
!;,,,,,!!/f/c!a,,,!//c;,,,,,;!cV0V/lf/.....l/;---
!./!/!/;/,/c! ;#s,///;,,,;!c/a#l;-.....,,-
;x/;,,,cslsac #a!,,,;/,,,casl,---..!cxfs/,,,!x#xl.....;/l,
-;,,,,;- s;!V.....;lca#/-...f#Vxcf#####s;.....;ssa.
xxl!,,,./c##l /c!,,,..c#/.;ac,.,---.,fs;a,
!0s;!!c,scfa##x- ---,.,,.,.###x#;.,!-!,-.lc/cfx/#f
x00s##l0##f;,,, ,-,.,.,.,;l#V;!/,,,;#-l#;!/,,,#!
- !##,.,.,. -ax;,,,,,f/...c//!./;-#,-##!,c! ,l;/-
/#c- -/, ##/;,,,,,;/,,,;lc1ll/ -.V##.,.,c! !.,;/
/##! -.;;#/,.,.,.,;l,.,.,sl,x### - ,#/,.,.,c; c,.,l/.
ls#,,,l.#f.,.,.ff/;,,,;0; ;l/s ---;#,,,;V ./l0cfl!.-
.lf#;,,,/!Vl,.,.,,xcacs/ ac/,-xf;,,,;# #/fff#l,
l/cflc/;.,al;lclx!, .f//---.,.,.,0 f/fcx##l.
;s///l-;f#;,,,;- s//f;,,,!,,,/ l#xf.c/!!
,;///!-;,,,c !///x.,.,,f- -c/f .,./

```

```

      ,fl!flll.,!/cc,      .ll//f;,,,;/;      !;;.    ,;/
      lf;!!0/;;c;l!fc      -fllclc/!!,,!l      -;;!-    ;;!
      ///cc;llVcl/f      #s/lccx;!/c/c.      .,lc!,    !;;
      ./,,,!c--.,,,,!      x//cf#fscf/c/f.      c,f;      ;;!
      -f///!!;-.-.,!,f      .cc!;/!f;///sf.      .fl!      !flls,
      - fxl;#;!V/fc/      l! ,!;;;c      ;l/lc-      ,#0lcs;
      - -!,-,l!;      x.;a!/cfc      ./ff;      !.-

```

{ TODO: what would ASCII art made of ASCII font look like??? ~drummyfish }

There are many tools for this, but for educational purposes this one was made using the following custom C program:

```

#include <stdio.h>

const char palette[] = "#0Vaxsflc/!;;,- ";

int readNum(void)
{
    int r = 0;

    while (1)
    {
        char c = getchar();

        if (c > '9' || c < '0')
            break;

        r = r * 10 + c - '0';
    }

    return r;
}

void skipAfterNewline(void)
{
    while (getchar() != '\n');
}

int main(void)
{
    skipAfterNewline(); // skip magic number
    skipAfterNewline(); // skip header

    int w = readNum(); // read width
    int h = readNum(); // read height

    skipAfterNewline(); // skip to pixels

    for (int j = 0; j < h; ++j) // read rows
    {
        for (int i = 0; i < w; ++i) // read columns
        {
            /* The following is a bit cryptic way of reading RGB, averaging it (giving
               higher significance to green, for human sight bias) to convert it to
               grayscale and then getting it to range 0 to 15 (palette size). */
            int v = ((getchar() + getchar() * 2 + getchar()) * 16) / (4 * 256);
            putchar(palette[v]);
        }

        putchar('\n');
    }

    return 0;
}

```

This program is extremely simple, it just reads an image in [PPM](#) format on standard input and outputs the image to terminal. Watch out, it won't work for all PPM images -- this one worked with a picture exported from [GIMP](#) in raw RGB PPM. Also note you have to scale the image down to a very small size AND its aspect ratio has to be highly stretched horizontally (because text characters, i.e. pixels, are much more tall than

wide). Also for best results you may want to mess with brightness, contrast, sharpness etc.

See Also

- [ANSI art](#)
 - [pixel art](#)
 - [plain text](#)
 - [cowsay](#)
 - [figlet](#)
-

ascii

ASCII

ASCII ([A](#)[m](#)[e](#)[r](#)[i](#)[c](#)[a](#)[n](#) standard code for information interchange) is a relatively simple standard for digital encoding of [text](#) that's one of the most basic and probably the most common format used for this purpose. For its simplicity and inability to represent characters of less common alphabets it is nowadays quite often replaced with more complex encodings such as [UTF-8](#) who are however almost always backwards compatible with ASCII (interpreting UTF-8 as ASCII will give somewhat workable results), and ASCII itself is also normally supported everywhere. ASCII is the [suckless/LRS/KISS](#) character encoding, recommended and [good enough](#) for most programs.

The ASCII standard assigns a 7 [bit](#) code to each basic text character which gives it a room for 128 characters -- these include lowercase and uppercase [English](#) alphabet, decimal digits, other symbols such as a question mark, comma or brackets, plus a few special control characters that represent instructions such as carriage return which are however often obsolete nowadays. Due to most computers working with 8 bit bytes, most platforms store ASCII text with 1 byte per character; the extra bit creates a room for **extending** ASCII by another 128 characters (or creating a variable width encoding such as [UTF-8](#)). These extensions include unofficial ones such as VISCII (ASCII with additional Vietnamese characters) and more official ones, most notably [ISO 8859](#): a group of standards by [ISO](#) for various languages, e.g. ISO 88592-1 for western European languages, ISO 8859-5 for Cyrillic languages etc.

The ordering of characters has been kind of cleverly designed to make working with the encoding easier, for example digits start with 011 and the rest of the bits correspond to the digit itself (0000 is 0, 0001 is 1 etc.). Corresponding upper and lower case letters only differ in the 6th bit, so you can easily convert between upper and lower case by negating it as `letter ^ 0x20`. { I think there is a few missed opportunities though, e.g. in not putting digits right before letters. That way it would be very easy to print hexadecimal (and all bases up to a lot) simply as `putchar('0' + x)`. UPDATE: seen someone ask this on some stack exchange, the answer said ASCII preferred easy masking or something, seems like there was some reason. ~drummyfish }

ASCII was approved as an [ANSI](#) standard in 1963 and since then underwent many revisions every few years. The current one is summed up by the following table:

dec	hex	oct	bin	symbol
000	00	000	0000000	NUL: null
001	01	001	0000001	SOH: start of heading
002	02	002	0000010	STX: start of text
003	03	003	0000011	ETX: end of text
004	04	004	0000100	EOT: end of stream
005	05	005	0000101	ENQ: enquiry
006	06	006	0000110	ACK: acknowledge
007	07	007	0000111	BEL: bell
008	08	010	0001000	BS: backspace
009	09	011	0001001	TAB: tab (horizontal)

dec	hex	oct	bin	symbol
010	0a	012	0001010	LF: new line
011	0b	013	0001011	VT: tab (vertical)
012	0c	014	0001100	FF: new page
013	0d	015	0001101	CR: carriage return
014	0e	016	0001110	SO: shift out
015	0f	017	0001111	SI: shift in
016	10	020	0010000	DLE: data link escape
017	11	021	0010001	DC1: device control 1
018	12	022	0010010	DC2: device control 2
019	13	023	0010011	DC3: device control 3
020	14	024	0010100	DC4: device control 4
021	15	025	0010101	NAK: not acknowledge
022	16	026	0010110	SYN: sync idle
023	17	027	0010111	ETB: end of block
024	18	030	0011000	CAN: cancel
025	19	031	0011001	EM: end of medium
026	1a	032	0011010	SUB: substitute
027	1b	033	0011011	ESC: escape
028	1c	034	0011100	FS: file separator
029	1d	035	0011101	GS: group separator
030	1e	036	0011110	RS: record separator
031	1f	037	0011111	US: unit separator
032	20	040	0100000	: space
033	21	041	0100001	!
034	22	042	0100010	"
035	23	043	0100011	#
036	24	044	0100100	\$
037	25	045	0100101	%
038	26	046	0100110	&
039	27	047	0100111	'
040	28	050	0101000	(
041	29	051	0101001)
042	2a	052	0101010	*
043	2b	053	0101011	+
044	2c	054	0101100	,
045	2d	055	0101101	-
046	2e	056	0101110	.
047	2f	057	0101111	/
048	30	060	0110000	0
049	31	061	0110001	1
050	32	062	0110010	2
051	33	063	0110011	3
052	34	064	0110100	4
053	35	065	0110101	5
054	36	066	0110110	6
055	37	067	0110111	7

dec	hex	oct	bin	symbol
-----	-----	-----	-----	--------

056	38	070	0111000	8
057	39	071	0111001	9
058	3a	072	0111010	:
059	3b	073	0111011	;
060	3c	074	0111100	<
061	3d	075	0111101	=
062	3e	076	0111110	>
063	3f	077	0111111	?
064	40	100	1000000	@
065	41	101	1000001	A
066	42	102	1000010	B
067	43	103	1000011	C
068	44	104	1000100	D
069	45	105	1000101	E
070	46	106	1000110	F
071	47	107	1000111	G
072	48	110	1001000	H
073	49	111	1001001	I
074	4a	112	1001010	J
075	4b	113	1001011	K
076	4c	114	1001100	L
077	4d	115	1001101	M
078	4e	116	1001110	N
079	4f	117	1001111	O
080	50	120	1010000	P
081	51	121	1010001	Q
082	52	122	1010010	R
083	53	123	1010011	S
084	54	124	1010100	T
085	55	125	1010101	U
086	56	126	1010110	V
087	57	127	1010111	W
088	58	130	1011000	X
089	59	131	1011001	Y
090	5a	132	1011010	Z
091	5b	133	1011011	[
092	5c	134	1011100	\
093	5d	135	1011101]
094	5e	136	1011110	^
095	5f	137	1011111	_
096	60	140	1100000	`: backtick
097	61	141	1100001	a
098	62	142	1100010	b
099	63	143	1100011	c
100	64	144	1100100	d
101	65	145	1100101	e

dec	hex	oct	bin	symbol
102	66	146	1100110	f
103	67	147	1100111	g
104	68	150	1101000	h
105	69	151	1101001	i
106	6a	152	1101010	j
107	6b	153	1101011	k
108	6c	154	1101100	l
109	6d	155	1101101	m
110	6e	156	1101110	n
111	6f	157	1101111	o
112	70	160	1110000	p
113	71	161	1110001	q
114	72	162	1110010	r
115	73	163	1110011	s
116	74	164	1110100	t
117	75	165	1110101	u
118	76	166	1110110	v
119	77	167	1110111	w
120	78	170	1111000	x
121	79	171	1111001	y
122	7a	172	1111010	z
123	7b	173	1111011	{
124	7c	174	1111100	`
125	7d	175	1111101	}
126	7e	176	1111110	~
127	7f	177	1111111	DEL

See Also

- [Unicode](#)
- [PETSCII](#)
- [ATASCII](#)
- [ASCII art](#)
- [base64](#)

assembly

Assembly

Assembly (also ASM) is, for any given hardware computing platform ([ISA](#), basically a [CPU](#) architecture), the lowest level [programming language](#) that expresses typically a linear, unstructured sequence of CPU instructions -- it maps (mostly) 1:1 to [machine code](#) (the actual [binary](#) CPU instructions) and basically only differs from the actual machine code by utilizing a more human readable form (it gives human friendly nicknames, or mnemonics, to different combinations of 1s and 0s). Assembly is converted by [assembler](#) into the the machine code, something akin a computer equivalent of the "[DNA](#)", the lowest level instructions for the computer. Assembly is similar to [bytecode](#), but bytecode is meant to be [interpreted](#) or used as an intermediate representation in [compilers](#) while assembly represents actual native code run by hardware. In ancient times when there were no higher level languages (like [C](#) or [Fortran](#)) assembly was used to write computer programs -- nowadays most programmers no longer write in assembly (majority of zoomer "[coders](#)" probably never even touch anything close to it) because it's hard (takes a long time) and not [portable](#), however programs written in assembly are known to be extremely fast as the programmer has

absolute control over every single instruction (of course that is not to say you can't fuck up and write a slow program in assembly).

Assembly is NOT a single language, it differs for every architecture, i.e. every model of CPU has potentially different architecture, understands a different machine code and hence has a different assembly (though there are some standardized families of assembly like x86 that work on wide range of CPUs); therefore **assembly is not portable** (i.e. the program won't generally work on a different type of CPU or under a different OS)! And even the same kind of assembly language may have several different syntax formats which may differ in comment style, order of writing arguments and even instruction abbreviations (e.g. x86 can be written in Intel and AT&T syntax). For the reason of non-portability (and also for the fact that "assembly is hard") you shouldn't write your programs directly in assembly but rather in a bit higher level language such as C (which can be compiled to any CPU's assembly). However you should know at least the very basics of programming in assembly as a good programmer will come in contact with it sometimes, for example during hardcore optimization (many languages offer an option to embed inline assembly in specific places), debugging, reverse engineering, when writing a C compiler for a completely new platform or even when designing one's own new platform. **You should write at least one program in assembly** -- it gives you a great insight into how a computer actually works and you'll get a better idea of how your high level programs translate to machine code (which may help you write better optimized code) and WHY your high level language looks the way it does.

OK, but why doesn't anyone make a portable assembly? Well, people do, they just usually call it a bytecode -- take a look at that. C is portable and low level, so it is often called a "portable assembly", though it still IS significantly higher in abstraction and won't usually give you the real assembly vibes. Forth may also be seen as close to such concept. ACTUALLY Dusk OS has something yet closer, called Harmonized Assembly Layer (see <https://git.sr.ht/~vdupras/duskos/tree/master/fs/doc/hal.txt>). Web assembly would also probably fit the definition.

The most common assembly languages you'll encounter nowadays are **x86** (used by most desktop CPUs) and **ARM** (used by most mobile CPUs) -- both are used by proprietary hardware and though an assembly language itself cannot (as of yet) be copyrighted, the associated architectures may be "protected" (restricted) e.g. by patents (see also IP cores). **RISC-V** on the other hand is an "open" alternative, though not yet so wide spread. Other assembly languages include e.g. AVR (8bit CPUs used e.g. by some Arduinos) and PowerPC.

To be precise, a typical assembly language is actually more than a set of nicknames for machine code instructions, it may offer helpers such as macros (something akin the C preprocessor), pseudoinstructions (commands that look like instructions but actually translate to e.g. multiple instructions), comments, directives, named labels for jumps (as writing literal jump addresses would be extremely tedious) etc.

Assembly is extremely low level, so you get no handholding or much programming "safety" (apart from e.g. CPU operation modes), you have to do everything yourself -- you'll be dealing with things such as function call conventions, interrupts, syscalls and their conventions, counting CPU cycles of individual instructions, looking up exact hexadecimal memory addresses, defining memory segments, dealing with endianness, raw goto jumps, call frames etc.

Note that just replacing assembly mnemonics with binary machine code instructions is not yet enough to make an executable program! More things have to be done such as linking libraries and converting the result to some executable format such as elf which contains things like header with metainformation about the program etc.

Typical Assembly Language

Assembly languages are usually unstructured, i.e. there are no control structures such as if or while statements: these have to be manually implemented using labels and jump (goto, branch) instructions. There may exist macros that mimic control structures. The typical look of an assembly program is however still a single column of instructions with arguments, one per line, each representing one machine instruction.

In assembly it is also common to blend program instructions and data, i.e. sometimes you create a label after which you just put bytes that will represent e.g. text strings or images and after that you start to write program instructions that work with these data, which will likely physically be placed this way (after the data)

in the final program. This may cause quite nasty bugs if you by mistake jump to a place where data reside and try to treat them as instructions.

The working of the language reflects the actual hardware architecture -- most architectures are based on registers so usually there is a small number (something like 16) of registers which may be called something like R0 to R15, or A, B, C etc. Sometimes registers may even be subdivided (e.g. in x86 there is an eax 32bit register and half of it can be used as the ax 16bit register). These registers are the fastest available memory (faster than the main RAM memory, they are literally INSIDE the CPU, even in front of the cache) and are used to perform calculations. Some registers are general purpose and some are special: typically there will be e.g. the FLAGS register which holds various 1bit results of performed operations (e.g. overflow, zero result etc.). Some instructions may only work with some registers (e.g. there may be kind of a "pointer" register used to hold addresses along with instructions that work with this register, which is meant to implement arrays). Values can be moved between registers and the main memory (with instructions called something like *move*, *load* or *store*).

Writing instructions works similarly to how you call a function in high level language: you write its name and then its arguments, but in assembly things are more complicated because an instruction may for example only allow certain kinds of arguments -- it may e.g. allow a register and immediate constant (kind of a number literal/constant), but not e.g. two registers. You have to read the documentation for each instruction. While in high level language you may write general expressions as arguments (like `myFunc(x + 2 * y, myFunc2())`), here you can only pass specific values.

There are also no complex data types, assembly only works with numbers of different size, e.g. 16 bit integer, 32 bit integer etc. Strings are just sequences of numbers representing ASCII values, it is up to you whether you implement null terminated strings or Pascal style strings. Pointers are just numbers representing addresses. It is up to you whether you interpret a number as signed or unsigned (some instructions treat numbers as unsigned, some as signed, some don't care because it doesn't matter).

Instructions are typically written as three-letter abbreviations and follow some unwritten naming conventions so that different assembly languages at least look similar. Common instructions found in most assembly languages are for example:

- **MOV** (move): move a number between registers and/or main memory (RAM).
- **JMP** (jump, also e.g. BRA for branch): unconditional jump to far away instruction.
- **JEQ** (jump if equal, also BEQ etc.): jump if result of previous comparison was equality.
- **ADD** (add): add two numbers.
- **NOP** (no operation): do nothing (used e.g. for delays or as placeholders).
- **CMP** (compare): compare two numbers and set relevant flags (typically for a subsequent conditional jump).
- ...

Fun note: HCF -- *halt and catch fire* -- is a humorous nickname for instructions that just stop the CPU and wait for restart.

How To

On Unices the objdump utility from GNU binutils can be used to **disassemble** compiled programs, i.e view the instructions of the program in assembly (other tools like `ndisasm` can also be used). Use it e.g. as:

```
objdump -d my_compiled_program
```

Let's now write a simple Unix program in x86 assembly (AT&T syntax). Write the following source code into a file named e.g. `program.s`:

```
.global  _start          # include the symbol in object file

str:
.ascii   "it works\n"    # the string data

.text
_start:                  # execution starts here
```

```

    mov    $5,    %rbx    # store loop counter in rbx

.loop:
    # make a Linux "write" syscall:
    # args to syscall will be passed in regs.
    mov    $1,    %rax    # says syscalls type (1 = write)
    mov    $1,    %rdi    # says file to write to (1 = stdout)
    mov    $str, %rsi    # says the address of the string to write
    mov    $9,    %rdx    # says how many bytes to write
    syscall                    # makes the syscall

    sub    $1,    %rbx    # decrement loop counter
    cmp    $0,    %rbx    # compare it to 0
    jne    .loop        # if not equal, jump to start of the loop

    # make an "exit" syscall to properly terminate:
    mov    $60, %rax    # says syscall type (60 = exit)
    mov    $0, %rdi    # says return value (0 = success)
    syscall                    # makes the syscall

```

The program just writes out it works five times: it uses a simple loop and a Unix system call for writing a string to standard output (i.e. it won't work on Windows and similar shit).

Now assembly source code can be manually assembled into executable by running assemblers like `as` or `nasm` to obtain the intermediate object file and then linking it with `ld`, but to assemble the above written code simply we may just use the `gcc` compiler which does everything for us:

```
gcc -nostdlib -no-pie -o program program.s
```

Now we can run the program with

```
./program
```

And we should see

```

it works
it works
it works
it works
it works

```

As an exercise you can `objdump` the final executable and see that the output basically matches the original source code. Furthermore try to disassemble some primitive C programs and see how a compiler e.g. `gcc` makes if statements or functions into assembly.

Example

Let's take the following C code:

```

#include <stdio.h>

char incrementDigit(char d)
{
    return // remember this is basically an if statement
        d >= '0' && d < '9' ?
        d + 1 :
        '?';
}

int main(void)
{
    char c = getchar();
    putchar(incrementDigit(c));
    return 0;
}

```

We will now compile it to different assembly languages (you can do this e.g. with `gcc -S my_program.c`). This assembly will be pretty long as it will contain boilerplate and implementations of `getchar` and `putchar` from standard library, but we'll only be looking at the assembly corresponding to the above written code. Also note that the generated assembly will probably differ between compilers, their versions, flags such as optimization level etc. The code will be manually commented.

{ I used this online tool: <https://godbolt.org>. ~drummyfish }

{ Also not sure the comments are 100% correct, let me know if not. ~drummyfish }

The x86 assembly may look like this:

```
incrementDigit:
    pushq    %rbp                # save base pointer
    movq     %rsp, %rbp          # move base pointer to stack top
    movl     %edi, %eax          # move argument to eax
    movb     %al, -4(%rbp)        # and move it to local var.
    cmpb     $47, -4(%rbp)        # compare it to '0'
    jle      .L2                 # if <=, jump to .L2
    cmpb     $56, -4(%rbp)        # else compare to '9'
    jg       .L2                 # if >, jump to .L4
    movzbl   -4(%rbp), %eax       # else get the argument
    addl     $1, %eax             # add 1 to it
    jmp      .L4                 # jump to .L4
.L2:
    movl     $63, %eax           # move '?' to eax (return val.)
.L4:
    popq     %rbp                # restore base pointer
    ret

main:
    pushq    %rbp                # save base pointer
    movq     %rsp, %rbp          # move base pointer to stack top
    subq     $16, %rsp           # make space on stack
    call     getchar             # push ret. addr. and jump to func.
    movb     %al, -1(%rbp)        # store return val. to local var.
    movsbl   -1(%rbp), %eax       # move with sign extension
    movl     %eax, %edi           # arg. will be passed in edi
    call     incrementDigit       # sign extend return val.
    movsbl   %al, %eax            # pass arg. in edi again
    movl     %eax, %edi
    call     putchar             # values are returned in eax
    movl     $0, %eax
    leave
    ret
```

The ARM assembly may look like this:

```
incrementDigit:
    sub     sp, sp, #16          // make room on stack
    strb    w0, [sp, 15]         // load argument from w0 to local var.
    ldrb    w0, [sp, 15]         // load back to w0
    cmp     w0, 47                // compare to '0'
    bls     .L2                  // branch to .L2 if <
    ldrb    w0, [sp, 15]         // load argument again to w0
    cmp     w0, 56                // compare to '9'
    bhi     .L2                  // branch to .L2 if >=
    ldrb    w0, [sp, 15]         // load argument again to w0
    add     w0, w0, 1             // add 1 to it
    and     w0, w0, 255           // mask out lowest byte
    b       .L3                  // branch to .L3
.L2:
    mov     w0, 63                // set w0 (ret. value) to '?'
.L3:
    add     sp, sp, 16           // shift stack pointer back
    ret

main:
    stp     x29, x30, [sp, -32]! // shift stack and store x regs
```

```

mov    x29, sp
bl     getchar
strb   w0, [sp, 31]           // store w0 (ret. val.) to local var.
ldrb   w0, [sp, 31]           // load it back to w0
bl     incrementDigit
and    w0, w0, 255             // mask out lowest byte
bl     putchar
mov    w0, 0                   // set ret. val. to 0
ldp    x29, x30, [sp], 32     // restore x regs
ret

```

The RISC-V assembly may look like this:

```

incrementDigit:
    addi    sp,sp,-32           # shift stack (make room)
    sw      s0,28(sp)           # save frame pointer
    addi    s0,sp,32            # shift frame pointer
    mv      a5,a0               # get arg. from a0 to a5
    sb      a5,-17(s0)          # save to local var.
    lbu     a4,-17(s0)          # get it to a4
    li      a5,47               # load '0' to a5
    bleu    a4,a5,.L2           # branch to .L2 if a4 <= a5
    lbu     a4,-17(s0)          # load arg. again
    li      a5,56               # load '9' to a5
    bgtu    a4,a5,.L2           # branch to .L2 if a4 > a5
    lbu     a5,-17(s0)          # load arg. again
    addi    a5,a5,1             # add 1 to it
    andi    a5,a5,0xff          # mask out the lowest byte
    j       .L3                 # jump to .L3
.L2:
    li      a5,63               # load '?'
.L3:
    mv      a0,a5               # move result to ret. val.
    lw      s0,28(sp)           # restore frame pointer
    addi    sp,sp,32            # pop stack
    jr      ra                   # jump to addr in ra

main:
    addi    sp,sp,-32           # shift stack (make room)
    sw      ra,28(sp)           # store ret. addr on stack
    sw      s0,24(sp)           # store stack frame pointer on stack
    addi    s0,sp,32            # shift frame pointer
    call    getchar
    mv      a5,a0               # copy return val. to a5
    sb      a5,-17(s0)          # move a5 to local var
    lbu     a5,-17(s0)          # load it again to a5
    mv      a0,a5               # move it to a0 (func. arg.)
    call    incrementDigit
    mv      a5,a0               # copy return val. to a5
    mv      a0,a5               # get it back to a0 (func. arg.)
    call    putchar
    li      a5,0                # load 0 to a5
    mv      a0,a5               # move it to a0 (ret. val.)
    lw      ra,28(sp)           # restore return addr.
    lw      s0,24(sp)           # restore frame pointer
    addi    sp,sp,32            # pop stack
    jr      ra                   # jump to addr in ra

```

assertiveness

Assertiveness

Assertiveness is an euphemism for being a dick.

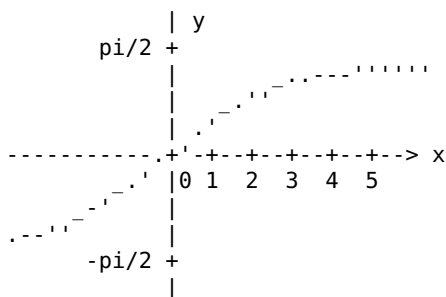
atan

Arcus Tangent

Arcus tangent, written as atan or \tan^{-1} , is the inverse function to the tangent function. For given argument x (any real number) it returns a number y (from $-\pi/2$ to $\pi/2$) such that $\tan(y) = x$.

Approximation: Near 0 $\text{atan}(x)$ can very roughly be approximated simply by x . For a large argument $\text{atan}(x)$ can be approximated by $\pi/2 - 1/x$ (as atan 's limit is $\pi/2$). The following formula { created by me ~drummyfish } approximates atan with a polynomial for non-negative argument with error smaller than 2%:

$$\text{atan}(x) \sim (x * (2.96088 + 4.9348 * x)) / (3.2 + 3.88496 * x + \pi * x^2)$$



plot of $\text{atan}(x)$

atheism

Atheism

"In this moment I am euphoric ..." --some retarded atheist

An atheist is someone who doesn't believe in god or any other similar supernatural beings. An especially annoying kind is the **reddit atheist** who will DESTROY YOU WITH FACTS AND LOGIC^(TM) while managing to throw around le 42 jokes. These atheists are 14 year old children who think they've discovered the secret of the universe and have to let the whole world know they're atheists who will destroy you with their 200 IQ logic and knowledge of all 10 argument logic fallacies, while in fact they reside at the mount stupid and many times involuntarily appear on other subreddits such as r/iamverysmart and r/cringe. They masturbate to Richard Dawkins, love to read soyentific studiis about how race has no biological meaning and think that religion is literally Hitler (oh noes, reduction to HITLER has been committed, game over). They love to write or even read the "rational" wiki. They like to pick easy targets such as flatearthers and cyberbully them on YouTube with the power of SCIENCE and their enormously large thesaurus (they will never use a word that's among the 100000 most common English words). They are so cringe you want to kill yourself, but their discussions are sometimes entertaining to read with a bowl of popcorn.

Such a specimen of atheist is one of the best quality examples of a pseudosceptic. See also this: <https://www.debunkingskeptics.com/Contents.htm>.

On a bit more serious note: we've all been there, most people in their teens think they're literal Einsteins and then later in life cringe back on themselves. However, some don't grow out of it and stay arrogant, ignorant fucks for their whole lives. The principal mistake of the stance they retain is they try to apply "science" (or whatever it means in their world) to EVERYTHING and reject any other approach to solving problems -- of course, science (the real one) is great, but it's just a tool, and just like you can't fix every problem with a hammer, you can't approach every problem with science. In your daily life you make a million of unscientific decisions and it would be bad to try to apply science to them; you cross the street not because you've read a peer-reviewed paper about it being the most scientifically correct thing to do, but because you feel like doing it, because you believe the drivers will stop and won't run you over. Beliefs, intuition, emotion, non-rationality and even spirituality are and have to be part of life, and it's extremely stupid to oppose these concepts just out of principle. With that said, there's nothing wrong about being a well behaved man who just doesn't feel a belief in any god in his heart, just you know, don't be an idiot.

Among the greatest minds it is hard to find true atheists, even though they typically have a personal and not easy to describe faith. Newton was a Christian. Einstein often used the word "God" instead of "nature" or "universe"; even though he said he didn't believe in the traditional personal God, he also said that the laws of physics were like books in a library which must have obviously been written by someone or something we can't comprehend. Nikola Tesla said he was "deeply religious, though not in the orthodox sense". There are also very hardcore religious people such as Larry Wall, the inventor of Perl language, who even planned to be a Christian missionary. The "true atheists" are mostly second grade "scientists" who make career out of the pose and make living by writing books about atheism rather than being scientists.

See Also

- stupidity
-

attribution

Attribution

In the world of intellectual works (such as programs, texts, images etc.) attribution means visibly and properly acknowledging the work of collaborators, i.e. usually mentioning the names or pseudonyms of others that somehow took part in creation of the work. Sometimes we distinguish between merely giving *credit*, i.e. just recording collaborators somewhere, even in a less visible place such as some documentation file, and *proper attribution* which may have further conditions, e.g. mentioning the authors in a visible place (e.g. game's main menu) along with a link to their website and so on. Attribution is something that's often a condition of a license, i.e. for example the Creative Commons Attribution (CC BY) license grants everyone rights to the work as long as the original author is properly attributed. However we at LRS see such license requirements as harmful; **forcing attribution by license is a very bad idea!** Never do it. Please consider the following:

- Forcing attribution may cause practical problems and make your work unusable. While it's no issue to give proper attribution to one guy who made music for your game, consider also a different scenario: e.g. in development of LMMS, a FOSS music making program, the authors had to collect hundreds of short sound samples for their virtual instruments -- here they couldn't use CC BY-SA samples because doing so would require anyone who made music with their program to also carry on proper attribution of all the author of every single sample that was used in the music, which is practically almost impossible.
- Forcing attribution can make you be force signed under things you don't want to be signed under. Consider you make a comics for children and license it CC BY-SA, i.e. require attribution. By free culture principles someone can take the characters from your story and make porn or terrorist supporting videos with them and even if those guys knew you wouldn't want to be signed under this (because you e.g. made it clear on your blog that you hate porn and terrorism) and even if they would be willing to not name you, your license will force them to write your name PROPERLY, i.e. visibly, under the thing they make.
- You're still playing the copyright game -- even if you relax copyright, you still acknowledge of the idea you keep some basic rights and have to enforce a "correct use" of your work. Even if the difference between CC0 and CC BY was practically of small importance, your mindset will likely be very different with each of them. There is a pattern of people who use CC0 being completely cool while the "CC BY-SA" people oftentimes changing their mind, trying to make trouble with "moral rights" and so on. Just don't do this.
- It is just legal bloat, it created friction, distract artists. It is unnecessary. Even if it's a small burden, it's still a burden for everyone -- the license has to be longer, it has to define what proper attribution means, what happens if it can't be technically achieved etc. You have to keep one more thing in your working memory, you have to observe if people respect this condition etc.
- It discourages many from using your work. For some of the mentioned reasons many people actually avoid reusing works that require attribution { Including me and many other people I know. ~drummyfish }. There exist dangers like attribution getting unintentionally lost in some copy paste by which you start violating the license, people are aware of this danger so they firstly look for works with no conditions at all, just to be safer. By releasing your work without requiring attribution you usually get "extra points" from the free culture community for saving other headaches and trouble.

- You will almost certainly be attributed even if you don't force it. People naturally credit others and there is basically no reason not to, it's in everyone's interest. In practice many people use licenses/waiver that don't force attribution and basically no "abuse" of this is seen -- firstly people are culturally very strongly taught to always attribute others and socially rewarded for doing so, but secondly it doesn't even make any sense to try to come up with any "abuse", there isn't a way to abuse this -- imagine someone wanted to take credit on social media for some work he didn't make: it would sooner or later be found he didn't make the work anyway -- the original author would comment or it would show the guy is incapable of producing more similar works etc., and this can be confirmed on the Internet by digging and finding the work posted previously by someone else. So the guy would just forever mark himself as a scammer, people just don't even try this. AND even if this happens -- e.g. with some nasty copycat Chinese scammers -- they just blatantly "steal" the work no matter the license, they literally don't care about licenses, they steal even proprietary Hollywood movie characters, license doesn't do anything here. { I've been using exclusively CC0 (which doesn't require credit) for many years and literally never encountered a single case when someone wouldn't credit me, nor have I heard of any malicious attempts at abusing this anywhere. ~drummyfish }
- ...

See Also

- copyleft
- NC
- ND

audiophilia

Audiophilia

Audiophilia is a mental disorder, similar to other diseases such as distrohopping and chronic ricing, that makes one scared of low or normal quality audio. Audiophiles are scared of lossy compression and so harm society by wasting storage space. Audiophilia, similarly to e.g. the business with mechanical keyboards, is the astrology of technology, it is an arbitrarily invented bullshit business creating an artificial need that makes people wanna buy golden cables and similar shit in belief that it will make their life happier, perpetuation consumerism and capitalism.

autostereogram

Autostereogram

Autostereogram is a cool sort of image that when viewed in a special way (with eyes crossed or "walled") enables the viewer to see a 3D structure within it by cheating human stereoscopic vision (it is therefore in a sense also an optical illusion). As the name suggests it is a special case of stereogram but unlike many traditional stereograms consisting of two side by side images, autostereogram is only a single image that forms the perceivable 3D pattern by being overlaid with itself (hence the prefix *auto*). These images are quite awesome for they implement stereoscopic 3D images without the need for special glasses or complex techniques like autostereoscopy or holography -- autostereograms can be made as long as we can draw plain 2D images, but of course they also suffer from some limitations. There are several types of autostereograms.

Viewing autostereograms is easy for some and difficult for others but don't worry, it can be trained. One trick that's used (for the "cross eyed" types of images) is putting a finger in front of the image, focusing your sight on it and then lowering the finger while keeping your eyes looking at the point where the finger was (for "walled" images you have to be looking beyond the image, i.e. try looking at a wall behind it). Also be careful about the possibility of crossing your eyes "too much" and seeing the image in incorrect way. Once you see the pattern, keep looking at it for a longer time, it becomes clearer and clear as the brain makes out more of the structure (it may also help to slightly move your head from side to side).

TODO

The "random dot" technique gives rise to an especially interesting type of autostereogram -- one whose creation can easily be automatized with a program and which lets us embed any depth image (or heightmap) into an image that consists of some repeating base pattern. And yes, it can even be animated! The pattern image may in theory be anything, even a photo, but it should have great variety, high frequencies and big contrast to work properly, so the typical pattern is just randomly generated color dots. This pattern is then horizontally deformed according to the embedded depth image. A disadvantage is, of course, that we can only embed the depth image, we cannot give it any texture.

[illegible]

The following is a C program that generates the above image.

Autostereogram

```

char buffer1[PATTERN_SIZE + 1];
char buffer2[PATTERN_SIZE + 1];

int charToDepth(char c)
{
    return c == ' ' ? 0 : (c - '0');
}

int main(void)
{
    const char *c = depth;
    char *lineCurrent, *linePrev;

    buffer1[PATTERN_SIZE] = 0;
    buffer2[PATTERN_SIZE] = 0;

    for (int j = 0; j < RES_Y; ++j)
    {
        for (int i = 0; i < PATTERN_SIZE; ++i) // initiate first pattern from seed
            buffer1[i] = patternSeed[(i + (j * 13)) % PATTERN_SEED_SIZE];

        lineCurrent = buffer1;
        linePrev = buffer2;

        for (int i = 0; i < RES_X; ++i)
        {
            if (i % PATTERN_SIZE == 0)
            {
                printf("%s", lineCurrent); // print the rendered line

                char *tmp = lineCurrent; // swap previous and current buffer
                lineCurrent = linePrev;
                linePrev = tmp;
            }

            lineCurrent[i % PATTERN_SIZE] = // draw the offset pixel
                linePrev[(PATTERN_SIZE + i + charToDepth(*c)) % PATTERN_SIZE];

            c++;
        }

        printf("%s\n", lineCurrent); // print also the last buffer
    }

    return 0;
}

```

autoupdate

Autoupdate

Autoupdate is a malicious software feature that frequently remotely modifies software on the user's device without asking, sometimes silently and many times in a forced manner without the possibility to refuse this modification (typically in proprietary software). This is a manifestation of update culture. These remote software modifications are called "updates" to make the user think they are a good thing, but in fact they usually introduce more bugs, bloat, security vulnerabilities, annoyance (forced reboots etc.) and malware (even in "open source", see e.g. the many projects on GitHub that introduced intentional malware targeted at Russian users during the Russia-Ukraine war).

avpd

Avoidant Personality Disorder

Avoiding the problem is in majority of cases the best solution to the problem.

Avoidant Personality Disorder (AVPD) is one of the great myriad of psychological personality "disorders" that's basically characterized by extreme shyness, social isolation and tendency to solve everything by avoidance -- people with this thing have no friends, social life, they isolate, don't go out, don't go to work (that's good), they constantly think about how they're judged by others, may try to adjust personality according to what the other people around seem to want etc. It could possibly be seen as the hardcore minimalist disease, AVPD positives just minimize their life to the a bare minimum of things they can't avoid, like eating, breathing etc. Of course this goes with anxiety, panic attacks and depression, sometimes self harm and so on.

axiom_of_choice

Axiom Of Choice

In mathematics (specifically set theory) axiom of choice is a possible axiom which basically states we can arbitrarily choose elements of sets and which is famous for being controversial and problematic because it causes trouble both when we accept or reject it. Now it's actually been included in ZFC, a kind of "commonly used base for mathematics", but its controversial nature stands. Note that this topic can go to a great depth and lead to philosophical debates, there is a huge rabbit hole and mathematicians can talk about this for hours; here we'll only state the very basic and quite simplified things, mostly for those who aren't professional mathematicians but need some overview of mathematics (e.g. programmers).

Indeed, **what really IS the axiom of choice?** It is an axiom, i.e. something that we can't prove but can either accept or reject as a basic fact so that we can use it to prove things. Informally it says that given any collection of sets (even an infinite collection of infinitely large sets), we can make an arbitrary selection of one element from each set. More mathematically it says: if we have a collection of sets, there always exists a function f such that for any set S from the collection $f(S)$ is an element of S .

This doesn't sound weird, does it? Well, in many normal situations it isn't. For example if we have finitely many sets, we can simply write out each element of the set, we don't need to define any selection function, so we don't need axiom of choice to make our choice of elements here. But also if we have infinitely many sets that are well ordered (we can compare elements), for example infinitely many sets of natural numbers, we can simply define a function that takes e.g. the smallest number from each set -- here we don't need axiom of choice either. The issues start if we have e.g. infinitely many sets of real numbers (which can't be well ordered without the axiom of choice, consider that e.g. open intervals don't have lowest number) -- here we can't say how a function should select one element from each set, so we have to either accept axiom of choice (we say it simply can be done "somehow", e.g. by writing each element out on an infinitely large paper) or reject it (we say it can't be done). Here it is again the case that what's normally completely non-problematic starts to get very weird once you involve infinity.

Why is it problematic? Once you learn about axiom of choice, your first question will probably be why should it pose any problems if it just seems like an obvious fact. Well, it turns out it leads to strange things. If we accept axiom of choice, then some weird things happen, most famously e.g. the Banach-Tarski paradox which uses the axiom of choice to prove that you can disassemble a sphere into finitely many pieces, then move and rotate them so that they create TWO new spheres, each one identical to the original (i.e. you duplicate the original sphere). But if we reject the axiom of choice, other weird things happen, for example we can't prove that every vector space has a basis -- it seems quite elementary that every vector space should have a basis, but this can't be proven without the axiom of choice and in fact accepting this implies the axiom of choice is true. Besides this great many number of proofs simply don't work without axiom of choice. So essentially either way things get weird, whether we accept axiom of choice or not.

So what do mathematicians do? How do they deal with this and why don't they kill themselves? Well, in reality most of them are pretty chill and don't really care, they try avoid it if they can (their proof is kind of stronger if it relies on fewer axioms) but they accept it if they really need it for a specific proof. Many elementary things in mathematics actually rely on axiom of choice, so there's no fuss when someone uses it, it's very normal. Turns out axiom of choice is more of something they argue over a beer, they usually disagree about whether it is INTUITIVELY true or false, but that doesn't really affect their work.

backgammon

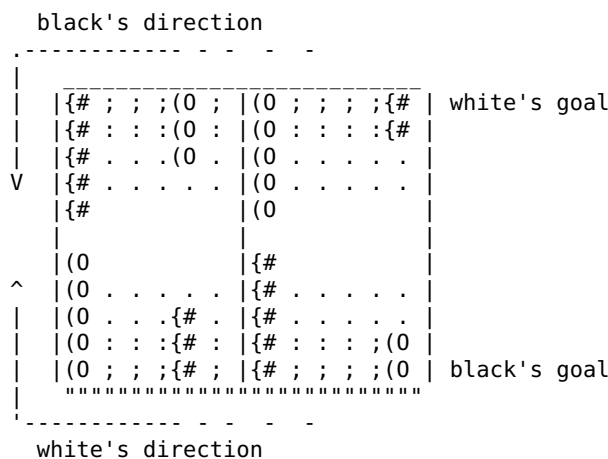
Backgammon

Backgammon is an old, very popular board game of both skill and chance (dice rolling) in which players race their stones from one side of the board to the other. It often involves betting (but can also be played without it) and is especially popular in countries of Near East such as Egypt, Syria etc. (where it is kind of what chess is to our western world or what shogi and go are to Asia). It is a very old game whose predecessors were played by old Romans and can be traced even as far as 3000 BC. Similarly to chess, go, shogi and other traditional board games backgammon is considered by us to be one of the best games as it is owned by no one, highly free, cheap, simple yet deep and entertaining and can be played even without a computer, just with a bunch of rocks; compared to the other mentioned board games backgammon is unique by involving an element of chance and being only played on 1 dimensional board; it is also relatively simple and therefore noob-friendly and possibly more relaxed (if you lose you can just blame it on rolling bad numbers).

Rules

Here we'll summarize the common rules, keep in mind there may be some variations, like extra rules on competitive level and so on. The rules seem quite complex and arbitrary at first, but by playing you'll see they're really pretty simple and sometimes quite intuitive (furthermore the game, at least on casual level, mostly doesn't require such hard thinking as e.g. chess, so it even feels more relaxed, you can focus on the rules well).

There are **two players**, black and white, each moving circular stone discs, or just **stones** of his color, here we'll use {# for black stones and (0 for white ones. There are **two six sided dice** in the game. The board has **24 places** (vertical lines, traditionally drawn as long triangles) which stones can occupy. The following shows the board, the initial setup of stones, the directions in which players move and their goals.



The **goal** of each player is to get all his stones to his goal -- the goal is one place beyond the last place on the board in the direction of his movement. Whoever does this the first wins.

The first six places on one's path are called the **home board**, the last six are called the **outer board**.

At start both players roll the dice (each one rolls one), whoever rolls the bigger number starts and has to use (details below) the numbers that were just rolled for his first turn (if the numbers were the same, they roll again). After the first player finishes his round, the other player rolls both dice, makes his turn, then the first player does the same again and so on, the players just take turns in rolling dice and playing.

A **turn** is played by rolling the two dice, resulting in numbers X (one die) and Y (the other one). The player then moves two stones (he can choose which), one by X places, the other by Y places. He can also move the same stone, but the move still counts as moving twice, i.e. first moving the stone by X, then moving it again by Y, or vice versa (this may be important in regards to rules explained later). If X and Y are the same, the numbers are doubled, so the player gets 4 numbers to play: X, X, X, X -- for example rolling 2 and 2, the player can move 4 stones, each by 2, or 1 stone by 8 (in separate steps) or 1 stone by 2 and other one by 6 and so on. Moves cannot be skipped by choice, the player has to move "as much as he can", i.e. if he can at least partially use the numbers he rolled, he has to (also if there is a choice between higher and lower

number rolled, he has to use the higher number etc.).

Movement: players move their stones in opposite directions by the number of steps they roll, in a kind of horseshoe shaped path (as shown above -- topologically the board is just a 1D line, it's just curved to nicely fill the board) -- notice that on one end the stones jump from one side of the board to the other side. Stones can walk over stones of same color and can even stay on the same place -- if more than one stones occupy the same place, they are "stacked" and protected against being taken. A stone can move over enemy stones (even if multiple stacked enemy stones), but can end on such place only if there is exactly one enemy stone, in which case it is taken -- it is removed and placed in the middle of the board. Remember that a stone that is moving by a sum of rolled numbers counts as several discrete moves, so if a stone is moving e.g. by 3 + 3 steps, it's not the same as moving by 6 because after the first 3 steps taken it mustn't land on stacked enemy stones (but it can land on one enemy stone and take it).

A stone that's been taken (placed in the middle of the board) is seen as being one place before the player's starting place (the opposite of one's goal), and can be returned to the game (appearing in the enemy home board) -- in fact it HAS TO be returned to the game before any other move can be made by the player whose stone it is, i.e. if a player has any stones out of the game because the opponent has taken them, he cannot move any other stones until he returns all his stones back to the game.

Once the player has all his stones in the enemy home board, he can start **bearing off**, i.e. getting the stones to the goal (i.e. before this his stones aren't allowed to reach the goal). The goal is seen as a place one after the final board square in the direction of the player's movement -- if the stone gets to the goal, it is placed on the board border. Here there are a bit more complex rules: normally a stone may reach the goal only if it steps on it exactly, i.e. a stone on the very last place can only get to the goal by rolling 1, the stone before it by rolling 2 etc. However the stone furthest away from the goal may also use a value higher than this, i.e. if there is a stone 3 places before the goal AND it is the last one back, it may finish with 3, 4, 5 or 6. During bearing off the player may also use the lower rolled value first, even if it wouldn't fully utilize the higher value (exception to a rule mentioned above).

Details

Despite chance playing some role, skill is highly important and there exist strategies and tactics that maximize one's chance of winning -- for example a basic realization is that the different sums you may roll don't have the same probabilities, e.g. 8 can be achieved by 2 + 6 or 2 + 2 + 2 + 2, but 3 only as 2 + 1 -- one can account for this. The highest probability to take the enemy stone with one's own stone is when the stones are 6 places apart. Taking enemy stone while having own stones stacked in all places in enemy home board makes opponent unable to play (he is required to return the stone to play but there is no number that can do it for him). There is also some opening theory.

The game is internationally governed by WBGF (World Backgammon Federation), similarly to how chess is governed by FIDE.

Who was the **best player ever**? There doesn't seem to be a clear consensus, but Masayuki Mochizuki (Japan) seems to come up very often as an answer to the question, other names include Paul Magriel, Nack Ballard etc.

Backgammon was the first board game in which the world champion at the time (Luigi Villa) was defeated by computer -- this happened in 1979. This was perhaps thanks to the element of chance.

As for backgammon **computer engines** the best free as in freedom one seems to be GNU backgammon, using neural networks, apparently beyond the strength of best human players. The Extreme Gammon engine is probably a bit stronger (currently said to be the strongest) but it is proprietary and therefore unusable.

Some statistics about the game: there are 18528584051601162496 legal positions. Average branching factor (considering all possible dice rolls) is very high, somewhere around 400, which is likely why space search isn't as effective as in chess and why neural networks greatly prevail. Average number of moves in a game seem to be slightly above 20.

TODO

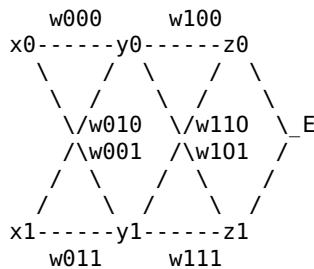
Backpropagation

{ Dunno if this is completely correct, I'm learning this as I'm writing it. There may be errors. ~drummyfish }

Backpropagation, or backprop, is an algorithm, based on the chain rule of derivation, used in training neural networks; it computes the partial derivative (or gradient) of the function of the network's error so that we can perform a gradient descent, i.e. update the weights towards lowering the network's error. It computes the analytical derivative (theoretically you could estimate a derivative numerically, but that's not so accurate and can be too computationally expensive). Backpropagation is one of the most common methods for training neural networks but it is NOT the only possible one -- there are many more such as evolutionary programming. It is called backpropagation because it works backwards and propagates the error from the output towards the input, due to how the chain rule works, and it's efficient by reusing already computed values.

Details

Consider the following neural network:



It has an input layer (neurons x_0 , x_1), a hidden layer (neurons y_0 , y_1) and an output layer (neurons z_0 , z_1). For simplicity there are no biases (biases can easily be added as input neurons that are always on). At the end there is a total error E computed from the networks's output against the desired output (training data).

Let's say the total error is computed as the squared error: $E = \text{squared_error}(z_0) + \text{squared_error}(z_1) = 1/2 * (z_0 - z_{0_desired})^2 + 1/2 * (z_1 - z_{1_desired})^2$.

We can see each non-input neuron as a function. E.g. the neuron z_0 is a function $z_0(x) = z_0(a(z_0s(x)))$ where:

- z_0s is the sum of inputs to the neuron, in this case $z_0s(x) = w100 * y_0(x) + w110 * y_1(x)$
- a is the activation function, let's suppose the normally used logistic function $a(x) = 1/(1 + e^{-x})$.

If you don't know what the fuck is going on see neural networks first.

What is our goal now? To find the **partial derivative of the whole network's total error function** (at the current point defined by the weights), or in other words the **gradient** at the current point. I.e. from the point of view of the total error (which is just a number output by this system), the network is a function of 8 variables (weights $w000$, $w001$, ...) and we want to find a derivative of this function in respect to each of these variables (that's what a partial derivative is) at the current point (i.e. with current values of the weights). This will, for each of these variables, tell us how much (at what rate and in which direction) the total error changes if we change that variable by certain amount. Why do we need to know this? So that we can do a gradient descent, i.e. this information is kind of a direction in which we want to move (change the weights and biases) towards lowering the total error (making the network compute results which are closer to the training data). So all in all the goal is to find derivatives (just numbers, slopes) with respect to $w000$, $w001$, $w010$, ... $w111$.

Could we do this without backpropagation? Yes -- we can use numerical algorithms to estimate derivatives, the simplest one would be to just try to change each weight, one by one, by some small number, let's say

dw , and see how much such change changes the output error. I.e. we would sample the error function in all directions which could give us an idea of the slope in each direction. However this would be pretty slow, we would have to reevaluate the whole neural network as many times as there are weights. Backpropagation can do this much more efficiently.

Backpropagation is based on the **chain rule**, a rule of derivation that equates the derivative of a function composition (functions inside other functions) to a product of derivatives. This is important because by converting the derivatives to a product we will be able to **reuse** the individual factors and so compute very efficiently and quickly.

Let's write derivative of $f(x)$ with respect to x as $D\{f(x),x\}$. The chain rule says that:

$$D\{f(g(x)),x\} = D\{f(g(x)),g(x)\} * D\{g(x),x\}$$

Notice that this can be applied to any number of composed functions, the product chain just becomes longer.

Let's get to the computation. Backpropagation work by going "backwards" from the output towards the input. So, let's start by computing the derivative against the weight w_{100} . It will be a specific number; let's call it ' w_{100} '. Derivative of a sum is equal to the sum of derivatives:

$$'w_{100} = D\{E,w_{100}\} = D\{squared_error(z_0),w_{100}\} + D\{squared_error(z_0),w_{100}\} = D\{squared_error(z_0),w_{100}\} + 0$$

(The second part of this sum became 0 because with respect to w_{100} it is a constant.)

Now we can continue and utilize the chain rule:

$$'w_{100} = D\{E,w_{100}\} = D\{squared_error(z_0),w_{100}\} = D\{squared_error(z_0(a(z_0s))),w_{100}\} = D\{squared_error(z_0),z_0\} * D\{a(z_0s),z_0s\} * d\{z_0s,w_{100}\}$$

We'll now skip the intermediate steps, they should be easy if you can do derivatives. The final results is:

$$'w_{100} = (z_0_desired - z_0) * (z_0s * (1 - z_0s)) * y_0$$

Now we have computed the derivative against w_{100} . In the same way can compute ' w_{101} ', ' w_{110} ' and ' w_{111} ' (weights leading to the output layer).

Now let's compute the derivative in respect to w_{000} , i.e. the number ' w_{000} '. We will proceed similarly but the computation will be different because the weight w_{000} affects both output neurons (' z_0 ' and ' z_1 '). Again, we'll use the chain rule.

$$w_{000} = D\{E,w_{000}\} = D(E,y_0) * D\{a(y_0s),y_0s\} * D\{y_0s,w_{000}\}$$

$$D(E,y_0) = D\{squared_error(z_0),y_0\} + D\{squared_error(z_1),y_0\}$$

Let's compute the first part of the sum:

$$D\{squared_error(z_0),y_0\} = D\{squared_error(z_0),z_0s\} * D\{squared_error(z_0s),y_0\}$$

$$D\{squared_error(z_0),z_0s\} = D\{squared_error(z_0),z_0\} * D\{a(z_0s),z_0s\}$$

Note that this last equation uses already computed values which we can reuse. Finally:

$$D\{squared_error(z_0s),y_0\} = D\{squared_error(w_{100} * y_0 + w_{110} * y_1),y_0\} = w_{100}$$

And we get:

$$D\{squared_error(z_0),y_0\} = D\{squared_error(z_0),z_0\} * D\{a(z_0s),z_0s\} * w_{100}$$

And so on until we get all the derivatives.

Once we have them, we multiply them all by some value (**learning rate**, a distance by which we move in the computed direction) and subtract them from the current weights by which we perform the gradient descent and lower the total error.

Note that here we've only used one training sample, i.e. the error E was computed from the network against a single desired output. If more example are used in a single update step, they are usually somehow averaged.

bazaar

Bazaar (The Cathedral And The Bazaar)

The Cathedral and the Bazaar (shortened to *catb*) is a very famous software engineering paper from 1997 by Eric S. Raymond (ESR, a famous oldschool hacker writer) which analyzes the development method of Linux, at the time a new way of mass developing FOSS software by many volunteers over the Internet with relatively little central planning -- this method is called the *Bazaar* (the word used for marketplace in middle east) and is contrasted with so called *Cathedral* method, i.e. the traditional, highly centralized development of software (not necessarily of proprietary software). This essay was later being expanded, updated and made into a whole book -- the short version of it can be read on ESR's website. It played a role in corporations adopting "open source" (Netscape, i.e. Firefox, was "open sourced" basically because of this essay).

Watch out: Raymond used to be an oldschool hacker who however, like many others, later turned to the evil side once he smelled money and fame; he basically became hardcore capitalist, promoting open source, free markets and even doing business himself. It can very well be seen in the essay -- it's not about programming, it is about software engineering, i.e. managing and manipulating masses of people to work like machines who will be continuously producing lines of code. It focused on things such as "productivity" and basically how to develop bloat in fastest way and for least cost. It takes things such as update culture, rapid development, gigantic software projects and existence of software companies for granted. Therefore *The Cathedral and the Bazaar* is of no use to less retarded software but it may be good to read for the big picture view.

{ The online version is not very long, the writing style is good and there are nice, catchy observations about software development, however it's still quite shitty, towards the end I was falling asleep, only the capitalist trigger words kept me awake eventually. But there are some nice things, like "plan to throw one away", i.e. when you want to write something, you'll probably have to write it once badly, by which you really understand the issue, then you throw it away and implement it again, this time well. ~drummyfish }

Here is a small **summary**: ESR used to believe software beyond some complexity threshold (e.g. operating system kernel or a big text editor) has to be developed mainly by a small team that closely communicates, carefully fixes bugs that users report and releases stable versions once in relatively long time -- yes, even if the software is FOSS and development is transparent. This is called the *Cathedral* method as the development is similar to the careful, highly centrally planned building of a cathedral -- one example was e.g. gcc (and any proprietary software, as they basically have no other option). However after seeing Linux (a very complex project) being developed by great many people in a very decentralized manner, with the central coordinator doing relatively little work, and having very short release cycles (even of buggy, unstable versions), he concluded it can work differently -- he called this the *Bazaar* method, i.e. one that looks a bit chaotic at first, but which statistically still converges to establishing good design in the end. He says the biggest invention of Linus Torvalds isn't Linux but its development model. He examines how and why it works because he sees it as the superior method, and he also tests the method on his own project (fetchmail) with which he immediately sees a great success. He notes several things, e.g. the following. Users being at the same time programmers (codevelopers) and vice versa is key because firstly programmers really care about what they write (because they use it) and secondly we get nice bug reports (in programmer terms). "Given enough eyeballs, all bugs are shallow" (*Linus's law*) says that with many users/programmers basically all bugs get spotted and fixed quickly, which is helped by the rapid release cycles -- if someone fixes it quickly, others see it's fixed and stop working on their more complicated fixes. This kind of parallelizes debugging (and also other things such as design change exploration). Quick releases reward contributors, they see their fixes immediately, contributors get motivated ("Treat your testers as your most valuable resource and they will respond by becoming your most valuable resource."), even the "work no one wants to do" gets done.

Bazaar project needs several things. Firstly good Internet (that's why Linux coincided with cheap access to Internet). Secondly it can't be started from scratch, someone has to make some basic project basically alone, and it should be some truly honest project (not something that just aims for profit), usually starting with a programmer "scratching his own itch" -- it's enough to make a project that shows promise so that people start jumping in. The "leader" doesn't have to be genius but he has to be able to recognize good design choices of contributors and he must be "good with people". Then he goes on to compare it to free market and other crap, he basically concludes managers are useless and they just pretend to be useful :D

bbs

BBS

{ I am too young to remember this shit so I'm just writing what I've read on the web. ~drummyfish }

Bulletin board system (BBS) is, or rather used to be, a kind of server that hosts a community of users who connect to it via terminal, who exchange messages, files, play games and otherwise interact -- BBSes were mainly popular before the invention of web, i.e. from about 1978 to mid 1990s, however some still exist today. BBSes are powered by special BBS software and the people who run them are called sysops.

Back then people connected to BBSes via dial-up modems and connecting was much more complicated than connecting to a server today: you had to literally dial the number of the BBS and you could only connect if the BBS had a free line. **Early BBSes weren't normally connected through Internet** but rather through other networks like UUCP working through phone lines. I.e. a BBS would have a certain number of modems that defined how many people could connect at once. It was also expensive to make calls into other countries so BBSes were more of a local thing, people would connect to their local BBSes. Furthermore these things ran often on non-multitasking systems like DOS so allowing multiple users meant the need for having multiple computers. The boomers who used BBSes talk about great adventure and a sense of intimacy, connecting to a BBS meant the sysop would see you connecting, he might start chatting with you etc. Nowadays the few existing BBSes use protocols such as telnet, nevertheless there are apparently about 20 known dial-up ones in north America. Some BBSes evolved into more modern communities based e.g. on public access Unix systems -- for example SDF.

A BBS was usually focused on a certain topic such as technology, fantasy roleplay, dating, warez etc., they would typically greet the users with a custom themed ANSI art welcome page upon login -- it was pretty cool. BBSes were used to share plain text files of all sorts, be it anarchist writings, computer manuals, poetry or recipes.

{ There's some documentary on BBS that's supposed to give you an insight into this shit, called literally *BBS: The documentary*. It's about 5 hours long tho. ~drummyfish }

{ According to <http://textfiles.com/law/ethics.txt> it seems like at least part of the BBS community frowned upon anonymity, the file advises to not use handles (at least in some situations), to properly describe one's place on connection and to restrain from private message unless absolutely necessary. And of course, no one probably even considered any encrypted connection back then. This is pretty nice, it's additional evidence for the privacy hysteria really being a new thing we could do without. ~drummyfish }

The first BBS was CBBS (computerized bulletin board system) created by Ward Christensen and Randy Suess in 1978 during a blizzard storm -- it was pretty primitive, e.g. it only allowed one user to be connected at the time. The ideas evolved from those of time sharing computers such as those running Unix, BBS just tried to make them more "user friendly" and so bring in more public to where there were mostly just professionals before, kind of an ancient Facebook-like mini revolution. After publication of their invention, BBSes became quite popular and the number of them grew to many thousands -- later there was even a magazine solely focused on BBSes (*BBS Magazine*). BBSes would later group into larger networks that allowed e.g. interchange of mail. The biggest such network was FidoNet which at its peak hosted about 35000 nodes.

{ Found some list of BBSes at <http://www.synchro.net/sbbslist.html>. ~drummyfish }

See Also

- [public access Unix](#)
 - [Usenet](#)
 - [modem world](#)
 - [tildeverse](#)
 - [multi user dungeon](#)
 - [imageboard](#)
 - [textboard](#)
 - [SDF](#)
 - [FidoNet](#)
-

beauty

Beauty

Beauty is the quality of being extremely appealing and pleasing. Though the word will likely invoke association with traditional [art](#), in [technology](#), [engineering](#), [mathematics](#) and other [science](#) beauty is, despite it's relative vagueness and subjectivity, an important aspect of design, and in fact this "mathematical beauty" has lots of times some clearly defined shapes -- for example [simplicity](#) is mostly considered beautiful. Beauty is similar to and many times synonymous with [elegance](#).

Beauty can perhaps be seen as a [heuristic](#), a touch of intuition that guides the expert in exploration of previously unknown fields, as we have come to learn that the greatest discoveries tend to be very beautiful (however there is also an opposite side: some people, such as Sabine Hossenfelder, criticize e.g. the pursuit of beautiful theories in modern physics as this approach seems to have led to stagnation). Indeed, beginners and [noobs](#) are mostly concerned with learning hard facts, learning standards and getting familiar with already known ways of solving known problems, they often aren't able to recognize what's beautiful and what's ugly. But as one gets more and more experienced and finds himself near the borders of current knowledge, there is suddenly no guidance but intuition, beauty, to suggest ways forward, and here one starts to get the feel for beauty. At this point the field, even if highly exact and rigorous, has become an [art](#).

What is beautiful then? As stated, there is a lot of subjectivity, but generally the following attributes are correlated with beauty:

- **[simplicity/minimalism](#)**, typically finding simplicity in complexity, e.g. a very short formula or algorithm that describes an infinitely complex [fractal](#) shape, a simple but valuable equation in physics ($e = m * c^2$), a short computer program that yields rich results ([demoscene](#), [code golfing](#), [suckless](#), [minimal viable program](#), ...).
- **[deepness](#)** -- if something starting very simple, e.g. a single small equation, leads to consequences that may be studied for millennia, for example [prime numbers](#).
- **[generality](#)**, i.e. if a simple equation can describe many problems, not just a specific case.
- **[lack of exceptions](#)**, i.e. when our equation works without having to deal with special cases (in programming represented by `if-then` branches).
- **[symmetry](#)**, i.e. when we can e.g. swap variables in the equation and get some kind of opposite result.
- **[unification](#)**, i.e. if multiple nice things meet, for example the [Euler's identity](#) brings together into one equation the most important numbers in mathematics: i , pi , 1 and 0.
- **[aesthetics](#)**, either of the equation itself or the generated thing (fractals, attractors, ...).
- TODO

Examples of beautiful things include:

- **[Euler's identity](#)**, an equation often cited as the most beautiful in mathematics: $e^{i\pi} + 1 = 0$. It is simple and contains many of the most important numbers: e , pi , i 1 and 0.
- **[minimalist software](#)**, **[Unix philosophy](#)**
- [fractals](#) TODO
- [bytebeat](#)


```

. . . . .
8 . . . . . 3

```

Let's say we first interpolate horizontally: we'll compute one value, a , on the top (between 1 and 5) and one value, b , at the bottom (between 8 and 3). When computing a we interpolate between 1 and 5 by the horizontal position of x ($4/7$), so we get $a = 1 + 4/7 * (5 - 1) = 23/7$. Similarly $b = 8 + 4/7 * (3 - 8) = 36/7$. Now we interpolate between a and b vertically (by the vertical position of x , $5/7$) to get the final value $x = 23/7 + 5/7 * (36/7 - 23/7) = 226/49 \approx 4.6$. If we first interpolate vertically and then horizontally, we'd get the same result (the value between 1 and 8 would be 6, the value between 5 and 3 would be $25/7$ and the final value $226/49$ again).

Here is a C code to compute all the inbetween values in the above, using fixed point (no float):

```

#include <stdio.h>

#define GRID_RESOLUTION 8

int interpolateLinear(int a, int b, int t)
{
    return a + (t * (b - a)) / (GRID_RESOLUTION - 1);
}

int interpolateBilinear(int topLeft, int topRight, int bottomLeft, int bottomRight,
    int x, int y)
{
#define FPP 16 // we'll use fixed point to prevent rounding errors

#if 1 // switch between the two versions, should give same results:
    // horizontal first, then vertical
    int a = interpolateLinear(topLeft * FPP, topRight * FPP, x);
    int b = interpolateLinear(bottomLeft * FPP, bottomRight * FPP, x);
    return interpolateLinear(a, b, y) / FPP;
#else
    // vertical first, then horizontal
    int a = interpolateLinear(topLeft * FPP, bottomLeft * FPP, y);
    int b = interpolateLinear(topRight * FPP, bottomRight * FPP, y);
    return interpolateLinear(a, b, x) / FPP;
#endif
}

int main(void)
{
    for (int y = 0; y < GRID_RESOLUTION; ++y)
    {
        for (int x = 0; x < GRID_RESOLUTION; ++x)
            printf("%d ", interpolateBilinear(1, 5, 8, 3, x, y));

        putchar('\n');
    }

    return 0;
}

```

The program outputs:

```

1 1 2 2 3 3 4 5
2 2 2 3 3 4 4 5
3 3 3 3 4 4 4 5
4 4 4 4 4 4 4 5
5 5 5 5 5 5 5 4
6 6 6 6 5 5 5 4
7 7 7 6 6 5 5 4
8 8 7 6 6 5 4 3

```

Cool hack to improve bilinear interpolation (from <https://iquilezles.org/articles/texture>): bilinear interpolation doesn't look as good as bicubic but bicubic is a lot more complex on hardware and bandwidth as it requires fetching more texels -- there is one trick which shader programmers use to improve the look of bilinear filtering while not requiring fetching more texels. They use the smoothstep function on the

interpolation parameter which eliminates instant "jumps" at edges between texels, it replaces straight lines with a smoother curve and so makes the derivative of the result continuous -- basically it looks a lot better. Still not as good as bicubic but close enough.

TODO: code for the above

billboard

Billboard

In 3D computer graphics billboard is a flat image placed in the scene that rotates so that it's always facing the camera. Billboards used to be greatly utilized instead of actual 3D models in old games thanks to being faster to render (and possibly also easier to create than full 3D models), but we can still encounter them even today and even outside retro games, e.g. particle systems are normally rendered with billboards (each particle is one billboard). Billboards are also commonly called sprites, even though that's not exactly accurate.

By axis of rotation there are two main types of billboards:

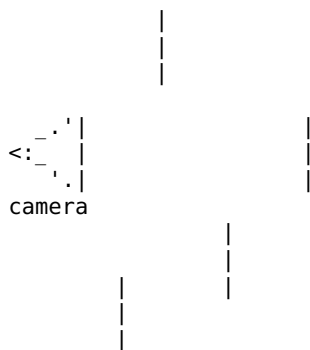
- Ones **rotating only about vertical axis**, i.e. billboards that change only their yaw, they only face the camera in a top-down view of the scene. Such sprite may deform on the screen (when the camera is at different height level) just like 3D models do and when viewed completely from above will disappear completely. This may in some situations look better than other options (e.g. in games enemies won't appear lying on their back when seen from above).
- **Freely rotating** ones, i.e. ones that change all three Euler angles so that they ALWAYS face the camera from any possible angle.

Furthermore there is another subdivision into two main types by HOW the billboards rotate:

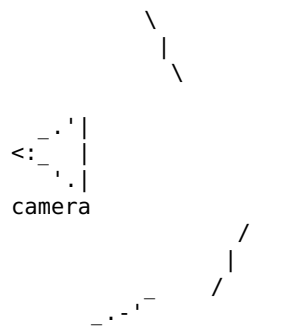
- **Projection plane aligned:** These billboards always align their orientation with the camera's projection plane (they simply rotate themselves in the same way as the camera) which always end up on the screen as an undeformed and unrotated image. This is simple to implement, we can simply blit a 2D image on the rendered 3D view.
- **Camera position facing:** These billboards face themselves towards the camera's position and copy the camera's roll.

Though the two types above may seem like two same things at first glance, they are in fact not, for the latter we need to know the camera and billboard's positions, for the former we only need the camera's rotation. For simplicity we usually choose to implement the former (projection plane aligned), though the latter may result in look closer to that which would be produced by an actual 3D model of the object.

projection plane aligned



position facing



Projection plane aligned vs position facing billboards.

Some billboards also choose their image based on from what angle they're viewed (e.g. an enemy in a game viewed from the front will use a different image than when viewed from the side, as seen e.g. in Doom). Also some billboards intentionally don't scale and keep the same size on the screen, for example health bars in

some games.

In older software billboards were implemented simply as image blitting, i.e. the billboard's scaled image would literally be copied to the screen at the appropriate position (this would implement the freely rotating billboard). Nowadays when rendering 3D models is no longer really considered harmful to performance and drawing pixels directly is less convenient, billboards are more and more implemented as so called textured quads, i.e. they are really a flat square 3D model that may pass the same pipeline as other 3D models (even though in some frameworks they may actually have different vertex shaders etc.) and that's simply rotated to face the camera in each frame (in modern frameworks there are specific functions for this).

Fun fact: in the old games such as Doom the billboard images were made from photographs of actual physical models from clay. It was easier and better looking than using the primitive 3D software that existed back then.

Implementation Details

The following are some possibly useful things for implementing billboards.

The billboard's position on the screen can be computed by projecting its center point in world coordinates with modelview and projection matrices, just as we project vertices of 3D models.

The billboard's size on the screen shall due to perspective be multiplied by $1 / (\tan(FOV / 2) * z)$ where *FOV* is the camera's field of view and *z* is the billboard's distance from camera's projection plane (which is NOT equal to the mere distance from the camera's position, that would create a fish-eye lens effect -- the distance from the projection plane can be obtained from the above mentioned projection matrix). (If the camera's FOV is different in horizontal and vertical directions, then also the billboard's size will change differently in these directions.)

For billboards whose images depends on viewing angle we naturally need to compute the angle. We may do this either in 2D or 3D -- most games resort to the simpler 2D case (only considering viewing angle in a single plane parallel to the floor), in which case we may simply use the combination of dot product and cross product between the normalized billboard's direction vector and a normalized vector pointing from the billboard's position towards the camera's position (dot product gives the cosine of the angle, the sign of cross product's vertical component will give the rest of the information needed for determining the exact angle). Once we have the angle, we quantize (divide) it, i.e. drop its precision depending on how many directional images we have, and then e.g. with a switch statement pick the correct image to display. For the 3D case (possible different images from different 3D positions) we may first transform the sprite's 3D facing vector to camera space with appropriate matrix, just like we transform 3D models, then this transformed vector will (again after quantization) directly determine the image we should use.

When implementing the free rotating billboard as a 3D quad that's aligning with the camera projection plane, we can construct the model matrix for the rotation from the camera's normalized directional vectors: *R* is camera's right vector, *U* is its up vector and *F* is its forward vector. The matrix simply transforms the quad's vertices to the coordinate system with bases *R*, *U* and *F*, i.e. rotates the quad in the same way as the camera. When using row vectors, the matrix is following:

```
R.x R.y R.z 0
U.x U.y U.z 0
F.x F.y F.z 0
0   0   0   1
```

bill_gates

Bill Gate\$

"Some people are so poor that all they've got is money."

William "Bill" Gaytes (28.10.1955 -- TODO) is a mass murderer and rapist (i.e. capitalist) who established and led the terrorist organization Micro\$oft. He is one of the most rich and evil individuals in history who took

over the world by force establishing the malware "operating system" Window\$ as the common operating system, nowadays being dangerous especially by hiding behind his "charity organization" (see charitywashing) which has been widely criticized (see e.g. http://techrights.org/wiki/Gates_Foundation_Critique, even such mainstream media as Wikipedia present the criticism) but which nevertheless makes him look as someone doing "public good" in the eyes of the naive brainless NPC masses.

```

      \_.._.._
     /  "  "  "  \
    /  /  ---  "  "  \
   ;  /  ---  "  "  \
  (= [ 0 ] " [ 0 ] =)
 \  |  "  "  "  "  | /
  |  |  .  _  ,  "  |
 \  |  .  _  ,  "  |
  |  |  .  _  ,  "  |
   \  |  .  _  ,  "  |
    \  |  .  _  ,  "  |
     \  |  .  _  ,  "  |
      \_.._.._

dead or alive

```

ASCII portrait of Bill Gaytes

He is really dumb, only speaks one language and didn't even finish university. He also has no moral values, but that goes without saying for any rich businessman. He was owned pretty hard in chess by Magnus Carlsen on some shitty TV show.

When Bill was born, his father was just busy counting dollar bills, so he named him Bill. Bill was mentally retarded as a child and as such had to attend a private school. He never really understood programming but with a below average intelligence he had a good shot at succeeding in business. Thanks to his family connections he got to Harvard where he met Steve Ballmer. Later he dropped out of the school due to his low intelligence.

In 1975 he founded Micro\$oft, a malware company named after his dick. By a sequence of extremely lucky events combined with a few dick moves by Bill the company then became successful: when around the year 1980 IBM was creating the IBM PC, they came to Bill because they needed an operating system. He lied to them that he had one and sold them a license even though at the time he didn't have any OS (lol). After that he went to a programmer named Tim Paterson and basically stole (bought for some penny) his OS named QDOS and gave it to IBM, while still keeping ownership of the OS (he only sold IBM a license to use it, not exclusive rights for it). He basically fucked everyone for money and got away with it, the American way. For this he is admired by Americans.

When Bill Gates and Steve Jobs saw how enormously rich they got by abusing the whole world, they got horny and had gay sex together, after which Bill legally changed his name to Bill Gaytes. This however gave Jobs ass cancer and he died.

binary

Binary

The word binary in general refers to having two choices or "two of a thing"; in computer science binary refers to the base 2 numeral system, i.e. a system of writing numbers with only two symbols, usually 1s and 0s, which are commonly interpreted *true* vs *false*. We can write any number in binary just as we can with our everyday decimal system (which uses ten digits, as opposed to two), but binary is more convenient for computers because this system is easy to implement in electronics (a switch can be on or off, i.e. 1 or 0; systems with more digits were tried but unsuccessful, they failed miserably in reliability -- see e.g. ternary computers). The word *binary* is also by extension used for non-textual computer files such as native executable programs or asset files for games.

One binary digit (a "place" for binary value in computer memory) can be used to store exactly 1 bit of information. We mostly use binary digits in two ways:

1. With single bits we represent basic logic values, i.e. *true* and *false*, and perform logic operations (e.g. AND, OR etc.) with so called Boolean algebra.

2. By grouping multiple bits together we create a **base-2 numeral system** that behaves in the same way as our decimal system and can be used to record numbers. We can build this numeral system with the above mentioned Boolean algebra, i.e. we extend our simple one bit system to multi bit system allowing to work not just with two values (*true* and *false*) but with many distinct values (whole numbers, from which we may later construct fractions etc.). Thanks to this we can implement algebraic operations such as addition, multiplication, square roots etc.

Of course the binary system didn't appear from nowhere, people in ancient times used similar systems, e.g. the poet Pingala (200 BC) created a system that used two syllables for counting, old Egyptians used so called Eye of Horus, a unit based on power of two fractions etc. Thomas Harriot used something very similar to today's binary in 1600s. It's just that until computers appeared there wasn't much practical use for it, so no one cared.

{ There is one classic but overplayed joke that became extremely cringe exactly by being too overplayed by wannabe haxors who think learning binary makes you Einstein, however since many noobs will likely be reading this and it helps understand the subject, it may be good to tell it anyway. It goes like this: There are 10 types of people -- those who understand binary and those who don't. Sometimes this is extended with: and those who don't know this joke is in base 3. You can also give people the finger by sending them "binary four". ~drummyfish }

Boolean Algebra ("True/False Logic")

In binary we start by working with single bits -- each bit can hold two values, 1 and 0. We may see bits now like "simple numbers", we'll want to do operations with them, but they can only ever be one of the two values. Though we can interpret these values in any way -- e.g. in electronics we see them as high vs low voltage -- in mathematics we traditionally turn to using logic and interpret them as meaning *true* (1) and *false* (0). This will further allow us to apply all the knowledge and theory we have gathered about logic, such as formulas that allow us to simplify binary expressions etc.

Next we want to define "operations" we can perform on single bits -- for this we use so called **Boolean algebra**, which is originally a type of abstract algebra that works with sets and their operations such as conjunction, disjunction etc. Boolean algebra can be seen as a sort of simplified version of what we do in "normal" elementary school algebra -- just as we can add or multiply numbers, we can do similar things with individual bits, we just have a bit different operations such as logic AND, logic OR and so on. Generally Boolean algebra can operate with more than just two values, however that's more interesting to mathematicians; for us all we need now is a binary Boolean algebra -- that's what programmers have adopted for their field. It is the case that in context of computers and programming we implicitly understand Boolean algebra to be the one working with 1s and 0s, i.e. the binary version, so the word "**boolean**" is essentially used synonymously with "binary" around computers. Many programming languages have a data type called `boolean` or `bool` that allows represents just two values (*true* and *false*).

The very basic operations, or logic functions, of Boolean algebra are:

- **NOT** (negation, !): Done with single bit, turns 1 into 0 and vice versa.
- **AND** (conjunction, /\): Done with two bits, yields 1 only if both input bits are 1, otherwise yields 0. This is similar to multiplication ($1 * 1 = 1$, $1 * 0 = 0$, $0 * 1 = 0$, $0 * 0 = 0$).
- **OR** (disjunction, \/): Done with two bits, yields 1 if at least one of the input bits is 1, otherwise yields 0. This is similar to addition ($1 + 1 = 1$, $1 + 0 = 1$, $0 + 1 = 1$, $0 + 0 = 0$).

There are also other function such as XOR (exclusive OR, is 1 exactly when the inputs differ) and negated versions of AND and OR (NAND and NOR, give opposite outputs of the respective non-negated function). The functions are summed up in the following table (we all these kinds of tables **truth tables**):

x	y	NOT x	x AND y	x OR y	x XOR y	x NAND y	x NOR y	x NXOR y
0	0	1	0	0	0	1	1	1
0	1	1	0	1	1	1	0	0
1	0	0	0	1	1	1	0	0
1	1	0	1	1	0	0	0	1

x y NOT x x AND y x OR y x XOR y x NAND y x NOR y x NXOR y

In fact there exists more functions with two inputs and one output (16 in total, computing this is left as exercise :]). However not all are named -- we only use special names for the commonly used ones, mostly the ones in the table above.

An interesting thing is that we may only need one or two of these functions to be able to create all other function (this is called *functional completeness*); for example it is enough to only have *AND* and *NOT* functions together to be able to construct all other functions. Functions *NAND* and *NOR* are each enough by themselves to make all the other functions! For example $NOT\ x = x\ NAND\ x$, $x\ AND\ y = NOT\ (x\ NAND\ y) = (x\ NAND\ y)\ NAND\ (x\ NAND\ y)$, $x\ OR\ y = (x\ NAND\ x)\ NAND\ (y\ NAND\ y)$ etc.

Boolean algebra further tells us some basic laws we can use to simplify our expressions with these functions, for example:

- trivial laws:
 - ◆ $x\ AND\ 0 = 0$
 - ◆ $x\ OR\ 1 = 1$
 - ◆ $x\ AND\ 1 = x$
 - ◆ $x\ OR\ 0 = x$
 - ◆ $x\ AND\ x = x$
 - ◆ $x\ OR\ x = x$
 - ◆ commutativity of OR: $x\ OR\ y = y\ OR\ x$
 - ◆ commutativity of AND: $x\ AND\ y = y\ AND\ x$
 - ◆ associativity of OR: $x\ OR\ (x\ OR\ x) = (x\ OR\ x)\ OR\ x$
 - ◆ associativity of AND: $x\ AND\ (x\ AND\ x) = (x\ AND\ x)\ AND\ x$
 - ◆ ...
- distributive laws:
 - ◆ $x\ AND\ (y\ OR\ z) = (x\ AND\ y)\ OR\ (x\ AND\ z)$
 - ◆ $x\ OR\ (y\ AND\ z) = (x\ OR\ y)\ AND\ (x\ OR\ z)$
- De Morgan's laws:
 - ◆ $NOT\ (x\ AND\ y) = NOT(x)\ OR\ NOT(y)$
 - ◆ $NOT\ (x\ OR\ y) = NOT(x)\ AND\ NOT(y)$
- ...

By combining all of these simple functions it is possible to construct not only operations with whole numbers and traditional algebra, but also a whole computer that renders 3D graphics and sends multimedia over the Internet. For more details see **logic circuits**.

Base-2 Numeral System

While we may use a single bit to represent two values, we can group more bits together and become able to represent more values; the more bits we group together, the more values we'll be able to represent as possible combinations of the values of individual bits. The number of bits, or "places" we have for writing a binary number is called a number of bits or **bit width**. A bit width N allows for storing 2^N values -- e.g. with 2 bits we can store $2^2 = 4$ values: 0, 1, 2 and 3, in binary 00, 01, 10 and 11. With 3 bits we can store $2^3 = 8$ values: 0 to 7, in binary 000, 001, 010, 011, 100, 101, 110, 111. And so on.

At the basic level binary works just like the decimal (base 10) system we're used to. While the decimal system uses powers of 10, binary uses powers of 2. Here is a table showing a few numbers in decimal and binary:

decimal binary

0	0
1	1
2	10
3	11
4	100

decimal binary

5	101
6	110
7	111
8	1000

... ..

Conversion to decimal: let's see an example that utilizes the facts mentioned above. Let's have a number that's written as 10135 in decimal. The first digit from the right (5) says the number of 10^0 s (1s) in the number, the second digit (3) says the number of 10^1 s (10s), the third digit (1) says the number of 10^2 s (100s) etc. Similarly if we now have a number **100101** in binary, the first digit from the right (1) says the number of 2^0 s (1s), the second digit (0) says the number of 2^1 s (2s), the third digit (1) says the number of 2^2 s (4s) etc. Therefore this binary number can be converted to decimal by simply computing $1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 + 0 * 2^4 + 1 * 2^5 = 1 + 4 + 32 = 37$.

```
100101 = 1 + 4 + 32 = 37
|||||
\\\\\\\\_number of 2^0s (= 1s)
\\\\\\\\\\_number of 2^1s (= 2s)
\\\\\\\\\\_number of 2^2s (= 4s)
\\\\\\\\\\_number of 2^3s (= 8s)
\\\\\\\\\\_number of 2^4s (= 16s)
\\\\\\\\\\_number of 2^5s (= 32s)
```

To **convert from decimal** to binary we can use a simple algorithm that's again derived from the above. Let's say we have a number X we want to write in binary. We will write digits from right to left. The first (rightmost) digit is the remainder after integer division of X by 2. Then we divide the number by 2. The second digit is again the remainder after division by 2. Then we divide the number by 2 again. This continues until the number is 0. For example let's convert the number 22 to binary: first digit = $22 \% 2 = 0$; $22 / 2 = 11$, second digit = $11 \% 2 = 1$; $11 / 2 = 5$; third digit = $5 \% 2 = 1$; $5 / 2 = 2$; $2 \% 2 = 0$; $2 / 2 = 1$; $1 \% 2 = 1$; $1 / 2 = 0$. The result is **10110**.

NOTE: once we start grouping bits to create numbers, we typically still also keep the possibility to apply the basic Boolean operations to these bits. You will sometimes encounter the term **bitwise** operation which signifies an operation that works on the level of single bits by applying given function to bits that correspond by their position; for example a bitwise AND of values 1010 and 1100 will give 1000.

Operations with binary numbers: again, just as we can do arithmetic with decimal numbers, we can do the same with binary numbers, even the algorithms we use to perform these operations with pen and paper work basically the same. For example the following shows multiplication of 110 (6) by 11 (3) to get 10010 (18):

```
110 *
  11
----
110
110
----
10010
```

All of these operations can be implemented just using the basic boolean functions -- see logic circuits and CPUs.

In binary it is very simple and fast to divide and multiply by powers of 2 (1, 2, 4, 8, 16, ...), just as it is simply to divide and multiply by powers of 10 (1, 10, 100, 1000, ...) in decimal (we just shift the radix point, e.g. the binary number 1011 multiplied by 4 is 101100, we just added two zeros at the end). This is why as a programmer **you should prefer working with powers of two** (your programs can be faster if the computer can perform basic operations faster).

Binary can be very easily converted to and from hexadecimal and octal because 1 hexadecimal (octal) digit always maps to exactly 4 (3) binary digits. E.g. the hexadecimal number F0 is 11110000 in binary (1111 is always equivalent to F, 0000 is always equivalent to 0). This doesn't hold for the decimal

base, hence programmers often tend to avoid base 10.

We can work with the binary representation the same way as with decimal, i.e. we can e.g. write negative numbers such as -110101 or rational numbers (or even real numbers) such as 1011.001101. However in a computer memory there are no other symbols than 1 and 0, so we can't use extra symbols such as - or . to represent such values. So if we want to represent more numbers than non-negative integers, we literally have to only use 1s and 0s and choose a specific **representation/format/encoding** of numbers -- there are several formats for representing e.g. signed (potentially negative) or rational (fractional) numbers, each with pros and cons. The following are the most common number representations:

- **two's complement**: Allows storing integers, both positive, negative and zero. It is **probably the most common representation** of integers because of its great advantages: basic operations (+, -, *) are performed exactly the same as with "normal" binary numbers, and there is no negative zero (which would be an inconvenience and waste of memory). Inverting a number (from negative to positive and vice versa) is done simply by inverting all the bits and adding 1. The leftmost bit signifies the number's sign (0 = +, 1 = -).
- **sign-magnitude**: Allows storing integers, both positive, negative and zero. It's pretty straightforward: the leftmost bit in a number serves as a sign (0 means +, 1 means -) and the rest of the number is the distance from zero in "normal" representation. So e.g. a 4 bit number 0011 is 3 while 1011 is -3 (note that we have to know the bit width of the number here, e.g. on 8 bits -3 would be 10000011). The disadvantage is there are two values for zero (positive, 0000 and negative, 1000) which wastes a value and presents a computational inconvenience, and operations with these numbers are more complicated and slower (checking the sign requires extra code).
- **one's complement**: Allows storing integers, both positive, negative and zero. The leftmost bit signifies a sign, in the same way as with sign-magnitude, but numbers are inverted differently: a positive number is turned into negative (and vice versa) by inverting all bits. So e.g. 0011 is 3 while 1100 is -3 (again, bit width matters). The disadvantage is there are two values for zero (positive, 0000 and negative, 1111) which wastes a value and presents a computational inconvenience, and some operations with these numbers may be more complex.
- **fixed point**: Allows storing rational numbers (fractions), i.e. numbers with a radix point (such as 1101.011), which can also be positive, negative or zero. It works by imagining a radix point at some fixed position in the binary representation, e.g. if we have an 8 bit number, we may consider 5 leftmost bits to represent the whole part and 3 rightmost bits to be the fractional part (so e.g. the number 11010110 represents 11010.110). The advantage here is extreme simplicity (we can use normal integer numbers as fixed point simply by imagining a radix point). The disadvantage may be low precision and small range of representable values.
- **floating point**: Allows storing rational numbers in great ranges, both positive, negative and zero, plus some additional values such as infinity and not a number. It allows the radix point to be shifted which gives a potential for storing extremely big and extremely small numbers at the same time. The disadvantage is that float is extremely complex, bloated, wastes some values and for fast execution requires a special hardware unit (which most "normal" computers nowadays have, but are missing e.g. in some embedded systems).
- ...

As anything can be represented with numbers, binary can be used to store any kind of information such as text, images, sounds and videos. See data structures and file formats.

Binary numbers can nicely encode sets: one binary number can be seen as representing a set, with each bit saying whether an object is or is not present. For example an 8 bit number can represent a set of whole numbers 0 to 7. Consider e.g. a value $S1 = 10000101$ and $S2 = 01001110$; $S1$ represents a set { 0, 2, 7 }, $S2$ represents a set { 1, 2, 3, 6 }. This is natural and convenient, no bits are wasted on encoding order of numbers, only their presence or absence is encoded, and many set operations are trivial and very fast. For example the basic operations on sets, i.e. union, intersection, complement are simply performed with boolean operators OR, AND and NOT. Also checking membership, adding or removing numbers to the set etc. are very simple (left as an exercise for the reader lol; also another exercise -- in a similar fashion, how would you encode a multiset?). This is actually very useful and commonly used, for example chess engines often use 64 bit numbers to represent sets of squares on a chessboard.

See Also

- [unary](#)
 - [ternary](#)
 - [logic circuit](#)
 - [bit](#)
 - [hexadecimal](#)
 - [De Morgan's laws](#)
 - [data structure](#)
 - [data type](#)
-

bit_hack

Bit Hack

Bit [hacks](#) (also bit tricks, bit [magic](#), bit twiddling etc.) are simple clever formulas for performing useful operations with [binary](#) numbers. Some operations, such as checking if a number is power of two or reversing bits in a number, can be done very efficiently with these hacks, without using loops, [branching](#) and other undesirably slow operations, potentially increasing speed and/or decreasing size and/or memory usage of code -- this can help us [optimize](#). Many of these can be found on the [web](#) and there are also books such as *Hacker's Delight* which document such hacks.

Basics

Basic bit manipulation techniques are common and part of general knowledge so they won't be listed under hacks, but for sake of completeness and beginners reading this we should mention them here. Let's see the basic bit manipulation operators in [C](#):

- `|` (bitwise [OR](#)): Performs the logical OR on all corresponding bits of two operands, e.g. `0b0110 | 0b1100` gives `0b1110` (14 in decimal). This is used to set bits and combine flags (options) into a single numeric value that can easily be passed to function etc. For example to set the lowest bit of a number to 1 just do `myNumber | 1`. Now consider e.g. `#define OPTION_A 0b0001`, `#define OPTION_B 0b0010` and `#define OPTION_C 0b0100`, now we can make a single number that represents a set of selected options e.g. as `OPTION_C | OPTION_B` (the value will be `0101` and says that options B and C have been selected).
- `&` (bitwise [AND](#)): Performs the logical AND on all corresponding bits of two operands, e.g. `0b0110 & 0b1100` gives `0b0100` (4 in decimal). This may be used to mask out specific bits, to check if specific bits are set (useful to check the set flags as mentioned above) or to clear (set to zero) specific bits. Consider the flag example from above, if we want to check if value `x` has e.g. the option B set, we simply do `x & OPTION_B` which results in non-zero value if the option is set. Another example may be `myNumber & 0b00001111` (in practice you'll see hexadecimal values, i.e. `myNumber & 0x0F`) which masks out the lowest 4 bits of *myNumber* (which is equivalent to the operation [modulo](#) 16).
- `~` (bitwise [NOT](#)): Flips every bit of the number -- pretty straightforward. This is used e.g. for clearing bits as `x & ~(1 << 3)` (clear 4th bit of `x`).
- `^` (bitwise [XOR](#)): Performs the logical XOR on all corresponding bits of two operands, e.g. `0b0110 ^ 0b1100` gives `0b1010` (10 in decimal). This is used to e.g. flip specific bits.
- `<<` and `>>` (binary shift left/right): Performs bitwise shift left or right (WATCH OUT: shifting by data type width or more is undefined behavior in C). This is typically used to perform fast multiplication (left) and division (right) by powers of two (2, 4, 8, 16, ...), just as we can quickly multiply/divide by 10 in decimal by shifting the decimal point. E.g. `5 << 3` is the same as `5 * 2^3 = 5 * 8 = 40`.
- We also sometimes use the logical (i.e. NOT bitwise) operators `&&` (AND), `||` (OR) and `!` (NOT); the difference against bitwise operators is that firstly they work with the whole value (i.e. not individual bits), considering 0 to be *false* and anything else to be *true*, and secondly they may employ a bit more complexity, e.g. [short circuit](#) evaluation.

Specific Bit Hacks

{ Work in progress. I'm taking these from various sources such as the *Hacker's Delight* book or web and rewriting them a bit, always testing. Some of these are my own. ~drummyfish }

TODO: stuff from this gophersite: [gopher://bitreich.org/0/thaumaturgy/bithacks](http://bitreich.org/0/thaumaturgy/bithacks)

Unless noted otherwise we suppose C syntax and semantics and integer data types, but of course we mainly want to express formulas and patterns you can use anywhere, not just in C. Keep in mind all potential dangers, for example it may sometimes be better to write an idiomatic code and let compiler do the optimization that's best for given platform, also of course readability will worsen etc. Nevertheless as a hacker you should know about these tricks, it's useful for low level code etc.

2^N: $1 \ll N$

absolute value of x (two's complement):

```
int t = x >> (sizeof(x) * 8 - 1);
x = (x + t) ^ t;
```

average x and y without overflow: $(x \& y) + ((x \wedge y) \gg 1)$ { TODO: works with unsigned, not sure about signed. ~drummyfish }

clear (to 0) Nth bit of x: $x \& \sim(1 \ll N)$

clear (to 0) rightmost 1 bit of x: $x \& (x - 1)$

conditionally add (subtract etc.) x and y based on condition c (c is 0 or 1): $x + ((0 - c) \& y)$, this avoids branches AND ALSO multiplication by c, of course you may replace + by another operators.

count 0 bits of x: Count 1 bits and subtract from data type width.

count 1 bits of x (8 bit): We add neighboring bits in parallel, then neighboring groups of 2 bits, then neighboring groups of 4 bits.

```
x = (x & 0x55) + ((x >> 1) & 0x55);
x = (x & 0x33) + ((x >> 2) & 0x33);
x = (x & 0x0f) + (x >> 4);
```

count 1 bits of x (32 bit): Analogous to 8 bit version.

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f);
x = (x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff);
x = (x & 0x0000ffff) + (x >> 16);
```

count leading 0 bits in x (8 bit):

```
int r = (x == 0);
if (x <= 0x0f) { r += 4; x <= 4; }
if (x <= 0x3f) { r += 2; x <= 2; }
if (x <= 0x7f) { r += 1; }
```

count leading 0 bits in x (32 bit): Analogous to 8 bit version.

```
int r = (x == 0);
if (x <= 0x0000ffff) { r += 16; x <= 16; }
if (x <= 0x00ffffff) { r += 8; x <= 8; }
if (x <= 0x0fffffff) { r += 4; x <= 4; }
if (x <= 0x3fffffff) { r += 2; x <= 2; }
if (x <= 0x7fffffff) { r += 1; }
```

divide x by 2^N: $x \gg N$

divide x by 3 (unsigned at least 16 bit, x < 256): $((x + 1) * 85) \gg 8$, we use kind of a fixed point multiplication by reciprocal (1/3), on some platforms this may be faster than using the divide instruction, but not always (also compilers often do this for you). { I checked this particular trick and it gives exact results for any $x < 256$, however this may generally not be the case for other constants than 3. Still even if not 100% accurate this can be used to approximate division. ~drummyfish }

divide x by 5 (unsigned at least 16 bit, x < 256): $((x + 1) * 51) \gg 8$, analogous to divide by 3.

get Nth bit of x: $(x \gg N) \& 0x01$

is x a power of 2?: $x \&\& ((x \& (x - 1)) == 0)$

is x even?: $(x \& 0x01) == 0$

is x odd?: $(x \& 0x01)$

isolate rightmost 0 bit of x: $\sim x \& (x + 1)$

isolate rightmost 1 bit of x: $x \& (\sim x + 1)$ (in two's complement equivalent to $x \& -x$)

log base 2 of x: Count leading 0 bits, subtract from data type width - 1.

maximum of x and y: $x \wedge ((0 - (x < y)) \& (x \wedge y))$

minimum of x and y: $x \wedge ((0 - (x > y)) \& (x \wedge y))$

multiply x by 2^N: $x \ll N$

multiply by 7 (and other numbers close to 2^N): $(x \ll 3) - x$

next higher or equal power of 2 from x (32 bit):

```
x--;
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
x = x + 1 + (x == 0);
```

parity of x (8 bit):

```
x ^= x >> 1;
x ^= x >> 2;
x = (x ^ (x >> 4)) & 0x01;
```

reverse bits of x (8 bit): We switch neighboring bits, then switch neighboring groups of 2 bits, then neighboring groups of 4 bits.

```
x = ((x >> 1) & 0x55) | ((x & 0x55) << 1);
x = ((x >> 2) & 0x33) | ((x & 0x33) << 2);
x = ((x >> 4) & 0x0f) | (x << 4);
```

reverse bits of x (32 bit): Analogous to the 8 bit version.

```
x = ((x >> 1) & 0x55555555) | ((x & 0x55555555) << 1);
x = ((x >> 2) & 0x33333333) | ((x & 0x33333333) << 2);
x = ((x >> 4) & 0x0f0f0f0f) | ((x & 0x0f0f0f0f) << 4);
x = ((x >> 8) & 0x00ff00ff) | ((x & 0x00ff00ff) << 8);
x = ((x >> 16) & 0x0000ffff) | (x << 16);
```

rotate x left by N (8 bit): $(x \ll N) \mid (x \gg (8 - N))$ (watch out, in C: $N < 8$, if storing in wider type also do $\& 0xff$)

rotate x right by N (8 bit): analogous to left rotation, $(x \gg N) \mid (x \ll (8 - N))$

set (to 1) Nth bit of x: $x \mid (1 \ll N)$

set (to 1) the rightmost 0 bit of x: $x \mid (x + 1)$

set or clear Nth bit of x to b: $(x \& \sim(1 \ll N)) \mid (b \ll N)$

sign of x (returns 1, 0 or -1): $(x > 0) - (x < 0)$

swap x and y (without tmp var.): $x \wedge= y; y \wedge= x; x \wedge= y;$ or $x -= y; y += x; x = y - x;$

toggle Nth bit of x: $x \wedge (1 \ll N)$

toggle x between A and B: $(x \wedge A) \wedge B$

x and y have different signs?: $(x > 0) == (y > 0), (x \leq 0) == (y \leq 0)$ etc. (differs on 0:0 behavior)

TODO: the ugly hacks that use conversion to/from float?

See Also

- [De Morgan's laws](#)
- [fast inverse square root](#)
- [optimization](#)

bit

Bit

Bit (for *binary digit*, symbol *b*, also *shannon*) is the lowest commonly used unit of [information](#), equivalent to a choice between two equally likely options (e.g. an answer to the question "Was the coin flip heads?"), in computers used as the smallest unit of [memory](#), with 1 bit being able to hold exactly one value that can be either 1 or 0. From bit a higher memory unit, [byte](#) (8 bits), is derived. In [quantum computing](#) the equivalent of a bit is [qubit](#), in [ternary](#) computers the analogy is [trit](#).

Can there exist a smaller quantity of information than 1 bit? Well, yes, for sure we can get zero information and it certainly also makes sense to speak of fractions of bits; for example one [decimal](#) digit carries $\log_2(10) \approx 3.32$ bits of information. [Entropy](#) is also measured in bits and can get smaller than 1 bit, e.g. for an unfair coin toss; an answer to the question "Will the Sun rise tomorrow?" gives less than 1 bit of information -- in fact it gives almost no information as we know the answer will most definitely be yes, though the certainty can never be absolute. Another idea: imagine there exist two people for whom we want to know, based on their sexes, whether they may reproduce together -- answer to this question takes 1 bit (yes or no) and to obtain it we have to know both of these people's sexes so we can say whether they differ. Now if we only know the sex of one of them, then in the context of the desired answer we might perhaps say we have a half of one bit of information, as if we also know the other one's sex (the other half of the bit), we get the whole 1 bit answer.

bitreich

Bitreich

{ Researching this on-the-go, send me corrections, thanks. ~drummyfish }

Bitreich is a small, obscure underground group/movement of programmers who greatly value minimalism/simplicity, oppose the evil and degeneration of modern mainstream technology and aim for making the world a better place mainly through simpler technology. They seem to belong to the cluster of "minimalist programmer groups", i.e. they are similar to suckless (which in their manifesto they see as a failed project), reactionary software and our very own LRS, sharing many values such as minimalism, Unix philosophy, preference and love of the C language, carrying on some of the hacker culture heritage, though of course they also have their own specifics that will make them different and even disagreeing with us and others on occasion, e.g. on copyleft (unlike us, they seem to greatly prefer the GPL), terminology (yeah, they seems to prefer "open source") and probably also things like privacy (though the craze doesn't seem to go too far, many have listed their real names and addresses) etc.

According to the gopherhole Bitreich started on 17.8.2016 -- the founder (or at least one of them?) seems to be 20h (Christoph Lohmann according to the user profile), a guy formerly active in suckless (can be found on their website), who even gave an interview about Bitreich to some radio/magazine/whatever. It seems Bitreich originated in Germany. As of 2023 they list 12 official member profiles (the number of lurker followers will of course be a much high number, there seem to be even bitreich subcommunities in other countries such as Italy). They are mostly present on gopher (gopher://bitreich.org), which they greatly promote, and IRC (ircs://irc.bitreich.org:6697/#bitreich-en). There are also Tor hidden services etc.; their website at bitreich.org seems to be purposefully broken in protest of the web horror.

Some of their ideas and philosophy seems to be very based, e.g. preference of KISS/older protocols (gopher, ftp, IRC, ...), "users are programmers" (opposing division into users as consumers and developers as overlords), "bug reports are patches", "programs can be finished" etc.

Bitreich is also about humor and fun (sometimes so much so that it's not clear if something is a joke or serious stuff -- maybe because it's partly both). They invented analgram, an authentication method based on analprints (alternative to fingerprint authentication). They put a snapshot of their source code into an actual Arctic vault in Greenland, to be preserved for millennia. Often there appear parodies of whatever is currently hyping in the mainstream, e.g. NFTs, "big data", AI, blockchain etc. { There's also some stuff going on with memes and cooking recipes but TBH I didn't get it. ~drummyfish }

Some interesting projects they do:

- **Bitreichcon**: annual conference, running since 2017. Their slides can be downloaded in plain text.
- **Bitreich radio**
- **Day Of The GrParazyd**: point and click adventure game. { Didn't even take a look at this yet, sorry, no idea what it really is :D ~drummyfish }
- **The Gopher Lawn**: directory/index of gopherspace, categorizing gopherhole links.
- **The Gopher Times**: a very cool printable magazine (in both pdf and plain text), git clone `git://bitreich.org/tgtimes`.
- A number of smaller utilities/programs and parody stuff (see their gopherhole).
- Keeping infrastructure to host stuff they see as valuable.
- ...

See Also

- suckless
- reactionary software
- less retarded software
- KISS

black

Black

Black, a color whose politically correct name is *afroamerican*, is a color that we see in absence of any light.

blender

Blender

Blender (also Blunder) is a greatly complex "open-source" 3D modeling and rendering software -- one of the most powerful and "feature-rich" ones, even compared to proprietary competition -- used not only by the FOSS community, but also the industry (commercial games, movies etc.), which is an impressive achievement in itself, however Blender is also a capitalist software suffering from many not-so-nice features such as bloat, update culture, hardware discrimination and centralized control.

After version 2.76 Blender started REQUIRING OpenGL 2.1 due to its "modern" EEVEE renderer, deprecating old machines and giving a huge fuck you to all users with incompatible hardware (for example the users of RYE laptops). This new version also stopped working with the free Nouveau driver, forcing the users to use NVidia's proprietary drivers. Blender of course doesn't at all care about this. { I've been forced to use the extremely low FPS software GL version of Blender after 2.8. ~drummyfish }

Are there good alternatives to Blender? Some programs to check out are wings3d (seems like the best candidate), k3d, meshlab and mm3d (looks like bloat), also for some things possibly FreeCAD. Remember you can also make models manually :-) Formats like obj can be hand-written.

See Also

- 3D modeling

bloat

Bloat

Bloat is a very wide term that in the context of software and technology means overcomplication, unnecessary complexity and/or extreme growth in terms of source code size, overall complexity, number of dependencies, redundancy, unnecessary and/or useless features (e.g. feature creep) and resource usage, all of which lead to inefficient, badly designed technology with bugs (crashes, unusable features, memory leaks, security vulnerabilities, ...), as well as great obscurity, ugliness, **loss of freedom** and waste of human effort. Simply put bloat is burdening bullshit. Bloat is extremely bad and one of the greatest technological issues of today. Creating bloat is bad engineering at its worst and unfortunately it is what's absolutely taking over all technology nowadays, mostly due to capitalism causing commercialization, consumerism, rushed "just works" products, creating demand for newer hardware and so on, also allowing incompetent people ("let's push more women/minorities into programming") trying to take on jobs they are in no way qualified to do.

A related but different term is **bloatware**; it's more commonly used among normie users and stands for undesirable programs that eat up computer resources, usually being preinstalled by the computer manufacturer etc. Further on we'll rather focus on bloat as defined before.

TODO: history of bloat?

LRS, suckless and some others rather small groups are trying to address the issue and write software that is good, minimal, reliable, efficient and well functioning. Nevertheless our numbers are very small and in this endeavor we are basically standing against the whole world and the most powerful tech corporations. The issue lies not only in capitalism pushing bloat but also in common people not seeing the issue (partly due to the capitalist propaganda promoting maximalism), no one is supporting the few people who are genuinely trying to create good tools, on the contrary such people often face hostility from the mainstream.

The issue of bloat may of course appear outside of the strict boundaries of computer technology, nowadays we may already observe e.g. **science bloat** -- science is becoming so overcomplicated (many times on purpose, e.g. by means of bullshit science) that 99% people can NOT understand it, they have to BELIEVE "scientific authorities", which does not at all differ from the dangerous blind religious behavior. Any time a new paper comes out, chances are that not even SCIENTISTS from the same field but with a different specialization will understand it in depth and have to simply trust its results. This combined with self-interest obsessed society gives rise to soyence and large scale brainwashing and spread of "science approved" propaganda.

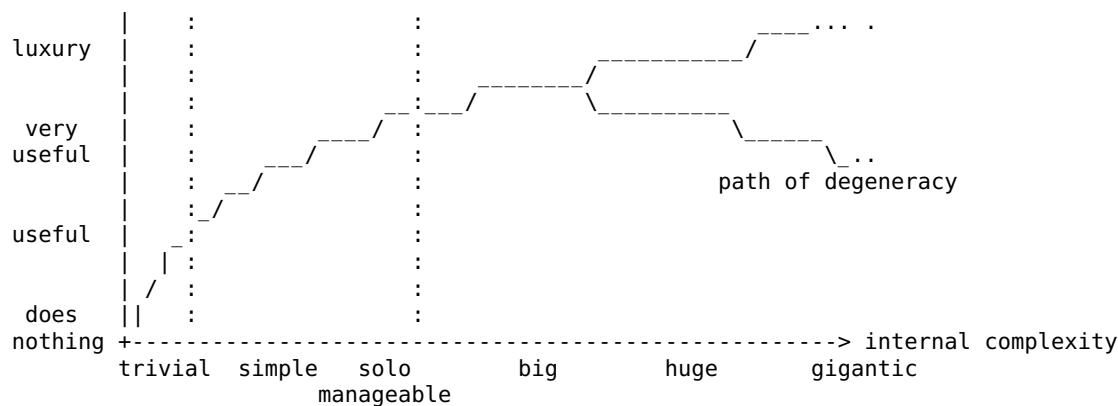
Some metrics traditionally used to measure bloat include **lines of source code**, **programming language used** (some languages are bloated themselves and inherently incapable of producing non-bloat, also choice of language indicates the developer's priorities, skills etc.), **number of dependencies** (packages, libraries, hardware, ...), **binary size** (size of the compiled program), **compile time**, **resource usage** (RAM, CPU, network usage, ...), **performance** (FPS, responsiveness, ...), **anti features** (GUI, DRM, auto updates, file formats such as XML, ...), **portability**, number of implementations, size of specification, number of developers and others. Some have attempted to measure bloat in more sophisticated ways, e.g. the famous *web bloat score* (<https://www.webbloatscore.com/>) measures bloat of websites as its total size divided by the page screenshot size (e.g. YouTube at 18.5 vs suckless.org at 0.386). It has been observed that **software gets slower faster than hardware gets faster**, which is now known as Wirth's law; this follows from Moore's law (speed of hardware doubles every 24 months) being weaker than Gate's law (speed of software halves every 18 months); or in other words: the stupidity of soydevs outpaces the brilliancy of geniuses.

Despite this there isn't any completely objective measure that would say "this software has exactly X % of bloat", bloat is something judged based on what we need/want, what tradeoffs we prefer etc. The answer to "how much bloat" there is depends on the answer to **"what really is bloat?"**. To answer this question most accurately we can't limit ourselves to simplifications such as lines of code or number of package dependencies -- though these are very good estimates for most practical purposes, a more accurate insight is obtained by carefully asking what *burdens* and *difficulties* of ANY kind come with given technology, and also whether and how much of a necessary evil they are. Realize for example that if your software doesn't technically require package X to run or be compiled, package X may be de facto required for your software to exist and work (e.g. a pure multiplayer game client won't have the server as a dependency, but it will be useless without a server, so de facto all bloat present in the server is now in a wider sense also the client's burden). So if you've found a program that's short and uses no libraries, you still have to check whether the language it is written in isn't bloated itself, whether the program relies on running on a complex platform that cannot be implemented without bloat, whether some highly complex piece of hardware (e.g. GPU or 8GB of RAM) is required, whether it relies on some complex Internet service etc. You can probably best judge the amount of bloat most objectively by asking the following: if our current technology instantly disappeared, how hard would it be to make this piece of technology work again? This will inevitably lead you to investigating how hard it would be to implement all the dependencies etc.

One of a very frequent questions you may hear a noob ask is **"How can bloat limit software freedom if such software has a free (or **"FOSS"**) license?"** Bloat de-facto limits some of the four essential freedoms (to use, study, modify and share) required for a software to be free. A free license grants these freedoms legally, but if some of those freedoms are subsequently limited by other circumstances, the software becomes effectively less free. It is important to realize that **complexity itself goes against freedom** because a more complex system will inevitably reduce the number of people being able to execute freedoms such as modifying the software (the number of programmers being able to understand and modify a trivial program is much greater than the number of programmers being able to understand and modify a highly complex million LOC program). This is not any made up reason, it is actually happening and many from the free software community try to address the issue, see e.g. HyperbolaBSD policies on accepting packages which rejects a lot of popular "legally free" software on grounds of being bloat (systemd, dbus, zstd, protobuf, mono, https://wiki.hyperbola.info/doku.php?id=en:philosophy:incompatible_packages). As the number of people being able to execute the basic freedom drops, we're approaching the scenario in which the software is de-facto controlled by a small number of people who can (e.g. due to the cost) effectively study, modify and maintain the program -- and a program that is controlled by a small group of people (e.g. a corporation) is by definition proprietary. If there is a web browser that has a free license but you, a lone programmer, can't afford to study it, modify it significantly and maintain it, and your friends aren't able to do that either, when the only one who can practically do this is the developer of the browser himself and perhaps a few other rich corporations that can pay dozens of full time programmers, then such browser cannot be considered free as it won't be shaped to benefit you, the user, but rather the developer, a corporation.

How much bloat can we tolerate? We are basically trying to get the most for the least price. The following diagram attempts to give an answer:

external			
"richness"			
A			
shiny		:	:
bullshit	NO :	YES	: NO



The **path of degeneracy** drawn in the graph shows how from a certain breaking point (which may actually appear at different places, the diagram is simplified) many software projects actually start getting less powerful and useful as they get more complex -- not all, some project really do stay on the path of increasing their "richness", but this requires great skills, experience, expertise and also a bit of lucky circumstances; in the zone of huge complexity projects start to get extremely difficult to manage -- non-primary tasks such as organization, maintenance and documentation start taking up so many resources that the primary task of actually programming the software suffers, the project crumbles under its own weight and the developers just try to make it fall slower. This happens mostly in projects made by incompetent soydevs, i.e. most today's projects. { Thanks to a friend for pointing out this idea. ~drummyfish }

Please note there may arise some disagreement among minimalist group about where the band is drawn exactly, especially old Unix hackers could be heard arguing for allowing (or even requiring) even trivial programs, maybe as long as the source code isn't shorter than the utility name, but then the discussion might even shift to questions like "what even is a program vs what's just a 10 characters long line" and so on.

As a quick heuristic for judging programs you can really take a look at the lines of code (as long as you know it's a simplification that ignores dependencies, formatting style, language used etc.) and use the following classes (basically derived from how suckless programs are often judged):

- < 10: Extremely small but may be useful, may be also too trivial for such small size to be justifiable, can aim to be completely bug-free. Example could be the cat program.
- 11 to 100: Very small, can be debugged to a great level, many greatly useful utilities, e.g. compression programs, may fit this class.
- 101 to 1000: Small "bigger" kinds of programs, often very minimalist implementations of programs that are usually not minimalist in nature like window managers, interactive text editors, web browsers and so on. Simplified version of comun language, called *minicomun*, fits here.
- 1001 to 5000: Still considered small, a bit more "feature rich" kind of previous class, you may find minimalist 3D games here, quite powerful programming languages, libraries handling complex file formats (that weren't designed to be minimalist) etc. Currently a lot of LRS programs like SAF, small3dlib and comun would fall here.
- 5001 to 10000: Often imposed upper limit on suckless programs, these programs aren't the smallest possible but may still be called minimalist, they are easily manageable by a single man without much hassle, anyone can modify them and there is a comfortable margin for implementing many "comfort" and fancy features that aren't complete BS. Anarch might be given as an example (if we subtract lines of code taken by game data and count only pure engine code).
- 10001 to 100000: Here things start to be called real bloat but may still be accepted as a compromise, not an "insanely bloated" program, we have to judge on a case by case basis as the transition towards bloat is gradual, but generally projects here must focus on not growing bigger, priority should be on minimizing. We have to consider anything here bloat unless proven otherwise. Minimalist projects end up here when trying to combine minimalism with some mainstream concept, e.g. implementing a complete operating system with all the standard features, trying to reimplement some mainstream, non-minimalist program etc. Example is tcc, the C compiler that has a little over 20000 LOC. Also many "good old" mainstream programs like Doom fall here.
- more: Basically just bloat, some operating systems can perhaps argue they are comparatively small even within this category, but as a matter of fact very few people can manage a codebase this big,

issues of bloat start to play a very significant role here, the project should most likely be split or rewritten from scratch in much more simplified way.

Yes, **bloat is also unecological** and no, it cannot be fixed by replacing fossil fuel cars with cars that run on grass and plastic computers by computers made from recycled cardboards mixed with composted horse shit. It is the immense volume of human ACTIVITY that's required by the bloated technology all around the globe that's inherently unecological by wasting so much effort, keeping focus on maximalism, growth and preventing us from frugality and minimizing resource waste. Just as any other bullshit that requires immense resources to just keep maintaining -- great complexity is just absolutely incompatible with ecology and as much as you dislike it, to achieve truly eco-friendly society we'll have to give up what we have now in favor of something orders of magnitude more simple and if you think otherwise, you are just yet too unexperienced (or remained purposefully ignorant) to have seen the big picture already. Consider that your program having bullshit dependencies such as Python, JavaScript, C++, Java, OpenGL, Vulkan, GPU, VR sets, gigabytes of RAM etcetc. requires having the inherently unecological system up, it needs millions of people doing bullshit jobs that are inherently wasting resources, increasing CO2 and making them not focus on things that have to be done -- yes, even if we replace plastic straws with paper straws. All those people that make the thousand pages standards that are updated every year, reviews of those standards, writing tons and tons of tests for implementations of those standards, electing other people to make those standards, testing their tests, implementing the standards themselves, optimizing them, all of that collectively needing many billions of lines of code and millions of hours of non-programming activities, it all requires complex bureaucracy, organization and management (complex version control systems, wikis, buildings, meeting spaces, ...) and communication tools and tons of other bullshit recursively spawning more and more waste -- all of these people require cars to go to work every day (even if some work from home, ultimately only a few can work from home 100% of the time and even so millions others need to physically go to factories to make all those computers, electricity, chair, food and other things those people need), they require keeping a high bandwidth 100% uptime global Internet network maintained, all of this requiring extra buildings, offices, factories, roads, buildings for governments overseeing the building of those buildings, maintenance of those roads etcetc. A newbie programmer (99.99999% programmers in the field nowadays) just don't see all this because they lack the big picture, a woman forced into programming has hard time comprehending an if statement, how do you expect her to see the deep interconnections between technology and society -- she may know that OpenGL is "something with graphics" and it's just there on every computer by default, she can't even picture the complexity that's behind what she sees on the screen. Hence the overall retardation. You just cannot have people living ecologically and at the same time have what we have now. So yes, by supporting and/or creating bloat you are killing the planet, whether you agree with it or not. No, you can't find excuses out of this, no, paper straws won't help, just admit you love point and click "programming without math" of your own shitty Minecraft clones in Godot even for the price of eliminating all life on Earth, that's fine (no it's not but it's better to just not bullshit oneself).

{ Fucking hell this shit has gone too far with the newest supershit gayme called Cities Skyline II, I literally can't anymore, apparently the game won't run smoothly even on Earth's most advanced supercomputer because, as someone analyzed, the retarddevs use billion poly models for pedestrians without any LOD, I bet they don't even know what it is, they probably don't even know what a computer is, these must be some extra retarded soy idiots making these games now. Though I knew it would come to this and that it will be getting yet much worse, I am always still surprised, my brain refuses to believe anyone would let such a piece or monstrous shit to happen. This just can't be real anymore. ~drummyfish }

Typical Bloat

The following is a list of software usually considered a good, typical example of bloat. However keep in mind that bloat is a relative term, for example vim can be seen as a minimalist suckless editor when compared to mainstream software (IDEs), but at the same time it's pretty bloated when compared to strictly suckless programs.

- Web since the onset of "web 2.0" has been steadily becoming more and more bloated with things such as Adobe Flash and JavaScript (and billions of its web frameworks). By today the situation about web bloat is reaching almost unbearable levels, especially in modern sites such as YouTube. For a great read see The Website Obesity Crisis.
- Ads, spyware, DRM, anti-cheats, anti-viruses, anti-sharing, anti-repair and other anti-user "features" are bloat.

- Desktop environments such as KDE and GNOME. The concept of a desktop environment itself is often considered bloat.
- Windows: one of the best examples of how software should NOT be done.
- Blender: quite useful FOSS 3D editor which however integrates things like a whole video editor, game engine, several renderers, scripting language with text editor and so on.
- CMake: gigantic build system that currently sits on top of a sky-high sandwich of other build systems, its number of dependencies is bigger than the number of retards in observable universe (known as drummyfish's number).
- D-Bus
- Docker
- Electron: GUI framework infamous for its huge resource consumption.
- flatpak: Absolutely horrible "application distribution/execution platform???" with package management, sandboxes and all that kind of shit.
- Systemd: Huge anti-unix do-it-all system taking over GNU/Linux.
- Virtual machines/environments/sandboxes, big abstraction sandwiches (e.g. program running in an interpreter running in a sandbox inside web browser that's running in a virtual machine that's running on an operating system).
- Firefox, Chromium and other mainstream web browsers.
- Java, Python and similar languages.
- IDEs such as VSCode or NetBeans.
- Big game engines such as Unreal, Unity or Godot.
- Practically all commercial games made in the 21st century such as World of Warcraft, Call of Duty etc.
- pulse audio
- office programs (e.g. M\$ Office and LibreOffice) and a lot of rich text
- Neural networks aka "AI" that is forced into everything nowadays.
- ...

Some of these programs may be replaced with smaller bloat that basically does the same thing (e.g. produces the same output) just with less bullshit around, for example Libreoffice with Ted, Godot with Irrlicht, Firefox with badwolf etc., however many times the spectacular pompous results these programs produce just cannot essentially be reproduced by anything minimal, wanting to achieve this is really a beginner mistake, the same as wanting to achieve the "Windows experience" on a GNU system. You will never be able to make an Unreal Engine style graphics with a minimalist game engine, just like you won't be able to shoot up your school with well written poetry (in both cases the former is something bad that however most Americans want to do, the latter is something truly good they should want instead). To truly get rid of bloat one has to become able to use truly minimal programs; this means unlearning the indoctrination that "bigger results are better", one has to understand that minimal results themselves are superior AND in addition allow using superior programs (i.e. minimal ones).

Medium And Small Bloat

Besides the typical big programs that even normies admit are bloated there exists also a smaller bloat which many people don't see as such but which is nevertheless considered unnecessarily complex by some experts and/or idealists and/or hardcore minimalists, including us.

Small bloat is a subject of popular jokes such as "OMG he uses a unicode font -- BLOAT!!!!". These are good jokes, it's nice to make fun out of one's own idealism. But watch out, this doesn't mean small bloat is only a joke concept at all, it plays an important role in designing good technology. When we identify something as *small bloat*, we don't necessarily have to completely avoid and reject that concept, we may just try to for example make it optional. In context of today's PCs using a Unicode font is not really an issue for performance, memory consumption or anything else, but we should keep in mind it may not be so on much weaker computers or for example post-collapse computers, so we should try to design systems that don't depend on Unicode.

Also remember that relatively small libraries for things that are easily done without a library, such as fixed point arithmetic, are also bloat.

Small/medium bloat includes for example:

- floating point (complex standard with design issues, requires special hardware for acceleration, fixed point is better)
- config files (and other unnecessary file I/O that requires a file I/O library, not all computers have file systems, configs should be part of source code)
- directories (just have all files on the same level and prefix their file names to organize them)
- library linking (header only libraries are better)
- any GPU, OpenGL (complex hardware and specifications, not all computers have complex GPUs, software rendering is better)
- Unicode (big specification requiring special libraries and big fonts, ASCII is better)
- antialiasing (just ignore aliasing, use low resolution textures etc.)
- 64 bit architectures (they only exist to allow ungodly amounts of RAM, 32 bits completely suffice for any computation, many times even 16 or 8 bits are enough)
- proportional font (fixed width font is better)
- linking, build systems/scripts, makefiles, directories and multiple source code files (just using a compiler or a few-line building shell script, single file source code, header only libraries and single compilation unit programs are better)
- infix notation (postfix notation is better)
- any GUI, window managers (pure text mode is better)
- operating system (bare metal is better)
- multithreading, parallelism, virtual memory, ...
- encryption, security, memory safety (just don't care and/or don't handle sensitive data with computers connected to the internet, don't live in a shitty society)
- X11 (just pure screen drawing is better)
- database software (plain files are better, see flatfile)
- C (something in between C and brainfuck would be likely ideal, e.g. comun or Forth)
- glibc, gcc, clang etc. (better alternatives are tcc, musl, uclibc etc.)
- letter accents/diacritics (can normally be ignored in most languages that use them)
- jpg, png, svg and similar formats (e.g. ppm or farbfeld is better)
- syntax highlight, text formatting, rich text and just colors anywhere they aren't absolutely necessary
- html, markdown (plain text is better)
- x86 instruction set (TODO: what's better? probably some RISC)
- any non-public-domain license (any legal burden introduced by a license is unnecessary bloat)
- dynamic linking/libraries (static linking is better, see Stali)
- web 1.0, gemini (gopher or FTP is better)
- mouse (keyboard is better)
- TCP (UDP is probably better)
- vim (things like ed are probably better?)
- sound (picture is usually enough)
- high resolution (640x480 is probably the maximum you'll ever need, lower resolution takes less RAM and makes rendering faster)
- true color (256 colors, e.g. 332 palette, is better, even 1 bit displays suffice for most things), high FPS (25 is more than enough), high resolution (320 x 240 is more than enough) etc.
- GNU Unix utils (things like busybox or sbash are probably better)
- data types (untyped or single type is better, everything can be just a number)
- package managers (just don't use them, install just a few programs manually, or at least make package managers as simple as possible)
- electricity (mechanical computers may be just fine)
- computers (pen and paper or counting with rocks is better)
- anything wireless (wifi, mice, ...)
- ...

Non-Computer Bloat

The concept of bloat can be applied even outside the computing world, e.g. to non-computer technology, art, culture, law etc. Here it becomes kind of synonymous with bullshit, but using the word *bloat* says we're seeing the issue from the point of view of someone acquainted with computer bloat. Examples include:

- clothes
- decorations (body, house, ...)
- cars

- using languages other than English or Esperanto
- luxury (big house, yacht with a swimming pool, ...)
- having electricity at home
- ...

See Also

- harmful
- maximalism
- obscurity
- shit

bloat_monopoly

Bloat Monopoly

Bloat monopoly is an exclusive control over or de-facto ownership of software or even a whole area of technology not by legal means but by means of bloat, or generally just abusing bloat in ways that lead to gaining monopolies, e.g. by establishing standards or even legal requirements (such as the EU mandatory content filters) which only the richest may conform to. Even if given software is FOSS (that is its source code is public and everyone has basic legal rights to it), it can be malicious due to bloat, for example it can still be made **practically** controlled exclusively by the developer because the developer is the only one with sufficient resources and/or know-how to be able to execute the basic rights such as meaningful modifications of the software, which goes against the very basic principle of free software.

Example: take a look at the web and how Google is gaining control over it by getting the search engine monopoly. It is very clear web along with web browsers has been becoming bloated to ridiculous levels -- this is not a coincidence, bloat is pushed by corporations such as Google to eliminate possible emerging competition. If practically all websites require JavaScript, CSS, HTTPS and similar nonsense, it is becoming much more difficult to crawl them and create a web index, leaving the possibility to crawl the web mostly to the rich, i.e. those who have enough money, time and know-how to do this. Alongside this there is the web browser bloat -- as websites have become extremely complex, it is also extremely complex to make and maintain a web browser, which is why there is only a few of them, all controlled (despite FOSS licenses) by corporations and malicious groups, one of which is Google itself. For these reasons Google loves bloat and encourages it, e.g. simply by ranking bloated webpages better in their search results, and of course by other means (sponsoring, lobbying, advertising, ...).

Bloat monopoly is capitalism's circumvention of free licenses and taking advantage of their popularity. With bloat monopoly capitalists can stick a FOSS license to their software, get an automatic approval (**openwashing**) of most "open-source" fanbois as well as their free work time, while really staying in control almost to the same degree as with proprietary software.

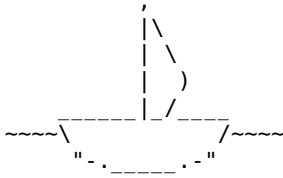
At the time of writing this if you want to compile so called "open source" Android, you will need a computer with at least 400 GB of space, 16 GB of RAM (but recommended is 32 or 64), a modern 64 bit CPU with multiple cores, and many hours of computational time. How long before we need a million dollar supercomputer to compile an "open source" program? Now ask yourself, is this still real freedom?

Examples of bloat monopoly include mainstream web browsers (furryfox, chromium, ...), Android, Linux, Blender etc. This software is characteristic by its difficulty to be even compiled, yet alone understood, maintained and meaningfully modified by a lone average programmer, by its astronomical maintenance cost that is hard to pay for volunteers, and by aggressive update culture.

boat

Boat Dock

WELCOME :) You find yourself on a strange island.



What is this? Boat is a [LRS](#) spinoff of [Tour Bus](#), a famous wiki [webring](#) -- see <http://meatballwiki.org/wiki/TourBus>. Why not join Tour Bus? Because we are antisocial and don't wanna talk to anyone, so we just start our new thing (also they would prolly [censor](#) us). Also our island is isolated from the [normieland](#) and no buses go here :)

On To The Island

You get greeted by a friendly [dog](#) -- *WOOF* --playfully wagging his tail he leads you around, along the beach. The [island](#) seems a bit empty but a few people can be seen here and there [loosely associating](#), looking very passionate about creating various things, some are writing, some constructing [weird machines](#), some copulating. Everyone is naked -- "clothes are [bloat](#)" says a [weirdo of caveman appearance](#) sitting in front of what appears to be his hut. "We are trying to create stuff, mostly with [computers](#)", he says, "also hiding here from the [hell](#) out there, trying to live a [better life](#)". He scratches his butt and adds: "Seeing you are a living being like myself -- that means you are welcome, come join us if you want."

sightseeing:

- [less retarded software](#): what we create
- [less retarded society](#): what we strive for
- [capitalism](#): what we oppose
- [jokes](#): we also try to have some fun [fun](#)

Continue Elsewhere

- **boat #1:** [Tour Bus Stop: meatballwiki](#), normieland (the main hub of Tour Bus)
- **boat #2:** [YOUR LINK HERE] :-) TODO, here will be some kinda site related to LRS

How To Join Our Boat Tour

Just link to this site from your site. If you want your site added here as a new departure boat, send [me](#) an email -- it's ideal if it's a wiki or something that has something to do with [LRS](#) (even remotely, no need to mention LRS, can be just software minimalism or whatever, ...). **I don't promise to add everything**, but it's pretty likely I'll add you if it's not a complete [shit](#) :D When (more like if) a few boats are here, other ones should be added further on to the chain, not here. Remember this isn't supposed to be a link dump but a selection of some kinda thematic quality links that form some nice webring.

body_shaming

Body Shaming

Your body sucks.

books

Books

Here there will be a constantly WIP list of [books](#) that might be of interest to supporters of [LRS](#):

{ Let's aim for quality rather than quantity here, don't put any book that has some connection to our cause here, but rather the ones you've read and which you judge as a quality book that enriched you in some way.

- **Blackout** (2017, Elsberg): Fiction, telling a story of a large blackout in Europe that shows to really be caused by bloated tech. For collapse enjoyers this is an interesting read if only for the detailed description of the consequences a sudden loss of electric power.
- **Computer Science: An Overview** (J. Glenn Brookshear): Cool bachelor level overview of whole computer science, including things like history of computers, their architecture, computer graphics, compression, encryption, AI, operating systems, complexity etc., with explanations that are neither too simplified nor too long and overcomplicated with equations, i.e. there is a nice balance, probably most useful to fresh university computer science students.
- **Day of the Triffids** (1951, Wyndham): Excellent sci-fi in which civilization comes to an end due to a disaster (won't spoil), very nice for collapse preps or just people enjoying a great story narrated in captivating way :-). The movie is a joke, don't even search for it. Also other books by Wyndham are awesome.
- **Einstein: His Life and Universe** (Isaacson, 2008): Einstein's biography, quite a nice read about a pretty awesome man who's image has been so distorted by the mainstream shit.
- **Encyclopedia Britannica 11th edition** (1911): Extremely large, old, uncensored encyclopedia, mostly digitized and fulltext searchable, also completely public domain, with very long articles on all topics up to the date of its publication. Great source of lesser known information and an alternative to modern censored sources. Also check out other similar encyclopedias.
- **Flatland** (Abbott, 1884): Absolutely amazing fantasy story set in two dimensional land with characters being geometric shapes, while being a critic of society to a big degree, it discusses practical and mathematical aspects of actually living in two dimensions, how the characters see, how they build their houses etc. It is now absolutely public domain!
- **Free as in Freedom** (Sam Williams, 2002): Free-licensed official biography of Richard Stallman, contains many historical details about how free software came to be, how open source spoiled it etc.
- **Free Culture** (Lessig, 2004): Creative-commons licensed (non-free but gratis) book by the founder of Creative Commons and free culture, goes into details on how copyright became abused by capitalism, why public domain is being smothered and why we must support free culture.
- **Game Engine Black Book: Doom** (Sanglard, 2019): Gratis, very nice book dissecting all the details about the legendary Doom engine and its internals -- how it worked, why was it so fast, what hacks went into it, written so that a reader of any programming skill (even none) will find something interesting. A must read for fans of oldschool game programming.
- **Game Engine Black Book: Wolfenstein 3D** (Sanglard, 2019): Same as the Doom engine book from the same author, just about the older game Wolfenstein 3D, also amazing.
- **Industrial Society and Its Future** (Kaczynski, 1995): A bit boring read by the famous Unabomber, criticizing rapid technology advancement, but an important read for those who are more into politics, if only for the memes :)
- **ISO/IEC 9899:1999** (1999): Specification of the version of C programming language that suckless/LRS very often uses. It's nice to skim over it to get an idea how a language is actually specified. You'll also probably learn something new about C in the process.
- **Just for Fun** (2001): Official biography of Linux Torvalds, the original creator of Linux. It recounts valuable historical moments with comments by Linus himself, revealing many interesting details and also a bit of Torvalds' personality (shows some of his evil side).
- **Larousse Desk Reference Encyclopedia** (1995): Very nice single-volume encyclopedia that's sorted by topic, with many nice illustrations, published back then when censorship wasn't so extreme, provides overview of all topics of human knowledge.
- **Masters of Doom** (Kushner, 2003): Another nice book for Doom fans, this time not really technical but rather just retelling the story of the game's development -- quite comfy, a lot of interesting trivia.
- **The Jargon File** (1975...): Hacker culture dictionary, a lot of wisdom, inside jokes, and things related to oldschool hacking.
- **Rebel Code** (Moody, 2001): A bit of a mainstream view at the whole "open source" history -- though it's a small brain business view which we have to keep in mind at all times, it's a nice introduction to the whole FOSS world for the newcomers, as the book covers most of the relevant projects and people.
- **The Country of the Blind** by H. G. Wells (1911): Very nice story, also in the public domain and digitized online, easily accessible. Though not related to technology, it's a great food for thought as it entertains an idea of a population of people who are completely blind which has interesting implications for their lives, and furthermore it shows that if you place someone too competent in a group of retards, they won't recognize his competence, in fact they'll see him as someone yet more

retarded than they are themselves.

- **The Nostalgia Nerd's Retro Tech**: Nice small database of all the old consoles/computers (SNES, Amiga, C64, ...), each one with high quality photos, short summary, specs and notable games. There is not much text, it's more like tl;drs of the most important stuff, it's an ideal overview of the old computers for newcomers but can also serve as a quick reference to anyone.
- older books by **Andreas Eschbach** { The new ones seemed to have some Feminist shit etc., had to stop reading it :D ~drummyfish }, mainly **Carpet Makers** and **Jesus Video**: This is not directly related to LRS but it feels right to mention one of the most underrated sci-fi authors here -- many LRS followers will probably appreciate high quality sci-fi dealing with super interesting topics that are at least loosely related to LRS. Really Eschbach is so superior to just 99% of all sci-fi you'll encounter, his books are extremely readable, believable and greatly interesting in choosing topics, he makes you think about society, religion etcetc. Spoilers probably won't help, just go check out the books.
- **The Pig and the Box** (MCM, 2009): A short story for kids showing the dangers of DRM, released under CC0!
- **The Tao of Programming** (James, 1987): Famous piece of hacker culture literature, wisdom of programming written in taoist style.
- **Tricks of the Game Programming Gurus** (1994): Very nice, readable book, that implements a whole 90s shooter game in C, without drowning the reader in tons of equations and smartass talk. It's written with the 90s mindset and in common language, contains many practical tricks for optimizing the code etc.
- ...

{ TODO (have to read first): Lisp From Nothing (implementing minimal self-hosted Lisp, CC0 code!).
~drummyfish }

boot

Boot

See bootstrapping.

bootstrap

Bootstrap/Boot

In general bootstrapping (from the idiom "pull yourself up by your bootstraps"), sometimes shortened to just *booting*, refers to a clever process of self-establishing some relatively complex system starting from something very small, without much external help. As an example imagine something like a "civilization bootstrapping kit" that contains only few primitive tools along with instructions on how to use those tools to mine ore, turn it into metal out of which one makes more tools which will be used to obtain more material and so on up until having basically all modern technology and factories set up in relatively short time (civboot is a project like this). The term *bootstrapping* is however especially relevant in relation to computer technology -- here it has two main meanings:

- The process by which a computer starts and sets up the operating system after power on, which often involves several stages of loading various modules, running several bootloaders etc. This is traditionally called **booting** (*rebooting* means restarting the computer).
- Utilizing the principle of bootstrapping for making greatly independent software, i.e. software that doesn't depend on other software as it can set itself up. This is usually what **bootstrapping** (the longer term) means. This is also greatly related to self hosting, another principle whose idea is to "implement technology using itself".

Bootstrapping: Making Dependency-Free Software

Bootstrapping as a general principle can aid us in creation of extremely free technology by greatly minimizing all its dependencies, we are able to create a small amount of code that will self-establish our whole computing environment with only very small effort during the process. The topic mostly revolves around designing programming language compilers, but in general we may be talking about bootstrapping

whole computing environments, operating systems etc.

Why be concerned with bootstrapping when we already have our systems set up? There are many reasons, one of the notable ones is that we may lose our current technology due to societal collapse, which is not improbable, it keeps happening throughout history over and over, so many people fear (rightfully so) that if by whatever disaster we lose our current computers, Internet etc., we will also lose with it all modern art, data, software we so painfully developed, digitized books, inventions and so on; not talking about the horrors that will follow if we're unable to quickly reestablish our computer networks we are so dependent on. Setting up what we currently have completely from scratch would be extremely difficult, a task for centuries -- just take a while to consider all the activity and knowledge that's required around the globe to create a single computer with all its billions of lines of code worth of software that makes it work. Knowledge of old technology gets lost -- to make modern computers we first needed older, primitive computers, but now that we only have modern computers no one remembers anymore how to make the older computers -- if we lose the current ones, we won't be able to make them, we will lack the tools. Another reason for bootstrapping is independence of technology which brings e.g. freedom (your operating system being able to be set up anywhere without some corporation's proprietary driver or hardware unit is pursued by many), robustness, simplicity, ability to bring existing software to new platforms and so on, i.e. things that are practical even in current world.

Forth has traditionally been used for making bootstrapping environments; Dusk OS is an example of such project. Similarly simple language such as Lisp and comun will probably work too.

How to do this then? To make a computing environment that can bootstrap itself you can do it like this:

1. **Make a simple programming language L.** You can choose e.g. the mentioned Forth but you can even make your own, just remember to keep it extremely simple -- simplicity of the base language is the key feature here. The language will serve as tool for writing software for your platform, i.e. it will provide some comfort in programming (so that you don't have to write in assembly) but mainly it will be an abstraction layer for the programs, it will allow them to run on any hardware/platform. The language therefore has to be portable; it should probably abstracts things like endianness, native integer size, control structures etc., so as to work nicely on all CPUs, but it also mustn't have too much abstraction (such as OOP) otherwise it will quickly get complicated. The language can compile e.g. to some kind of very simple bytecode that will be easy to translate to any assembly. At first you'll have to temporarily implement L in some already existing language, e.g. C. NOTE: in theory you could just make bytecode, without making L, and just write your software in that bytecode, but the bytecode has to focus on being simple to translate, i.e. it will e.g. likely have few opcodes, which will be in conflict with making it at least somewhat comfortable to program on your platform. However one can try to make some compromise and it will save the complexity of translating language to bytecode, so it can be considered (uxn seems to be doing this).
2. **Write L in itself, i.e. self host it.** This means you'll use L to write a compiler of L that outputs L's bytecode. Once you do this, you have a completely independent language and can throw away the original compiler of L written in another language. Now compile L with itself -- you'll get the bytecode of L compiler. At this point you can bootstrap L on any platform as long as you can execute the L bytecode on it -- this is why it was crucial to make L and its bytecode very simple. In theory it's enough to just interpret the bytecode but it's better to translate it to the platform's native machine code so that you get maximum efficiency (the nature of bytecode should make it so that it isn't really more difficult to translate it than to interpret it). If for example you want to bootstrap on an x86 CPU, you'll have to write a program that translates the bytecode to x86 assembly; if we suppose that at the time of bootstrapping you will only have this x86 computer, you will have to write the translator in x86 assembly manually. If your bytecode really is simple and well made, it shouldn't be hard though (you will mostly be replacing your bytecode opcodes with given platform's machine code opcodes).
3. **Further help make L bootstrappable.** This means making it even easier to execute the L bytecode on any given platform -- you may for example write the bytecode translators for common platforms like x86, ARM, RISC-V and so on. At this point you have L bootstrappable without any work on the platform you have translators for and on others it will just take a tiny bit of work to write its own translator.
4. **Write everything else in L.** This means writing the platform itself and software such as various tools and libraries. You can potentially even use L to write a higher level language for yet more comfort in programming. Since everything here is written in L and L can be bootstrapped, everything here can be bootstrapped as well.

Booting: Computer Starting Up

Booting as in "starting computer up" is also a kind of setting up a system from the ground up -- we take it from granted but remember it takes some work to get a computer from being powered off and having all RAM empty to having an operating system loaded, hardware checked and initialized, devices mounted etc.

Starting up a **simple computer** -- such as some MCU-based embedded open console that runs bare metal programs -- isn't as complicated as booting up a mainstream PC with an operating system.

First let's take a look at the simple computer. It may work e.g. like this: upon start the CPU initializes its registers and simply starts executing instructions from some given memory address, let's suppose 0 (you will find this in your CPU's data sheet). Here the memory is often e.g. flash ROM to which we can externally upload a program from another computer before we turn the CPU on -- in game consoles this can often be done through USB. So we basically upload the program (e.g. a game) we want to run, turn the console on and it starts running it. However further steps are often added, for example there may really be some small, permanently flashed initial boot program at the initial execution address that will handle some things like initializing hardware (screen, speaker, ...), setting up interrupts and so on (which otherwise would have to always be done by the main program itself) and it can also offer some functionality, for example a simple menu through which the user can select to actually load a program from SD card to flash memory (thanks to which we won't need external computer to reload programs). In this case we won't be uploading our main program to the initial execution address but rather somewhere else -- the initial bootloader will jump to this address once it's done its work.

Now for the PC (the "IBM compatibles"): here things are more complicated due to the complexity of the whole platform, i.e. because we have to load an operating system first, of which there can be several, each of which may be loadable from different storages (harddisk, USB stick, network, ...), also we have more complex CPU that has to be set in certain operation mode, we have complex peripherals that need complex initializations etcetc. Generally there's a huge bloated boot sequence and PCs infamously take longer and longer to start up despite skyrocketing hardware improvements -- that says something about state of technology. Anyway, it usually works like this:

{ I'm not terribly experienced with this, verify everything. ~drummyfish }

1. Computer is turned on, the CPU starts executing at some initial address (same as with the simple computer).
2. From here CPU jumps to an address at which **stage one bootloader** is located (bootloader is just a program that does the booting and as this is the first one in a line of potentially multiple bootloaders, it's called *stage one*). This address is in the motherboard ROM and in there typically **BIOS** (or something similar that may be called e.g. UEFI, depending on what standard it adheres to) is uploaded, i.e. BIOS is stage one bootloader. BIOS is the first software (we may also call it firmware) that gets run, it's uploaded on the motherboard by the manufacturer and isn't supposed to be rewritten by the user, though some based people still rewrite it (ignoring the "read only" label :D), often to replace it with something more free (e.g. libreboot). BIOS is the most basic software that serves to make us able to use the computer at the most basic level without having to flash programs externally, i.e. to let us use keyboard and monitor, let us install an operating system from a CD drive etc. (It also offers a basic environment for programs that want to run before the operating system, but that's not important now.) BIOS is generally different on each computer model, it normally allows us to set up what (which device) the computer will try to load next -- for example we may choose to boot from harddisk or USB flash drive or from a CD. There is often some countdown during which if we don't intervene, the BIOS automatically tries to load what's in its current settings. Let's suppose it is set to boot from harddisk.
3. BIOS performs the *power on self test* (POST) -- basically it makes sure everything is OK, that hardware works etc. If it's so, it continues on (otherwise halts).
4. BIOS loads the **master boot record** (MBR, the first sector of the device) from harddisk (or from another mass storage device, depending on its settings) into RAM and executes it, i.e. it passes control to it. This will typically lead to loading the **second stage bootloader**.
5. The code loaded from MBR is limited by size as it has to fit in one HDD sector (which used to be only 512 bytes for a long time), so this code is here usually just to load the bigger code of the second stage bootloader from somewhere else and then again pass control to it.

6. Now the second stage bootloader starts -- this is a bootloader whose job it is normally to finally load the actual operating system. Unlike BIOS this bootloader may quite easily be reinstalled by the user -- oftentime installing an operating system will also cause installing some kind of second stage bootloader -- example may be **GRUB** which is typically installed with GNU/Linux systems. This kind of bootloader may offer the user a choice of multiple operating systems, and possibly have other settings. In any case here the OS kernel code is loaded and run.
 7. Voila, the kernel now starts running and here it's free to do its own initializations and manage everything, i.e. Linux will start the PID 1 process, it will mount filesystems, run initial scripts etcetc.
-

brainfuck

Brainfuck

Brainfuck is an extremely simple, minimalist untyped esoteric programming language; simple by its specification (consisting only of 8 commands) but very hard to program in (it is so called Turing tarpit). It works similarly to a pure Turing machine. In a way it is kind of beautiful by its simplicity, it is very easy to write your own brainfuck interpreter (or compiler) -- in fact the Brainfuck author's goal was to make a language for which the smallest compiler could be made.

There exist **self-hosted Brainfuck interpreters and compilers** (i.e. themselves written in Brainfuck) which is pretty fucked up. The smallest one is probably the one called dbfi which has only slightly above 400 characters, that's incredible!!! (Esolang wiki states that it's one of the smallest self interpreters among imperative languages). Of course, **Brainfuck quines** (programs printing their own source code) also exist, but it's not easy to make them -- one example found on the web was a little over 2100 characters long.

The language is based on a 1964 language PÂ'Â' which was published in a mathematical paper; it is very similar to Brainfuck except for having no I/O. Brainfuck itself was made in 1993 by Urban Muller, he wrote a compiler for it for Amiga, which he eventually managed to get under 200 bytes.

Since then Brainfuck has seen tremendous success in the esolang community as the **lowest common denominator language**: just as mathematicians use Turing machines in proofs, esolang programmers use brainfuck in similar ways -- many esolangs just compile to brainfuck or use brainfuck in proofs of Turing completeness etc. This is thanks to Brainfuck being an actual, implemented and working language with I/O and working on real computers, not just some abstract mathematical model. For example if one wants to encode a program as an integer number, we can simply take the binary representation of the program's Brainfuck implementation. Brainfuck also has many derivatives and modifications (esolang wiki currently lists over 600 such languages), e.g. Brainfork (Brainfuck with multithreading), Boolfuck (has only binary cells), Brainfuck++ (adds more features like networking), Pi (encodes Brainfuck program in error against pi digits), Unary (encodes Brainfuck with a single symbol) etcetc.

In LRS programs brainfuck may be seriously used as a super simple scripting language.

Brainfuck can be trivially translated to comun like this: remove all comments from brainfuck program, then replace +, -, >, <, ., ,, [and] with ++, --, \$>0, \$<0, ->', \$<0 <- , @' and . , respectively, and prepend \$>0 .

Specification

The "vanilla" brainfuck operates as follows:

We have a linear memory of **cells** and a **data pointer** which initially points to the 0th cell. The size and count of the cells is implementation-defined, but usually a cell is 8 bits wide and there is at least 30000 cells.

A program consists of these possible commands:

- +: increment the data cell under data pointer
- -: decrement the data cell under data pointer
- >: move the data pointer to the right
- <: move the data pointer to the left

- [: jump after corresponding] if value under data pointer is zero
-]: jump after corresponding [if value under data pointer is not zero
- .: output value under data pointer as an ASCII character
- ,: read value and store it to the cell under data pointer

Characters in the source code that don't correspond to any command are normally ignored, so they can conveniently be used for comments.

Brainfuck source code files usually have *.bf* or *.b* extension.

Implementation

This is a very simple C implementation of brainfuck:

```
#include <stdio.h>

#define CELLS 30000

const char program[] = "[.-]"; // your program here

int main(void)
{
    char tape[CELLS];
    unsigned int cell = 0;
    const char *i = program;
    int bDir, bCount;

    while (*i != 0)
    {
        switch (*i)
        {
            case '>': cell++; break;
            case '<': cell--; break;
            case '+': tape[cell]++; break;
            case '-': tape[cell]--; break;
            case '.': putchar(tape[cell]); fflush(stdout); break;
            case ',': scanf("%c", tape + cell); break;
            case '[':
            case ']':
                if ((tape[cell] == 0) == (*i == ']'))
                    break;

                bDir = (*i == '[') ? 1 : -1;
                bCount = 0;

                while (1)
                {
                    if (*i == '[')
                        bCount += bDir;
                    else if (*i == ']')
                        bCount -= bDir;

                    if (bCount == 0)
                        break;

                    i += bDir;
                }

                break;

            default: break;
        }

        i++;
    }
}
```

TODO: comun implementation

Programs

Here are some simple programs in brainfuck.

Print HI:

```
+++++ . + .
```

Read two 0-9 numbers (as ASCII digits) and add them:

```
,>,[<+>-]<-----.
```

TODO: more

Variants

TODO

See Also

- [False](#) (a very similar esolang)
- [comun](#)

brain_software

Brain Software

Brain software, also brainware, is kind of a fun idea of software that runs on the human brain as opposed to a typical electronic computer. This removes the dependency on computers and highly increases freedom. Of course, this also comes with a huge drop of computational power :) However, aside from being a fun idea to explore, this kind of software and "architectures" may become interesting from the perspective of freedom and primitivism (especially when the technological collapse seems like a real danger).

Primitive tools helping the brain compute, such as pen and paper or printed out mathematical tables, may be allowed.

Example of brain software can be the game of chess. Chess masters can easily play the game without a physical chess board, only in their head, and they can play games with each other by just saying the moves out loud. They may even just play games with themselves, which makes chess a deep, entertaining game that can be 100% contained in one's brain. Such game can never be taken away from the man, it can't be altered by corporations, it can't become unplayable on new hardware etc., making it free to the greatest extent. Many other board games and pen and pencil games, such as racetrack (pen and pencil racing game suitable for one or many players).

One may think of a pen and paper computer with its own simple instruction set that allows general purpose programming. This instruction set may be designed to be well interpretable by human and it may be accompanied by tables printed out on paper for quick lookup of operation results -- e.g. a 4 bit computer might provide a 16x16 table with precomputed multiplication results which would help the individual execute the multiplication instruction within mere seconds.

Yet another idea is to make a computer with architecture similar to the typical electronic computers but powered by human brains -- let's call this a human computer (not to be confused with people whose job was to perform computations!). Imagine that after a societal collapse we lose our computer technology (i.e. the ability to manufacture transistors and similar key components), but we retain our knowledge of computer architecture, algorithms and the usefulness of computers. As a temporary solution for performing computations we may create a "computer made of humans", a room with several men, each one performing

a role of some computer component, for example an ALU, cache and memory controller. Again, a special instruction set and a set of tools (such as physical lookup tables for results of instructions) could be made to make such a human computer relatively fast. It might not run Doom, but it could possibly e.g. compute some mathematical constants to a high precision or perhaps help find optimal structure of cities, compute stresses in big building etc. In such conditions even a slow calculator could be immensely useful.

bs

BS

Bullshit.

build_engine

Build Engine

For now see Duke Nukem.

bullshit

Bullshit

Bullshit (BS) is nonsense, arbitrary unnecessary shit and/or something made up. Typical example are e.g. so called *bullshit jobs* -- jobs that are arbitrarily created just to keep people employed and don't actually serve anything.

Simplified example of capitalist bullshit: under capitalism basically the whole society is based on bullshit of many kinds, small and big, creating many bullshit clusters that are all intertwined and interact in complex ways, creating one huge bullshit. For simplicity let's consider an educational isolated bullshit cluster (that we won't see that often in reality), a hypothetical car factory. No matter how the factory came to be, it now ended up making cars for people so that these people can drive to work at the car factory to make more cars for other people who will work to the car factory to make cars etc. -- a bullshit cycle that exists just for its own sake, just wastes natural resources and lives of people. Of course at one point all the factory employees will own a car and the factory will have no more demand for the cars, which threatens its existence. Here capitalism employs adding more bullshit: let's say they create new bullshit jobs they call something like "smart car research center" -- this will create new work position, i.e. more people who will need cars to drive to work, but MAINLY the job of these people will be adding artificial obsolescence to the cars, which will make them last much shorter time and regularly break so that they will need repairs using parts manufactured at the factory, creating more work that will need to be done and more bullshit jobs in the car repair department. Furthermore the team will make the cars completely dependent on subscription software, employing consumerism, i.e. the car will no longer be a "buy once" thing but rather something one has to keep feeding constantly (fuel, software subscription, insurance, repairs, cleaning, tire changes, and of course once in a few years just buying a new non-obsolete model), so that workers will still need to drive to work every day, perpetuating their need for being preoccupied with owning and maintaining a car. This is a bullshit cluster society could just get rid of without any cost, on the contrary it would gain many free people who could do actually useful things like curing diseases, eliminating world hunger, creating art for others to enjoy. However if you tell a capitalist any part of this system is bullshit, he will defend it by its necessity in the system as a whole ("How will people get to work without cars?!", "Factories are needed for the economy!", "Economy is needed to drive manufacturing of cars!") -- in reality the bullshit clusterfuck spans the whole world to incredibly deep levels so you just can't make many people see it, especially when they're preoccupied with maintaining their own existence and just get by within it.

Some things that are bullshit include:

- antiviruses
- army
- bureaucracy

- capitalism
- censorship
- clothes
- consumerism
- countries
- crypto
- disclaimers
- DRM
- economy
- management
- fashion
- "game design" (it's just part of programming games)
- gender studies
- guns
- insurance
- jobs
- law
- licenses
- "life coaching" Imao
- marketing, ads
- money
- police
- political correctness
- prisons
- privacy
- productivity
- property (including copyright etc.)
- security
- states
- UML
- ...

OK then, what's not bullshit? Well, things that matter, for example food, health, education, love, fun, art, technology, knowledge about the world, science, morality, exploration, ...

bytebeat

Bytebeat

Bytebeat is a procedural chiptune/8bit style music generated by a short expression in a programming language; it was discovered/highlighted in 2011 by Viznut (author of countercomplex blog) and others, and the technique capable of producing quite impressive music by single-line code has since caught the attention of many programmers, especially in demoscene. There has even been a paper written about bytebeat. Bytebeat can produce music similar (though a lot simpler) to that created e.g. with music trackers but with a lot less complexity and effort.

This is a beautiful hack for LRS/suckless programmers because it takes quite a tiny amount of code, space and effort to produce nice music, e.g. for games (done e.g. by Anarch).

8bit samples corresponding to unsigned char are typically used with bytebeat. The formulas take advantage of overflows that create rhythmical patterns with potential other operations such as multiplication, division, addition, squaring, bitwise/logical operators and conditions adding more interesting effects.

Bytebeat also looks kind of cool when rendered as an image (outputting pixels instead of audio samples).

How To

Quick experiments with bytebeat can be performed with online tools that are easy to find on the [web](#), these usually use [JavaScript](#).

Nevertheless, traditionally we use `C` for bytebeat. We simply create a loop with a *time* variable (`i`) and inside the loop body we create our bytebeat expression with the variable to compute a char that we output.

A simple "workflow" for bytebeat "development" can be set up as follows. Firstly write a C program:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10000; ++i)
        putchar(
            i / 3 // < bytebeat formula here
        );

    return 0;
}
```

Now compile the program and play its output e.g. like this:

```
gcc program.c && ./a.out | aplay
```

Now we can just start experimenting and invent new music by fiddling with the formula indicated by the comment.

General tips/tricks and observations are these:

- Outputting the variable `i` creates a periodical saw-shaped beat, **multiplication/division decreases/increases the speed, addition/subtraction shifts the phase backward/forward**.
- Squaring (and other powers) create a **wah-wah effect**.
- Crazier patterns can be achieved by **using the variable in places of numerical constants**, e.g. `i << ((i / 512) % 8)` (shifting by a value that depends on the variable).
- Modulo (%) increases the frequency and **decreases volume** (limits the wave peak).
- So called **Sierpinski harmonies** are often used melodic expressions of the form `i*N & i >> M`.
- Bitwise and (&) can add distortion (create steps in the wave).
- A **macro structure** of the song (silent/louds parts, verse/chorus, ...) can be achieved by combining multiple patterns with some low-frequency pattern, e.g. this alternates a slower and faster beat: `int cond = (i & 0x8000) == 0; cond * (i / 16) + !cond * (i / 32)`
- **Extra variables** can add more complexity (e.g. precompute some variable `a` which will subsequently be used multiple times in the final formula).

Copyright

It is not exactly clear whether, how and to what extent [copyright](#) can apply to bytebeat: on one hand we have a short formula that's uncopyrightable (just like mathematical formulas), on the other hand we have music, an artistic expression. Many authors of bytebeat "release" their creations under [free licenses](#) such as [CC-BY-SA](#), but such licenses are of course not applicable if copyright can't even arise.

We believe copyright doesn't and **SHOULDN'T** apply to bytebeat. To ensure this, it is good to stick [CC0](#) to any released bytebeat just in case.

Examples

A super-simple example can be just a simple:

- `i / 16`

The following more complex examples come from the LRS game Anarch (these are legally safe even in case copyright can apply to bytebeat as Anarch is released under CC0):

- distortion guitar rhythmical beat: `~((((i >> ((i >> 2) % 32)) | (i >> ((i >> 5) % 32))) & 0x12) << 1) | (i >> 11)`
- electronic/techno: `((0x47 >> ((i >> 9) % 32)) & (i >> (i % 32))) | (0x57 >> ((i >> 7) % 32)) | (0x06 >> ((i >> (((i * 11) >> 14) & 0x0e) % 32)) % 32))`
- main theme, uses an extra variable: `((i & 65536) ? (a & (((i * 2) >> 16) & 0x09)) : ~a),` where `uint32_t a = ((i >> 7) | (i >> 9) | (~i << 1) | i)`

See Also

- [music tracker](#)
- [MIDI](#)

bytecode

Bytecode

Bytecode (BC, also *P-code*, "portable code") is a type of binary format for executable programs usually intended to be interpreted or to serve as an intermediate representation in compilers (i.e. meant to be translated to some other language); it is quite similar to machine code, however machine code is meant to be directly run by some physical hardware while bytecode is more of a virtual, machine independent code preferring things like portability, speed of interpretation, retaining meta information or being easy to translate.

TODO: moar

Example

Let's consider a simple algorithm that tests the Collatz conjecture (which says that applying a simple operation from any starting number over and over will always lead to number 1). The program reads a number (one digit for simplicity) and then prints the sequence until reaching the final number 1. The algorithm in C would look as follows:

```
// Collatz conjecture
#include <stdio.h>

int next(int n)
{
    return n % 2 ? // is odd?
        3 * n + 1 :
        n / 2;
}

int main(void)
{
    int n = getchar() - '0'; // read input ASCII digit

    while (1)
    {
        printf("%d\n", n);

        if (n == 1)
            break;

        n = next(n);
    }

    return 0;
}
```

C will be normally compiled to machine code, however we can take a look at some immediate representation bytecode that compilers internally use to generate the machine code. The following is LLVM, a widely used bytecode that can be produced from the above C code with clang compiler (e.g. as `clang -cc1 tmp.c -S -emit-llvm -o -`):

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone
define i32 @next(i32 %n) #0 {
entry:
    %n.addr = alloca i32, align 4
    store i32 %n, i32* %n.addr, align 4
    %0 = load i32, i32* %n.addr, align 4
    %rem = srem i32 %0, 2
    %tobool = icmp ne i32 %rem, 0
    br i1 %tobool, label %cond.true, label %cond.false

cond.true:                                     ; preds = %entry
    %1 = load i32, i32* %n.addr, align 4
    %mul = mul nsw i32 3, %1
    %add = add nsw i32 %mul, 1
    br label %cond.end

cond.false:                                    ; preds = %entry
    %2 = load i32, i32* %n.addr, align 4
    %div = sdiv i32 %2, 2
    br label %cond.end

cond.end:                                      ; preds = %cond.false, %cond.true
    %cond = phi i32 [ %add, %cond.true ], [ %div, %cond.false ]
    ret i32 %cond
}

; Function Attrs: noinline nounwind optnone
define i32 @main() #0 {
entry:
    %retval = alloca i32, align 4
    %n = alloca i32, align 4
    store i32 0, i32* %retval, align 4
    %call = call i32 (...) @getchar()
    %sub = sub nsw i32 %call, 48
    store i32 %sub, i32* %n, align 4
    br label %while.body

while.body:                                    ; preds = %entry, %if.end
    %0 = load i32, i32* %n, align 4
    %call1 = call i32 (i8*, ...) @printf(i8* ... )
    %1 = load i32, i32* %n, align 4
    %cmp = icmp eq i32 %1, 1
    br i1 %cmp, label %if.then, label %if.end

if.then:                                       ; preds = %while.body
    br label %while.end

if.end:                                        ; preds = %while.body
    %2 = load i32, i32* %n, align 4
    %call2 = call i32 @next(i32 %2)
    store i32 %call2, i32* %n, align 4
    br label %while.body

while.end:                                    ; preds = %if.then
    ret i32 0
}

declare i32 @getchar(...) #1

declare i32 @printf(i8*, ...) #1
```

```

attributes #0 = { ... }
attributes #1 = { ... }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{"clang version 7.0.1-8+deb10u2 (tags/RELEASE_701/final)"}

```

TODO: analyze the above

Now let's rewrite the same algorithm in comun, a different language which will allow us to produce another kind of bytecode (obtained with comun -T program.cmn):

```

# Collatz conjecture

next:
  $0 2 % ? # is odd?
  3 * 1 +
  ;
  2 /
  .
.

<-      # read input ASCII digit
"0" -   # convert it to number

@@
# print:
$0 10 / "0" + ->
$0 10 % "0" + ->
10 ->

$0 1 = ?
!@
.

next
.

```

Here is annotated comun bytecode this compiles to:

000000:	DES	00 0111	# func	\ next:
000001:	JMA	00 0100...	# 20 (#14)	
000002:	COC	00 0001		
000003:	MGE	00 0000		\$0
000004:	CON'	00 0010	# 2 (#2)	2
000005:	MOX	00 0000		%
000006:	DES	00 0001	# if	\ ?
000007:	JNA	00 0000...	# 16 (#10)	
000008:	COC	00 0001		
000009:	CON'	00 0011	# 3 (#3)	3
00000a:	MUX	00 0000		*
00000b:	CON'	00 0001	# 1 (#1)	1
00000c:	ADX	00 0000		+
00000d:	DES	00 0010	# else	< ;
00000e:	JMA	00 0011...	# 19 (#13)	
00000f:	COC	00 0001		
000010:	CON'	00 0010	# 2 (#2)	2
000011:	DIX	00 0000		/
000012:	DES	00 0011	# end if	/ .
000013:	RET	00 0000		/ .
000014:	INI	00 0000		
000015:	INP	00 0000		<-
000016:	CON'	00 0000...	# 48 (#30)	"0"
000017:	COC	00 0011		
000018:	SUX	00 0000		-
000019:	DES	00 0100	# loop	\ @@
00001a:	MGE	00 0000		\$0
00001b:	CON'	00 1010	# 10 (#a)	10

```

00001c: DIX 00 0000 | /
00001d: CON' 00 0000... # 48 (#30) | "0"
00001e: COC 00 0011 |
00001f: ADX 00 0000 | +
000020: OUT 00 0000 | ->
000021: MGE 00 0000 | $0
000022: CON' 00 1010 # 10 (#a) | 10
000023: MOX 00 0000 | %
000024: CON' 00 0000... # 48 (#30) | "0"
000025: COC 00 0011 |
000026: ADX 00 0000 | +
000027: OUT 00 0000 | ->
000028: CON' 00 1010 # 10 (#a) | 10
000029: OUT 00 0000 | ->
00002a: MGE 00 0000 | $0
00002b: CON' 00 0001 # 1 (#1) | 1
00002c: EQX 00 0000 | =
00002d: DES 00 0001 # if | \ ?
00002e: JNA 00 0100... # 52 (#34) | |
00002f: COC 00 0011 | |
000030: DES 00 0101 # break | | !@
000031: JMA 00 1000... # 56 (#38) | |
000032: COC 00 0011 | |
000033: DES 00 0011 # end if | / .
000034: CAL 00 0011 # 3 (#3) | next
000035: DES 00 0110 # end loop / .
000036: JMA 00 1010... # 26 (#1a)
000037: COC 00 0001
000038: END 00 0000

```

TODO: analyze the above, show other bytecodes (python, java, ...)

Let's try the same in Python. The code we'll examine will look like this:

```

# Collatz conjecture

def next(n):
    return 3 * n + 1 if n % 2 != 0 else n / 2

n = ord(raw_input()[0]) - ord('0')

while True:
    print(n)

    if n == 1:
        break

    n = next(n)

```

And the bytecode we get (e.g. with `python -m dis program.py`):

```

3      0 LOAD_CONST          0 (<code object next at ...>)
      3 MAKE_FUNCTION          0
      6 STORE_NAME            0 (next)

6      9 LOAD_NAME            1 (ord)
     12 LOAD_NAME            2 (raw_input)
     15 CALL_FUNCTION          0
     18 LOAD_CONST            1 (0)
     21 BINARY_SUBSCR
     22 CALL_FUNCTION          1
     25 LOAD_NAME            1 (ord)
     28 LOAD_CONST            2 ('0')
     31 CALL_FUNCTION          1
     34 BINARY_SUBTRACT
     35 STORE_NAME            3 (n)

8      38 SETUP_LOOP          43 (to 84)
  >> 41 LOAD_NAME            4 (True)
     44 POP_JUMP_IF_FALSE     83

```

9	47 LOAD_NAME	3 (n)
	50 PRINT_ITEM	
	51 PRINT_NEWLINE	
11	52 LOAD_NAME	3 (n)
	55 LOAD_CONST	3 (1)
	58 COMPARE_OP	2 (==)
	61 POP_JUMP_IF_FALSE	68
12	64 BREAK_LOOP	
	65 JUMP_FORWARD	0 (to 68)
14 >>	68 LOAD_NAME	0 (next)
	71 LOAD_NAME	3 (n)
	74 CALL_FUNCTION	1
	77 STORE_NAME	3 (n)
	80 JUMP_ABSOLUTE	41
>>	83 POP_BLOCK	
>>	84 LOAD_CONST	4 (None)
	87 RETURN_VALUE	

TODO: make sense of it and analyze it

TODO: web assembly

byte

Byte

Byte (symbol: B) is a basic unit of information, nowadays already practically always consisting of 8 bits (for which it's also called an **octet**), that allow it to store $2^8 = 256$ distinct values (for example a number in range 0 to 255). It is commonly the smallest unit of computer memory a CPU is able to operate on; memory addresses are assigned by steps of one byte. We use bytes to measure the size of memory and derive higher memory units such as a kilobyte (kB, 1000 bytes), kibibyte (KiB, 1024 bytes), megabyte (MB, 10^6 bytes) and so forth. In conventional programming a one byte variable is seen as very small and used if we are really limited by memory constraints (e.g. embedded) or to mimic older 8bit computers ("retro games" etc.): one byte can be used to store very small numbers (while in mainstream processors numbers nowadays mostly have 4 or 8 bytes), text characters (ASCII, ...), very primitive colors (see RGB332, palettes, ...) etc.

Historically *byte* was used to stand for the basic addressable unit of memory capable of storing one text character or another "basic value" and could therefore have a different size than 8 bits: for example ASCII machines might have had a 7bit byte, 16bit machines a 16bit byte etc.; in C (standard 99) `char` is the "byte" data type, its byte size is always 1 (`sizeof(char) == 1`), though its number of bits (`CHAR_BIT`) can be greater or equal to 8; if you need an exact 8bit byte use types such as `int8_t` and `uint8_t` from the standard `stdint` library. From now on we will implicitly talk about 8bit bytes.

Value of one byte can be written exactly with two hexadecimal digits with each digit always corresponding to higher/lower 4 bits, making mental conversions very easy; this is very convenient compared to decimal representation, so programmers prefer to write byte values in hexadecimal. For example a byte whose binary value is `11010010` is `D2` in hexadecimal (`1101` is always `D` and `0010` is always `2`), while in decimal we get 210.

Byte frequency/probability: it may be interesting and/or useful (e.g. for compression) to know how often different byte values appear in the data we process with computers -- indeed, this always DEPENDS; if we are working with plain ASCII text, we will never encounter values above 127, and on the other hand if we are processing photos from a polar expedition, we will likely mostly encounter byte values of 255 (as snow will cause most pixels to be completely white). In general we may expect values such as 0, 255, 1 and 2 to be most frequent, as many times these are e.g. assigned special meanings in data encodings, they may be cutoff values etc. Here is a table of measured byte frequencies in real data:

{ Measured by me. ~drummyfish }

type of data	least c.	2nd least c.	3rd least c.	3rd most c.	2nd most c.	most c.
GNU/Linux x86 executable	0x9e (0%)	0xb2 (0%)	0x9a (0%)	0x48 (2%)	0xff (3%)	0x00 (32%)
bare metal ARM executable	0xcf (0%)	0xb7 (0%)	0xa7 (0%)	0xff (2%)	0x01 (3%)	0x00 (15%)
UTF8 English txt book	0x00 (0%)	0x01 (0%)	0x02 (0%)	0x74 (t, 6%)	0x65 (e, 8%)	0x20 (, 14%)
C source code	0x00 (0%)	0x01 (0%)	0x02 (0%)	0x31 (1, 6%)	0x20 (, 12%)	0x2c (, , 16%)
raw 24bit RGB photo image	0x07 (0%)	0x09 (0%)	0x08 (0%)	0xdd (0%)	0x00 (1%)	0xff (25%)

cache

Cache

Cache is a very small but fast computer memory that helps make communication between computer components much more efficient (typically by making it much faster or taking less bandwidth) by remembering recent requests and answers so that they don't have to be expensively repeated. The concept of cache memory is extremely important and one of the very basics for designing and optimizing hardware and software (as cache may be implemented both in hardware and software). A cache may also help prevent expensively recomputing results of functions in the same way, by remembering the recent results of the function (we may see this as a more abstract CPU-function communication). Though caches find wide use almost everywhere, without further specifying the context or type of cache the word *cache* most often refers to the CPU cache -- cache memory found in a CPU (nowadays in all PC CPUs, however still NOT in all embedded CPUs), which is typically further subdivided into multiple levels (L1, L2 etc.) -- here we will be using the term cache the same way, but keep in mind the principles apply everywhere and caches really are used in many places. Cache is not to be confused with a buffer (which also helps optimize communication but rather by means of creating bigger chunks to be transferred at once).

Basic principle: cache can be seen as a black box, "man in the middle" component that's placed in the line of communication between a CPU and main memory (RAM). (Physically it is nowadays part of the CPU itself, but we may imagine it as a separate component just sitting "on the wire" between CPU and RAM.) When reading from memory, we have a pretty simple situation -- once CPU requests something from the memory, the request first goes to the cache; if the cache has the result stored, it just quickly returns it -- we call this a **cache hit** (this is good, we saved time!). A **cache miss** happens when the cache doesn't have the result stored -- in such case the cache has to expensively forward the request to the memory and retrieve the data; usually the cache retrieves a whole smaller block of memory because it can be expected the CPU will access something in nearby memory in the near future (see the principle of locality below). When writing data to memory the situation is a bit more complex as the cache may choose different strategies of behavior: for simplicity it may just write the data through every time, but a more efficient (and also more complicated) approach is to just store the data for itself and write it to the main memory only when necessary (e.g. when it needs to load a different block of memory). Here we get into things such as cache coherence etc., which may cause pretty nasty bugs and headaches.

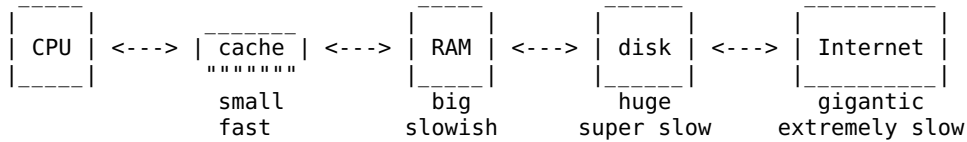
Programmers often try to optimize their programs by making them "cache friendly", i.e. they try to minimize long jumps in memory which causes a lot of cache misses and slows down the program. A typical example is e.g. storing image data in the order by which it will be written to the screen.

A cache is related and/or exploits some observations and concepts related to computers such as:

- **principle of locality:** Computers (/CPUs) tend to more often than not access data that are close to each other in memory, i.e. a CPU doesn't typically make random jumps in memory but rather e.g. reads a sequence of bytes one after another from an array or struct. For this reason when a CPU pulls something out of memory, there is a high probability of accessing an address that is nearby to this memory next time -- a cache helps us get ready for this by prefetching this nearby data and having it ready for very fast access.
- **memory hierarchy:** Mostly because of the principle of locality computer memory is divided into different levels, a chain of memories that get progressively further away from the CPU, increasing their size (decreasing price for capacity) as they get further away but also decreasing their speed. Here a cache can be seen as the closest memory to the CPU (except for the registers), i.e. being the

smallest, most expensive but also fastest memory. By extension we can see that RAM can in many cases be seen as a "cache" for the hard drive, hard drive can be seen as "cache" for the network (after all web browsers ARE caching websites into files on the disk) etc.

- **dynamic programming:** Dynamic programming is a programming technique revolving around remembering already calculated results so that we don't have to compute them again in the future -- this is basically what caches do, they remember results we obtained in relatively expensive way so that next time we can get them cheaper.
- ...



Cache resides very close to the CPU within the memory hierarchy.

TODO: code

cancer

Cancer

Cancer is similar to shit but is even worse because it spreads itself and infects anything else it touches (it is a subset of shit).

Examples of cancer are:

- human civilization
- capitalism and many of its parts like consumerism, corporations, marketing, industrialization etc.
- pseudoleftist movements like LGBT or Feminism
- the mainstream
- fashion
- soynet, soytech
- political correctness
- nationalism, fascism, militarism, ...
- revolutions
- fight culture
- hero culture
- politics
- biological cancer
- ...

See Also

- shit

capitalism

\$\$\$Capitalism\$\$\$

Capitalism is how you enslave a man with his approval.

Capitali\$ism is the worst socioeconomic system we've yet seen in history, ^source based on pure greed, culture of slavery and artificially sustained conflict between everyone in society (so called competition), abandoning all morals and putting money and profit (so called capital) above everything else including preservation of life itself, capitalism fuels the worst in people and forces them to compete and suffer for basic resources, even in a world where abundance of resources is already possible to achieve -- of course,

capitalism is a purely rightist idea. Capitalism goes against progress (see e.g. antivirus paradox), good technology and freedom, it supports immense waste of resources, wars, abuse of people and animals, destruction of environment, decline of morals, deterioration of art, invention of bullshit (bullshit jobs, bullshit laws, ...), utilizing and perfecting methods of torture, brainwashing, ensorship and so on. In a sense capitalism can be seen as **slavery 2.0** or *universal slavery*, a more sophisticated form of slavery, one which denies the label by calling itself the polar opposite ("freedom") and manipulates people into them approving and voluntarily partaking in their own enslavement (capitalist slaves are called wage slaves or *wagies*) -- this new form of slavery which enslaves everyone evolved because the old form with strictly separated classes of slaves and masters was becoming unsustainable, with the enslaved majority revolting, causing civil wars etc. This alone already seems to many like a good reason for suicide, however wage and consumption slavery is still only a small part of capitalist dystopia -- capitalism brings on destruction basically to every part of civilization. It is also often likened to a cancer of society; one that is ever expanding, destroying everything with commercialism, materialism, waste and destruction, growing uncontrollably with the sole goal of just never stop an ever accelerating growth. Nevertheless, it's been truthfully stated that "it is now easier to imagine the end of all life than any substantial change in capitalism." Another famous quote is that "capitalism is the belief that the worst of men driven by the nastiest motives will somehow work for the benefit of everyone", which describes its principle quite well.

{ Some web bashing capitalism I just found: <http://digdeeper.club/articles/capitalismcancer.xhtml>, read only briefly, seems to contain some nice gems capturing the rape of people. ~drummyfish }

Capitalism is fundamentally flawed and CANNOT be fixed -- capitalists build on the idea that competition will drive society, that market will be self sustaining, however capitalism itself works for instating the rule of the winners who eliminate their competition, capitalism is self destabilizing, i.e. the driving force of capitalism is completely unsustainable and leads to catastrophic results as those who get ahead in working competition are also in advantage -- as it's said: money makes money, therefore money flow from the poor to the rich and create a huge imbalance in which competition has to be highly forced, eventually completely arbitrarily and in very harmful ways (invention of bullshit jobs, creating artificial needs and hugely complex state control and laws). It's as if we set up a race in which those who get ahead start to also go faster, and those become the ones who oversee and start to create the rules of the race -- expecting a sustained balance in such a race is just insanity. Society tries to "fight" this emerging imbalance with various laws and rules of market, but this effort is like trying to fight math itself -- the system is mathematically destined to be unstable, pretending we can win over laws of nature themselves is just pure madness.

Capitalism is practically equivalent to the terms free market and free trade -- today's extreme, catastrophic form of capitalism is just sufficiently evolved free market, i.e. it is impossible to support free market without supporting what we see today as long as you believe there will ever be any progress in society; against beliefs of great many unintelligent individuals, it is for example impossible to be a true anarchist as long as you believe in form of free market.

Capitalism produces the worst imaginable technology and rewards people for being cruel to each other. It points the direction of society towards a collapse and may very likely be the great filter of civilizations; in capitalism people de-facto own nothing and become wholly dependent on corporations which exploit this fact to abuse them as much as possible. This is achieved by slowly boiling the frog and leading the pig to the slaughterhouse. Capitalism further achieves enslavement of society while staying accepted by **deflecting responsibility** from the big picture to insignificant details: it says "Look, this politician fucked up your society! This one CEO of this corporation did it! This law here fucked your society! This one immigrant minority is responsible for it! Social media is to blame!" while in fact all of these are just symptoms of the underlying cancer of capitalism; it is relied on an average idiot's inability to see the big picture (society made mostly of idiots is achieved by indoctrination, propaganda and brainwashing that teaches one to only care about the immediate, himself and his daily food; big picture related concepts such as ethics and morality are laughed at). No one owns anything, products become services (your car won't drive without Internet connection and permission from its manufacturer), all independence and decentralization is lost in favor of a highly fragile and interdependent economy and infrastructure of services, each one controlled by the monopoly corporation. Then only a slight break in the chain is enough to bring the whole civilization down in a spectacular domino effect.

The underlying issue of capitalism is competition and conflict -- competition is the root of all evil in any social system, however capitalism is the absolute glorification of competition, amplification of this evil to maximum. It is implemented by setting and supporting a very stupid idea that **everyone's primary and only goal is to be self-benefit**, i.e. maximization of capital. This is combined with the fact that the

environment of free market is an **evolutionary system** which through natural selection extremely effectively and quickly optimizes the organisms (corporations) for achieving this given goal, i.e. generating maximum profit, on the detriment of all other values such as wellbeing of people, sustainability or morality. In other words capitalism has never promised a good society, it literally only states that everyone should try to benefit oneself as much as possible, i.e. defines the fitness function purely as the ability to seize as many resources as possible, and then selects and rewards those who best implement this function, i.e. those we would call sociopaths or "dicks", and to those is given the power in society. Yes, this is how nature works, but it must NOT be how a technologically advanced civilization with unlimited power of destruction should work. In other words we simply get what we set to achieve: find entities that are best at making profit at any cost. The inevitable decline of society can not possibly be prevented by laws, any effort of trying to stop evolution by inventing artificial rules on the go is a battle against nature itself and is extremely naive, the immense power of the evolutionary system that's constantly at work to find ways to bypass or cancel laws in the way of profit and abuse of others will prevail just as life will always find its way to survive and thrive even in the worst conditions on Earth. Trying to stop corporations with laws is like trying to stop a train by throwing sticks in its path. The problem is not that "people are dicks", it is that we choose to put in place a system that rewards the dicks, a system that fuels the worst in people and smothers the best in them.

Even though nowadays quite a lot of time has passed since times of Marx and capitalism has evolved to a stage with countless disastrous issues Marx couldn't even foresee, it is useful to mention one of the basic and earliest issues identified by Marx, which is that economically capitalism is based on **stealing the surplus value**, i.e. abuse of workers and consumers by owners of the means of production (factories, tools, machines etc.) -- a capitalist basically takes money for doing nothing, just for letting workers use tools he proclaims to own (a capitalist will proclaim to "own" land that he never even visited, machines he didn't make as they were developed over centuries, nowadays he even claims to own information and ideas) -- as Kropotkin put it: the working man cannot purchase with his wage the wealth he has produced. This allows a capitalist oppressor to make exponentially more money for nothing and enables existence of monstrously rich and powerful individuals in a world where millions are starving -- consider for example that nowadays there are people who own hundreds of buildings and cars plus a handful of private planes and a few private islands. It is not possible for any single human to work an equivalent of effort that's needed to produce what such an individual owns, even if he worked 24 hours a day for his whole life, he wouldn't get even close to matching the kind of effort that's needed to build the hundreds of buildings he owns -- any such great wealth is always stolen from countless workers whose salary is less than what's adequate for their work and also from consumers who pay more than it really costs to manufacture the goods they buy. Millions of people are giving their money (resources) for free to someone who just proclaims to "own" tools and even natural resources that have been there for billions of years. The difference in wealth and privileges this wealth provides divides society into antagonist classes that are constantly at war -- traditionally these classes are said to be the **bourgeoisie** (business owners, the upper class) and the **proletariat** (workers, the lower class), though under modern capitalism the division of society is not so simple anymore -- there are more classes (for example small businesses work for larger businesses) but they are still all at war.

Nowadays **capitalism is NOT JUST an economic system** anymore. Technically perhaps, however in reality it takes over society to such a degree that it starts to redefine very basic social and moral values to the point of taking the role of a religion, or better said a brainwashing cult in which people are since childhood taught (e.g. by constant daily exposure to private media) to worship economy, brands, engage in cults of personalities (see myths about godlike entrepreneurs) and productivity (i.e. not usefulness, morality, efficiency or similar values, just the pure ability to produce something for its own sake). Close minded people will try to counter argue in shallow ways such as "but religion has to have some supernatural entity called God" etc. Again, technically speaking this may be correct, but if we don't limit our views by arbitrary definitions of words, we see that the effects of capitalism on society are de facto of the same or even greater scale than those of religion, and they are certainly more negative. Capitalism itself works towards suppressing traditional religions (showing it is really competing with them and therefore aspiring for the same role) and their values and trying to replace them with worship of money, success and self interest, it permeates society to the deepest levels by making every single area of society a subject of business and acting on the minds of all people in the society every single day which is an enormously strong pressure that strongly shapes mentality of people, again mostly negatively towards a war mentality (constant competition with others), egoism, materialism, fascism, pure pursuit of profit etc. **Capitalism in a society ultimately leaves place for nothing but capitalism**, it seizes every place for itself, just like cancer, it will eventually smother religion, ethics, art, science, technology, culture, ... whatever it is you love you will have to give up to capitalism that in the end will only be making money for the sake of being able to make money; so a capitalist can really only be that who is either too stupid to see this or just loves the purely self serving

existence of money more than existence of anything else.

From a certain point of view capitalism is not really a traditional socioeconomic system, it is **the failure to establish one** -- capitalism is the failure to prevent the establishment of capitalism, and it is also the punishment for this failure. It is the continuation of the jungle to the age when technology for mass production, mass surveillance etc. has sufficiently advanced -- capitalism will arise with technological progress unless we prevent it, just as cancer will grow unless we treat it in very early stages. This is what people mean when they say that capitalism simply works or that it's *natural* -- it's the least effort option, one that simply lets people behave like animals, except that these animals are now equipped with weapons of mass destruction, tools for implementing slavery, world wide surveillance etc. It is natural in the same way in which wars, murders, bullying and deadly diseases are. It is the most primitive system imaginable, it is uncontrolled, leads to suffering and self-destruction.

Under capitalism you are not a human being, you are a resource, at best a machine that's useful for some time but becomes obsolete and undesired once it outlives its usefulness and potential to be exploited. Under capitalism you are a slave that's forced to live the life of 3 Cs: **conform, consume, compete**. Or, as Encyclopedia Dramatica puts it: work, buy, consume, die.

Who invented capitalism? Well, it largely developed on its own, society is just responsible for not stopping it. Capitalism as seen today has mostly evolved from the tradition of small trade, slavery, markets, competition, evil, war and abuse due to societal hierarchy (e.g. peasants by noblemen, poor by rich etc.), combined with technological progress of industrial revolution (18th. - 19th century) which allowed mass production and mass abuse of workers, as well as the information revolution (20th - 21st century) which allowed mass surveillance, unlimited corporate control, acceleration of bullshit business and extreme mass brainwashing, reaching capitalist singularity. Adam Smith (18th century), a mentally retarded egoist with some extra chromosomes who tried to normalize and promote self-interest and torture of others for self-benefit, is often called the **"father of capitalism"** (which is about the same honor as being called the father of holocaust), although he didn't really invent capitalism, he merely supported its spread (saying he invented capitalism would be like saying Hitler invented killing) -- by the same spirit this man is to be also largely credited for the future extermination of all life.

{ My brother who's into movies shared with me a nice example of how capitalism ruined the art of movie dubbing (which my country has a big tradition in) -- it's just one example which however reflects many other areas that got ruined and shows why we just see this huge decline of all art and craft. Back in the day (here during a non-capitalist regime) movie dubbing was done like a play, dubbing was performed scene by scene, all actors were present, they all watched the scene together, then rehearsed it several times and then dubbed it together (on a single microphone); if the result wasn't satisfactory, they tried another take until they were happy. The voice actors got time, creative freedom and were interacting together -- movie dubbing from these times are excellent works of art that sometimes even elevate the original works higher. Nowadays dubbing is done by each actor separately (no interaction between actors), each one scheduled at different time, they work without rehearsal, on first take, the translation is done on tight schedule by the cheapest translator the company finds (usually some student who's doing it as a side job at nights, soon this will probably just be done by AI), the actors are tired as hell as they have to voice many movies in a single day, they are pushed to work quickly and produce as much material as possible and to keep it safe so as to not have to risk additional takes (time loss = money loss), i.e. artistic freedom completely disappears. As different performances are recorded separately, the equipment is also more expensive (there has to be minimum noise as many records will be added together, which will amplify noise, and also someone has to do the mixing etc.). So not only are these dubbing complete and absolute soulless sterile shit without any true emotion and with laughable translation errors, they are also more expensive to make. Capitalism killed the art, humiliated it and in addition made us pay more for it. ~drummyfish }

If we continue along the lines of the valid analogy between capitalism and cancer, we notice that in the past our society used to have a kind of autoimmunity system against this cancer -- people themselves. In human body cancerous cells appear quite regularly, but the immunity system is able to kill those cells before they start growing uncontrollably, as has been happening in our society. In the past we used to have this kind of immunity too, it was the people themselves who would revolt whenever capitalist pressure became too bad -- this has amounted for a great deal of revolutions in history. The capitalism of today however already represent a malignant tumor as we're most likely beyond capitalist singularity, i.e. our society has a tumor we failed to remove at an early stage (we instead decided to feed it), it got out of hand and it can no longer be fixed now, the defensive mechanism such as revolutions are already prevented by capitalism itself, all

communication between is completely controlled, thinking of people is under control too and even if people by a miracle decided to revolt, today's military is so powerful they can't even hope to stand a chance.

On capitalism and Jews: rightists believe the issues caused by capitalism are really caused by Jews and that somehow getting rid of Jews will fix society -- actually this is not entirely accurate; white rightists want to remove Jews so that they (the white race) can take their place in ruling the society, so they don't actually want to fix or remove capitalism (on the contrary, they love its presence and its mechanisms), they just want to become the masters instead of slaves. It is definitely true Jews are overrepresented in high positions of a capitalist society, but that's just because Jews as a race really developed the best "skills" to succeed in capitalism as they historically bet on the right cards (focus on trade and money, decentralization of business, spread across the world and globalization, ...) and really evolved to the race best suited for the winners of the capitalist game. So while the rightist may be correct in the observation that Jews are winning the game, we of course cannot agree with their supposed "fix" -- we do not want to remove the slave masters and replace them with different ones, we want to get rid of capitalism, the unethical system itself which enables slavery in the first place.

{ There is a famous 1988 movie called *They Live* which, while being a funny alines'n'stuff B movie, actually deeply analyzes and criticizes capitalism and for its accurate predictions of the future we now live in became a cult classic. It's been famously said that *They Live* is rather a documentary. I highly recommend giving it a watch. ~drummyfish }

Attributes Of Capitalism

The following is a list of just SOME attributes of capitalism -- note that not all of them are present in initial stages but capitalism will always converge towards them.

- **slavery, oppression, loss of freedom:** In capitalism people are slaves firstly as workers -- in work time, so called wage slavery -- and secondly as consumers -- in "free" time. Banks create inflation to devalue money people save so that they have to work constantly for their whole lives as products are getting progressively more expensive. More and more essentially unnecessary spending purchases are forced on people -- new smartphone every year, mortgages, gas and maintenance of cars, new clothes according to fashion, insurance etc. Practically no one has a truly free time anymore.
- **extreme waste:** Bullshit products, bullshit jobs and the need for constant dynamics of the market force to waste energy, material and human work just to keeping everything in motion, even if purely arbitrarily. Corporations keep reinventing and reselling slightly modified version of already existing products, one group of people is creating something while another group is destroying it, just to keep everyone occupied. Byproduct physical waste such as plastics and chemicals are dumped in the environment and pollute it for decades, even centuries to come. At the moment we are already drowning in physical waste, we just export it to third world and hope they will have infinite space to store more.
- **antivirus paradox:** Sustaining and artificially creating undesirable phenomena so as to build a business in fighting it, to keep and create jobs ("firefighters starting fires").
 - ♦ **artificial scarcity:** In order to be able to sell something, that something has to be scarce, an abundant resource such as air cannot be sold. Once technology emerges to make some resource abundant, it threatens those who have a business in selling that resource. This creates the huge interest in keeping resources scarce, sometimes by force. Corporations are known to routinely destroy food that can still be eaten, and other goods as well. Corporations indirectly conspire on keeping resources scarce by artificial obsolescence, outlawing old products as "unsafe", using copyright to prevent people from recycling old intellectual works etc.
 - ♦ **artificial obsolescence:** To keep businesses running companies have an interest in making people consume even things that could otherwise last them even whole lives, so we see phenomena like people being forced to buy new phones every every year. There used to be the famous light bulb cartel (Phoebus cartel) that fined any bulb manufacturer that made long lasting light bulbs, bulbs were forcefully made to last for a short time. Apple has remotely decreased the performance of older iPhones when new ones came out. There are countless examples.
 - ♦ **artificial crippling of technology:** It is nowadays the norm to create a high tier product, such as a CPU or a car, and then artificially cripple some of the manufactured units (limit car engine power by software, burn parts of the CPU, ...) so as to sell them as a lower tier of that

product. It is cheaper than to separately invent several tiers of the product. So it costs the same (actually less) to create a high end CPU as the low end one -- we could all be using high end CPUs, but the poorer of us are forced to use the forcefully crippled versions, because "capitalism".

- ♦ **purposeful incompatibility in technology:** In market competition products of one company will often be incompatible with products of the competition on purpose, so as to discourage consumers from buying it. Technology corporations create their own "ecosystems" for consumers into which they are trying to lock them.
- ♦ **bullshit jobs, invention of bullshit products/needs:** As automatization takes people's jobs, people try to keep jobs by creating artificial bullshit, e.g. "lack of women in tech" leads to creation of "diversity departments", politicians try to *create more jobs* by increasing bureaucracy etc. This is of course in direct conflict with the base goal of civilization itself of eliminating the need for human work. One online company even successfully sold literal excrement (which had no actual use, it was just marketed as "funny and cool").
- ♦ **preventing progress, sustaining status quo:** Capitalism is extremely hostile towards social progress (more leisure time, more social security, ...), i.e. the main kind of progress (all progress should eventually serve well being of people, otherwise it's just artificial self-serving burden). It is also, contrary to popular belief, against technological progress -- the established corporations want to perpetuate their established businesses and will attack and destroy new ideas that endanger it (i.e. electric cars vs fuel powered cars, food corporations vs the solution of world hunger etc.). Capitalism prevents realization of any idea that's physically possible but which is **economically impossible**, ruling out e.g. many solutions to global heating etc.
- ♦ ...
- **fascism:** Capitalism is based on fascism, i.e. extreme hierarchy and "tribes" of which each fights to death for its own self interest. This fight happens between companies themselves, between state and companies, different departments inside companies, between workers and employers, between brands on the market etc. Capitalism is a constant war against everyone else -- not even jungle has this much conflict.
- **no long term planning, irresponsibility:** Companies need to make immediate profit, managers hired to new positions are expected to immediately increase profits and they don't come to stay for long, they have no responsibility, so they simply do whatever it takes to create immediate profit without considering any long term consequence such as pollution etc.
- **extreme lowering of quality of products, deterioration of art:** Despite capitalist propaganda, capitalism doesn't lead to increased quality of products -- on the contrary it seeks to find the MINIMUM quality that will be accepted by the consumer. In seeking to minimize manufacturing cost of a single unit, companies save money wherever they can and rather invest in marketing etc. -- for example instead of paying several experts to produce a good, well fact-checked documentary, only one man will be paid to create the documentary with the focus on it being "fun and engaging" rather than factually correct. Art is hastened, scheduled, produced on short deadlines, littered with product placement etc.
- **plutocracy, i.e. loss of (true) democracy:** In capitalism only illusion of democracy is sustained, there is no rule of the people, there is rule of the rich THROUGH people, as the rich are those who make the laws and actually take the ruling positions and who have a tremendous power to manipulate masses via private media. State is becoming more and more the tool of corporations rather than a protection against them. USA, the worst case of capitalism, is infamous for having no voting freedom, there exists just a laughable choice of two parties which are exactly the same.
- **monopolies with unlimited power, degeneration of competition:** The naive ideas of capitalists that markets will magically regulate themselves quickly falls apart, basically no one even tries to believe it anymore. In a competitive market monopolies arise in a short time who will prevent any competition from even arising. Can a tiny starting company compete with an established corporations with billions of dollars and thousand lawyers? No. The corporation can defeat them by gigantic marketing, unfair practices (unfair prices etc.) despite fines, by simply buying them, legal trolling, media trolling (negative internet reviews, ...), even physical attacks if necessary (just anonymously pay a bunch of hackers to DDOS competition's servers etc.). Once a monopoly without competition exists, the few advantages of competition disappear completely. A corporation doesn't respond to demand, it creates the demand. It can do whatever it likes, it can set arbitrarily high prices, create arbitrarily shitty products and so on, no competition is pressuring it to do otherwise, people have no choice than to subvert.
- **poverty:** Despite capitalist propaganda, not everyone can be successful in capitalism (if everyone could retire at 20, why doesn't everyone just do it?), and it is a fact that because money makes

money, the gap between the poor and the rich is becoming wider and wider (as of 2020, 8 richest people owned as much wealth as the whole poorer half of the population). Poor people are pushed into loans, getting into debts, trapping themselves, working multiple jobs, while their health deteriorates increasing their debt on medical bills, decreasing their ability to work more etc.

- **torture and killing of people:** The poorest, mostly in third world countries, including children, are forced to hard labor that destroys their lives. Whole cities live off of processing waste coming from first world countries, e.g. disintegrating used ships with primitive tools, no work safety, breathing cancerous fumes etc.
- **materialism: there exists nothing but money:** By definition capitalism advises ONLY to maximize one's profit, any other values such as human well being, peace, preservation of life environment or progress are subverted to the goal of profit. As other values are often in conflict with profit, profit wins and people suffer. People are being attacked, exploited, caught in traps, hunted like animals; see e.g. usury, business with poverty and so on.
- **fight culture, fascism, extreme hostility between people, disappearance of morality:** The very basis of capitalism -- competition -- nurtures people towards self interest, self centeredness and hostility towards others while suppressing good attributes such as sharing, love for others and altruism. With this morals decline and fascist groups arise. Furthermore the system of overcomplicated laws are starting to replace morals, people ask "is it legal?" rather than "is it a good thing to do?". This creates a society of dicks and psychopaths who are additionally rewarded for their immoral behavior by becoming "successful" and wealthy. In long term this serves as a natural selection in Darwinian evolution, immorally behaving people are actually more likely to survive and reproduce, which leads to genes of psychopathic behavior becoming more and more common in society -- under capitalism good people quite literally become extinct in the long run.
- **fear culture:** To keep people consuming and constantly engaged a tension has to be kept, comfortable people are undesirable in capitalism. So there is constantly a propaganda of some threat, be it viruses, terrorism, pedophiles on the internet, computer viruses, killing bees etc.
- **consumerism:** To keep businesses running people need to consume everything, even things that shouldn't be consumed and that could last for very long such as computers and cars. This leads to creation of hasted low quality products (even art such as TV series) that are meant to be used and thrown away, repairing is no longer considered.
- **commerce infects absolutely everything:** In advanced capitalism there is no such thing as a commerce free zone, everything is privatized eventually and serves selfish interests. There is **nowhere to hide**, capitalism has to work towards eliminating escape places as abused people will want to naturally retreat from a place of abuse somewhere safe. Nowadays even such areas as health care, welfare or education of children is permeated by money, ads and corporate propaganda. Even nonprofits have to make money. Educational videos in schools are preceded with ads (as they are played on YouTube), propagandists even legally go to school and brainwash little children (they call it "education in financial literacy" and teach children that they should e.g. create bank accounts in the propagandist's specific bank).
- **destruction of life environment:** This is nowadays already pretty clear, global heating is attributed mainly to capitalism and is seen as maybe the most likely doom that's probably already unavoidable. Lack of long term planning and any concern for anything but money, along with consumerism and extreme waste (of energy, physical waste such as plastic, toxic chemicals etc.) lead to building bullshit factories and performing unnecessary activity for economic reasons (e.g. transporting materials over the globe for assembly, then transporting it back), leading to extreme pollution of air (visible air smog already makes it hard to breathe in many cities), water (it is no longer safe to drink rain water as it used to be) and food (microplastic particles are already basically EVERYWHERE, eating them can't be avoided). Forests that are necessary for cleaning air, host many precious life forms and are overall a key part of ecosystem are being destroyed rapidly, entire species are disappearing very quickly. And that's just a quick sum up.
- **rule of idiots:** Under capitalism the incompetent become successful as success isn't a matter of competence at art but rather willingness to win for any cost, matter of persevering despite being untalented, succumbing to unethical behavior, investing into "promoting" oneself through marketing, social media etc. The truly skilled and intelligent often see the system is bullshit, the skilled are skilled before they want to do their art rather than engage in fights, so they get depressed and disgusted and leave to live in the underground, they live only for their art, opening the way for the unskilled, stupid and at best average thirsty for success. That's why there are so many "professional" wedding photographers who know absolutely nothing about photography, so many elementary school drop outs who become celebrities on TikTok and YouTube who go on to advise the masses on who to vote for in the elections, so many shitty movies, music and games, so many "programmers" and "security

experts" who can't do elementary school math etc.

- **loss of ethical and moral behavior:** Ethical behavior is a disadvantage in a competitive environment of the market, it is a limitation. Those trying to behave ethically (e.g. fair prices or good treatment of employees) will simply lose to the unethically behaving ones and be eliminated from the market. Eventually there only remain unethically behaving entities, which is exactly what we are seeing nowadays -- there basically doesn't exist a single ethically behaving corporation in the world (which has however already been normalized and is no longer seen as an issue). { Where I live there is an old proverb that says "self praise stinks", it's an old wisdom that correctly states people who aren't humble are always evil idiots. Capitalism stands on massive marketing and basically goes all in on this evil, marketing schools are nothing but teaching self praise. ~drummyfish }
- **anti-people design:** By definition in capitalism technology is not to serve people, it is to serve companies to make profit and abuse people, so technology spies on its users, refuses to work (DRM, ...) shows ads, forces children into purchases (predatory games), breaks on purpose so as to enforce a paid repair etc.
- **censorship:** One kind of capitalist censorship is so called intellectual property (allowing "ownership" of ideas, art etc.), but there are many more, e.g. so called moderation of social media which censors specific political views (deemed "politically incorrect" and hence "dangerous" for the advertising potential or brand of the platform) or sharing of certain facts (e.g. those revealing unethical practice of the platform itself, negative reviews of its products etc.). Privately owned media lawfully censor and manipulate information so as to manipulate people in whichever way they see will bring them most profit. While "intellectual property" is marketed as "protecting intellectual workers", in practice it serves corporations and states to do whatever they want, from political censorship, deception and implementing surveillance (justified by "antipiracy") to legal bullying and implementing artificial scarcity ("no, you can't grow this type of food on your field as the plant is patented; only we can grow it and you have to buy it from us in order to live").
- **surveillance:** Companies want to analyze behavior of people, manipulate them, target ads, spam, train neural networks on their data etc., so there is a huge interest in applying surveillance. Indeed, this is exactly what we see in practice -- this is not even a conspiracy theory, cases of revealed mass surveillance are almost daily news.
- **extreme brainwashing and propaganda:** Marketing reaches extreme levels in capitalism, utilizing advanced psychological tricks and repetition to the point of becoming a torture, with the goal of teaching people brand loyalty, consumerist behavior etc. Application of this brainwashing even on children has already been normalized. Entrepreneurs create cults of personalities. There is now even a "legit" job called an influencer whose sole purpose is in spreading corporate propaganda on the Internet.
- **criminality:** This is a direct consequence of poverty, horrible working conditions, fight culture and overall diminishing morality. Poor people (the absolute majority in the system) become desperate and do desperate things -- of course, all blame is put on them, not on those who are responsible for their poverty.
- **instability:** Lack of long term planning, extreme interdependence, monopolies over essential resources, fragility of the market, increasing wealth gap, pushing workers to poverty while taking away social security, eliminating self sufficiency, extreme waste and other phenomena pose great dangers of market collapses, violent strikes and revolutions, running out of resources, destruction of living environment and other disasters and even societal collapse.
- **need for extremely complex market control and laws, burdening society:** As corporations are absolutely unethical and pursue evil goals such as enslaving workers and abuse of consumers, there have to be an extremely complex set of constantly evolving laws and bureaucracy just to somehow "make corporations behave". However laws are imperfect and corporations work 24/7 on bypassing them as well as on attacking and eliminating the laws themselves via lobbyist etc. This creates a constant, extremely expensive legal war-like game in which everyone has to take part, which is completely arbitrary and unnecessary and which eventually corporations will likely win.
- **uncontrolled growth:** Capitalism is likened to cancer as it requires a constant uncontrolled growth which on a planet of limited resources inevitably leads to a catastrophic scenario.
- **hyperspecialization, loss of self sufficiency:** Entities (people, cities, companies, states, ...) lose self sufficiency as they hyperspecialize in some task and in everything else rely on someone else. This complete dependence creates slavery and danger -- in an event of a blackout for example people cannot survive as they cannot make their own food, they can't repair their basic tools etc.
- **loss of humanity:** In capitalism humans are just consumers, machines for production and "resources" -- corporations now routinely use these terms (*human resources department* etc.). People are brainwashed to no longer even see it as concerning to be called such terms. People's worth is only

in how much they can work or consume.

- **abuse of animals:** In capitalism animals are just products and resources, they are kept in very bad conditions just to be killed for meat or other purpose. They are slaughtered by millions just so we can overeat to morbid obesity. Maximizing profit dictates no money should be spent on animal comfort.
- **productivity cult:** People are brainwashed and forced into becoming robots whose sole purpose is to produce. Working overtimes, skipping lunch, minimizing sleep etc. has already become part of the work culture for example in USA and Japan.
- **financial crises:** Crises are as regular and certain in capitalism as rain is in the nature, and possibly much more unpredictable; every crisis hurts everyone but the strongest corporations, whom it in turn makes stronger. People become poorer and great many small and mid-size businesses, i.e. potential competition to the big guys, either die to a bankrupt or are forced to let themselves be devoured by the big guys. This further accelerated the scissors effect, making poor poorer (i.e. better abusable) and rich richer. This is also yet another reason why the small "good guy"/"not only for profit" companies always lose, they simply refuse to steal the food of others to eat themselves obese before a famine and so they will die during the next famine or the next or the one after it. Only the bad guys survive many series of crises.
- **everything is fake:** Everything is rotten and corrupted on the inside with an extreme effort towards putting up a misleading good looking facade. TV shows, including "news", are all staged (even those swearing not to be, no producer is going to invest money in something depending on pure luck), smiles and emotion of people you meet in workplace or see on ads, women's tits, butts and faces, men's muscles, photos on social media, food that has basically no food in it, even news and facts, everything is fake. Investigating any area (government, working conditions, technology, healthcare, education, charities, academia and "science", ...) a bit in depth practically always leads to unrevealing how corrupt it actually is despite it looking nice and ideal at the first look.
- **loss of ownership:** Even that which is most saint to a capitalist -- property and the ability to own things -- is greatly lost under capitalism. This happens with the trend of everything becoming a service, a typical example of which is so called "software as a service". While in the past one would buy a physical copy of a program that he would at least physically own forever, nowadays many programs become a subscription service, e.g. games, movies and music are no longer something you buy but just buy a one time ticket for. But this trend is everywhere, artificial obsolescence tries to limit durability of physical goods such electronic devices so that one has to keep consuming them like a service.
- **rape effect:** The mechanisms of capitalism work in such ways that everyone gets progressively more raped with any further advance of capitalism.
- **endangering existence of all life:** The mentioned destruction of environment, lack of long term planning, irresponsibility and instability along with creating a dangerous overdependence on technology and interconnections of self insufficient states and cities is just waiting for a disaster such as CME that immediately collapses the civilization and consequently endangers all life on Earth e.g. by meltdowns in nuclear plants or possible nuclear wars as a consequence of panic. The absolute preference of immediate profit is hostile towards investments in future and precautions. RIP.
- ...

How It Works

The "old" capitalism, or perhaps its basic forms, that socialist writers have analyzed very well is characterized mainly by abuse of workers by capitalists who declare to "own" means of production such as factories, land and machines -- as e.g. Kropotkin has written in *The Conquest of Bread*, it is **poverty** that drives capitalism because only a poor man who just needs ANY salary for himself and his family will accept horrible working conditions and low pay, simply because he has no other choice -- a capitalist exploits this, "employs" (enslaves) the poor and then only pays them as much as to keep the barely alive and working for him, and he further has the audacity of calling himself an "altruist" who "feeds" people and "gives them a work"; a capitalist employs workers in his factory like he employs chicken in egg factories or pigs in slaughterhouses -- in modern days many may fall to the illusion that workers aren't poor anymore as they may posses smartphones and big screen TVs, but in essence a worker still lives salary to salary and is in desperate need of it; without a salary he will quickly end up starving in the street. Workers do labor that's in itself worth a lot but the capitalist only gives him a small salary, firstly to gain own profit and secondly to keep the worker poor because again, only a poor man will work for him. This is also why capitalists are against anything that would end poverty, such as universal basic income. If the workers owned the factory collectively and didn't have to cut the profit off their labor, they wouldn't have to work so many hours in such harsh conditions at all, it's only because there is a capitalist leech at the top that everyone has to slave

himself to death so that the leech can get enormously rich.

Here a capitalist says to the worker: "I am not forcing you to slavery, if you don't like the working conditions, go elsewhere". Of course, this is a laughable insult -- the capitalist knows very well there is nowhere else to go; wherever you go work in capitalism, you get exploited -- you can only do as much as choose your slavemaster. A capitalist will then say: "start your own business then", which again is a complete idiocy -- it's extremely hard to succeed in business, not everyone can do it, those who have established businesses won't let anyone on the market, and of course it's the immoral thing to do, the capitalist is just telling you to start doing what he's doing: abuse others. If you do start your business, he will be sure to attack you as a competition and with his power he will very likely be able to stop your business. So this advice is similar to that of "go start your own country if you don't like this one" -- he might as well tell you to move to another planet.

The **new**, modern capitalism is yet worse as it takes full advantage of technology never before seen in history which allows extreme increase of exploitation of both workers and consumers -- now there are cameras and computers watching each worker's individual production, there are "smart" devices spying on people and then forcing ads on them, there are loud speakers and screens everywhere full of propaganda and brainwashing, nowhere to escape. Now a single capitalist can watch over his factories all over the world through Internet, allowing for such people to get yet much richer than we could ever imagine.

While the old capitalism was more of a steady slavery and the deterioration of society (life environment, morality, art, ...) by it was relatively slow (i.e. it seemed to be somewhat "working"), nowadays, in the new capitalism the downfall of society accelerates immensely. In countries where capitalism is newly instated, e.g. after the fall of an old regime, it indeed seem to be "working" for a short time, however it will never last -- initially when more or less everyone is at the same start line, when there are no highly evolved corporations with their advanced methods of oppression, small businesses grow and take their small shares of the market, there appears true innovation, businesses compete by true quality of products, people are relatively free and it all feels natural because it is, it's the system of the jungle, i.e. as has been said, capitalism is the failure to establish a controlled socioeconomic system rather than a presence of a purposefully designed one. Its benefits for the people are at this point only a side effect, people see it as good and continue to support it. However the system has other goals of its own, and that is the development and constant growth that's meant to create a higher organism just like smaller living cells formed us, multi cell organisms. The system will start being less and less beneficial to the people who will only become cells in a higher organism to which they'll become slaves. A cell isn't supposed to be happy, it is supposed to sacrifice its life for the good of the higher organism.

{ This initial prosperous stage appeared e.g. in Czechoslovakia, where I lived, in the 90s, after the fall of the totalitarian regime. Everything was beautiful, sadly it didn't last longer than about 10 years. ~drummyfish }

Slowly "startups" evolve to medium sized businesses and a few will become the big corporations. These are the first higher entities that have an intelligence of their own, they are composed of humans and technology who together work solely for the corporation's further growth and profit. A corporation has a super human intelligence (combined intelligence of its workers) but has no human emotion or conscience (which is suppressed by the corporation's structure), it is basically the rogue AI we see in sci-fi horror movies. Corporation selects only the worst of humans for the management positions and has further mechanisms to eliminate any effects of human conscience and tendency for ethical behavior; for example it works on the principle of "I'm just doing my job": everyone is just doing a small part of what the whole company is doing so that no one feels responsible for the whole or sometimes doesn't even know what he's part of. If anyone protests, he's replaced with a new hire. Of course, many know they're doing something bad but they have no choice if they want to feed their families, and everyone is doing it.

Deterioration of society is fast now but people are kept in a false sense of a feeling that "it's just a temporary thing", "it's this individual's fault (not the system's)" and that "it's slowly getting better", mainly with the help of 24/7 almighty media brainwashing. Due to heavy greenwashing, openwashing etc. most people are for example naively convinced that corporations are becoming more "environment friendly", "responsible", "open source" ("Microsoft isn't what it used to be", ...) etc., as if a corporation had something akin emotion instead of pure desire for profit which is its only goal by definition. A corporation will repeat ads telling you it is paying black handicapped gays to plant trees but internally no one gives a shit about anything but making more money, a manager's job is just to increase profit, waste is increasing and dumped to oceans when no one is looking, bullshit is being invented to kickstart more bullshit business which leads to more need for

energy wasting (unnecessary transportation, upkeep of factories and workplaces, invention of bullshit technology to solve artificial problems arising from artificial bullshit). A lie repeated 1000 times a day will beat even truth that's evident to naked eye, basic logic and common sense. Even when sky is littered with ads, cities are burning and people are working 20 hours a day, a capitalist will keep saying "this is a good society", "we are just in a temporary crisis", "it is getting better" and "I care about the people", and people will take it as truth.

Corporations make calculated decisions to eliminate any competition, they devour or kill smaller businesses with unfair practices (see e.g. the Microsoft's infamous EEE), more marketing and by other means, both legal and illegal. They develop advanced psychological methods and exert extreme pressure such as brainwashing by ads to the population to create an immensely powerful propaganda that bends any natural human thinking. With this corporations no longer need to satisfy the demand, they **create the demand** arbitrarily. They create artificial scarcity, manipulate the market, manipulate the people, manipulate laws (those who make laws are nowadays mostly businessmen who want to strengthen corporations whose shares they hold and if you believe voters can somehow prevent such psychopaths getting this power, just take a look literally at any parliament of any country). At this point they've broken the system, competition no longer works as idealized by theoretical capitalists, corporations can now do practically anything they want.

This is an evolutionary system in which the fitness function is simply the ability to make capital. Entities involved in the market are simply chosen by natural selection to be the ones that best make profit, i.e. who are best at circumventing laws, brainwashing, hiding illegal activities etc. Ethical behavior is a disadvantage that leads to elimination; if a business decides to behave ethically, it is outrun by the one who doesn't have this weakness.

The unfair, unethical behavior of corporations is still supposed to be controlled by the state, however corporations become stronger and bigger than states, they can manipulate laws by lobbying, financially supporting preferred candidates, favoring them with their propaganda etc. States are the only force left supposed to protect people from this pure evil, but they are too weak; a single organization of relatively few people who are, quite importantly, often corporation share holder, won't compete against a plethora of the best warriors selected by the extremely efficient system of free market. Furthermore voters, those who are supposed to choose their protectors, are just braindead zombies now who literally do what their cellphones shows them on its display. By all this states slowly turn to serving corporations, becoming their tools and then slowly dissolve (see how small role the US government already plays). Capitalist brainwashing is so strong that **it even makes people desire more torture** -- see so called "anarcho" capitalism which the stupidest of our population have already fallen for and which is basically about saying "let's get rid of anything that protects us against absolute capitalist apocalypse". "Anarcho" capitalism is the worst stage of capitalism where there is no state, no entity supposed to protect the people, there is only one rule and that is the unlimited rule of the strongest corporation which has at its hands the most advanced technology there ever was.

Here the strongest corporation takes over the world and starts becoming the higher organism of the whole Earth, capitalist singularity has been reached. The world corporation doesn't have to pretend anything at this point, it can simply hire an army, it can use physical force, chemical weapons, torture, unlimited surveillance, anything to achieve further seize of remaining bits of power and resources.

People will NOT protest or revolt at this point, they will accept anything that comes and even if they suffer everyday agony and the system is clearly obviously set up for their maximum exploitation, they will do nothing -- in fact they will continue to support the system and make it stronger and they will see more slavery as more freedom; this tendency is already present in rightists today. You may ask why, you think that at some point people will have enough and will seize back their power. This won't happen, just as the billions of chicken and pigs daily exploited at factories won't ever revolt -- firstly because the system will have absolute control over people at this point, they will be 100% dependent on the system even if they hate it, they will have proprietary technology as part of their bodies (which they willingly admitted to in the past as part of bigger comfort while ignoring our warnings about loss of freedom), they will be dependent on drugs of the system (called "vaccines" or "medicine"), air that has to be cleaned and is unbreathable anywhere one would want to escape, 100% of communication will be monitored to prevent any spark of revolution etc. Secondly the system will have rewritten history so that people won't see that life used to be better and bearable -- just as today we think we live in the best times of history due to the interpretation of history that was force fed us at schools and by other propaganda, in the future a human in every day agony will think history was even worse, that there is no other option than for him to suffer every day and it's a privilege he

can even live that way.

We can only guess what will happen here, a collapse due to instability or total destruction of environment is possible, which would at least save the civilization from the horrendous fate of being eternally tortured. If the system survives, humans will be probably be more and more genetically engineered to be more submissive, further killing any hope of a possible change, surveillance chips will be implanted to everyone, reproduction will be controlled precisely and finally perhaps the system will be able, thanks to an advanced AI, to exist and work more efficiently without humans completely, so they will be eliminated. This is how the mankind ends.

{ So here you have it -- it's all here for anyone to read, explained and predicted correctly and in a completely logical way, we even offer a way to prevent this and fix the system, but no one will do it because this will be buried and censored by search engines and the 0.00000000000001% who will find this by chance will dismiss it due to the amount of brainwashing that's already present today. It's pretty sad and depressive, but what more can we do? ~drummyfish }

Capitalist Propaganda And Fairy Tales

Capitalist brainwashing is pretty sophisticated -- unlike with centralized oppressive regimes, capitalism has a decentralized way of creating and spreading propaganda, in ways similar to for example self-replicating and self-modifying malware in the world of software. Creators and promoters of capitalist propaganda are mostly people who are unaware of doing so, they have been brainwashed and programmed by the system itself to behave that way, for example just by being exposed to hearing the capitalist fairy tales since they were born. Some examples of common capitalist propaganda you will probably encounter are the following:

- *"Capitalism is freedom."* -- This is of course a complete twist of the word freedom: capitalism promotes freedom of market which goes against the freedom of people. I.e. the freedom enabled by capitalism is the freedom to abuse others; the freedom to restrict freedoms of others, which is of almost the polar opposite of genuine freedom.
- *"In capitalism everyone can make it if he only works hard."* -- This is firstly of course not even logically possible, it's just as claiming that everyone can win the Olympic games; to reach the top in an extremely competitive system not only do you have to work hard, you also have to be born with the right talent, in the right place, to the right family, and be immensely lucky to be in right places in right times and make correct guess decisions in situations in which it is impossible to know the correct decision. Even if you work 24/7 without sleep, there will be thousands of others who do the same, your success is a pure bet on lottery. **Capitalist fairy tales make heavy use of the survivorship bias** -- you will see movies only about the successful people who are asked how they achieved success and who answer along the lines "I just worked hard". Indeed they did, but this doesn't imply that anyone working hard will succeed, this is the same as taking an advice from a lottery winner; if you ask a lottery winner how he won the lottery, he will simply say "I bet all I had on a random number". Following such advice is of course just about the worse decision you can make. If everyone can make it, why doesn't everyone do it, why don't we have a world consisting exclusively of billionaires?
- *"Capitalism just works, capitalism is natural, capitalism means progress etc."* -- **Capitalism doesn't work, it's just hard to get rid of.** Progress isn't dependent on capitalism, progress can't be stopped and will be here even without capitalism. If it "works and is natural", then it works and is natural natural in the same way as for example cancer or wars. This doesn't mean we should support it or see it as something positive. Most of other similar lies are discussed in sections above.
- Fairy tales about about the rich capitalist altruist: you will hear stories about famous capitalists that paint them as nice guys who give to poor for free, who do manual work despite not having to etc. This is just part of cult of personality propaganda and applying cheap populist tricks to deceive masses. A rich guy giving \$100 to a poor kid on camera is an extreme powerful marketing that costs \$100, which for a billionaire is of course a laughable cost. The fact is that every billionaire is the best player of the most dirty game humanity has invented and researching any such guy reveals, basically in 100% of cases, that he was in fact the worst imaginable psychopath -- Edison killed animals with electric current as part of marketing, the owner of McDonald's stole the idea and know-how from McDonald brothers ALONG WITH their own name, similar thing happened with KFC, Steve Jobs was infamous for his psychological pressure on workers equating torture, the working condition's in Ford's factories were basically the same as those for black African slaves, etcetc. **Trying to find a rich man that's good is like trying to find a shark who's a vegetarian.**

- *"Capitalism isn't perfect but it's the best system we know."*: complete bullshit, capitalism is probably the worst system that there ever was. See this whole article.
- *"Successful capitalists are the most intelligent people."*: on the contrary, mostly the stupidest people become successful -- success in capitalism depends mostly on luck and lack of moral obstacles, i.e. doing whatever it takes to succeed such as stabbing friends in the back, public lying, exploitation of workers etc. High intelligence is actually a disadvantage in this manner as it makes one see all the negative consequences of his behavior, a smart man sees that by engaging in capitalism he is not just hurting and even killing other people, but even e.g. working towards destruction of art, culture, living environment and possibly life itself. Being a successful businessman in capitalism is like steering a plane full of people, including oneself, towards ground -- only an idiot can do it.
- TODO: moar

So What To Replace Capitalism With?

At this point basically anything else is better. But we here advocate less retarded society.

capitalist_singularity

Capitalist Singularity

Capitalist singularity is a point in time at which capitalism becomes irreversible and the cancerous growth of society unstoppable due to corporations taking absolute control over society. It is when people lose any power to revolt against corporations as corporations become stronger than states and any other collective effort towards their control.

This is similar to the famous technological singularity, the difference being that society isn't conquered by a digital AI but rather a superintelligent entity in a form of corporation. While many people see the danger of superintelligent AIs, surprisingly not many have noticed that we've already seen rise of such AIs -- corporations. A corporation is an entity much more intelligent than any single individual, with the single preprogrammed goal of profit. A corporation doesn't have any sense of morals as morals are an obstacle towards making profit. A corporation runs on humans but humans don't control them; there are mechanisms in place to discourage moral behavior of people inside corporations and anyone exhibiting such behavior is simply replaced.

capitalist_software

Capitalist Software

Capitalist software is software that late stage capitalism produces and is practically 100% shitty modern bloat and malware hostile to its users, made with the sole goal of benefiting its creator (often a corporation). Capitalist software is not just proprietary corporate software, but a lot of times "open source", indie software and even free software that's just infected by the toxic capitalist environment -- this infection may come deep even into the basic design principles, even such things as UI design, priorities and development practices and subtle software behavior which have simply all been shaped by the capitalist pressure on abusing the user.

{ Seriously I don't have enough brain to understand how anyone can accept this shit. ~drummyfish }

Capitalist software largely mimics in technology what capitalist economy is doing in society -- for example it employs huge waste of resources (computing resources such as RAM and CPU cycles as an equivalent to natural resources) in favor of rapid growth (accumulation of "features"), it creates hugely complex, interdependent and fragile ever growing networks (tons of library of hardware dependencies as an equivalent of import/export dependencies of countries) and employs consumerism (e.g. in form of mandatory frequent updates). These effects of course bring all the negative implications along and lead to highly inefficient, fragile, bloated, unethical software.

Basically everyone will agree that corporate software such as Windows is to a high degree abusive to its users, be it by its spying, unjustified hardware demands, forced non customizability, price etc. A mistake a lot

of people make is to think that sticking a free license to similar software will simply make it magically friendly to the user and that therefore most FOSS programs are ethical and respect its users. This is sadly not the case, a license is only the first necessary step towards freedom, but not a sufficient one -- other important steps have to follow.

A ridiculous example of capitalist software is the most consumerist type: games. AAA games are pure evil that no longer even try to be good, they just try to be addictive like drugs. Games on release aren't even supposed to work correctly, tons of bugs are the standard, something that's expected by default, customers aren't even meant to receive a finished product for their money. They aren't even meant to own the product or have any control over it (lend it to someone, install it on another computer, play it offline or play it when it gets retired). These games spy on people (via so called anti-cheat systems), are shamelessly meant to be consumed and thrown away, purposefully incompatible ("exclusives"), bloated, discriminative against low-end computers and even targeting attacks on children ("lootboxes"). Game corporations attack and take down fan modification and remakes and show all imaginable kinds of unethical behavior such as trying to steal rights for maps/mods created with the game's editor (Warcraft: Reforged).

But how can possibly a FOSS program be abusive? Let's mention a few examples:

- Being a **bloat monopoly**.
- **Allowing maintenance cost to be high** and prioritizing e.g. features over maintainability leads to programs being expensive to maintain which discriminizes against developers unable to pay this maintenance cost. If a rich corporation intentionally makes their program bloated and expensive to just maintain, it ensures no one poor will be able to fork the software and maintain it (let alone shape it into something better), which effectively removes the possibility of an ethical competition being made out of their "open source" program.
- **Bloat, intentional obscurity and update culture may lead to de-facto (as opposed to de-jure) limitations of basic freedom conditions, despite a free license.** Specifically freedom 1 (to study the software, which may be unnecessarily difficult and **expensive**) and 2 (to modify the software, which requires its understanding, unnecessarily high cost of dealing with bad code and the ability to compile it which may be non-trivial). Therefore a company may, on paper, provide the rights to study and modify their program, but keep the actual know-how of the program's working and modification private, de-facto becoming the program's owner and sole controlling entity.
- **Allowing proprietary dependencies**, which happens especially in open source. While free software usually avoids this, open source is happy with e.g. Windows-only programs which of course requires the users to run abusive code in order for the program to function.
- **Unnecessarily high hardware demands and dropping support for old hardware** which drives consumerism and discriminates against poor people and people who just don't want to "consoom" hardware. A group can make "open source" software that intentionally requires the latest hardware that they just happen to sell (e.g. gaymes with "AAA graphics"), even if the software might in theory run on older hardware. Possible "fixes" of this by third parties can be prevented by the above mentioned techniques.
- **Allowing bloat to increase the risk of security vulnerabilities and bugs** (which may in some cases be fatal and lead to literal deaths).
- **Obscurity and interdependence may be used to successfully hide malicious features even within publicly accessible code.** See for example the anti-Russian "protestware" cases such as node-ipc, an "open source" package that introduced malicious file-wiping code and infected all software depending on it.
- **Introducing dangerous dependencies:** for example a fully free software may be unnecessarily designed as cloud software which increases the risk of its non functionality e.g. in cases of Internet blackouts (or just any loss of connection).
- **Licenses can be bypassed**, e.g. copyleft was legally eliminated by Google's Android which is based on copylefted Linux: their proprietary Play Store is a 3rd party program to which the copyleft doesn't apply but which is essential for Android and serves to control Android (which should have been prevented by the copyleft). This is an example of a FOSS "protection mechanism" failing under capitalist pressure.
- Setting up **discriminatory, fascist and toxic centralized development communities** that de-facto own and control the software and use discriminatory practices and censorship, e.g. with codes of conduct. This allows to bully and "cancel" developers who are, for political or any other reason, unwelcome.

- **Even free software may behave in unethical ways.** For example a company that profits from gambling may create a completely "FOSS" game for children that however teaches them gambling so that when they grow up they'll be more likely to become their victims.

The essential issue of capitalist software is in its goal: profit. This doesn't have to mean making money directly, profit can also mean e.g. gaining popularity and political power. This goal goes before and eventually against goals such as helping and respecting the users. A free license is a mere obstacle on the way towards this goal, an obstacle that may for a while slow down corporation from abusing the users, but which will eventually be overcome just by the sheer power of the market environment which works on the principles of Darwinian evolution: those who make most profit, by any way, survive and thrive.

Therefore "fixing" capitalist software is only possible via redefinition of the basic goal to just developing selfless software that's good for the people (as opposed to making software for profit). This approach requires eliminating or just greatly limiting capitalism itself, at least from the area of technology. We need to find other ways than profit to motivate development of software and yes, other ways do exist (morality, social status, fun etc.).

cat_v

Cat-v

Cat-v.org (accessible at <http://cat-v.org>) is a minimalist hacker website describing itself as a *random contrarian insurgent organization* which promotes critical thinking, free speech, examines technology from minimalist point of view, opposes orthodoxy and talks about wider context of technology such as politics, society and philosophy; the site hosts a few "subsites", e.g. those related to Plan 9 OS and Go language, however most famous is its encyclopedia of things considered harmful (<http://harmful.cat-v.org/>). The whole site, especially the "harmful" section (which was the first one), revolves around the phrase "**considered harmful**" -- this is basically a computer science academic meme that started with a 1968 paper named "Go To Statement Considered Harmful" which was later followed by dozens of similarly named articles; cat-v is taking this to the next level by building a whole website about all things *considered harmful*. The name of the site itself comes from Rob Pike's 1983 presentation "UNIX Style, or cat -v Considered Harmful" that criticized the -v flag of the Unix cat program as such flag, strictly speaking, violates the Unix philosophy (cat should only concatenate files, the flag makes it do something that should rather be done by another program). Though maybe coincidental, the name is also similar to CatB (a short for famous hacker essay/book Cathedral and Bazaar). The site is very nice, made in plain HTML minimalist style, working with HTTP and besides others contains a ton of great quotes on every topic, there is also an IRC, mailing list and a blog.

The section "considered harmful" contains many things, even quite general ones, probably to provoke thought -- one should likely not see a thing present on the list as something we have to always necessarily get rid of -- though many times we should! -- sometimes we just may think about how to improve the thing or minimize its negative impact; try to think of harmful things like "things that suck"; everything sucks, some things just suck less. Among things listed under the *harmful* section are besides others all software, OOP, GNU, Linux, C++, dynamic linking, Java, XML, vim, Emacs, GPL (one recommended alternative being CC0 instead), Perl, standards, Sweden, gay marriage, marriage, children, words, intellectual property, religion, science, minimum wage, the Avatar movie, Wikileaks, people, economics, global warming scaremongering, security theater etc.

Cat-v has existed since at least 2005 (according to Internet Archive) and was started by Uriel M. Pereira, a minimalist hacker who greatly contributed to a lot of suckless software and who committed suicide in 2012. Suckless and cat-v seem to be pretty close -- suckless.org has its own section of harmful things called simply "sucks".

From LRS point of view cat-v is based in great many ways, mainly its focus on the big picture and wider context or technology, promotion of minimalism, freedom of speech and thinking and anti-orthodoxy -- it is not a soyboy site, good quality sites without SJWery are greatly appreciated. However we would also find disagreements e.g. on Plan 9 and Go, which we consider greatly harmful. And of course some politics etc.

See Also

- [suckless](#)
-

cc0

CC0

CC0 is a [waiver](#) (similar to a [license](#)) of [copyright](#), created by [Creative Commons](#), that can be used to dedicate one's work to the [public domain](#) (kind of).

Unlike a license, a waiver such as this *removes* (at least effectively) the author's copyright; by using CC0 the author willingly gives up his own copyright so that the work will no longer be owned by anyone (while a license preserves the author's copyright while granting some rights to other people). It's therefore the most [free](#) and [permissive](#) option for releasing intellectual works. CC0 is designed in a pretty sophisticated way, it also waives "neighboring rights" (e.g. [moral rights](#); waving these rights is why we prefer CC0 over other waivers such as [unlicense](#)), and also contains a fallback license in case waiving copyright isn't possible in a certain country. For this CC0 is one of the best ways, if not the best, of truly and completely dedicating works to public domain world-wide (well, at least in terms of copyright). In this world of extremely fucked up [intellectual property](#) laws it is not enough to state "my work is public domain" -- you need to use something like CC0 to achieve legally valid public domain status.

WATCH OUT: don't confuse CC0 with Creative Commons Public Domain Mark (apart from name the symbols are also a bit similar), the latter is not a license or waiver, just a tag, i.e. CC0 is used to release something to the public domain, while PD mark is used to mark that something is already in the public domain (mostly due to being old).

CC0 is recommended by [LRS](#) for both programs and other art -- however for programs additional waivers of [patents](#) should be added as CC0 doesn't deal with patents. CC0 is endorsed by the [FSF](#) but not [OSI](#) (who rejected it because it explicitly states that trademarks and patents are NOT waived).

It's nice that CC0 became quite widely used and you can find a lot of material under this waiver, but **BEWARE**, if you find something under CC0, do verify it's actually valid, normies often don't know what CC0 means and happily post derivative works of proprietary stuff under CC0.

Some **things under CC0** include Librivox audiobooks, [Dusk OS](#), [Wikidata](#) database, great many things on sites like [Wikimedia Commons](#), [opengameart](#) (see e.g. Kenney), [Blendswap](#), [freesound](#) etc., whole [Esolang Wiki](#), [OSdev Wiki](#) (since 2011), [Encyclopedia Dramatica](#), [LRS](#) software ([Anarch](#), [small3dlib](#), [raycastlib](#), [SAF](#), [comun](#)) and [LRS wiki](#), [books](#) like *The Pig and the Box* (anti [DRM](#) child story) or *Cost of Freedom*, some [fonts](#) by dotcolon, [Lix](#) (libre game), [evlisp](#) minimalist [Lisp](#) (from book "Lisp From Nothing") and many others.

cc

CC

CC can stand for:

- [Creative Commons](#)
 - [C compiler](#)
 - ...
-

censorship

Censorship

This page is not accessible in your country... NOT :)

Censorship means intentional effort towards preventing exchange of certain kind of information among other individuals, for example suppression of free speech, altering old works of art for political reasons, forced takedowns of copyrighted material from the Internet etc. Note that thereby censorship does **NOT** include some kinds of data or information filtering, for example censorship does not include filtering out noise such as spam on a forum or static from audio (as noise is a non-information) or PERSONAL avoidance of certain information (e.g. using adblock or hiding someone's forum posts ONLY FOR ONESELF). Censorship often **hides under euphemisms** such as "moderation", "safe space", "filter", "protection", "delisting", "review" etc. **Censorship is always wrong** -- in a good society there is never a slightest reason to censor anything, therefore whenever censorship is deemed the best solution, something within the society is deeply fucked up. In current society censorship, along with propaganda, brainwashing and misinformation, is extremely prevalent and growing -- it's being pushed not only by governments and corporations but also by harmful terrorist groups such as LGBT and feminism who force media censorship (e.g. that of Wikipedia or search engines) and punishment of free speech (see political correctness and "hate speech").

Sometimes you can actually exploit censorship to get to good content -- look up a block list (e.g. https://en.wikipedia.org/wiki/Category:Blocked_websites_by_country), then you have a list of interesting places you probably want to visit :)

Sometimes it is not 100% clear which action constitutes censorship: for example categorization such as moving a forum post from one thread to another (possibly less visible) thread may or may not be deemed censorship -- this depends on the intended result of such action; moving a post somewhere else doesn't remove it completely but can make it less visible. Whether something is censorship always depends on the answer to the question: "does the action prevent others from information sharing?".

Modern censorship is much more sophisticated; in old days, e.g. those of USSR pseudocommunist regimes, it was simple: stuff was reviewed and it either got censored or it passed, governments even openly admitted to censorship and stated it was simply necessary for the advancement of society. People wanted to talk but the government didn't want to let them. Not so nowadays, it got much advance in several ways:

1. Censorship is no longer done just by the state, but by corporations, various social subgroups and even individuals as well, as so called self censorship, often automatically and subconsciously. In wanting to talk you are not just standing against one big bad guy who wants you silent, there are hundreds of sneaky bastards waiting to sue you, report you, ban you, cancel you, even physically terminate you if you touch anything controversial in one way or another.
2. **NO ONE ADMITS TO CENSORSHIP NOWADAYS, no matter how blatantly obvious their censorship is**, exactly in the capitalist "deny EVERYTHING" spirit -- Wikipedia explicitly states "we are not censored" and then literally removes and blocks inclusion of legitimate information it deems "harmful". You point it out, they ban you. They will say "no, it's not censorship, it is MODERATION, PROTECTION, DELISTING, free speech has its limits, it is not a ban, it is deplatformization, blocking of hate speech is not censorship blablabla ..." -- they are inventing hundreds of new terms so that they don't have to use the word *censorship*.
3. There is a lot of soft, undercover and hard to prove censorship -- no longer is something either censored or not censored, but it may be shadowbanned, hugely underranked in search, censored only to specific eyes, modified rather than deleted etc. For example Google censors thousands of websites; you WILL find those websites if Google sees you are looking specifically for those to test their censorship, but it won't ever show it to people who don't know about the site and are legitimately looking for the information they contain. Maybe they will show the site on the 100th page of the search results, which is equivalent to just blocking it completely, but they can say "haha we are not actually censoring it, gotcha". TV series and movies are silently edited retroactively in the cloud to no longer include scenes deemed politically incorrect, no one notices as no one owns physical copies anymore. In the endgame capitalists will just be constantly updating history, let's say they will just change the characters in Godfather to LGBTQ queer black women and since the movie will only be streamed from the cloud, without any old copied of the original existing, they will just say "the movie has always been like that, the author supported our politics". And so on.

There exist **tools for bypassing censorship**, e.g. proxies or encrypted and/or distributed, censorship-resistant networks such as Tor, Freenet, I2P or torrent file sharing. Watch out: using such tools may be illegal or at least make you look suspicious and be targeted harder by the surveillance.

Examples

TODO: wikipedia, google, fediblock, DVD codes, copyright, ...

See Also

- [free speech](#)
-

chaos

Chaos

In [mathematics](#) chaos is a phenomenon that makes it extremely difficult to predict, even approximately, the outcome of some process even if we completely know how the process works and what state it starts in. In more technical terms chaos is a property of a [nonlinear deterministic system](#) in which even a very small change in input creates a great change in the output, i.e. the system is very sensitive to [initial conditions](#). Chaos is a topic studied by the field called **chaos theory** and is important in all [science](#). In [computer science](#) it is important for example for the generation of [pseudorandom](#) numbers or in [cryptography](#). Every programmer should be familiar with the existence of chaotic behavior because in mathematics (programming) it emerges very often, it may pose a problem but, of course, it may be taken advantage of as well.

Perhaps the most important point is that a chaotic system is difficult to predict NOT because of [randomness](#), lack of information about it or even its incomprehensible complexity (many chaotic systems are defined extremely simply), but because of its inherent structure that greatly amplifies any slight nudge to the system and gives any such nudge a great significance. This may be caused by things such as [feedback loops](#) and [domino effects](#). Generally we describe this behavior as so called **butterfly effect** -- we liken this to the fact that a butterfly flapping its wings somewhere in a forest can trigger a sequence of events that may lead to causing a tornado in a distant city a few days later.

Examples of chaotic systems are the double pendulum, weather (which is why it is so difficult to predict it), dice roll, [rule 30](#) cellular automaton, [logistic map](#), [Baker's map](#), gravitational interaction of [N bodies](#) or [Lorenz differential equations](#). [Langton's ant](#) sometimes behaves chaotically. Another example may be e.g. a billiard table with multiple balls: if we hit one of the balls with enough strength, it'll shoot and bounce off of walls and other balls, setting them into motion and so on until all balls come to stop in a specific position. If we hit the ball with exactly the same strength but from an angle differing just by 1 degree, the final position would probably end up being completely different. Despite the system being deterministic (governed by exact and predictable laws of motion, neglecting things like quantum physics) a slight difference in input causes a great different in output.

A simple example of a chaotic equation is also the function $\sin(1/x)$ for x near 0 where it oscillates so quickly that just a tiny shift along the x axis drastically changes the result. See how unpredictable results a variant of the function can give:

$$x \quad 1000 * \sin(10^9 / x)$$

4.001 455,...

4.002 818,...

4.003 -511,...

4.004 -974,...

4.005 -335,...

Logistic map is often given as the typical example of a chaotic system. It is the series defined as $x[n + 1] = r * x[n] * (1 - x[n])$, which for some constant r (interpreted as speed of population increase) says how a population evolves from some starting value $x[0]$; for low $x[n]$ the population will be increasing proportionally by the rate of r but once it reaches a higher value, it will start decreasing (as if by starvation), resulting in oscillation. Now if we only start to be interested in changing the value r and then seeing at what value the population stabilizes (for a big n), we make some interesting discoveries. This is best seen by plotting the stable values (let's say $x[1000]$) depending on r . For r approximately between 3.57 and 4 we start to see a

chaotic behavior, with results greatly depending on the initial population value ($x[0]$). This demonstrates chaotic behavior.

The following is a fixed point C implementation of the above:

```
#include <stdio.h>

#define FP_UNIT 256
#define DOWNSCALE_X 4
#define DOWNSCALE_Y 25
#define LINE_LENGTH (FP_UNIT / DOWNSCALE_X)
#define GENERATIONS 1000

char stablePoints[LINE_LENGTH + 1];

int main(void)
{
    stablePoints[LINE_LENGTH] = 0; // string terminator

    for (int i = 0; i <= FP_UNIT * 4; i += DOWNSCALE_Y) // for different rs
    {
        for (int j = 0; j < LINE_LENGTH; ++j)
            stablePoints[j] = ' ';

        for (int j = 0; j < FP_UNIT; ++j) // for different starting population sizes
        {
            int population = j;

            for (int k = 0; k < GENERATIONS; ++k)
                population = (i * population * (FP_UNIT - population)) / (FP_UNIT * FP_UNIT);

            population /= DOWNSCALE_X;

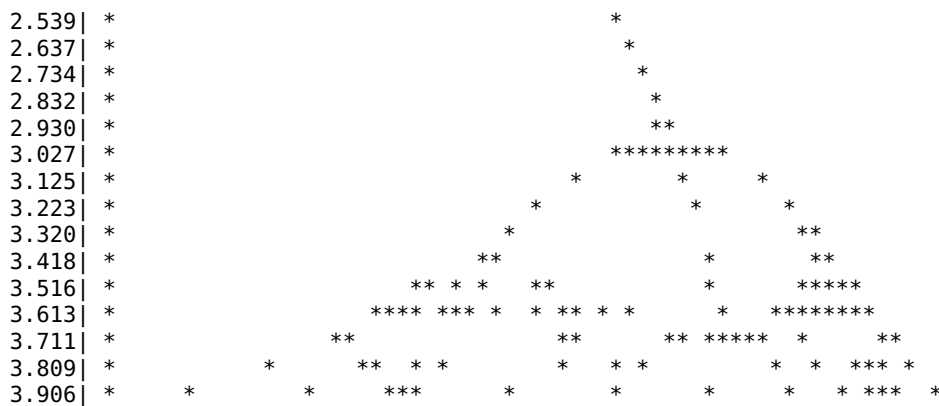
            if (population >= 0 && population < LINE_LENGTH)
                stablePoints[population] = '*';
        }

        printf("%.3f| %s\n", i / ((float) FP_UNIT), stablePoints);
    }

    return 0;
}
```

It outputs the following:

```
0.000| *
0.098| *
0.195| *
0.293| *
0.391| *
0.488| *
0.586| *
0.684| *
0.781| *
0.879| *
0.977| *
1.074| *****
1.172| **      ***
1.270| **      **
1.367| *      **
1.465| *      *
1.562| *      **
1.660| *      *
1.758| *      *
1.855| *      *
1.953| *      *
2.051| *      *
2.148| *      *
2.246| *      *
2.344| *      *
2.441| *      *
```



Vertical axis is the r parameter, i.e. the population growth speed. Horizontal axis shows stable population size after 1000 generations, starting with different initial population sizes. We can see that up until about $r = 3$ the stable population size always stabilizes at around the same size, which gradually increases with r . However then the line splits and after around $r = 3.56$ the stable population sizes are quite spread out and unpredictable, greatly depending on the initial population size. Pure CHAOS!

charity_sex

Charity Sex

We define charity sex as sex selflessly provided for free just to make the other one happy -- this would mostly be done by a women as women decide if sex happens or not, but in rare cases Charity sex can also be provided by an extremely handsome man. If a girl goes around providing a lot of charity sex to guys that are desperate for getting laid (such as incels), she might be called a *charity whore* -- this is a greatly admirable activity, like someone going around buying food for the homeless. If you are girl please do this.

chasm_the_rift

Chasm: The Rift

Poor man's Quake.

Chasm: The Rift is a 1997 FPS game, developed basically by a few Ukrainian basement nerds to be a cheaper version of Quake, which was of course completely overshadowed by this giant and therefore isn't much known but which is nevertheless quite technically impressive upon closer inspection.

{ TODO: do more research about the engine, attempt to translate the Russian YT video, reach the devs.
~drummyfish }

The game requirements were a 486 CPU (which reached 100 MHz at most), 16 MB RAM and 75 MB storage space.

The engine is possibly the most interesting part of the game as it used software rendering that combined a "2.5D" level rendering and "true 3D" polygonal models for things like level decorations, enemies and weapon view model. Not much is known about the internals as the whole code is proprietary and "closed source", we may only inspect it visually and through reverse engineering. To us it is not known if environment rendering uses BSP rendering, portal rendering, raycasting, something similar or whether it just utilizes its 3D model renderer for levels too, however there are some 2.5D simplifications going on as levels are defined as 2D (no room-above-room) and looking up/down is faked (even for the environment inserted "true 3D" models, the up/down look is limited to just a small offset probably to mask the "2.5D" nature of the engine). In fact it isn't even possible to have different height levels of floor, all levels just have the same floor height (ceiling height can be set though)! This is masked a bit by using 3D models onto which it is indeed possible to jump. The game's level editor shows levels use a square grid on which however it is possible to place even non-90 degree walls. There is also a lightmap lighting system present allowing dynamic lights -- a pretty advanced feature, though the lightmap only seems to be 2D, just as the level itself. Destructible environment is also

faked in some levels by having a 3D model behaving like part of a wall, then disappearing when destroyed.

Apart from the engine the game was also nice for being quite KISS, taking similar approach to e.g. Anarch by using very minimal menu and controls: for example there is no door opening or "use" button, items just activate by proximity, weapon switching is also performed by single button. This is actually quite nice as setting up controls and learning them is many times something that just puts you off.

See Also

- Gloom

cheating

Cheating

Cheating means circumventing or downright violating rules, usually while trying to keep such behavior secret. You can cheat on your partner, in games, in business and so forth, however despite cheating seeming like purely immoral behavior at first glance, it may be relatively harmless or even completely moral, for instance in computer graphics we occasionally "cheat" our sense of sight and fake certain visual phenomena which leads to efficient rendering algorithms. In capitalism cheating is demonized and people are brainwashed to partake in **cheater witch hunts** as part of fear culture, arbitrary drama in fight for attention, trying to monopolize game platforms with bloat monopoly "anti cheat" systems etc. These so called "anti cheat" systems introduce unimaginable bloat and bullshit and provide excuse for things like spying (e.g. monitoring OS processes) and proprietary technology (so that "cheaters can't study the system to trick it").

The truth is that **cheating is only an issue in a shitty society** that is driven by competition. Indeed, in such society there is a huge motivation for cheating (sometimes literally physical survival) as well as potentially disastrous consequences of it. Under the tyranny of capitalism we are led to worship heroes and high achievers and everyone gets pissed when we get fooled. Corporations go "OH NOES our multi billion dollar entertainment industry is going to go bankrupt if consoomers get annoyed by cheaters! People are gonna lose their bullshit jobs! Someone is going to get money he doesn't deserve! Our customers may get butthurt!!!" (as if corporations themselves weren't basically just stealing money and raping people lol). So they start a huge brainwashing propaganda campaign, a cheater witch hunt. States do the same, communities do the same, everyone wants to stone cheaters to death but at the same time the society pressures all of us to compete to death with others or else we'll starve. We reward winners and torture the losers, then bash people who try to win -- and no, many times there is no other choice than to cheat, the top of any competition is littered with cheaters, most just don't get caught, so in about 99% of cases the only way to the top is to cheat and try to not get caught, just to have a shot at winning against others. It is proven time after time, legit looking people in the top leagues of sports, business, science and other areas are constantly being revealed as cheaters, usually by pure accident (i.e. the number of actual cheater is MANY times higher). Take a look for instance at the Trackmania cheating scandal in which after someone invented a replay analysis tool he revealed that a great number or top level players were just cheaters, including possibly the GOAT of Trackmania Riolu (who just ragequit and never showed again lol). Of course famous cases like Neil Armstrong don't even have to be mentioned. { I just randomly found out that in the world of Pokemon tournaments cheating at top level also showed to be a huge issue lol. ~drummyfish } Cheater detection systems are (and always will be) imperfect and try to minimize false positives, so only the cheaters who REPEATEDLY make MANY very OBVIOUS mistakes get caught, the smart cheaters stay and take the top places in the competitive system, just as surely as natural selection leads to the evolution of organisms that best adapt to the environment. Even if perfect cheat-detection systems existed, the problem would just shift from cheating to immoral unsportmanship, i.e. abuse of rules that's technically not cheating but effectively presents the same kind of problems. How to solve this enormously disgusting mess? We simply have to stop desperately holding to the system itself, we have to ditch it.

In a good society, such as LRS, cheating is not an issue at all, there's no incentive for it (people don't have to prove their worth by their skills, there are no money, people don't worship heroes, ...) and there are no negative consequences of cheating worse than someone ragequitting an online game -- which really isn't an issue of cheating anyway but simply a consequence of unskilled player facing a skilled one (whether the pro's skill is natural or artificial doesn't play a role, the nub will ragequit anyway). In a good society cheating

can become a mild annoyance at worst, and it can really be a positive thing, it can be fun -- seeing for example a skilled pro face and potentially even beat a cheater is a very interesting thing. If someone wants to win by cheating, why not let him? Valid answers to this can only be given in the context of a shit society that creates cults of personality out of winners etc. In a good society choosing to cheat in a game is as if someone chooses to fly to the top of a mountain by helicopter rather than climbing it -- the choice is everyone's to make.

The fact that cheating isn't after all such an issue is supported by the hilariously vastly different double standards applied e.g. by chess platforms in this matter, on one hand they state in their TOS they have absolutely 0% tolerance of any kind of cheating/assistance and will lifeban players for the slightest suspicion of cheating yelling "WE HAVE TO FIGHT CHEATING", on the other hand they allow streamers literally cheat on a daily basis on live stream where everyone is seeing it, of course because streamers bring them money -- ALL top chess streamers (chessbrah, Nakamura, ...), including the world champion Magnus Carlsen himself, have videos of themselves getting advice on moves from the chat or even from high level players present during the stream, Magnus Carlsen is filmed taking over his friend's low rated account and winning a game which is the same as if the friend literally just used an engine to win the game, and Magnus is also filmed getting an advice from a top grandmaster on a critical move in a tournament that won him the game and granted him a FINANCIAL PRIZE. **World chess champion is literally filmed winning money by cheating and no one cares** because it was done as part of a highly lucrative stream "in a fun/friendly mood". Chessbrah streams ordinarily consist of many viewers in the room just giving advice on moves to the one who is currently playing, of course they censor all comments that try to bring up the fact that this is 100% cheating directly violating the platform's TOS. People literally have no brains, they only freak out about cheating when they're told to by the industry, when cheating is good for business people are told to shut up because it's okay and indeed they just shut up and keep consuming.

chess

Chess

Chess (from Persian *shah*, *king*) is a very old two-player board game, perhaps most famous and popular among all board games in history. In video game terms we could call it a turn-based strategy, in mathematical terms it's a zero sum, complete information game with no element of randomness, that simulates a battle of two armies on an 8x8 board with different battle pieces, also called *chessmen* or just *men*. Chess is also called the King's Game, it has a world-wide competitive community and is considered an intellectual sport but it's also been a topic of research and programming (many chess engines, Als and frontends are being actively developed). Chess is similar to games such shogi ("Japanese chess"), xiangqi ("Chinese chess") and checkers. As the estimated number of chess games is bigger than googol, it is unlikely to ever be solved; though the complexity of the game in sheer number of possibilities is astronomical, among its shogi, go and xiangqi cousins it is actually considered one of the "simplest" (the board is relatively small and the game tends to simplify as it goes on as there are no rules to get men back to the game etc.).

{ There is a nice black and white indie movie called *Computer Chess* about chess programmers of the 1980s, it's pretty good, very oldschool, starring real programmers and chess players, check it out. ~drummyfish }

Drummyfish has created a suckless/LRS chess library smallchesslib which includes a simple engine called *smolchess* (and also a small chess game in SAF with said library).

At LRS we consider chess to be one of the best games for the following reasons:

- It is just a greatly interesting and deep game in which luck plays minimal role.
- **It is greatly suckless**, the rules are very simple, it can be implemented on simple 8bit computers. Of course the game doesn't even require a computer, just a board and a few men -- chess masters don't even need a board to play (they can completely visualize the games in memory). In the end one can in theory just play against himself in his head, achieving the ultimate freedom: the only dependency of the game is one's brain, i.e. it becomes a brain software. Chess is extremely inexpensive, doesn't discriminate against poor people and will survive even the most extreme technological collapse.
- **No one owns chess**, the game is hundreds of years old and many books about it are also already in the public domain. It is extremely free.

- It is a basis for other derived games, for example many different chess variants or chess puzzles which can be considered a "singleplayer chess game".
- It is a source of many interesting mathematical and programming challenges.
- It seems to strike the right balance of simplicity and complexity, it is very simple but not so trivial as to be ever solved in a foreseeable future.

Many however see go as yet a more beautiful game: a more minimal, yet more difficult one, with a completely unique experience.

Where to play chess online? There exist many servers such as <https://chess.com> or <https://chess24.com> -- however these ones are proprietary, so don't use them. For us a possible one is **Lichess** (libre chess) at <https://lichess.org> which not only FOSS, but is also gratis (it also allows users to run bots under special accounts which is an amazing way of testing engines against people and other engines), however it requires JavaScript. Another server, a more suckless one, is **Free Internet Chess Server** (FICS) at <https://www.freechess.org/> -- on this one you can play through telnet (`telnet freechess.org 5000`) or with graphical clients like pychess. Online servers usually rate players with Elo/Glicko just like FIDE, sometimes there are computer opponents available, chess puzzles, variants, analysis tools etc.

Chess as a game is not and cannot be copyrighted, but **can chess games (moves played in a match) be copyrighted?** Thankfully there is a pretty strong consensus and precedence that say this is not the case, even though capital worshippers try to play the intellectual property card from time to time (e.g. 2016 tournament organizers tried to stop chess websites from broadcasting the match moves under "trade secret protection", unsuccessfully).

Chess and IQ/intelligence: there is a debate about how much of a weight general vs specialized intelligence, IQ, memory and pure practice have in becoming good at chess. It's not clear at all, everyone's opinion differs. A popular formula states that *highest achievable Elo* = $1000 + 10 * IQ$, though its accuracy and validity are of course highly questionable. All in all this is probably very similar to language learning: obviously some kind of intelligence/talent is needed to excel, however chess is extremely similar to any other sport in that putting HUGE amounts of time and effort into practice (preferably from young age) is what really makes you good -- without practice even the biggest genius in the world will be easily beaten by a casual chess amateur, and even a relatively dumb man can learn chess very well under the right conditions (just like any dumbass can learn at least one language well); many highest level chess players admit they sucked at math and hated it. As one starts playing chess, he seems to more and more discover that it's really all about studying and practice more than anything else, at least up until the highest master levels where the genius gives a player the tiny nudge needed for the win -- at the grandmaster level intelligence seems to start to matter more. Intelligence is perhaps more of an accelerator of learning, not any hard limit on what can be achieved, however also just having fun and liking chess (which may be just given by upbringing etc.) may have similar accelerating effects on learning. Really the very basics can be learned by literally ANYONE, then it's just about learning TONS of concepts and principles (and automatizing them), be it tactical patterns (forks, pins, double check, discovery checks, sacrifices, smothered mates, ...), good habits, positional principles (pawn structure, king safety, square control, piece activity, ...), opening theory (this alone takes many years and can never end), endgame and mating patterns, time management etcetc.

Fun fact: chess used to be played over telegraph, first such game took place probably in 1844.

How to play chess with yourself? If you have no computer or humans to play against, you may try playing against yourself, however playing a single game against yourself doesn't really work, you know what the opponent is trying to do -- not that it's not interesting, but it's more of a search for general strategies in specific situations rather than actually playing a game. One way around this could be to play many games at once (you can use multiple boards but also just noting the positions on paper as you probably won't be able to set up 100 boards); every day you can make one move in some selected games -- randomize the order and games you play e.g. with dice rolls. The number of games along with the randomized order should make it difficult for you to remember what the opponent (you) was thinking on his turn. Of course you can record the games by noting the moves, but you may want to cover the moves (in which case you'll have to be keeping the whole positions noted) until the game is finished, so that you can't cheat by looking at the game history while playing. If this method doesn't work for you because you can keep up with all the games, at least you know got real good at chess :) { This is an idea I just got, I'm leaving it here as a note, haven't tried it yet. ~drummyfish }

Chess In General

Chess evolved from ancient board games in India (most notably Chaturanga) in about 6th century -- some sources say that in chess predecessor games dice was used to determine which man a player was allowed to move but that once dice were banned because of hazard games, we got the variant without any element of chance. Nowadays the game is internationally governed by **FIDE** which has taken the on role of an authority that defines the official rules: FIDE rules are considered to be the standard chess rules. FIDE also organizes tournaments, promotes the game and keeps a list of registered players whose performance it rates with so called Elo system --â based on the performance it also grants titles such as **Grandmaster** (GM, strongest), **Internation Master** (IM, second strongest) or **Candidate Master** (CM). A game of chess is so interesting in itself that chess is usually not played for money like many other games (poker, backgammon, ...).

The mastery of chess is often divided into two main areas (it is also common to divide strong players into these two categories depending on where their main strength lies):

- **positional play**: Long term, big picture patterns that offer many advantages and opportunities for playing good moves, trying to get a "good position" with men on strong squares, controlling key parts of the board, putting pressure on enemy, ensuring safety of own king etc.
- **tactical play**: Short term, quick action, tricks and calculation skills that win advantages, often material, with tools like forks, pins, discovery checks, sacrifices etc.

Of course this is not the only possible division, another one may be for example offensive vs defensive play etc., but generally chess revolves around position and tactics.

A single game of chess is seen as consisting of three stages: **opening** (starting, theoretical "book" moves, developing men), **middlegame** (seen as the pure core of the game) and **endgame** (ending in which only relatively few men remain on the board). There is no clear border between these stages and they are sometimes defined differently, however each stage plays a bit differently and may require different skills and strategies; for example in the endgame king becomes an active man while in the opening and middlegame he tries to stay hidden and safe.

The study of chess openings is called **opening theory** or just *theory*. Playing the opening stage is special by being based on memorization of this theory, i.e. hundreds or even thousands of existing opening lines that have been studied and analyzed by computers, rather than by performing mental calculation (logical "thinking ahead" present in middlegame and endgame). Some see this as weakness of chess that makes players spend extreme energy on pure memorization. One of the best and most famous players, Bobby Fischer, was of this opinion and has created a chess variant with randomized starting position that prevents such memorization, so called *chess 960*.

Elo rating is a mathematical system of numerically rating the performance of players (it is used in many sports, not just chess). Given two players with Elo rating it is possible to compute the probability of the game's outcome (e.g. white has 70% chance of winning etc.). The FIDE set the parameters so that the rating is roughly this: < 1000: beginner, 1000-2000: intermediate, 2000-3000: master. More advanced systems have also been created, namely the Glicko system.

The rules of chess are quite simple (easy to learn, hard to master) and can be found anywhere on the Internet. In short, the game is played on a 8x8 board by two players: one with **white** men, one with **black** (LOL IT'S RACIST :D). Each man has a way of moving and capturing (eliminating) enemy men, for example bishops move diagonally while pawns move one square forward and take diagonally. The goal is to **checkmate** the opponent's king, i.e. make the king attacked by a man while giving him no way to escape this attack. There are also lesser known rules that noobs often miss and ignore, e.g. so called en-passant or the 50 move rule that declares a draw if there has been no significant move for 50 moves.

At the competitive level **clock** (so called *time control*) is used to give each player a limited time for making moves: with unlimited move time games would be painfully long and more a test of patience than skill. Clock can also nicely help balance unequal opponent by giving the stronger player less time to move. Based on the amount of time to move there exist several formats, most notably **correspondence** (slowest, days for a move), **classical** (slow, hours per game), **rapid** (faster, tens of minutes per game), **blitz** (fast, a few seconds per move) and **bullet** (fastest, units of seconds per move).

Currently the best player in the world is pretty clearly Magnus Carlsen from Norway with Elo rating 2800+.

During covid chess has experienced a small boom among normies and YouTube chess channels have gained considerable popularity. This gave rise to memes such as the bong cloud opening popularized by a top player and streamer Hikaru Nakamura; the bong cloud is an intentionally shitty opening that's supposed to taunt the opponent (it's been even played in serious tournaments lol).

White is generally seen as having a slight advantage over black (just like in real life lol). It is because he always has the first move -- statistics also seems to support this as white on average wins a little more often. This doesn't play such a big role in beginner and intermediate games but starts to become apparent in master games. How big the advantages is is a matter of ongoing debate, most people are of the opinion there exists a slight advantage for the white (with imperfect play, i.e. that white plays easier, tolerates slightly less accurate play), though most experts think chess is a draw with perfect play (pro players can usually quite safely play for a draw and secure it if they don't intend to win; world championships mostly consist of drawn games as really one player has to make a mistake to allow the other one to win). Minority of experts think white has theoretical forced win. Probably only very tiny minority of people think white doesn't have any advantage. Some people argue black has some advantages over white, as it's true that sometimes the obligation to make a move may be a disadvantage. Probably no one thinks black has a forced win, though that's not disproved yet so maybe someone actually believes it.

On **perfect play**: as stated, chess is unlikely to be ever solved so it is unknown if chess is a theoretical forced draw or forced win for white (or even win for black), however many simplified endgames and some simpler chess variants have already been solved. Even if chess was ever solved, it is important to realize one thing: **perfect play may be unsuitable for humans** and so even if chess was ever solved, it might have no significant effect on the game played by humans. Imagine the following: we have a chess position in which we are deciding between move *A* and move *B*. We know that playing *A* leads to a very good position in which white has great advantage and easy play (many obvious good moves), however if black plays perfectly he can secure a draw here. We also know that if we play *B* and then play perfectly for the next 100 moves, we will win with mathematical certainty, but if we make just one incorrect move during those 100 moves, we will get to a decisively losing position. While computer will play move *B* here because it is sure it can play perfectly, it is probably better to play *A* for human because human is very likely to make mistakes (even a master). For this reason humans may willingly choose to play mathematically worse moves -- it is because a slightly worse move may lead to a safer and more comfortable play for a human.

Fun fact: there seem to be **almost no black people in chess** :D the strongest one seems to be Pontus Carlsson which rates number 1618 in the world; even women seem to be much better at chess than black people. But how about black women? LMAO, it seems like there haven't even been any black female masters :D The web is BLURRY on these facts, but there seems to be a huge excitement about one black female, called Rochelle Ballantyne, who at nearly 30 years old has been sweating for a decade to reach the lowest master rank (the one which the nasty oppressive white boys get at like 10 years old) and MAYBE SHE'LL DO IT, she seems to have with all her effort and support of the whole Earth overcome the 2000 rating, something that thousands of amateurs on the net just causally do every day without even trying too much. But of course, it's cause of the white male oppression =3 lel

Chess And Computers

{ This is an absolutely amazing video about weird chess algorithms :) ~drummyfish }

Chess is a big topic in computer science and programming, computers not only help people play chess, train their skills, analyze positions and perform research of games, but they also allow mathematical analysis of chess as such and provide a platform for things such as artificial intelligence.

Chess software is usually separated to **libraries**, **chess engines** and **frontends**. Chess engine is typically a CLI program capable of playing chess but also doing other things such as evaluating arbitrary position, hinting best moves, saving and loading games etc. -- commonly the engine has some kind of custom CLI interface (flags, interactive commands it understands, ...) plus a support of some standardized text communication protocol, most notably XBoard (older one, more KISS) and UCI (newer, more bloated). There is also typically support for standardized formats such as FEN (way of encoding a chess position as a text string), PGN (way of encoding games as text strings) etc. Frontends on the other hand are usually GUI programs (in this case also called *boards*) that help people interact with the underlying engine, however

there may also be similar non-GUI programs of this type, e.g. those that automatically run tournaments of multiple engines.

Computers have already surpassed the best humans in their playing strength (we can't exactly compute an engine's Elo as it depends on hardware used, but generally the strongest would rate high above 3000 FIDE). As of 2023 the strongest chess engine is widely agreed to be the FOSS engine Stockfish, with other strong engines being e.g. Leela Chess Zero (also FOSS), AlphaZero (proprietary by Google) or Komodo Dragon (proprietary). GNU Chess is a pretty strong free software engine by GNU. There are world championships for chess engines such as the *Top Chess Engine Championship* or *World Computer Chess Championship*. CCRL is a list of chess engines along with their Elo ratings deduced from tournaments they run. Despite the immense strength of modern engines, there are still some specific artificial situations in which a human beats the computer (shown e.g. in this video); this probably won't last long though.

The first chess computer that beat the world champion (at the time Gary Kasparov) was famously Deep Blue in 1997. Alan Turing himself has written a chess playing algorithm but at his time there were no computers to run it, so he executed it by hand -- nowadays the algorithm has been implemented on computers (there are bots playing this algorithm e.g. on lichess).

Playing strength is not the only possible measure of chess engine quality, of course -- for example there are people who try to make the **smallest chess programs** (see countercomplex and golfing). As of 2022 the leading programmer of smallest chess programs seems to be Åscar Toledo G. (<https://nanochess.org/chess.html>). Unfortunately his programs are proprietary, even though their source code is public. The programs include Toledo Atomchess (392 x86 instructions), Toledo Nanochess (world's smallest C chess program, 1257 non-blank C characters) and Toledo Javascript chess (world's smallest javascript chess program). He won the IOCCC. Another small chess program is micro-Max by H. G. Muller (<https://home.hccnet.nl/h.g.muller/max-src2.html>, 1433 C characters, Toledo claims it is weaker than his program). Other engines try to be strong while imitating human play (making human moves, even mistakes), most notably Maia which trains several neural networks that play like different rated human players.

{ Nanochess is actually pretty strong, in my testing it easily beat smallchesslib Q_Q ~drummyfish }

Programming Chess

NOTE: our smallchesslib/smolchess engine is very simple, educational and can hopefully serve you as a nice study tool to start with :)

There is also a great online wiki focused on programming chess engines:
<https://www.chessprogramming.org>.

Programming chess is a fun and enriching experience and is therefore recommended as a good exercise. There is nothing more satisfying than writing a custom chess engine and then watching it play on its own.

The core of chess programming is writing the AI. Everything else, i.e. implementing the rules, communication protocols etc., is pretty straightforward (but still a good programming exercise). Nevertheless, as the chess programming wiki stresses, one has to pay a great attention to eliminating as many bugs as possible; really, the importance of writing automatic tests can't be stressed enough as debugging the AI will be hard enough and can become unmanageable with small bugs creeping in. Though has to go into choosing right data structures so as to allow nice optimizations, for example board representation plays an important role (two main approaches are a 64x64 2D array holding each square's piece vs keeping a list of pieces, each one recording its position).

The AI itself works traditionally on the following principle: firstly we implement so called static **evaluation function** -- a function that takes a chess position and outputs its evaluation number which says how good the position is for white vs black (positive number favoring white, negative black, zero meaning equal, units usually being in pawns, i.e. for example -3.5 means black has an advantage equivalent to having extra 3 and a half pawns; to avoid fractions we sometimes use centipawns, i.e. rather -350). This function considers a number of factors such as total material of both players, pawn structure, king safety, men mobility and so on. Traditionally this function has been hand-written, nowadays it is being replaced by a learned neural network (NNUE) which showed to give superior results (e.g. Stockfish still offers both options); for starters you probably want to write a simple evaluation function manually.

Note: if you could make a perfect evaluation function that would completely accurately state given position's true evaluation (considering all possible combinations of moves until the end of game), you'd basically be done right there as your AI could just always make a move that would lead to position which your evaluation function rated best, which would lead to perfect play. Though neural networks got a lot closer to this ideal than we once were, as far as we can foresee ANY evaluation function will always be just an approximation, an estimation, heuristic, many times far away from perfect evaluation, so we cannot stop at this. We have to program yet something more.

So secondly we need to implement a so called **search** algorithm -- typically some modification of minimax algorithm, e.g. with alpha-beta pruning -- that recursively searches the game tree and looks for a move that will lead to the best result in the future, i.e. to position for which the evaluation function gives the best value. This basic principle, especially the search part, can get very complex as there are many possible weaknesses and optimizations. Note now that this search kind of improves on the basic static evaluation function by making it dynamic and so increases its accuracy greatly (of course for the price of CPU time spent on searching).

Exhaustively searching the tree to great depths is not possible even with most powerful hardware due to astronomical numbers of possible move combinations, so the engine has to limit the depth quite greatly and use various hacks, approximations, heuristics etc.. Normally it will search all moves to a small depth (e.g. 2 or 3 half moves or *plys*) and then extend the search for interesting moves such as exchanges or checks. Maybe the greatest danger of searching algorithms is so called **horizon effect** which has to be addressed somehow (e.g. by detecting quiet positions, so called *quiescence*). If not addressed, the horizon effect will make an engine miscalculate certain moves by stopping the evaluation at certain depth even if the played out situation would continue and lead to a vastly different result (imagine e.g. a queen taking a pawn which is guarded by another pawn; if the engine stops evaluating after the pawn take, it will think it's a won pawn, when in fact it's a lost queen). There are also many techniques for reducing the number of searched tree nodes and speeding up the search, for example pruning methods such as **alpha-beta** (which subsequently works best with correctly ordering moves to search), or **transposition tables** (remembering already evaluated position so that they don't have to be evaluated again when encountered by a different path in the tree).

Alternative approaches: most engines work as described above (search plus evaluation function) with some minor or bigger modifications. The simplest possible stupid AI can just make random moves, which will of course be an extremely weak opponent (though even weaker can be made, but these will actually require more complex code as to play worse than random moves requires some understanding and searching for the worst moves) -- one might perhaps try to just program a few simple rules to make it a bit less stupid and possibly a simple training opponent for complete beginners: the AI may for example pick a few "good looking" candidate moves that are "usually OK" (pushing a pawn, taking a higher value piece, castling, ...) and aren't a complete insanity, then pick one at random only from those (this randomness can further be improved and gradually controlled by scoring the moves somehow and adding a more or less random value from some range to each score, then picking the moves with highest score). One could also try to just program in a few generic rules such as: checkmate if you can, otherwise take an unprotected piece, otherwise protect your own unprotected piece etc. -- this could produce some beginner level bot. Another idea might be a "Chinese room" bot that doesn't really understand chess but has a huge database of games (which it may even be fetching from some Internet database) and then just looking up what moves good players make in positions that arise on the board, however a database of all positions will never exist, so in case the position is not found there has to be some fallback (e.g. play random move, or somehow find the "most similar position" and use that, ...). As another approach one may try to use some **non neural network machine learning**, for example genetic programming, to train the evaluation function, which will then be used in the tree search. Another idea that's being tried (e.g. in the Maia engine) is **pure neural net AI** (or another form of machine learning) which doesn't use any tree search -- not using search at all has long been thought to be impossible as analyzing a chess position completely statically without any "looking ahead" is extremely difficult, however new neural networks have shown to be extremely good at this kind of thing and pure NN AIs can now play on a master level (a human grandmaster playing ultra bullet is also just a no-calculation, pure pattern recognition play). Next, **Monte Carlo tree search** (MCTS) is an alternative way of searching the game tree which may even work without any evaluation function: in it one makes many random playouts (complete games until the end making only random moves) for each checked move and based on the number of wins/losses/draws in those playouts statistically a value is assigned to the move -- the idea is that a move that most often leads to a win is likely the best. Another Monte Carlo approach may just make random playouts, stop at random depth and then use normal static evaluation function (horizon

effect is a danger but hopefully its significance should get minimized in the averaging). However MCTS is pretty tricky to do well. MCTS is used e.g. in Komodo Dragon, the engine that's currently among the best. Another approach may lie in somehow using several methods and heuristics to vote on which move would be best.

Many other aspects come into the AI design such as opening books (databases of best opening moves), endgame tablebases (precomputed databases of winning moves in simple endgames), clock management, pondering (thinking on opponent's move), learning from played games etc. For details see the above linked chess programming wiki.

Notable Chess Engines/Computers/Entities

See also ratings of computer engines at <https://www.computerchess.org.uk/ccrl/4040/>.

Here are some notable chess engines/computers/entities, as of 2024:

- **Stockfish** (SF): FOSS engine (written in C++), without any doubt **the strongest chess engine** that's been reliably winning all the computer tournaments for years now; its strength is far beyond any human, even if run on quite a weak device -- it actually caused some trouble because it's extremely easy to just download onto a cellphone and cheat even in OTB tournaments. Currently the engine is using a neural network for evaluating positions but still also uses the tree search algorithm (a greatly optimized one so that it searches gigantic numbers of positions per second). Important part of the development is so called *Fishtest*, a distributed framework for testing and improving the engine's performance, it's one of the reasons why it good so strong. Stockfish's current CCRL Elo rating is 3639 (warning: this is incomparable to human Elos).
- **Magnus Carlsen**: Human, most likely the strongest player ever, has been quite comfortably winning every tournament he entered including the world championship until he quit, basically because he got "bored". His top FIDE Elo was 2882.
- **Komodo Dragon**: Proprietary, currently seems to be the second strongest engine, it's main feature is [Monte Carlo] ("randomized") search algorithm. Current CCRL Elo is 3624.
- **Leela Chess Zero** (lc0): FOSS engine (written in C++), among top strongest engines (currently top 50 on CCRL), it is interesting mainly for how it works: it is a neural network engine that's **completely self-taught** from the ground up, i.e. it didn't learn chess by watching anyone else play, it was only allowed to learn by playing against itself. Current CCRL Elo is 3441.
- **Deep Blue**: A historically famous supercomputer, the first one to have beaten the human world chess champion in 1997.
- **GNU chess**: Free engine by GNU, not among absolute top by strength but still very strong. Current CCRL Elo is 2825.
- **Maia**: FOSS engine, or rather neural network, notable by not trying to be the strongest, but rather most human-like, i.e. tries to imitate human play, even with errors. There are several versions, each trained for different strength. It is also notable by using pure neural network, i.e. it doesn't perform any search, it's a pure "pattern recognition"/static engine that still manages to play quite well.
- **Toledo Nanochess**: Seems to be the world's smallest C chess engine, with only 1257 non-blank characters of source code.
- **smallchesslib/smolchess**: Tiny LRS C library/engine, very weak but is very simple, small and portable, may be good enough in many situations.
- **Chessmaster**: A famous proprietary chess video games with its own engine, it was strong for a video game of its time (around 2000 Elo) but nowadays would be considered rather weak for an engine -- its significance is cultural, it's used for comparisons, many people played against it and still use it to test their engines against.
- **Turochamp**: Probably the first chess program ever, made by David Champernowne and Alan Turing himself in 1948, in times when computers still couldn't execute it! It was very primitive, looking only two moves ahead, and was only ever executed manually -- of course, it got raped pretty bad the human opponent.
- ...

Stats And Records

Chess stats are pretty interesting.

{ Some chess world records are here: <https://timkr.home.xs4all.nl/records/records.htm>. ~drummyfish }

Number of possible games is not known exactly, Shannon estimated it at 10^{120} (lower bound, known as *Shannon number*). Number of possible games by plies played is 20 after 1, 400 after 2, 8902 after 3, 197281 after 4, 4865609 after 5, and 2015099950053364471960 after 15.

Similarly the **number of possibly reachable positions** (position for which so called *proof game* exists) is not known exactly, it is estimated to at least 10^{40} and 10^{50} at most. Numbers of possible positions by plies is 20 after 1, 400 after 2, 5362 after 3, 72078 after 4, 822518 after 5, and 726155461002 after 11.

Shortest possible checkmate is by black on ply number 4 (so called *fool's mate*). As of 2022 the **longest known forced checkmate** is in 549 moves -- it has been discovered when computing the Lomonosov Tablebases.

Average game of chess lasts 40 (full) moves (80 plies). **Average branching factor** (number of possible moves at a time) is around 33. **Maximum number of possible moves in a position** seems to be 218 (FEN: R6R/3Q4/1Q4Q1/4Q3/2Q4Q/Q4Q2/pp1Q4/kBNN1KB1 w - - 0 1).

White wins about 38% of games, black wins about 34%, the remaining 28% are draws (38.7%, 31.1%, 30.3% respectively on Computer Chess Rating Lists).

What is the **longest possible game**? It depends on the exact rules and details we set, for example if a 50 move rule applies, a player MAY claim a draw but also doesn't have to -- but if neither player ever claims a draw, a game can be played infinitely -- so we have to address details such as this. Nevertheless the longest possible chess game upon certain rules has been computed by Tom7 at 17697 half moves in a paper for SIGBOVIK 2020. Chess programming wiki states 11798 half moves as the maximum length of a chess game which considers a 50 move rule (1966 publication).

The **longest game played in practice** is considered to be the one between Nikolic and Arsovic from 1989, a draw with 269 moves lasting over 20 hours. For a shortest game there have been ones with zero moves; serious decisive shortest game has occurred multiple times like this: 1.d4 Nf6 2.Bg5 c6 3.e3 Qa5+ (white resigned).

Best players ever: a 2017 paper called *Who is the Master?* analyzed 20 of the top players of history based on how good their moves were compared to Stockfish, the strongest engine. The resulting **top 10** was (from best): Carlsen (born 1990 Norway, peak Elo 2882), Kramnik (born 1975 Russia, peak Elo 2817), Fischer (born 1943 USA, peak Elo 2785), Kasparov (born 1963 Russia, peak Elo 2851), Anand (born 1969 India, peak Elo 2817), Khalifman, Smyslov, Petrosian, Karpov, Kasimdzhanov. It also confirmed that the quality of chess play at top level has been greatly increasing. The **best woman player** in history is considered to be Judit Polgar (born 1976 Hungary, peak Elo 2735), which still only managed to reach some 49th place in the world; by Elo she is followed by Hou Yifan (born 1994 China, peak Elo 2686) and Koneru Humpy (born 1987 India, peak Elo 2623). **Strongest players of black race** (NOT including brown, e.g. India): lol there don't seem to be many black players in chess :D The first black GM only appeared in 1999 (!!!) -- Maurice Ashley (born 1966 Jamaica, peak rating 2504) who is also probably the most famous black chess player, though more because of his commentator skills; Pontus Carlsson (peak Elo 2531) may be strongest. { Sorry if I'm wrong about the strongest black player, this information is pretty hard to find as of course you won't find a race record in any chess player database. So thanks to political correctness we just can't easily find good black players. ~drummyfish }

What's the **most typical game**? We can try to construct such a game from a game database by always picking the most common move in given position. Using the lichess database at the time of writing, we get the following incomplete game (the remainder of the game is split between four games, 2 won by white, 1 by black, 1 drawn):

```
1. e4 e5 2. Nf3 Nc6 3. Bc4 Bc5 4. c3 Nf6 5. d4 exd4
6. cxd4 Bb4+ 7. Nc3 Nxe4 8. O-O Bxc3 9. d5 Bf6 10. Re1 Ne7
11. Rxe4 d6 12. Bg5 Bxg5 13. Nxg5 h6 14. Qe2 hxd3
15. Re1 Be6 16. dxe6 f6 17. Re3 c6 18. Rh3 Rxh3
19. gxh3 g6 20. Qf3 Qa5 21. Rd1 Qf5 22. Qb3 O-O-O
23. Qa3 Qc5 24. Qb3 d5 25. Bf1
```

You can try to derive your own stats, there are huge free game databases such as the Lichess [CC0](#) database of billions of games from their server.

{ TODO: Derive stats about the best move, i.e. for example "best move is usually by queen by three squares" or something like that. Could this actually help the play somehow? Maybe could be used for move ordering in alpha-beta. ~drummyfish }

Rules

The exact rules of chess and their scope may depend on situation, this is just a sum up of rules generally used nowadays.

The start setup of a chessboard is following (lowercase letters are for black men, uppercase for white men, on a board with colored squares A1 is black):

```

      /8 | r n b q k b n r |
r | 7 | p p p p p p p p |
a | 6 | . . . . . . . . |
n | 5 | . . . . . . . . |
k | 4 | . . . . . . . . |
s | 3 | . . . . . . . . |
  | 2 | P P P P P P P P |
  \1 | R N B Q K B N R |
      " " " " " " " " " "
      A B C D E F G H
      \-----/
          files

```

Players take turns in making moves, white always starts. A move consists of moving one (or in special cases two) of own men from one square to another, possibly capturing (removing from the board) one opponent's man -- except for a special *en passant* move capturing always happens by moving one man to the square occupied by the opposite color man (which gets removed). Of course no man can move to a square occupied by another man of the same color. A move can NOT be skipped. A player wins by giving a **checkmate** to the opponent (making his king unable to escape attack) or if the opponent resigns. If a player is to move but has no valid moves, the game is a draw, so called **stalemate**. If neither player has enough men to give a checkmate, the game is a draw, so called **dead position**. There are additional situation in which game can be drawn (threefold repetition of position, 50 move rule). Players can also agree to a draw. A player may also be declared a loser if he cheated, if he lost on time in a game with clock etc.

The individual men and their movement rules are (no man can move beyond another, except for knight who jumps over other men):

man	symbol	~value	movement	comment
pawn	P	1	1F, may also 2F from start, captures 1F1L or 1F1R, also <i>en passant</i>	promotes on last row
knight	N	3	L-shape (2U1L, 2U1R, 2R1U, 2R1D, 2D1R, 2D1L, 2L1U, 2L1D), jumps over	
bishop	B	3.25	any distance diagonally	stays on same color sq.
rook	R	5	any distance orthogonally (U, R, D or L)	can reach all sq.
queen	Q	9	like both bishop and rook	strongest piece
king	K	inf	any of 8 neighboring squares	

{ Cool players call knights *horses* or *ponies* and pawns *peasants*, rook may be called a *tower* and bishop a *sniper* as he often just sits on the main diagonal and shoot pieces that wonder through. Also pronounce *en passant* as "en peasant". Nakamura just calls all pieces a *juicer*. ~drummyfish }

Check: If the player's king is attacked, i.e. it is immediately possible for an enemy men to capture the king, the player is said to be in check. A player in check has to make such a move as to not be in check after that move.

A player cannot make a move that would leave him in check!

Castling: If a player hasn't castled yet and his king hasn't been moved yet and his kingside (queenside) rook hasn't been moved yet and there are no men between the king and the kingside (queenside) and the king isn't and wouldn't be in check on his square or any square he will pass through or land on during castling, short (long) castling can be performed. In short (long) castling the king moves two squares towards the kingside (queenside) rook and the rook jumps over the king to the square immediately on the other side of the king.

Promotion: If a pawn reaches the 1st or 8th rank, it is promoted, i.e. it has to be switched for either queen, rook, bishop or knight of the same color.

Checkmate: If a player is in check but cannot make any move to get out of it, he is checkmated and lost.

En passant: If a pawn moves 2 squares forward (from the start position), in the immediate next move the opponent can take it with a pawn in the same way as if it only moved 1 square forward (the only case in which a man captures another man by landing on an empty square).

Threefold repetition is a rule allowing a player to claim a draw if the same position (men positions, player's turn, castling rights, en passant state) occurs three times (not necessarily consecutively). The 50 move rule allows a player to claim a draw if no pawn has moved and no man has been captured in last 50 moves (both players making their move counts as a single move here).

Variants

Besides similar games such as shogi there are many variants of chess, i.e. slight modifications of rules, foremost worth mentioning is for example chess 960. The following is a list of some variants:

- **antichess** (suicide, ...): The goal is to lose all men or get stalemated, rules are a bit changed, e.g. castling and checks are removed and taking is forced.
- **chess 960** aka **Fischer's random**: Starting position is randomly modified by shuffling the non-pawn rows (with these rules: king must be between rooks, bishops on opposite colors and black/white's positions are mirrored). The rules are the same with a slight modification to castling. This was invented by Bobby Fischer to emphasize pure chess skill as opposed to memorizing the best opening moves, he saw the opening theory as harmful to chess. Chess 960 is nowadays even advocated by some to become the "main" version of chess.
- **chess boxing**: Chess combined with box, players switch between the two games, one wins either by checkmate or knockout.
- **crazyhouse**: When a player captures a man, it goes into his reserve. From the reserve a man can be dropped (as a man of the current player's color) to an empty square instead of making a normal move. This is a rule taken from shogi.
- **different men**: Some variants use different men, e.g. empress (moves like rook and knight) or amazon (queen/knight).
- **duck chess**: After each move players place a duck on an empty square, the duck blocks the square. The duck cannot be left on the same square, it has to be moved. There are no checks, players win by capturing the king.
- **fog of war**: Makes chess an incomplete-information game by allowing players to only see squares they can immediately move to (this is similarly to some strategy video games).
- **horde chess**: Asymmetric starting position: large number of black pawns vs a white army of traditional men. Rules are slightly modified, e.g. black can only be defeated by having all pawns captured (there is no black king).
- **infinite chess**: Infinite chessboard. { Huge rabbit hole with things like "mate in omega" etc. ~drummyfish }
- **minichess**: Smaller chessboard, e.g. 4x4, 4x8 etc. Los Alamos chess is played at 6x6 board without bishops (also no promotion to bishop, no pawn double step, no en passant, no castling). Some are already solved (e.g. 3x3).
- **more players**: E.g. 3 man chess or 4 player chess allow more than two players to play, some use different boards.
- **old chess**: The rules of chess itself have been changing over time (e.g. adding the 50 move rule etc.). The older rule sets can be seen as variants as well.

- **puzzle:** For single player, chess positions are presented and the player has to find the best move or sequence of moves.
- **racing kings:** The starting position has both players on the same side, the goal is to get one's king to the other side first.
- **different board geometries/topologies:** e.g. non-Euclidean (hyperbolic, spherical, torus, ...), hexagonal chess (had some considerable following) etc.
- **3D chess:** 3D generalization of chess, possible are also other dimensions (4D, 5D, ... maybe even 1D?).
- **randomly chosen variant:** Here a chess variant to be played is chosen at random before the game, e.g. by dice roll. { This is an idea I got, not sure if this exists or has a different name. ~drummyfish }

Playing Tips

Some general tips and rules of thumb, mostly for beginners:

- Try to control the center of the board (D4, D5, E4, E5).
- Don't bring the queen out too early, the opponent can harass it and get ahead in development.
- Learn some universal setup openings or "systems" to play, e.g. London, King's Indian, the hippo etc.
- Develop your men before attacking, usually knights go out before bishops, bishops are well placed on the longest diagonals as "snipers".
- Learn basic tactics, especially **forks** (attacking two or more men at once so that one of them cannot escape capture) and **pins** (attack one man so that if he moves out of the way he will expose another one to be captured).
- King safety is extremely important until endgame, castle very early but not extremely early. In the endgame (with queens out) king joins the battle as another active man.
- Pawn structure is very important.
- Watch out for back rank checkmates, make an escape square for your king.
- Rooks want to be on open files, you also want to CONNECT them (have both guard each other). Also a rook in the opponents second row (2nd/7th rank) is pretty good.
- Bishops are generally seen a bit more valuable than knights, especially in pairs -- if you can trade your knight for opponent's bishop, it's often good. If your opponent has two bishops and you only have one, you want to trade yours for his so he doesn't have the pair. A knight pair is also pretty powerful though, especially when the knights are guarding each other.
- "Knight on a rim is dim" (knights are best placed near the center).
- An extremely strong formation is both rooks and the queen on the same open file.
- Blocking the opponents man so that it can't move is almost as good as taking it. And vice versa: you want to activate all your men if possible.
- Nubs are weak against long range bishops, they can't see them. Place a bishop to corner on the long diagonal and just snipe the opponent's material.
- Don't play "hope chess", always suppose your opponent will play the best move he can.
- If you can achieve something with multiple men, usually it's best to do it with the weakest one.
- TODO: moar

How To Disrespect Your Opponent And Other Lulz In Chess

see also [unsportmanship](#)

WORK IN PROGRESS, pls send me more tips :)

- OTB (over the board) only:
 - ♦ Turn your knights to face backwards or in another weird way (always face the opponent's king etc.). Also place the pieces unevenly on the squares to piss off opponents with OCD and autism.
 - ♦ Behave weird, make weird faces, walk extremely far away from the board and walk in circles (or just get up and stand up directly behind your opponent in a completely upright position staring into the distance without moving at all like a robot lol), constantly sneeze (try to sneeze every time the opponent touches a piece), make very long unbroken eye contact with the opponent while smiling as if you know what he's thinking, call the referee constantly, go to the toilet after every move, pretend to fall asleep from boredom etc. Overeat on beans before

the game so you fart a lot and always try to fart as loud as possible. Wear nice clothes but right before the game go sweat to the gym so that you smell like a pig and distract the opponent with toxic fume. If you're a wimmin behave sexually, keep grabbing your boobs, lick your lips and opponent's captured pieces and silently moan sometimes as if you're having an orgasm, pretend to masturbate under the table; if your opponent is male he is almost definitely smarter than you, you gotta use your woman weapons, but it will probably work easily on the chess virgins.

- ♦ In a tournament change play based on opponent's race or sex, for example play only one opening against white people and another opening against black people, see if anyone notices the pattern :D
- ♦ Outside tournament take advantage of the fact that you can do whatever the fuck you want: have one hand constantly on the clock and play with the other hand (considered rude and often forbidden), touch and knock over your opponent's pieces, take back your moves, ... and of course when you're losing, "accidentally" knock over the whole board and be like "oops, let's consider it a draw then" :D
- ♦ Trash talk the referee.
- ♦ Correct the opponent's pronunciation of *en passant*, insist it's pronounced "en peasant".
- ♦ ...
- online only:
 - ♦ Be annoying and offensive in chat, if opponent blunders write gg, spam ez when you win. If he wins say it was a shit game and accuse him of cheating.
 - ♦ Constantly ask for takebacks, offer draws, report legit opponents for cheating and offensive behavior.
 - ♦ ...
- Play the bongcloud, fool's mate, 1. h3 or similar offensive opening, especially against a stronger player. Offer a draw after 1st move. Just play knight F3 and back constantly. Castle manually even if you don't have to. Play the exact mirror of opponent's moves -- if he tries to break it then just always try to get back to mirrored position.
- When losing constantly offer draws, prolong the game AS MUCH AS POSSIBLE, before the very last move just let the clock run out.
- Repeatedly try to make swastikas on the board, especially against colored opponents.
- Underpromote pawns e.g. to knights or bishops.
- When playing a noob, don't just mate him but absolutely rape him, promote all pawns to knights before winning, then say you didn't even have to try and that he should look into another game as chess is clearly not his game.
- Look up chess etiquette and do the exact opposite of what it says.
- ...

LRS Chess

{ Has someone already made this tho? Seems like a pretty obvious simplification to make. ~drummyfish }

Chess is only mildly bloated but what if we try to unbloot it completely? Here we propose the LRS version of chess. The rule changes against normal chess are:

- No castling.
- No en passant.
- Promotion is always to queen.
- No checks or checkmates, king is just another man.
- Whoever takes the opponent's king first wins.
- If a player has no available moves, he loses.
- Only a single draw rule: if game doesn't end in 1024 half moves or fewer, it is a draw. I.e. there are no weird draw rules (50 move, repetition, ...). Of course players may still agree on draw anytime.
- Random: optionally random variant of LRS chess can be played. Here we randomly shuffle the white player's back row men in the starting position and mirror it for black (no weird conditions on men positions like in chess 960).

See Also

- [shogi](#)
 - [go](#)
 - [hexapawn](#)
 - [hex game](#)
 - [checkers](#)
 - [advance wars](#)
 - [backgammon](#)
 - [Catan](#)
 - [Deep Blue](#)
 - [stockfish](#)
 - [anal bead](#)
-

chinese

Chinese

Chinese is one of the most [bloated](#) natural human languages, spoken in China.

Any text in chinese basically looks like this:

```
#####  
#####  
#####  
#####
```

It is so bloated that some Chinese people literally don't understand other Chinese people.

cloud

"Cloud Computing"

Cloud is just someone else's computer.

Cloud computing, more accurately known as clown computing, means giving up an autonomous computer by storing one's data as well as running one's programs on someone else's (often a corporation's) [computer](#), known as *the cloud*, through the [Internet](#), becoming wholly [dependent](#) on *someone else* to which one gives all the power. While the general idea of [server computers](#) and remote [terminals](#) is not bad in itself and may be utilized in very good ways, the term *cloud computing* stands for abusing this idea e.g. by [capitalists](#) or states to take away autonomous computers from the people as well as to restrict freedoms of people in other ways, for example by pushing [DRM](#), making it impossible to truly own a copy of software or other data, to run computations privately, isolated from the Internet or run non-approved, [user-respecting](#) software. Moreover clown computing as applied nowadays is mostly a very bad engineering approach that wastes [bandwidth](#), introduces [lag](#), requires complex and expensive infrastructure etc.

Despite all this "cloud" is the mainstream nowadays, it is the way of computing among [normies](#), even despite regular leaks and losses of their personal data etc., simply because they're constantly being pushed to it by the [big tech](#) ([Apple](#), [Google](#), [Micro\\$oft](#), ...) -- many times they don't even have a choice, they are simply supposed to SHUT UP AND CONSUME. And of course they wouldn't even have an idea about what's going on in the first place, all that matters to a normie is ["comfort"](#), ["everyone does it"](#), "I just need my [TikTok](#)" etc. [Zoomers](#) probably aren't even aware of the cloud, they simply have phones with apps that show their photos if Apple approves of it, they don't even care how shit works anymore.

In the [future](#) non-cloud computers will most likely become illegal. This will be justified by autonomous computers being "dangerous", only needed by [terrorists](#), [pirates](#) and [pedophiles](#). An autonomous computer will be seen as a [gun](#), the right to own it will be greatly limited.

C

{ We have a C tutorial! ~drummyfish }

C is an old low level structured statically typed imperative compiled programming language, it is very fast and currently mostly used by less retarded software. Though by very strict standards it would still be considered bloated, compared to any mainstream modern language it is very bullshitless, KISS and greatly established and "culturally stable", so it is also the go-to language of the suckless community as well as most true experts, for example the Linux and OpenBSD developers, because of its good, relatively simple design, **uncontested performance, wide support**, great number of compilers, level of control and a greatly established and tested status. C is **perhaps the most important language in history**; it influenced, to smaller or greater degree, basically all of the widely used languages today such as C++, Java, JavaScript etc., however it is not a thing of the past -- in the area of low level programming C is still the number one unsurpassed language. C is by no means perfect but it is currently probably the best choice of a programming language (along with comun, of course). Though C is almost always compiled, there have appeared some C interpreters as well.

{ Look up *The Ten Commandments for C Programmers* by Henry Spencer. Also the *Write in C* song (parody of *Let it Be*). ~drummyfish }

It is usually **not considered an easy language to learn** because of its low level nature: it requires good understanding of how a computer actually works and doesn't prevent the programmer from shooting himself in the foot. Programmer is given full control (and therefore responsibility). There are things considered "tricky" which one must be aware of, such as undefined behavior of certain operators and raw pointers. This is what can discourage a lot of modern "coding monkeys" from choosing C, but it's also what inevitably allows such great performance -- undefined behavior allows the compiler to choose the most efficient implementation. On the other hand, C as a language is pretty simple without modern bullshit concepts such as QQP, it is not as much hard to learn but rather hard to master, as any other true art. In any case **you have to learn C** even if you don't plan to program in it regularly, it's the most important language in history and lingua franca of programming, you will meet C in many places and have to at least understand it: programmers very often use C instead of pseudocode to explain algorithms, C is used for optimizing critical parts even in non-C projects, many languages compile to C, it is just all around and you have to understand it like you have to understand English.

Some of the typical traits of C include great reliance on and utilization of preprocessor (macros, the underlying C code is infamously littered with "#ifdefs" all over the place which modify the code just before compiling -- this is mostly used for compile-time configuration and/or achieving better performance and/or for portability), pointers (direct access to memory, used e.g. for memory allocation, this is infamously related to "shooting oneself in the foot", e.g. by getting memory leaks) and a lot of **undefined behavior** (many things are purposefully left undefined in C to allow compilers to generate greatly efficient code, but this sometimes lead to weird bugs or a program working on one machine but not another, so C requires some knowledge of its specification). You can also infamously meet complicated type declarations like `void (*float(int,void (*n)(int)))(int)`, these are frequently a subject of jokes ("look, C is simple").

Unlike many "modern" languages, C by itself doesn't offer too much advanced functionality such as displaying graphics, working with network, getting keyboard state and so on -- the base language doesn't even have any input/output, it's a pure processor of values in memory. The standard library offers things like basic I/O with standard input/output streams, basic operations with files, strings, time, math functions and other things, but for anything more advanced you will need an external library like SDL or Posix libraries.

C is said to be a "**portable assembly**" because of its low level nature, great performance etc. -- though C is structured (has control structures such as branches and loops) and can be used in a relatively high level manner, it is also possible to write assembly-like code that operates directly with bytes in memory through pointers without many safety mechanisms, so C is often used for writing things like hardware drivers. On the other hand some restrain from likening C to assembly because C compilers still perform many transformations of the code and what you write is not necessarily always what you get.

Mainstream consensus acknowledges that C is among the best languages for writing low level code and code that requires **performance**, such as operating systems, drivers or games. Even scientific libraries with normie-language interfaces -- e.g. various machine learning Python libraries -- usually have the performance critical core written in C. Normies will tell you that for things outside this scope C is not a good language, with which we disagree -- we recommend using C for basically everything that's supposed to last, i.e. if you want to write a good website, you should write it in C etc.

Is C low or high level? This depends on the context. Firstly back in the day when most computers were programmed in assembly, C was seen as high level, simply because it offered the highest level of abstraction at the time, while nowadays with languages like Python and JavaScript around people see C as very low level by comparison -- so it really depends on if you talk about C in context of "old" or "modern" programming and which languages you compare it to. Secondly it also depends on HOW you program in C -- you may choose to imitate assembly programming in C a lot, avoid using libraries, touch hardware directly, avoid using complex features and creating your own abstractions -- here you are really doing low level programming. On the other hand you can emulate the "modern" high-level style programming in C too, you can even mimic OOP and make it kind of "C++ with different syntax", you may use libraries that allow you to easily work with strings, heavy macros that pimp the language to some spectacular abomination, you may write your own garbage collector etc. -- here you are basically doing high level programming in C.

Fun: `main[-1u]={1};` is a C compiler bomb :) it's a short program that usually makes the compiler produce a huge binary.

History and Context

C was developed in 1972 at Bell Labs alongside the Unix operating system by Dennis Ritchie and Brian Kernighan, as a successor to the B language (portable language with recursion) written by Denis Ritchie and Ken Thompson, which was in turn inspired by the the ALGOL language (code blocks, lexical scope, ...). C was for a while called NB for "new B". C was intimately interconnected with Unix and its hacker culture, both projects would continue to be developed together, influencing each other. In 1973 Unix was rewritten in C. In 1978 Keninghan and Ritchie published a book called *The C Programming Language*, known as *K&R*, which became something akin the C specification. In March 1987 Richard Stallman along with others released the first version of GNU C compiler -- the official compiler of the GNU project and the compiler that would go on to become one of the most widely used. In 1989, the ANSI C standard, also known as C89, was released by the American ANSI -- this is a very well supported and overall good standard. The same standard was also adopted a year later by the international ISO, so C90 refers to the same language. In 1999 ISO issues a new standard that's known as C99, still a very good standard embraced by LRS. Later in 2011 and 2017 the standard was revised again to C11 and C17, which are however no longer considered good.

Standards

C is not a single language, there have been a few standards over the years since its inception in 1970s. The notable standards and versions are:

- **K&R C:** C as described by its inventors in the book *The C Programming Language*, before official standardization. This is kind of too ancient nowadays.
- **C89/C90 (ANSI/ISO C):** First fully standardized version, usable even today, many hardcore C programmers stick to this version so as to enjoy maximum compiler support.
- **C95:** A minor update of the previous standard, adds wide character support.
- **C99:** Updated standard from the year 1999, striking a nice balance between "modern" and "good old". This is a good version to use in LRS programs, but will be a little less supported than C89, even though still very well supported. Notable new features against C89 include `//` comments, stdint library (fixed-width integer types), float and `long long` type, variable length stack-allocated arrays, variadic macros and declaration of variables "anywhere" (not just at function start).
- **C11:** Updated standard from the year 2011. This one is too bloated and isn't worth using.
- **C17/C18:** Yet another update, yet more bloated and not worth using anymore.
- ...

Quite nice online reference to all the different standards (including C++) is available at <https://en.cppreference.com/w/c/99>.

LRS should use C99 or C89 as the newer versions are considered bloat and don't have such great support in compilers, making them less portable and therefore less free.

The standards of C99 and older are considered pretty future-proof and using them will help your program be future-proof as well. This is to a high degree due to C having been established and tested better than any other language; it is one of the oldest languages and a majority of the most essential software is written in C, C compiler is one of the very first things a new hardware platform needs to implement, so C compilers will always be around, at least for historical reasons. C has also been very well designed in a relatively minimal fashion, before the advent of modern feature-creep and and bullshit such as OOP which cripples almost all "modern" languages.

Compilers

C is extreme well established, standardized and implemented so there is a great number of C compilers around. Let us list only some of the more notable ones.

- gcc: The main "big name" that can compile all kinds of languages including C, used by default in many places, very bloated and can take long to compile big programs, but is pretty good at optimizing the code and generating fast code. Also has number of frontends and can compile for many platforms. Uses GENERIC/GIMPLE intermediate representation.
- clang: Another big bloated compiler, kind of competes with gcc, is similarly good at optimization etc. Uses LLVM intermediate representation.
- tcc: Tiny C compiler, suckless, orders of magnitude smaller (currently around 25 KLOC) and simpler than gcc and clang, doesn't use any intermediate representation, cannot optimize nearly as well as the big compilers so the generated executables can be a bit slower and/or bigger (though sometimes they may be smaller), however besides its internal simplicity there are many advantages, mainly e.g. fast compilation (claims to be 9 times faster than gcc) and small tcc executable (about 100 kB). Seems to only support x86 at the moment.
- scc: Another small/suckless C compiler, currently about 30 KLOC.
- DuskCC: Dusk OS C compiler written in Forth, focused on extreme simplicity, probably won't adhere to standards completely.
- 8c, 8cc, chibicc: Some other small compilers.
- c2bf: Partially implemented C to brainfuck compiler.
- lcc: Proprietary, source available small C compiler, about 20 KLOC.
- pcc: A very early C compiler that was later developed further to support even the C99 standard.
- Borland Turbo C: old proprietary compiler with IDE.
- sdcc (small device C compiler): For small 8 bit microcontrollers.
- msvc (Micro\$oft visual C++): Badly bloated proprietary C/C++ compiler by a shitty corporation. Avoid.
- ...

Standard Library

Besides the pure C language the C standard specifies a set of libraries that have to come with a standard-compliant C implementation -- so called standard library. This includes e.g. the *stdio* library for performing standard input/output (reading/writing to/from screen/files) or the *math* library for mathematical functions. It is usually relatively okay to use these libraries as they are required by the standard to exist so the dependency they create is not as dangerous, however many C implementations aren't completely compliant with the standard and may come without the standard library. Also many stdlib implementations suck or you just can't be sure what the implementation will prefer (size? speed?) etc. So for sake of portability it is best if you can avoid using standard library.

The standard library (libc) is a subject of live debate because while its interface and behavior are given by the C standard, its implementation is a matter of each compiler; since the standard library is so commonly used, we should take great care in assuring it's extremely well written, however we ALWAYS have to choose our priorities and make tradeoffs, there just mathematically CANNOT be an ultimate implementation that will be all extremely fast and extremely memory efficient and extremely portable and extremely small. So choosing your C environment usually comprises of choosing the C compiler and the stdlib implementation. As you probably guessed, the popular implementations (glibc et al) are bloat and also often just shit. Better

alternatives thankfully exist, such as:

- musl
- uclibc
- not using the standard library :)
- ...

Good And Bad Things About C

Firstly let's sum up some of the reasons why C is so good:

- **C as a language is relatively simple:** Though strictly speaking it's not in the league of most minimal languages like Forth and Lisp, C is the next best thing in terms of minimalism and the small amount of bloat it contains is usually somehow justified at least, the language (or its subset) can be implemented in a quite minimal way if one so desires. It employs little abstraction. This all helps performance, freedom and encourages many implementations. C's standard library also isn't gigantic, the important parts basically just provide I/O and help with simple things like manipulating strings and memory allocation, so new C implementations aren't burdened by having to implement tons of libraries.
- **It is extremely fast and efficient:** Owing to other mentioned points such as good specification, simplicity, lack of bullshit and having a good balance between low and high level attributes, C is known for being possibly the fastest portable language in existence, also greatly efficient with memory etc.
- **C doesn't limit you or hold (tie) your hands:** This is bad for the beginner but great for the expert, most of the times C won't "protect" you from doing anything, even crashing your program -- this kind of freedom is necessary to achieve truly marvelous things, C is like a race car, it doesn't have speed limiters and automatic transmission, nothing that would tie your hands or increase the car weight, it trusts in you being a good driver.
- **C is highly standardized:** Many languages have some kind of "online specification", however C is on the next level by literally being officially standardized by the forefront standardizing organizations like ANSI and ISO, by full time paid experts over many years and iterations, so the language is extremely well defined and described, down to saying which exact things are left undefined/unspecified, leaving freedom of implementation that leads to the language's great performance.
- **It's extremely well establishes, optimized, stable and time tested, with many tools:** Being among the oldest languages, the language of the old time hackers and the language of Unix, maybe the most important piece of software in history, C has been so widely adopted, reimplemented, optimized and tested over and over that it's considered to be among the most essential pieces of software any platform has to have. Everything on the low level is written in C, so you essentially first have to have C to be able to run anything else. Many companies have invested great many resources to making C fast as it benefited them. While other languages come and go, or at least mutate and become something else over time, C stands as one of very few stable things in computer technology. There are also tons and tons of tools that help with C development, various static analyzers, debuggers, code beautifiers etcetc.
- **It doesn't have any modern bullshit:** There is no OOP, generics, garbage collection, no package manager etc.
- **There is a huge number of compilers:** While a "modern" language has some kind of main reference implementation and then maybe one of two alternative implementations, C has dozens (maybe even hundreds) of compilers. You'll find compilers under all the licenses, huge ones with many features and great optimizations, small ones that will run on tiny devices, ones that compile very fast, ones that translate C to other languages etcetc.
- **It is elitist:** The relatively higher difficulty of learning the language has a nice effect of keeping idiots out of its community, keeping the language less intoxicated by retarded ideas. { NOTE: The word "elitist" here is not to really mean inherently "discriminating" of course, but rather "unpopular" because it's quite different from the mainstream and requires some effort on unlearning bad mainstream habits, i.e. nowadays it needs some dedication, you can't just join in effortlessly. It's elitist in the same way in which Unix systems or suckless software are elitist. ~drummyfish }
- **C is close to the hardware, reflecting how computers work:** This has many advantages: firstly efficiency, as code that maps well to hardware is predictable and efficient, lacking magic in translation. It simplifies implementations, making the language more free. Then also the programmer himself is close to the machine, he has to learn how it works, what it likes and dislikes -- a knowledge

every programmer has to have.

- **There is a great balance between low and high level (minimalism vs "features"):** C seems to have hit a sweet spot at which it offers just enough high level features for comfortable programming, such as data types, functions and expressions, while not crossing the line beyond which it would have to pay a high cost for this comfort, i.e. it managed to buy us a lot practically for free. Things like this cannot really be planned well, it takes a genius and intuition to design a language this way, this shows the greatness of the old master programmers.
- **It is old, written only by white male hackers, at times when capitalism was weaker:** No women were probably involved in the development (of course we aren't racists or sexists, it's just a fact that white men are best at programming), the development was largely part of genuine research, at the time when computers weren't mainstream and computer technology wasn't being raped as hard as today. C developers didn't even think of embedding any political message in the language. Times like this will never be repeated.
- ...

Now let's admit that nothing is perfect, not even C; it was one of the first relatively higher level languages and even though it has showed to have been designed extremely well, some things didn't age great, or were simply bad from the start. We still prefer this language as usually the best choice, but it's good to be aware of its downsides or smaller issues, if only for the sake of one day designing a better language. Please bear in mind all here are just suggestions, they made of course be a subject to counter arguments and further discussion. Here are some of the **bad things** about the language:

- **C specification (the ISO standard) is proprietary** :(The language itself probably can't be copyrighted, nevertheless this may change in the future, and a proprietary specs lowers C's accessibility and moddability (you can't make derivative versions of the spec).
- **The specification is also long as fuck** (approx. 500 pages, out of that 163 of the pure language), indicating bloat/complexity/obscurity. A good, free language should have a simple definition. It could be simplified a lot by simplifying the language itself as well as dropping some truly legacy considerations (like BCD systems?) and removing a lot of undefined behavior.
- **Some behavior is weird and has unnecessary exceptions**, for example a function can return anything, including a struct, except for an array. This makes it awkward to e.g. implement vectors which would best be made as arrays but you want functions to return them, so you may do hacks like wrapping them inside a struct just for this.
- **Some things could be made simpler**, e.g. using reverse polish notation for expressions, rather than expressions with brackets and operator precedence, would make implementations much simpler, increasing sucklessness (of course readability is an argument).
- **Some things could be dropped entirely** (enums, bitfields, possibly also unions etc.), they can be done and imitated in other ways without much hassle.
- **The preprocessor isn't exactly elegant**, it has completely different syntax and rules from the main language, not very suckless -- ideally preprocessor uses the same language as the base language.
- **The syntax is sucky sometimes**, e.g. case with variable inside it HAS TO be enclosed in curly brackets but other ones don't, data type names may consist of multiple tokens (long long int etc.), multiplication uses the same symbol as pointer dereference (*), many preprocessor commands need to be on separate lines (makes some one liners impossible), also it's pretty weird that the condition after if has to be in brackets etc., it could all be designed better. Keywords also might be better being single chars, like ? instead of if etc. (see comun). A shorter source code that doesn't try to imitate English would be probably better.
- **Some undefined/unspecified behavior is probably unnecessary** -- undefined behavior isn't bad in general of course, it is what allows C to be so fast and efficient in the first place, but some of it has shown to be rather cumbersome; for example the unspecified representation of integers, their binary size and behavior of floats leads to a lot of trouble (unknown upper bounds, sizes, dangerous and unpredictable behavior of many operators, difficult testing etc.) while practically all computers have settled on using 8 bit bytes, two's complement and IEEE754 for floats -- this could easily be made a mandatory assumption which would simplify great many things without doing basically any harm. New versions of C actually already settle on two's complement. This doesn't mean C should be shaped to reflect the degenerate "modern" trends in programming though!
- Some basic things that are part of libraries or extensions, like fixed width types and binary literals and possibly very basic I/O (putchar/readchar), could be part of the language itself rather than provided by libraries.

- All that stuff with .c and .h files is unnecessary, there should just be one file type probably.
- ...

Basics

This is a quick overview, for a more in depth tutorial see [C tutorial](#).

A simple program in C that writes "welcome to C" looks like this:

```
#include <stdio.h> // standard I/O library

int main(void)
{
    // this is the main program

    puts("welcome to C");

    return 0; // end with success
}
```

You can simply paste this code into a file which you name e.g. program.c, then you can compile the program from command line like this:

```
gcc -o program program.c
```

Then if you run the program from command line (./program on Unix like systems) you should see the message.

Cheatsheet/Overview

Here is a quick reference cheatsheet of some of the important things in C, also a possible overview of the language.

data types (just some):

data type	values (size)	printf	notes
int (signed int, ...)	integer, at least -32767 to 32767 (16 bit), often more	%d	native integer, fast (prefer for speed)
unsigned int	integer, non-negative, at least 0 to 65535, often more	%u	same as int but no negative values
signed char	integer, at least -127 to 127, mostly -128 to 127	%c, %hhi	char forced to be signed
unsigned char	integer, at least 0 to 255 (almost always the case)	%c, %hhu	smallest memory chunk, byte
char	integer, at least 256 values	%c	signed or unsigned, used for string characters
short	integer, at least -32767 to 32767 (16 bit)	%hd	like int but supposed to be smaller
unsigned short	integer, non-negative, at least 0 to 65535	%hu	like short but unsigned
long	integer, at least -2147483647 to 2147483647 (32 bit)	%ld	for big signed values
unsigned long	integer, at least 0 to 4294967295 (32 bit)	%lu	for big unsigned values
long long	integer, at least some $-9 * 10^{18}$ to $9 * 10^{18}$ (64 bit)	%lld	for very big signed values
unsigned long long	integer, at least 0 to 18446744073709551615 (64 bit)	%llu	for very big unsigned values
float		%f	

data type	values (size)	printf	notes
	floating point, some $-3 * 10^{38}$ to $3 * 10^{38}$		<u>float</u> , tricky, bloat, can be slow, avoid
double	floating point, some $-1 * 10^{308}$ to 10^{308}	%lf	like float but bigger
T [N]	array of N values of type T		array , if T is char then string
T *	memory address	%p	pointer to type T, (if char then string)
uint8_t	0 to 255 (8 bit)	PRIu8	exact width, two's compl., must include <stdint.h>
int8_t	-128 to 127 (8 bit)	PRId8	like uint8_t but signed
uint16_t	0 to 65535 (16 bit)	PRIu16	like uint8_t but 16 bit
int16_t	-32768 to 32767 (16 bit)	PRId16	like uint16_t but signed
uint32_t	-2147483648 to 2147483647 (32 bit)	PRIu32	like uint8_t but 32 bit
int32_t	0 to 4294967295 (32 bit)	PRId32	like uint32_t but signed
int_least8_t	at least -128 to 127	PRIdLEAST8	signed integer with at least 8 bits, <stdint.h>
int_fast8_t	at least -128 to 127	PRIdFAST8	fast signed int. with at least 8 bits, <stdint.h>
struct			structured data type

There is no **bool** (true, false), use any integer type, 0 is false, everything else is true (there may be some bool type in the stdlib, don't use that). A **string** is just array of chars, it has to end with value 0 (NOT ASCII character for "0" but literally integer value 0)!

main program structure:

```
#include <stdio.h>

int main(void)
{
    // code here
    return 0;
}
```

branching aka if-then-else:

```
if (CONDITION)
{
    // do something here
}
else // optional
{
    // do something else here
}
```

for loop (repeat given number of times):

```
for (int i = 0; i < MAX; ++i)
{
    // do something here, you can use i
}
```

while loop (repeat while CONDITION holds):

```
while (CONDITION)
{
    // do something here
}
```

do while loop (same as *while* but CONDITION at the end), not used that much:

```
do
{
    // do something here
} while (CONDITION);
```

function definition:

```
RETURN_TYPE myFunction (TYPE1 param1, TYPE2 param2, ...)
{ // return type can be void
    // do something here
}
```

compilation (you can replace gcc with another compiler):

- quickly compile and run: `gcc myprogram.c && ./a.out.`
- compile more properly: `gcc -std=c99 -Wall -Wextra -pedantic -O3 -o myprogram myprogram.c.`

To **link** a library use `-llibrary`, e.g. `-lm` (when using `<math.h>`), `-lSDL2` etc.

The following are some symbols (functions, macros, ...) from the **standard library**:

symbol	library	description	example
<code>putchar(c)</code>	<code>stdio.h</code>	Writes a single character to output.	<code>putchar('a');</code>
<code>getchar()</code>	<code>stdio.h</code>	Reads a single character from input.	<code>int inputChar = getchar();</code>
<code>puts(s)</code>	<code>stdio.h</code>	Writes string to output (adds newline at the end).	<code>puts("hello");</code>
<code>printf(s, a, b, ...)</code>	<code>stdio.h</code>	Complex print func., allow printing numbers, their formatting etc.	<code>printf("value is %d\n",var);</code>
<code>scanf(s, a, b, ...)</code>	<code>stdio.h</code>	Complex reading func., allows reading numbers etc.	<code>scanf("%d",&var);</code>
<code>fopen(f,mode)</code>	<code>stdio.h</code>	Opens file with given name in specific mode, returns pointer.	<code>FILE *myFile = fopen("myfile.txt","r");</code>
<code>fclose(f)</code>	<code>stdio.h</code>	Closes previously opened file.	<code>fclose(myFile);</code>
<code>fputc(c,f)</code>	<code>stdio.h</code>	Writes a single character to file.	<code>fputc('a',myFile);</code>
<code>fgetc(f)</code>	<code>stdio.h</code>	Reads a single character from file.	<code>int fileChar = fgetc(myFile);</code>
<code>fputs(s,f)</code>	<code>stdio.h</code>	Writes string to file (without newline at end).	<code>fputs("hello",myFile);</code>
<code>fprintf(s, a, b, ...)</code>	<code>stdio.h</code>	Like <code>printf</code> but outputs to a file.	<code>fprintf(myFile,"value is %d\n",var);</code>
<code>fscanf(f, s, a, b, ...)</code>	<code>stdio.h</code>	Like <code>scanf</code> but reads from a file.	<code>fscanf(myFile,"%d",&var);</code>
<code>fread(data,size,n,f)</code>	<code>stdio.h</code>	Reads <i>n</i> elems to <i>data</i> from <i>file</i> , returns no. of elems read.	<code>fread(myArray,sizeof(item),1,myFile);</code>
<code>fwrite(data,size,n,f)</code>	<code>stdio.h</code>	Writes <i>n</i> elems from <i>data</i> to <i>file</i> , returns no. of elems writ.	<code>fwrite(myArray,sizeof(item),1,myFile);</code>
<code>EOF</code>	<code>stdio.h</code>	<u>End of file</u> value.	<code>int c = getchar(); if (c == EOF) break;</code>
<code>rand()</code>	<code>stdlib.h</code>		<code>char randomLetter = 'a' + rand() % 26;</code>

symbol	library	description	example
		Returns pseudorandom number.	
<i>srand(n)</i>	<i>stdlib.h</i>	Seeds pseudorandom number generator.	<code>srand(time(NULL));</code>
<i>NULL</i>	<i>stdlib.h</i> , ...	Value assigned to pointers that point "nowhere".	<code>int *myPointer = NULL;</code>
<i>malloc(size)</i>	<i>stdlib.h</i>	Dynamically allocates memory, returns pointer to it (or <i>NULL</i>).	<code>int *myArr = malloc(sizeof(int) * 10);</code>
<i>realloc(mem,size)</i>	<i>stdlib.h</i>	Resizes dynamically allocates memory, returns pointer (or <i>NULL</i>).	<code>myArr = realloc(myArr,sizeof(int) * 20);</code>
<i>free(mem)</i>	<i>stdlib.h</i>	Frees dynamically allocated memory.	<code>free(myArr);</code>
<i>atof(str)</i>	<i>stdlib.h</i>	Converts string to floating point number.	<code>double val = atof(answerStr);</code>
<i>atoi(str)</i>	<i>stdlib.h</i>	Converts string to integer number.	<code>int val = atoi(answerStr);</code>
<i>EXIT_SUCCESS</i>	<i>stdlib.h</i>	Value the program should return on successful exit.	<code>return EXIT_SUCCESS;</code>
<i>EXIT_FAILURE</i>	<i>stdlib.h</i>	Value the program should return on exit with error.	<code>return EXIT_FAILURE;</code>
<i>sin(x)</i>	<i>math.h</i>	Returns <u>sine</u> of angle in <u>RADIANS</u> .	<code>float angleSin = sin(angle);</code>
<i>cos(x)</i>	<i>math.h</i>	Like <i>sin</i> but returns cosine.	<code>float angleCos = cos(angle);</code>
<i>tan(x)</i>	<i>math.h</i>	Returns <u>tangent</u> of angle in <u>RADIANS</u> .	<code>float angleTan = tan(angle);</code>
<i>asin(x)</i>	<i>math.h</i>	Returns arcus sine of angle, in <u>RADIANS</u> .	<code>float angle = asin(angleSine);</code>
<i>ceil(x)</i>	<i>math.h</i>	Rounds a floating point value up.	<code>double x = ceil(y);</code>
<i>floor(x)</i>	<i>math.h</i>	Rounds a floating point value down.	<code>double x = floor(y);</code>
<i>fmod(a,b)</i>	<i>math.h</i>	Returns floating point reminded after division.	<code>double rem = fmod(x,3.5);</code>
<i>isnan(x)</i>	<i>math.h</i>	Checks if given float value is NaN.	<code>if (!isnan(x))</code>
<i>NAN</i>	<i>math.h</i>	Float quiet <u>NaN</u> (not a number) value, don't compare!	<code>if (y == 0) return NAN;</code>
<i>log(x)</i>	<i>math.h</i>	Computes natural <u>logarithm</u> (base <u>e</u>).	<code>double x = log(y);</code>
<i>log10(x)</i>	<i>math.h</i>	Computes decadic <u>logarithm</u> (base 10).	<code>double x = log10(y);</code>
<i>log2(x)</i>	<i>math.h</i>	Computes binary <u>logarithm</u> (base 2).	<code>double x = log2(y);</code>
<i>exp(x)</i>	<i>math.h</i>	Computes exponential function (e^x).	<code>double x = exp(y);</code>
<i>sqrt(x)</i>	<i>math.h</i>	Computes floating point <u>square root</u> .	<code>double dist = sqrt(dx * dx + dy * dy);</code>
<i>pow(a,b)</i>	<i>math.h</i>		<code>double cubeRoot = pow(var,1.0/3.0);</code>

symbol	library	description	example
		Power, raises <i>a</i> to <i>b</i> (both floating point).	
<i>abs(x)</i>	<i>math.h</i>	Computes <u>absolute value</u> .	<code>double varAbs = abs(var);</code>
<i>INT_MAX</i>	<i>limits.h</i>	Maximum value that can be stored in int type.	<code>printf("int max: %d\n",INT_MAX);</code>
<i>memset(mem,val,size)</i>	<i>string.h</i>	Fills block of memory with given values.	<code>memset(myArr,0,sizeof(myArr));</code>
<i>memcpy(dest,src,size)</i>	<i>string.h</i>	Copies bytes of memory from one place to another, returns dest.	<code>memcpy(destArr,srcArr,sizeof(srcArr);</code>
<i>strcpy(dest,src)</i>	<i>string.h</i>	Copies string (zero terminated) to dest, unsafe.	<code>char myStr[16]; strcpy(myStr,"hello");</code>
<i>strncpy(dest,src,n)</i>	<i>string.h</i>	Like strcpy but limits max number of bytes to copy, safer.	<code>strncpy(destStr,srcStr,sizeof(destStr));</code>
<i>strcmp(s1,s2)</i>	<i>string.h</i>	Compares two strings, returns 0 if equal.	<code>if (!strcmp(str1,"something"))</code>
<i>strlen(str)</i>	<i>string.h</i>	Returns length of given string.	<code>int l = strlen(myStr);</code>
<i>strstr(str,substr)</i>	<i>string.h</i>	Finds substring in string, returns pointer to it (or NULL).	<code>if (strstr(cmdStr,"quit") != NULL)</code>
<i>time(t)</i>	<i>time.h</i>	Stores calendar time (often Unix t.) in t (can be NULL), returns it.	<code>printf("tstamp: %d\n",(int) time(NULL));</code>
<i>clock()</i>	<i>time.h</i>	Returns approx. CPU cycle count since program start.	<code>printf("CPU ticks: %d\n",(int) clock());</code>
<i>CLOCKS_PER_SEC</i>	<i>time.h</i>	Number of CPU ticks per second.	<code>int sElapsed = clock() / CLOCKS_PER_SEC;</code>

Some Programs In C

TODO

See Also

- [B](#)
- [D](#)
- [comun](#)
- [C tutorial](#)
- [C pitfalls](#)
- [C programming style](#)
- [C++](#)
- [IOCCC](#)
- [HolyC](#)
- [QuakeC](#)
- [Pascal](#)
- [Fortran](#)
- [LISP](#)
- [FORTH](#)
- [memory management](#) in C

coc

Code of Conduct

COC is like a tiny little guillotine of the "open \$ource revolution".

Code of conduct (COC), also *code of coercion* or *code of censorship*, is a shitty invention of SIW fascists that's put up in projects (e.g. software) and which declares how developers of a specific project must behave socially (typically NOT just within the context of the development but also outside of it), generally pushing toxic woke concepts such as forced inclusivity, exclusivity of people with unapproved political opinions or use of politically correct language (newspeak). Sometimes a toxic COC hides under a different name such as *social contract* or *mission statement*, though not necessarily. COC is typically placed in the project repository as a CODE_OF_CONDUCT file. In practice COCs are used to establish dictatorship and allow things such as kicking people out of development because of their political opinions expressed anywhere, inside or outside the project, and to push political opinions through software projects. COCs are an indication of tranny software. See also <http://techrights.org/2019/04/23/code-of-coercion/>.

COCs are extremely controversial and opposed by many, for example Alexandre Oliva, one of the top people at Free Software Foundation, the maintainer of Linux libre who was a serious candidate of the FSF president, himself identifying as "neurodivergent" and being rather politically correct, has expressed extreme criticism (here) of forcing a COC into GNU Binutils. He criticized not only forceful push of the decision to put a COC (they basically just announced it without asking if people agree with it), but also stated very true observations such as that "[the power of] enforcement and exclusion tends to attract people with authoritarian leanings" and that he himself feels "vulnerable and unsafe" -- again, someone who basically plays by the pseudoleftist rules. Expanding on the guillotine comparison, Oliva knows that the French revolution (and all other revolutions) devoured many of its children, and though he probably feels part of this revolution, he still criticized the guillotine.

{ Also here's some site: <https://nocodeofconduct.com/>, tho it's not shitting on it hard enough. ~drummyfish }

LRS must never employ any COC, with possible exceptions of anti-COC (such as NO COC) or parody style COCs, not because we dislike genuine inclusivity, but because we believe COCs are bullshit and mostly harmful as they support bullying, censorship and exclusion of people.

Anyway it's best to avoid any kind of COC file in the repository, it just takes up space and doesn't serve anything. We may simply ignore this shitty concept completely. You may argue why we don't ignore e.g. copyright in the same way and just not use any licenses? The situation with copyright is different: it exists by default, without a license file the code is proprietary and our neighbors don't have the legal safety to execute basic freedoms, they may be bullied by the state -- for this we are forced to include a license file to get rid of copyright. With COC there simply isn't any such implicit issues to be solved (because COCs are simply inventing their own issues), so we just don't try to solve non-issues.

Should you avoid software with COC? Well, yes if possible, though it's very difficult nowadays. A COCed software is not really legally non-free, you can really fork such project and just delete the COC file, there is no problem in that; the issue is not the file itself but more what it signifies -- it indicates a very toxic community around the project and often also harmful properties of the projects which is most likely tranny software and therefore loat using shit languages like Rust etc. Even if you can legally take a copy of the software from the toxic community and "make it your own", strip it off their propaganda, you probably won't rewrite it from scratch. Even if you do something with the software you are legally allowed to do, the community may bully you online because well, they are just toxic. COC is just a sign of trouble to come and a smell of bad technology. Rather seek for something better.

coding

Coding

Coding nowadays means low quality attempt at programming, usually practiced by soydevs and barely qualified coding monkeys. Coder is to programmer what a bricklayer is to architect.

Traditionally it means encoding and decoding of information as in e.g. video coding -- this is the only non-gay

collapse

Collapse

Collapse of our civilization is a concerning scenario in which basic structures of society relatively rapidly fall apart and cause unusually large, possibly world-wide horrors such as chaos, wars, famine and loss of advanced technology. It is something that will very likely happen very soon due to uncontrolled growth and societal decline by capitalism: we, the LRS, are especially focusing on a very probable **technological collapse** (caused by badly designed technology as well as its wrong application and extreme overuse causing dangerous dependencies) but of course clues point to collapse are coming from many directions (ecological, economical, political, natural disasters such as a coronal mass ejection etc.). Some have said that a society can deal with one crisis, but if multiple crises hit at once this hit may be fatal; however the dependence of current society on computer technology is so great that its collapse could be enough to deliver a fatal blow alone. Recently (around 2015) there has even appeared a specific term **collapsology** (also *collapse informatics* etc.) referring to the study of the potential collapse.

There is a reddit community for discussing the collapse at <https://reddit.net/r/collapse>. WikiWikiWeb has a related discussion under *ExtinctionOfHumanity*.

Collapse of civilizations has been a repeated theme throughout history, it is nothing new or exceptional, see e.g. Maya empire collapse, Bronze age collapse, the fall of Rome etc. It usually comes when a civilization reaches high complexity and becomes "spoiled", morally corrupt and socially divided, which may also be "helped" by technological advancement (e.g. the Bronze age collapse is speculated to have been partially caused by the new technology of iron which has broken the old established structures) -- basically just what we are seeing today. Economic interdependence is especially dangerous, and since we are currently living under extreme form of capitalism, we are extremely subjected to this threat: everyone is hyperspecialized and so practically no one is self sufficient, people don't know how to make food, build homes, make tools, factories are unable to produce anything without dozens of other companies providing material, technology and services for them; everything depends on highly complex and extremely fragile production chains. Besides dependence on economy, an equal or even greater danger may be our **absolute dependence on computer technology**: nothing works without computers and Internet, even that which could and should: factories, traffic, governments, hospitals, even many basic tools ("smart" ones, "tools as a service", ...). It is extremely likely we'll sooner or later sustain a blow that will paralyze the Internet and computers, be it a natural disaster such as coronal mass ejection, an economic disaster (supply chains collapsing, ...), political disaster (war, cyber attacks, ...), unintentional or intentional "accident" ("oops, I just turned off all computers in the world that are running Windows" --Microsoft employee), simple unsustainability of maintenance of the exponentially growing complexity of computers or anything similar (AI taking over the internet? :)). Thinking about it deeper, it seems like a miracle we are still here.

In **technological world** more and more people are concerned about the collapse, notably e.g. the Collapse OS/Dusk OS, operating systems meant to run on simple hardware after the technological supply chain collapses and renders development of modern computers impossible. Collapse technology has its specific focus and areas of interest, for example bootstrapping (a kind of self containment that allows the technology to set itself up), self hosting, simplicity, repairability, durability, offline work, low power consumption, usage of improvised parts (see e.g. salvage computing). Collpase OS website predicted the collapse would happen before 2030. The chip shortage, financial, climatic and energetic crisis and beginning of war in Europe as of early 2020s are one of the first warnings showing how fragile the systems really is. { I also believe it to be most probable the collapse will come before 2030, but of course there is still a good chance it will be another decade or two late, this is impossible to predict with accuracy. In any case we have to get ready as soon as possible. ~drummyfish }

Ted Kaczynski (a famous primitivist mathematician that committed mass murderer to warn about the decline of society due to complex technology) has seen the collapse as a possible option. Internet bloggers/vloggers such as Luke Smith and *no phone man* advocate (and practice) simple, independent off-grid living, possibly to be prepared for such an event. Even proprietary normies like Jonathan Blow warn of a coming disaster (in his talk *Preventing the Collapse of Civilization*). Viznut is another programmer warning about the collapse.

The details of the collapse cannot of course be predicted exactly -- it may come in a relatively quick, violent form (e.g. in case of a disaster causing a blackout) or as a more agonizing slow death. CollapseOS site talks about two stages of the slow collapse: the first one after the collapse of the supply chain. i.e. when the production of modern computers halts, and the second (decades after) when the last modern computer stops working. It most likely won't happen overnight -- that's a very extreme case. A typical collapse may take decades during which all aspects of society see a rapid decline. Of course, a collapse doesn't mean extinction of humans either, just deaths of many and great losses of what has been achieved culturally and technologically.

There also appeared a new area of study, so called **salvage computing**, which instead of trying to find ways to design new technology better (without discouraging this of course) rather focuses on making use on what's already been produced, i.e. even potentially "bad" technology which is already around and just became artificially obsolete. This is not in conflict with trying to design new and better technology, just additionally trying to maximize the use of what's already there.

{ I've read a book called *Blackout* by Marc Elsberg whose story revolves around a fictional large collapse of power supply in Europe. A book called *The World Without Us* explores what the world would look like if humans suddenly disappeared. Also the podcast called *Fall of Civilizations* by Paul Cooper is awesome.
~drummyfish }

Live Documenting The Current Collapse Of Our Civilization

This section will record the current in-progress collapse of our civilization as seen by me, drummyfish. It may serve later generations and historians, as I know any information left behind is important for avoiding repeating the mistakes, though I also know you will not learn and will repeat the mistakes anyway. A man can dream I guess. If you can take away just one thing though, please be it this: **FOR THE LOVE OF GOD, DO NOT EVER ALLOW COMPETITION TO BE THE BASE OF SOCIETY AGAIN.**

Late 2022 Report

It seems like the collapse may have already begun. After the worldwide Covid pandemic the Russia-Ukraine war has begun with talks of nuclear war already going on. A great economic crisis has begun, possibly as a result of the pandemic and the war, inflation is skyrocketing and breaking all records, especially gas and energy prices are growing to extremes and as a result basically prices of everything go up as well. Russia isolated itself, new cold war has begun. Many big banks have gone bankrupt. War immigrants from Ukraine are flooding into Europe and European fascists/nationalists seem to be losing their patience about it. People in European first world countries are now actually concerned about how not to freeze during the winter, this talk is all over TV and radio. The climate disaster has also started to show, e.g. in Czech Republic there was the greatest forest fire in its history as well an extremely hot summer, even tornados that destroyed some villages (tornados in this part of world are basically unheard of), winters have almost no snow unlike some two decades ago. Everything is shitty, food costs more and is of much lower quality as basically everything else, newly bought technology cannot be expected to last longer than a few months. Society is spoiled to an unimaginable level, extreme hostility, competition and aggressive commerce is everywhere, kids are addicted to cellphones and toxic social media, mental health of population rapidly deteriorates. Art such as movies and music is of extremely low quality, people hate every single new movie or video game that comes out. A neofascist party has won elections in Italy, in Czech Republic all socialist parties were eliminated from the parliament: only capitalists rule now -- all social securities are being cancelled, people are getting poorer and poorer and forced to work more and to much higher ages. Ads are everywhere and equate psychological torture. The situation now definitely seems extremely bad.

Late 2023 Report

Yep, the collapse is happening. All previously mentioned issues just deepen, though I stopped watching the news and just avoid the negative info to enjoy the last few years I have on this Earth without much stress. Russian-Ukraine war is still happily ongoing (despite all the predictions that Russia will soon run out of resources lol) AND there is a brand new war in Israel, new immigrants are gonna flood Europe, then USA is probably gonna invade the weakened countries or something. AI is currently breaking the Internet, Google became absolutely unusable which became noticeable even by normies now, it is just flooded by AI articles, news are an endless pile of AI generated nonsense. Technology is yet much worse than before, NOTHING

BUYIT BUY IT BUUUUUUUUUUY CONSOOOOOOOOOOOME THIIIII
CONSOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOMMMMMMMMMMMMMMME). Watching it
for 5 minutes literally makes you kill yourself. Hmm what else. NOTHING WORKS lol, you buy something, it is
already broken, you pay for repair, they take money and return it unrepaired LMAO :D I am not kidding, it is
literally the norm now, my bosses business is sinking due to all machines just breaking. Absolutely
unqualified people now do all the jobs lol, there are literally teachers who CANNOT READ OR WRITE correctly,
they just use Google to check how everything's spelled (I am NOT kidding, I have this from first hand
sources). "Programmers" literally can't program, they just use AI. People are generally braindead, controlled
by religions such as economy worship, productivity cult, women fascism (it has been officially declared now
that women are physically stronger than man AND also smarter LMAO, every movie is obliged by law to
include a scene where this is confirmed). I dunno man, this can't last much longer than a few years.

See Also

- capitalist singularity

collision_detection

Collision Detection

Collision detection is an essential problem e.g. of simulating physics of mechanical bodies in physics engines (but also elsewhere), it tries to detect whether (and also how) geometric shapes overlap. Here we'll be talking about the collision detection in physics engines, but the problem appears in other contexts too (e.g. frustum culling in computer graphics). Collision detection potentially leads to so called *collision resolution*, a different stage that tries to deal with the detected collision (separate the bodies, update their velocities, make them "bounce off"). Physics engines are mostly divided into 2D and 3D ones so we also normally either talk about 2D or 3D collision detection (3D being, of course, a bit more complex).

There are two main types of collision detection:

- **discrete**: Detecting collisions only at one point in time (each engine tick or "frame") -- this is easier but can result in detecting the collisions in wrong ways or missing them completely (imagine a fast flying object that in one moment is wholly in front of a wall and at the next instant wholly behind it). Nevertheless this is completely usable, one just has to be careful enough about the extreme cases.
- **continuous**: Detecting collisions considering the continuous motion of the bodies (still done at discrete ticks but considering the whole motion since the last tick) -- this is more difficult to program and more costly to compute, but also correctly detects collisions even in extreme cases. Sometimes engines perform discrete detection by default and use continuous detection in special cases (e.g. when speeds become very high or in other error-prone situations). Continuous detection can be imagined as a collision detection of a higher dimensional bodies where the added dimension is time -- e.g. detecting collisions of 2D circles becomes detecting collisions of "tubes" in 3D space. If you don't want to go all the way to implementing continuous collisions, you may consider an in-between solution by detecting collisions in smaller steps (which may also be done only sometimes, e.g. only for high speed bodies or only when an actual discrete collision is detected).

Collision detection is non-trivial and in many cases really hard because we need to detect NOT JUST the presence of the collision but also its parameters which are typically the exact **point of collision, collision depth and collision normal** (but potentially also other ones, depending on what we're doing, e.g. volume of overlap etc.) -- these are needed for subsequently resolving the collision (typically the bodies will be shifted along the normal by the collision depth to become separated and impulses will be applied at the collision point to update their velocities). We also need to detect **general cases**, i.e. not just collisions of surfaces but of WHOLE VOLUMES (imagine e.g. a tiny cuboid inside an arbitrarily rotated bigger cone). This is very hard and/or expensive for some complex shapes such as general 3D triangle meshes (which is why we approximate them with simpler shapes), but even for relatively simple shapes like arbitrarily rotated 3D

boxes the solution is not easy. We also want the detection algorithm to be at least reasonably **fast** -- for this reason collision detection mostly happens in two phases:

- **broad phase:** Quickly estimates which bodies MAY collide, usually with bounding volumes (such as spheres or axis aligned bounding boxes) or space indexing and algorithms such as sweep and prune. This phase quickly opts-out of checking collision of objects that definitely CANNOT collide because they're e.g. too far away.
- **narrow phase:** Applying the precise, expensive collision detection on the potentially colliding pairs of bodies determined in the broad phase. This yields the real collisions.

In many cases it is also important to correctly detect the **order of collisions** in time -- it may well happen a body collides not with one but with multiple bodies at the time of collision detection and the computed behavior may vary widely depending on the order in which we consider them. Imagine that body A is colliding with body B and body C at the same time; in real life A may have first collided with B and be deflected so that it would have never hit C, or the other way around, or it might have collided with both. In continuous collision detection we know the order as we also have exact time coordinate of each collision (even though the detection itself is still computed at discrete time steps), i.e. we know which one happened first. With discrete collisions we may use heuristics such as the direction in which the bodies are moving, but this may fail in certain cases (consider e.g. collisions due to rotations).

On shapes: general rule is that **mathematically simpler shapes are better for collision detection**. Spheres (or circles in 2D) are the best, they are stupidly simple -- a collision of two spheres is simply decided by their distance (i.e. whether the distance of their center points is less than the sum of the radii of the spheres), which also determines the collision depth, and the collision normal is always aligned with the vector pointing from one sphere center to the other. So **if you can, use spheres** -- it is even worth using multiple spheres to approximate more complex shapes if possible. Capsules ("extruded spheres"), infinite planes, half-planes, infinite cylinders (distance from a line) and axis-aligned boxes are also pretty simple. Cylinders and cuboids with arbitrary rotation are bit harder. Triangle meshes (the shape most commonly used for real-time 3D models) are very difficult but may be approximated e.g. by a convex hull which is manageable (a convex hull is an intersection of a number of half-spaces) -- if we really want to precisely collide full 3D meshes, we may split each one into several convex hulls (but we need to write the non-trivial splitting algorithm of course). Also note that you need to write a detection algorithm for any possible pair of shape types you want to support, so for N supported shapes you'll need $N * (N + 1) / 2$ detection algorithms.

{ In theory we may in some cases also think about using iterative/numerical methods to find collisions, i.e. starting at some point between the bodies and somehow stepping towards their intersection until we're close enough. Another idea I had was to use signed distance functions for representing static environments, I kind of implemented it in tinyphysicsengine. ~drummyfish }

TODO: some actual algorithms

collision

Collision

Collision, sometimes also *conflict*, happens when two or more things want to occupy the same spot. This situation usually needs to be addressed somehow; then we talk about **collision resolution**. In programming there are different types of collisions, for example:

- **hash collision:** When two items produce the same hash, they will map to the same index in a hash table. Typical solution is to have a list at each table index so that multiple items can fit there.
- **collision of bodies in a physics engine:** See collision detection. These collisions are resolved by separating the bodies and updating their velocities so that they "bounce off" as in real life.
- **request collision:** General situation in which multiple clients request access to something that can be used only by one client at a time, e.g. a communication bus. Resolution is usually done by some kind of arbiter who decides, by whatever algorithm, who to grant the access to.
- **name collision:** When e.g. the same identifier is used in two separate libraries that are included at the same time, the compiler doesn't know which one is intended. This is addressed by namespaces.
- ...

Color

Color (also *colour*, from *celare*, "to cover") is the perceived visual quality of light that's associated with its wavelength/frequency (or mixture of several); for example red, blue and yellow are colors. Electromagnetic waves with wavelength from about 380 to 750 nm (about 400 to 790 THz) form the **visible spectrum**, i.e. waves our eyes can see -- combining such waves with different intensities and letting them fall on the retina of our eyes gives rise to the perception of color in our brain. There is a hugely deep *color theory* concerned with the concept of color (its definition, description, reproduction, psychological effect etc.). Needless to say colors are extremely important in anything related to visual information such as art, computer graphics, astrophysics, various visualizations or just everyday perception of our world. Color support is sometimes used as the opposite of systems that are extremely limited in the number of colors they can handle, which may be called monochromatic, 1bit (distinguishing only two colors), black&white or grayscale. Color can be seen to be in the same relation to sight as pitch is to hearing.

How many colors are there? The number of colors humans can distinguish is of course individual (color blindness makes people see fewer colors but there are also conditions that make one see more colors), then also we can ask what color really means (see below) but -- approximately speaking -- various sources state we are able to distinguish millions or even over 10 million different colors on average. In computer technology we talk about **color depth** which says the number of bits we use to represent color -- the more bits, the more colors we can represent. 24 bits are nowadays mostly used to record color (8 bits for each red, green and blue component, so called *true color*), which allows for 16777216 distinct colors, though even something like 16 bits (65536 colors) is mostly enough for many use cases. Some advanced systems however support many more colors than true color, especially extremely bright and dim ones -- see HDR.

What gives physical objects their color? Most everyday objects get its color from reflecting only specific parts of the white light (usually sunlight), while absorbing the opposite part of the spectrum, i.e. for example a white object reflects all incoming light, a black one absorbs all incoming light (that's why black things get hot in sunlight), a red one reflects the red light and absorbs the rest etc. This is determined by the qualities of the object's surface, such as the structure of its atoms or its microscopic geometry.

TODO

What Is Color?

This is actually a non-trivial question, or rather there exist many varying definitions of it and furthermore it is a matter of subjective experience, perception of colors may differ between people. When asking what color really is, consider the following:

- Are non-primary colors true colors, or just mixtures of the primary colors? Red, green and blue are the three primary colors, the ones we can mix all other colors from. Many will say yes, non-primary colors are colors. But hold on.
- Are non-spectral colors colors or just mixtures of spectral colors? Spectral colors are the colors with a single wavelength (e.g. red, orange or violet), other colors (like pink) are just mixtures of these. Again, probably yes.
- Is saturation part of color, or a separate attribute? I.e. are e.g. green and greenish gray different colors, or same colors with different saturation? Now it depends.
- Is black a color, or rather a lack of a color? E.g. in computers it is usually treated just as another color, but real world black is really the absence of any light.
- Is white a color? If we are using a subtractive color model, the argument is the same as for black (white paper is really just lack of any color on it).
- Is e.g. gold a color? Or just yellow with a lot of specular reflection? In real world many things may be called to have a gold color, but in computer graphics we would likely separate the color from the light reflective attribute (such as metalicity).
- Is transparent a color?
- Is intensity part of color (especially in context of e.g. HDR)? For example we might say both Sun and paper are white, but still Sun's color is much "stronger" -- is it therefore a "whiter white" than that of a

- paper?
- Are colors not perceivable by average human colors? Many animals see colors we can't see (e.g. those in infrared spectrum), but there are also rare cases of humans (so called tetrachromats) who see many more colors than usual thanks to a mutation.
- Are impossible colors colors? Interestingly there exist colors perceivable by average humans which however cannot naturally be seen due to "physics" -- they can however be seen with "eye hacks". Do we count these too?
- ...

combinatorics

Combinatorics

Combinatorics is an area of math that's basically concerned with counting possibilities. As such it is very related to probability theory (as probability is typically defined in terms of ratios of possible outcomes). It explores things such as permutations and combinations, i.e. question such as how many ways are there to order N objects or how many ways are there to choose k objects from a set of N objects.

The two basic quantities we define in combinatorics are permutations and combinations.

Permutation (in a simple form) of a set of objects (lets say A, B and C) is one possible ordering of such set (i.e. ABC, ACB, BAC etc.). I.e. here by permutation of a number n , which we'll write as $P(n)$, we mean the number of possible orderings of a set of size n . So for example $P(1) = 1$ because there is only one way to order a set containing one item. Similarly $P(3) = 6$ because there are six ways to order a set of three objects (ABC, ACB, BAC, BCA, CAB, CBA). $P(n)$ is computed very simply, it is factorial of n , i.e. $P(n) = n!$.

Combination (without repetition) of a set of objects says in how many ways we can select given number of objects from that set (e.g. if there are 4 shirts in a drawer and we want to choose 2, how many possibilities are there?). I.e. given a set of certain size a combination tells us the number of possible subsets of certain size. I.e. there are two parameters of a combination, one is the size of the set, n , and the other is the number of items (the size of the subset) we want to select from that set, k . This is written as nCk , $C(n,k)$ or

$$\begin{array}{c} / \quad n \quad \backslash \\ | \quad \quad | \\ \backslash \quad k \quad / \end{array}$$

A combination is computed as $C(n,k) = n! / (k! * (n - k)!)$. E.g. having a drawer with 4 shirts (A, B, C and D) and wanting to select 2 gives us $C(4,2) = 4! / (2! * (4 - 2)!) = 6$ possibilities (AB, AC, AD, BC, BD, CD).

Furthermore we can define combinations with repetitions in which we allow ourselves to select the same item from the set more than once (note that the selection order still doesn't matter). I.e. while combinations without repetition give us the number of possible subsets, a combinations WITH repetitions gives us the number of possible multisubsets of a given set. Combinations with repetition is computed as $Cr(n,k) = C(n + k - 1, k)$. E.g. having a drawer with 4 shirts and wanting to select 2 WITH the possibility to choose one shirt multiple times gives us $Cr(4,2) = C(5,2) = 5! / (2! * (5 - 2)!) = 10$ possibilities (AA, AB, AC, AD, BB, BC, BD, CC, CD, DD).

Furthermore if we take combinations and say that order matters, we get generalized permutations that also take two parameters, n and k , and there are two kinds: without and with repetitions. I.e. permutations without repetitions tell us in how many ways we can choose k items from n items when ORDER MATTERS, and is computed as $P(n,k) = n! / (n - k)!$ (e.g. $P(4,2) = 4! / (4 - 2)! = 12$, AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, DC). Permutations with repetitions tell us the same thing but we are allowed to select the same thing multiple times, it is computed as $Pr(n,k) = n^k$ (e.g. $P(4,2) = 4^2 = 16$, AA, AB, AC, AD, BA, BB, BC, BD, CA, CB, CC, CD, DA, DB, DC, DD).

To sum up:

quantity	order matters?	repetition allowed?	formula
permutation (simple)	yes		$P(n) = n!$

quantity	order matters?	repetition allowed?	formula
permutation without rep.	yes	no	$P(n,k) = n!/(n - k)!$
permutation with rep.	yes	yes	$Pr(n,k) = n^k$
combination without rep.	no	no	$C(n,k) = n! / (k! * (n - k)!)$
combination with rep.	no	yes	$Cr(n,k) = C(n + k - 1, k)$

Here is an example of applying all the measures to a three item set ABC (note that selecting nothing from a set counts as 1 possibility, NOT 0):

quantity	possibilities (for set ABC)	count
P(3)	ABC ACB BAC BCA CAB CBA	$3! = 6$
P(3,0)		$3!/(3 - 0)! = 1$
P(3,1)	A B C	$3!/(3 - 1)! = 3$
P(3,2)	AB AC BA BC CA CB	$3!/(3 - 2)! = 6$
P(3,3)	ABC ACB BAC BCA CAB CBA	$3!/(3 - 3)! = 6$
Pr(3,0)		$3^0 = 1$
Pr(3,1)	A B C	$3^1 = 3$
Pr(3,2)	AA AB AC BA BB BC CA CB CC	$3^2 = 9$
Pr(3,3)	AAA AAB AAC ABA ABB ABC ACA ACB ACC ...	$3^3 = 27$
C(3,0)		$3!/(0! * (3 - 0)!) = 1$
C(3,1)	A B C	$3!/(1! * (3 - 1)!) = 3$
C(3,2)	AB AC BC	$3!/(2! * (3 - 2)!) = 3$
C(3,3)	ABC	$3!/(3! * (3 - 3)!) = 1$
Cr(3,0)		$C(3 + 0 - 1, 0) = 1$
Cr(3,1)	A B C	$C(3 + 1 - 1, 1) = 3$
Cr(3,2)	AA AB AC BB BC CC	$C(3 + 2 - 1, 2) = 6$
Cr(3,3)	AAA AAB AAC ABB ABC ACC BBB BBC BCC CCC	$C(3 + 3 - 1, 3) = 10$

comment

Comment

Comment is part of computer code that doesn't affect how the code is interpreted by the computer and is intended to hold information for humans that read the code (even though comments can sometimes contain additional information for computers such as metadata and autodocumentation information). There are comments in basically all programming languages, they usually start with `//`, `#`, `/*` and similar symbols, sometimes parts of code that don't fit the language syntax are ignored and as such can be used for comments.

Even though you should write nice, self documenting code, **you should comment you source code** as well. General tips on commenting:

- ALWAYS put a **global file comment** at the top of a file to make it self-contained. It should include:
 - ◆ **Description of what the file actually does.** This is extremely important for readability, documentation and quick orientation. If a new programmer comes looking for a specific part of the code, he may waste hours on searching the wrong files just because the idiotic author couldn't be bothered to include fucking three sentences at the start of the file. Modern program just don't fucking do this anymore, this is just shit.
 - ◆ License/waiver, either full text or link. Even if your repo contains a global license (which it should), it's good for the file to carry the license because the file may just be copy pasted on its own into some other project and then it will appear as having no license.
 - ◆ **Name/nick of the author(s)** and roughly the date of creation (year is enough). This firstly helps legally assess copyright (who and for how long holds the copyright) and secondly helps others contact the author in case of encountering something weird in the code.

- Comment specific blocks of code with **keywords** -- this will help searching the code with tools like grep. E.g. in game's code add comment `// player: shoot, fire` to the part of code that handles player's shooting so that someone searching for any one of these two words will be directed here.
- Functions (maybe with some exceptions like trivial one-liners) should come with a comment documenting:
 - ♦ **Behavior** of the function, what it does and also how it does that (Is the function slow? Is it safe? Does it perform checks of arguments? Does it have side effects? How are errors handled? ...).
 - ♦ **Meaning of all arguments** and if needed their possible values.
 - ♦ **Return value meaning**.
- You may decide to use comment format of some autodoc system such as doxygen -- it costs nothing and helps firstly unify the style of your comments and secondly, obviously, generate automatic documentation of your code, as well as possibly automatically process it in other ways.
- TODO: moar

communism

Communism

"*Imagine no possession*" --John Lennon

Communism (from *communis* -- common, shared) is a very wide term which most generally stands for the idea that sharing and equality should be the basic values and principles of a society; as such it is a leftist idea which falls under socialism (i.e. basically focusing on people at large). There are very many branches, theories, political ideologies and schools of thought somewhat based on communism, for example Marxism, Leninism, anarcho communism, primitive communism, Christian communism, Buddhist communism etc. -- of course, some of these are good while others are evil and only abuse the word communism as a kind of *brand* (as also happens e.g. with anarchism). Sadly after the disastrous failure of the violent pseudocommunist revolutions of the 20th century, most people came to equate the word communism with oppressive militant regimes, however we have to stress that **communism is NOT equal to USSR, Marxism-Leninism, Stalinism or any other form of pseudocommunism**, on the contrary, such regimes were rather hierarchical, violent and pseudocommunist, often downright fascist. We ourselves embrace true communism and base our LRS and less retarded society on ideas of unconditional sharing. **Yes, large communist societies have existed and worked**, for example the Inca empire worked without money and provided FREE food, clothes, houses, health care, education and other products of collective work to everyone, according to his needs. Many other communities also work on more or less communist principles, see e.g. Jewish kibbutz, Sikhist langar, free software, or even just most families for that matter. Of course, no one says the mentioned societies and groups are or were ideal, just that the principles of communism DO work, that communism should be considered a necessary attribute of an ideal society and that ideal society is not impossible due to impossibility of communism because as we see, it is indeed possible. The color red is usually associated with communism and the "hammer and sickle" (U+262D) is taken as its symbol, though that's mostly associated with the evil communist regimes and so its usage by LRS supporters is probably better be avoided.

Common ideas usually associated with communism are (please keep in mind that this may differ depending on the specific flavor of communism):

- Ending capitalism and similar rightist oppressive hierarchical systems which are the polar opposite of communism and are incompatible with it. Along with these also things like consumerism, worker exploitation, crime and poverty will disappear.
- Abolishment of private property, establishing common ownership, for example a factory shouldn't have a single owner who makes profit off of it, it should rather be collectively managed by those who work in the factory and they should collectively share what they make there.
- Sharing and collaboration as opposed to competition.
- Equality, seizing of division of people into social classes (such as workers, bourgeoisie, rich, poor, aristocracy, ...).
- Eventual abolishment of state, as again in a good society that benefits people there shouldn't be any crime, theft, abuse of workers etc. However some "communists" see state and its control of economy as a necessary intermediate step towards this goal.

- Abolishment of money as that is a means of dividing people into classes (rich and poor), means of abuse (wage slavery) and a tool of systems such as capitalism. In a good society money is unnecessary, everyone gets what he needs.
- Sometimes revolution (and even war, temporary dictatorship etc.) is seen by some "communists" as a necessary way of achieving a change, however many others oppose this as revolution means violence, dominating man by another man (inequality) etc. -- peaceful voluntary evolutionary approach is also an option of achieving communism.
- Focus on workers and common people.
- Intellectual endeavor and idealism -- many communists are intellectuals, scientifically examining society and seeking models of an ideal society, a "utopia" as opposed to accepting life in a dystopia.
- ...

TODO

See Also

- less retarded society
- socialism
- capitalism

competition

Competition

Competition is a situation of conflict in which several entities try to overpower or otherwise win over each other. It is the opposite of collaboration. Competition is connected to pursuing self interest.

Competition is the absolute root cause of most evil in society. Society must never be based on competition. Unfortunately our society has decided to do the exact opposite with capitalism, the glorification of competition -- this will most certainly lead to the destruction of our society, possibly even to the destruction of all life.

Competition is to society what a drug is to an individual: competition makes a situation become better quickly and start achieving technological "progress" but for the price of things going downwards from then on, competition quickly degenerates and kills other values in society such as altruism and morality; society that decides to make unnaturally fast "progress" and base itself on competition is equivalent to someone deciding to take steroids to grow muscles quickly -- corporations that arise in technologically advanced society take over the world just like muscle cancer that grows from taking steroids. A little bit of competition can be helpful in small doses just as painkillers can on occasion help lower suffering of an individual, but one has to be extremely careful to not take too many of them... even smoking a joint from time to time can have a positive effect, however with capitalism our society has become someone who has started to take heroin and only live for that drug alone, take as much of it as he can. Invention of bullshit jobs just to keep competition running, extreme growing hostility of people, productivity cults, overworking, wage slavery, extreme waste that's destroying our environment, all of these are signs our society is dying from overdose, living from day to day, trying to get a few bucks for the next dose of its drug.

Is all competition bad? As a mechanism in society yes. But as concept outside these boundaries it may on occasion be good, it may for example be used in genetic programming to evolve good computer programs. People also have a NEED for at least a bit of competition as this need was necessary to survive in the past and is hard wired in us -- this need has to be satisfied, so we create artificial, mostly harmless competition e.g. with games and sports -- please note that people playing games doesn't mean competition is part of basic mechanics of society (this overlook in the thought process often happens), just as singing in a shower isn't part of how democracy works for example. This kind of competition happening between people (but not withing mechanisms of society) is not so bad as long as we are aware of the dangers of overapplying it (just as we have to be careful with any kind of drug for example). What IS bad is making competition the basis of a society, in a good society people must never compete for basic needs such as food, shelter or health care. People must never see other people as enemies. Furthermore after sufficient technological progress, competition is no longer just a bad basis for society, it becomes a fatal one because society gains means for complete annihilation of all life such as nuclear weapons or factories poisoning our environment that in the

heat of competition will sooner or later destroy the society. I.e. in a technologically advanced society it is necessary to give up competition so as to prevent own destruction. Sadly we are probably past the point now.

Why is competition so prevalent if it is so bad? Because it is natural and it has been with us since we as life came to existence. It is immensely difficult to let go of such a basic instinct but it has to be done not only because competition has become obsolete and is now only artificially sustaining suffering without bringing in any benefits (we, humans, have basically already won at evolution), but because, as has been said, sustaining competition is now simply fatal.

How to achieve letting go of competition in society? The only way is a voluntary choice achieved through our intellect, i.e. through education. Competition is something we naturally want to do, but we can rationally decide not to do it once we see and understand it is bad -- such behavior is already occurring, for example if we know someone is infected with a sexually transmitting disease, we rationally overcome the strong natural instinct to have sex with him.

compiler_bomb

Compiler Bomb

Compiler bomb is a type of software bomb (similar to fork bombs, zip bombs, tar bombs etc.) exploiting compilers, specifically it's a short program (written in the compiler's programming language) which when compiled produces an extremely large compiled program (i.e. executable binary, bytecode, transpiled code etc.). Effectiveness of such a bomb can be measured as the size of output divided by the size of input. Of course compiler bombs usually have to be targeted at a specific compiler (its weaknesses, optimizations, inner mechanisms, ...), the target platform and so on, however some compiler bombs are quite universal as many compilers employ similar compiling strategies and produce similar outputs. Alternatively a compiler bomb can be defined to do other malicious things, like maximizing the amount of RAM and time needed for compilation etc.

{ Found here: <https://codegolf.stackexchange.com/questions/69189/build-a-compiler-bomb>. ~drummyfish }

Compiler bombs in various languages:

- C: `main[-1u]={1};`, creates 16 GB executable, works by defining a huge array and initializes its first element so the whole array will be explicitly stored in the executable.
 - Rust: every program :D
 - comun: `TODO :-}`
 - ...
-

complexity

Complexity

Complexity may stand for several things, for example:

- opposite of simplicity, general complexity of technology, see e.g. bloat
 - computational complexity, mathematical study of computer resource usage
 - ...
-

compression

Compression

Compression means encoding data (such as images or texts) in a different way so that the data takes less space (memory) while keeping all the important information, or, in plain terms, it usually means "making files smaller". Compression is pretty important so that we can utilize memory or bandwidth well -- without it our

hard drives would be able to store just a handful of videos, internet would be slow as hell due to the gigantic amount of transferred data and our RAM wouldn't suffice for things we normally do. There are many algorithms for compressing various kinds of data, differing by their complexity, performance, efficiency of compression etc. The reverse process to compression (getting the original data back from the compressed data) is called **decompression**. The ratio of the compressed data size to the original data size is called **compression ratio** (the lower, the better). The science of data compression is truly huge and complicated AF, here we'll just mention some very basics. Also watch out: compression algorithms are often a patent mine field.

{ I've now written a tiny LRS compression library/utility called shitpress, check it out at <https://codeberg.org/drummyfish/shitpress>. It's fewer than 200 LOC, so simple it can nicely serve educational purposes. The principle is simple, kind of a dictionary method, where the dictionary is simply the latest output 64 characters; if we find a long word that occurred recently, we simply reference it with mere 2 bytes. It works relatively well for most data! ~drummyfish }

{ There is a cool compressing competition known as Hutter Prize that offers 500000 pounds to anyone who can break the current record for compressing Wikipedia. Currently the record is at compressing 1GB down to 115MB. See <http://prize.hutter1.net> for more. ~drummyfish }

{ LMAO retard patents are being granted on impossible compression algorithms, see e.g. <http://gailly.net/05533051.html>. See also Sloot Digital Coding System, a miraculous compression algorithm that "could store a whole movie in 8 KB" lol. ~drummyfish }

Let's keep in mind compression is not applied just to files on hard drives, it can also be used e.g. in RAM to utilize it more efficiently.

Why don't we compress everything? Firstly because compressed data is slow to work with, it requires significant CPU time to compress and decompress data, it's a kind of a space-time tradeoff (we gain more storage space for the cost of CPU time). Compression also obscures data, for example compressed text file will typically no longer be human readable, any code wanting to work with such data will have to include the nontrivial decompression code. Compressed data is also more prone to corruption because redundant information (which can help restoring corrupted data) is removed from it -- in fact we sometimes purposefully do the opposite of compression and make our data bigger to protect it from corruption (see e.g. error correcting codes, RAID etc.). And last but not least, many data can hardly be compressed or are so small it's not even worth it.

The basic division of compression methods is to:

- **lossless**: No information contained in the original data will be lost in the compressed data, i.e. the original file can be restored in its entirety from the compressed file.
- **lossy**: Some information contained in the original data is lost during compression, i.e. for example a compressed image will be of slightly worse quality. This usually allows for much greater compression. Lossy compressors usually also additionally apply lossless compression as well.

Furthermore we may divide compression e.g. to offline (compresses a whole file, may take long) and streaming (compressing a stream of input data on-the-go and in real-time), by the type of input data (binary, text, audio, ...), basic principle (RLE, dictionary, "A", ...) etc.

The following is an example of how well different types of compression work for an image (screenshot of main page of Wikimedia Commons, 1280x800):

{ Though the website screenshot contained also real life photos, it still contained a lot of constant color areas which can be compressed very well, hence quite good compression ratios here. A general photo won't be compressed as much. ~drummyfish }

compression	~size (KB)	ratio
none	3000	1
general lossless (lz4)	396	0.132
image lossless (PNG)	300	0.1

compression	~size (KB)	ratio
image lossy (JPG), nearly indistinguishable quality	164	0.054
image lossy (JPG), ugly but readable	56	0.018

Mathematically there cannot exist a lossless compression algorithm that would always reduce the size of any input data -- if it existed, we could just repeatedly apply it and compress ANY data to zero bytes. And not only that -- **every lossless compression will inevitably enlarge some input files**. This is also mathematically given -- we can see compression as simply mapping input binary sequences to output (compressed) binary sequences, while such mapping has to be one-to-one (bijjective); it can be simply shown that if we make any such mapping that reduces the size of some input (maps a longer sequence to a shorter one, i.e. compresses it), we will also have to map some short code to a longer one. However we can make it so that our compression algorithm enlarges a file at most by 1 bit: we can say that the first bit in the compressed data says whether the following data is compressed or not; if our algorithm fails to reduce the size of the input, it simply sets the bit to says so and leaves the original file uncompressed (in practice many algorithms don't do this though as they try to work as streaming filters, without random access to data, which would be needed here).

Dude, how does compression really work tho? The basic principle of lossless compression is **removing redundancy** (correlations in the data), i.e. that which is explicitly stored in the original data but doesn't really have to be there because it can be reasoned out from the remaining data. This is why a completely random noise can't be compressed -- there is no correlated data in it, nothing to reason out from other parts of the data. However human language for example contains many redundancies. Imagine we are trying to compress English text and have a word such as "computer" on the input -- we can really just shorten it to "computr" and it's still pretty clear the word is meant to be "computer" as there is no other similar English word (we also see that compression algorithm is always specific to the type of data we expect on the input -- we have to know what nature of the input data we can expect). Another way to remove redundancy is to e.g. convert a string such as "HELLOHELLOHELLOHELLOHELLO" to "5xHELLO". Lossy compression on the other hand tries to decide what information is of low importance and can be dropped -- for example a lossy compression of text might discard information about case (upper vs lower case) to be able to store each character with fewer bits; an all caps text is still readable, though less comfortably.

{ A quick intuitive example: encyclopedias almost always have at the beginning a list of abbreviations they will use in the definition of terms (e.g. "m.a. -> middle ages", ...), this is so that the book gets shorter and they save money on printing. They compress the text. ~drummyfish }

OK, but how much can we really compress? Well, as stated above, there can never be anything such as a universal uber compression algorithm that just makes any input file super small -- everything really depends on the nature of the data we are trying to compress. The more we know about the nature of the input data, the more we can compress, so a general compression program will compress only a little, while an image-specialized compression program will compress better (but will only work with images). As an extreme example, consider that **in theory we can make e.g. an algorithm that compresses one specific 100GB video to 1 bit** (we just define that a bit "1" decompresses to this specific video), but it will only work for that one single video, not for video in general -- i.e. we made an extremely specialized compression and got an extremely good compression ratio, however due to such extreme specialization we can almost never use it. As said, we just cannot compress completely random data at all (as we don't know anything about the nature of such data). On the other hand data with a lot of redundancy, such as video, can be compressed A LOT. Similarly video compression algorithms used in practice work only for videos that appear in the real world which exhibit certain patterns, such as two consecutive frames being very similar -- if we try to compress e.g. static (white noise), video codecs just shit themselves trying to compress it (look up e.g. videos of confetti and see how blocky they get). All in all, some compression benchmarks can be found e.g. at <https://web.archive.org/web/20110203152015/http://www.maximumcompression.com/index.html> -- the following are mentioned types of data and their best measured compression ratios: English text 0.12, image (lossy) 0.76, executable 0.24.

Methods

The following is an overview of some most common compression techniques.

Lossless

RLE (run length encoding) is a simple method that stores repeated sequences just as one element of the sequence and number of repetitions, i.e. for example "abcabcabc" as "3abc".

Entropy coding is another common technique which counts the frequencies (probabilities) of symbols on the input and then assigns the shortest codes to the most frequent symbols, leaving longer codes to the less frequent. The most common such codings are **Huffman coding** and **Arithmetic coding**.

Dictionary (substitutional) methods try to construct a dictionary of relatively long symbols appearing in the input and then only store short references to these symbols. The format may for example choose to first store the dictionary and then the actual data with pointers to this dictionary, or it may just store the data in which pointers are stored to previously appearing sequences.

Predictor compression is based on making a *predictor* that tries to guess following data from previous values (which can be done e.g. in case of pictures, sound or text) and then only storing the difference against such a predicted result. If the predictor is good, we may only store the small amount of the errors it makes.

A famous family of dictionary compression algorithms are **Lempel-Ziv (LZ)** algorithms -- these two guys first proposed LZ77 in (1977, sliding window) and LZ78 (explicitly stored dictionary, 1978). These were a basis for improved/remix algorithms, most notably LZW (1984, Welch). Additionally these algorithms are used and combined in other algorithms, most notably gif and DEFLATE (used e.g. in gzip and png).

An approach similar to the predictor may be trying to find some general mathematical model of the data and then just find and store the right parameters of the model. This may for example mean vectorizing a bitmap image, i.e. finding geometrical shapes in an image composed of pixels and then only storing the parameters of the shapes -- of course this may not be 100% accurate, but again if we want to preserve the data accurately, we may additionally also store the small amount of errors the model makes. Similar approach is used in vocoders used in cellphones that try to mathematically model human speech (however here the compression is lossy), or in fractal compression of images. A nice feature we gain here is the ability to actually "increase the resolution" (or rather generate detail) of the original data -- once we fit a model onto our data, we may use it to tell us values that are not actually present in the data (i.e. we get a fancy interpolation/extrapolation).

Another property of data to exploit may be its sparsity -- if for example we have a huge image that's prevalently white, we may say white is the implicit color and we only somehow store the pixels of other colors.

Some more wild techniques may include genetic programming that tries to evolve a small program that reproduces the input data, or using "AI" in whatever way to compress the data (in fact compression is an essential part of many neural networks as it forces the network to "understand", make sense of the data -- many neural networks therefore internally compress and decompress the data so as to filter out the unimportant information; large language models are now starting to beat traditional compression algorithms at compression ratios).

Note that many of these methods may be **combined or applied repeatedly** as long as we are getting smaller results.

Furthermore also take a look at procedural generation, a related technique that allows to embed a practically infinite amount of content with only quite small amount of code.

Lossy

In lossy compression we generally try to limit information that is not very important and/or to which we aren't very sensitive, typically by dropping precision by quantization, i.e. basically lowering the number of bits we use to store the "not so important" information -- in some cases we may just drop some information altogether (decrease precision to zero). Furthermore we finally also apply lossless compression to make the result even smaller.

For **images** we usually exploit the fact that human sight is less sensitive to certain visual information, such as specific frequencies, colors, brightness etc. Common methods used here are:

- Convert image from RGB to YUV, leave the Y channel (brightness) as is and reduce resolution of the U and V (color) channels. This works because human eye is less sensitive to color than brightness.
- Convert the image to frequency domain (e.g. with DCT or some wavelet transform) and quantize (allocate fewer bits to) higher frequencies. This exploits the fact that human eye is less sensitive to higher frequencies. This is the basis of e.g. jpeg.
- Reduce the number of possible colors -- traditional RGB uses 8 bits for each R, G and B component and so each pixel takes 3 bytes, which allows for about 6 million colors. However using just 2 bytes (65 thousand colors) many times suffices and saves 1/3rd of the size -- see RGB565. We may also utilize an image-specific palette and save the image in indexed mode, i.e. compute a palette of let's say 256 most common colors in the image, then encode the image as the palette plus pixels, of which each will only take one byte! This saves almost 2/3rds of the size. The drop of quality can further be made less noticeable with dithering.
- Reduce resolution -- plain simple. However this can be made smarter by e.g. trying to detect areas with few details and only reducing the resolution there.

In **video** compression we may reuse the ideas from image compression and further employ exploiting temporal redundancy, i.e. the fact that consecutive video frames look similar, so we may only encode some kind of delta (change) against the previous (or even next) frame. The most common way is to fully record only one key frame in some time span (so called I-frame, further compressed with image compression methods), then divide it to small blocks and estimate the movement of those blocks so that they approximately make up the following frames -- we then record only the motion vectors of the blocks. This is why videos look "blocky". In the past interlacing was also used -- only half of each frame was recorded, every other row was dropped; when playing, the frame was interlaced with the previous frame. Another cool idea is keyframe superresolution: you store only some keyframes in full resolutions and store the rest of them in smaller size; during decoding you can use the nearby full scale keyframes to upscale the low res keyframes (search for matching subblocks in the low res image and match them to those in the big res image).

In **audio** we usually straight remove frequencies that humans can't hear (usually said to be above 20 kHz), for this we again convert the audio from spatial to frequency domain (using e.g. Fourier transform). Furthermore it is very inefficient to store sample values directly -- we rather use so called *differential PCM*, a lossless compression that e.g. stores each sample as a difference against the previous sample (which is usually small and doesn't use up many bits). This can be improved by a predictor, which tries to predict the next values from previous values and then we only save the difference against this prediction. *Joint stereo coding* exploits the fact that human hearing is not so sensitive to the direction of the sound and so e.g. instead of recording both left and right stereo channels in full quality rather records the sum of both and a ratio between them (which can get away with fewer bits). *Psychoacoustics* studies how humans perceive sound, for example so called *masking*: certain frequencies may for example mask nearby (both in frequency and time) frequencies (make them unheardable for humans) so we can drop them. See also vocoders.

TODO: LZW, DEFLATE etc.

Compression Programs/Utils/Standards

Here is a list of some common compression programs/utilities/standards/formats/etc:

util/format	extensions	free?	media	lossless?	notes
<u>bzip2</u>	.bz2	yes	general	yes	Burrows-Wheeler alg.
<u>flac</u>	.flac	yes	audio	yes	super free lossless audio format
<u>gif</u>	.gif	now yes	image/anim.	no	limited color palette, patents expired
<u>gzexe</u>		yes	executable bin.	yes	makes self-extracting executable
<u>gzip</u>	.gz	yes	general	yes	by GNU, DEFLATE, LZ77, mostly used by Unices
<u>jpeg</u>	.jpg, .jpeg	yes?	raster image	no	common lossy format, under patent fire
<u>lz4</u>	.lz4	yes	general	yes	high compression/decompression speed, LZ77

util/format	extensions	free?	media	lossless?	notes
<u>mp3</u>	.mp3	now yes	audio	no	popular audio format, patents expired
<u>png</u>	.png	yes	raster image	yes	popular lossless image format, transparency
<u>rar</u>	.rar	NO	general	yes	popular among normies, PROPRIETARY
<u>vorbis</u>	.ogg	yes	audio	no	was a free alternative to mp3, used with ogg
<u>zip</u>	.zip	yes?	general	yes	along with encryption may be patented
<u>7-zip</u>	.7z	yes	general	yes	more complex archiver

Code Example

Let's write a simple lossless compression utility in `C`. It will work on binary files and we will use the simplest RLE method, i.e. our program will just shorten continuous sequences of repeating bytes to a short sequence saying "repeat this byte N times". Note that this is very primitive (a small improvement might be actually done by looking for sequences of longer words, not just single bytes), but it somewhat works for many files and demonstrates the basics.

The compression will work like this:

- We will choose some random, hopefully not very frequent byte value, as our special "marker value". Let's say this will be the value `0xF3`.
- We will read the input file and whenever we encounter a sequence of 4 or more same bytes in a row, we will output these 3 bytes:
 - ♦ the marker value
 - ♦ byte whose values is the length of the sequence minus 4
 - ♦ the byte to repeat
- If the marker value is encountered in input, we output 2 bytes:
 - ♦ the marker value
 - ♦ value `0xFF` (which we won't be able to use for the length of the sequence)
- Otherwise we just output the byte we read from the input.

Decompression is then quite simple -- we simply output what we read, unless we read the marker value; in such case we look whether the following value is `0xFF` (then we output the marker value), else we know we have to repeat the next character this many times plus 4.

For example given input bytes

```
0x11 0x00 0x00 0xAA 0xBB 0xBB 0xBB 0xBB 0xBB 0xBB 0x10 0xF3 0x00
                        \-----/      \-/
                        long repeating sequence  marker!
```

Our algorithm will output a compressed sequence

```
0x11 0x00 0x00 0xAA 0xF3 0x02 0xBB 0x10 0xF3 0xFF 0x00
                        \-----/      \-----/
                        compressed seq.  encoded marker
```

Notice that, as stated above in the article, there inevitably exists a "danger" of actually enlarging some files. This can happen if the file contains no sequences that we can compress and at the same time there appear the marker values which actually get expanded (from 1 byte to 2).

The nice property of our algorithm is that both compression and decompression can be streaming, i.e. both can be done in a single pass as a filter, without having to load the file into memory or randomly access bytes in files. Also the memory complexity of this algorithm is constant (RAM usage will be the same for any size of the file) and time complexity is linear (i.e. the algorithm is "very fast").

Here is the actual code of this utility (it reads from stdin and outputs to stdout, a flag `-x` is used to set decompression mode, otherwise it is compressing):

```
#include <stdio.h>
```

```

#define SPECIAL_VAL 0xf3 // random value, hopefully not very common

void compress(void)
{
    unsigned char prevChar = 0;
    unsigned int  seqLen = 0;
    unsigned char end = 0;

    while (!end)
    {
        int c = getchar();

        if (c == EOF)
            end = 1;

        if (c != prevChar || c == SPECIAL_VAL || end || seqLen > 200)
        { // dump the sequence
            if (seqLen > 3)
                printf("%c%c%c", SPECIAL_VAL, seqLen - 4, prevChar);
            else
                for (int i = 0; i < seqLen; ++i)
                    putchar(prevChar);

            seqLen = 0;
        }

        prevChar = c;
        seqLen++;

        if (c == SPECIAL_VAL)
        {
            // this is how we encode the special value appearing in the input
            putchar(SPECIAL_VAL);
            putchar(0xff);
            seqLen = 0;
        }
    }
}

void decompress(void)
{
    unsigned char end = 0;

    while (1)
    {
        int c = getchar();

        if (c == EOF)
            break;

        if (c == SPECIAL_VAL)
        {
            unsigned int seqLen = getchar();

            if (seqLen == 0xff)
                putchar(SPECIAL_VAL);
            else
            {
                c = getchar();

                for (int i = 0; i < seqLen + 4; ++i)
                    putchar(c);
            }
        }
        else
            putchar(c);
    }
}

int main(int argc, char **argv)
{
    if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'x' && argv[1][2] == 0)

```

```

    decompress();
else
    compress();

return 0;
}

```

How well does this perform? If we try to let the utility compress its own source code, we get to 1242 bytes from the original 1344, which is not so great -- the compression ratio is only about 92% here. We can see why: the only repeating bytes in the source code are the space characters used for indentation -- this is the only thing our primitive algorithm manages to compress. However if we let the program compress its own binary version, we get much better results (at least on the computer this was tested on): the original binary has 16768 bytes while the compressed one has 5084 bytes, which is an EXCELLENT compression ratio of 30%! Yay :-)

See Also

- [procedural generation](#)
- [minification](#)

compsci

Computer Science

Computer science, abbreviated as "compsci", is (surprise-surprise) a [science](#) studying [computers](#). The term is pretty wide, a lot of it covers very formal and theoretical areas that neighbor and overlap with [mathematics](#), such as [formal languages](#), [cryptography](#) and [machine learning](#), but also more practical/applied and "softer" disciplines such as [software engineering](#), [programming hardware](#), computer networks or even [user interface](#) design. This science deals with such things as [algorithms](#), [data structures](#), [artificial intelligence](#) and [information](#) theory. The field has become quite popular and rapidly growing after the coming of the 21st century computer/[Internet](#) revolution and it has also become quite spoiled and abused by its sudden lucrativity.

Overview

Notable fields of computer science include:

- [artificial intelligence](#)
- [computer graphics](#)
- [databases](#)
- [hardware](#) design
- [networking](#)
- [security](#) and [cryptography](#)
- [software engineering](#)
- theoretical computer science
- [user interface](#)
- smaller field or subfields such as [operating systems](#), [compiler](#) design, formal verification, speech recognition etc.

Computer science also figures in interdisciplinary endeavors such as [bioinformatics](#) and [robotics](#).

In the industry there have arisen fields of [art](#) and study that probably shouldn't be included in computer science itself, but are very close to it. These may include e.g. [web design](#) (well, let's include it for the sake of completeness), [game](#) design, [system administration](#) etc.

computational_complexity

Computational Complexity

Computational complexity is a formal (mathematical) study of resource usage (usually time and memory) by computers as they're solving various types of problems. For example when using computers to sort arrays of numbers, computational complexity can tell us which algorithm will be fastest as the size of the array grows, which one will require least amount of memory (RAM) and even what's generally the fastest way in which this can be done. While time ("speed", number of steps) and memory (also *space*) are generally the resources we are most interested in, other can be considered too, e.g. network or power usage. Complexity theory is extremely important and one of the most essential topics in computer science; it is also immensely practically important as it helps us optimize our programs, it teaches us useful things such as that we can trade time and space complexity (i.e. make program run faster on detriment of memory and vice versa) etc.

Firstly we have to distinguish between two basic kinds of complexity:

- **algorithm complexity**: Complexity of a specific algorithm. For example quick sort and bubble sort are both algorithms for sorting arrays but quick sort has better time complexity. (Sometimes we may extend this meaning and talk e.g. about memory complexity of a data structure etc.)
- **problem complexity**: Complexity of the best algorithm that can solve particular problem; e.g. time complexity of sorting an array is given by time complexity of the algorithm that can sort arrays the fastest.

Algorithm Complexity

Let us now focus on algorithm complexity, as problem complexity follows from it. OK, so **what really is the "algorithm complexity"**? Given resource R -- let's consider e.g. time, or the number of steps the algorithm needs to finish solving a problem -- let's say that complexity of a specific algorithm is a function $f(N)$ where N is the size of input data (for example length of the array to be sorted), which returns the amount of the resource (here number of steps of the algorithm). However we face issues here, most importantly that the number of steps may not only depend on the size of input data but also the data itself (e.g. with sorting it may take shorter time to sort and already sorted array) and on the computer we use (for example some computers may be unable to perform multiplication natively and will emulate it with SEVERAL additions, increasing the number of steps), and also the exact complexity function will be pretty messy (it likely won't be a nice smooth function but rather something that jumps around a bit). So this kind of sucks. We have to make several steps to get a nice, usable theory.

The **solution** to above issues will be achieved in several steps.

FIRSTLY let's make it more clear what $f(N)$ returns exactly -- when computing algorithm complexity we will always be interested in one of the following:

- **best case** scenario: Here we assume $f(N)$ always returns the best possible value for given N , usually the lowest (i.e. least number of steps, least amount of memory etc.). So e.g. with array sorting for each array length we will assume the input array has such values that the given algorithm will achieve its best result (fastest sorting, best memory usage, ...). I.e. this is the **lower bound** for all possible values the function could give for given N .
- **average case** scenario: Here $f(N)$ returns the average, i.e. taking all possible inputs for given input size N , we just average the performance of our algorithm and this is what the function tells us.
- **worst case** scenario: Here $f(N)$ return the worst possible values for given N , i.e. the opposite of best case scenario. This is the **upper bound** for all possible value the function could give for given N .

This just deals with the fact that some algorithms may perform vastly different for different data -- imagine e.g. linear searching of a specific value in a list; if the searched value is always at the beginning, the algorithm always performs just one step, no matter how long the list is, on the other hand if the searched value is at the end, the number of steps will increase with the list size. So when analyzing an algorithm **we always specify which kind of case we are analyzing** (WATCH OUT: do not confuse these cases with differences between big O, big Omega and big Theta defined below). So let's say from now on we'll be implicitly examining worst case scenarios.

SECONDLY rather than being interested in PRECISE complexity functions we will rather focus on so called **asymptotic complexity** -- this kind of complexity is only concerned with how fast the resource usage generally GROWS as the size of input data approaches big values (infinity). So again, taking the example of array sorting, we don't really need to know exactly how many steps we will need to sort any given array, but rather how the time needed to sort bigger and bigger arrays will grow. This is also aligned with practice in another way: we don't really care how fast our program will be for small amount of data, it doesn't matter if it takes 1 or 2 microseconds to sort a small array, but we want to know how our program will scale -- if we have 10 TB of data, will it take 10 minutes or half a year to sort? If this data doubles in size, will the sorting time also double or will it increase 1000 times? This kind of complexity also no longer depends on what machine we use, the rate of growth will be the same on fast and slow machine alike, so we can conveniently just consider some standardized computer such as Turing machine to mathematically study complexity of algorithms.

Rather than exact value of resource usage (such as exact number of steps or exact number of bytes in RAM) asymptotic complexity tells us a **class** into which our complexity falls. These classes are given by mathematical functions that grow as fast as our complexity function. So basically we get kind of "tiers", like *constant*, *linear*, *logarithmic*, *quadratic* etc., and our complexity simply falls under one of them. Some common complexity classes, from "best" to "worst", are following (note this isn't an exhaustive list):

- **constant**: Given by function $f(x) = 1$ (i.e. complexity doesn't depend on input data size). Best.
- **logarithmic**: Given by function $f(x) = \log(x)$. Note the base of logarithm doesn't matter.
- **linear**: Given by function $f(x) = x$.
- **linearithmic**: Given by function $f(x) = x * \log(x)$.
- **quadratic**: Given by function $f(x) = x^2$.
- **cubic**: Given by function $f(x) = x^3$.
- **exponential**: Given by function $f(x) = n^x$. This is considered very bad, practically unusable for larger amounts of data.

Now we just put all the above together, introduce some formalization and notation that computer scientists use to express algorithm complexity, you will see it anywhere where this is discussed. There are the following:

- **big O (Omicron) notation**, written as $O(f(N))$: Says the algorithm complexity (for whatever we are measuring, i.e. time, space etc. and also the specific kind of case, i.e. worst/best/average) is asymptotically bounded from ABOVE by function $f(N)$, i.e. says the **upper bound** of complexity. This is probably the most common information regarding complexity you will encounter (we usually want this "pessimistic" view). More mathematically: complexity $f(x)$ belongs to class $O(g(y))$ if from some N_0 (we ignore some initial oscillations before this value) the function f always stays under function g multiplied by some positive constant C . Formally: $f(x) \text{ belongs to } O(g(y)) \Rightarrow \text{exists } C > 0 \text{ and } N_0 > 0: \text{ for all } n \geq N_0: 0 \leq f(n) \leq C * g(n)$.
- **big Omega notation**, written as $\Omega(f(N))$: Says the algorithm complexity **lower bound** is given by function $f(N)$. Formally: $f(x) \text{ belongs to } \Omega(g(y)) \Rightarrow \text{exists } C > 0 \text{ and } N_0 > 0: \text{ for all } n \geq N_0: 0 \leq C * g(n) \leq f(n)$.
- **big Theta notation**, written as $\Theta(f(N))$: This just means the complexity is both $O(f(N))$ and $\Omega(f(N))$, i.e. the complexity is tightly bounded by given function.

Please note that big O/Omega/Theta are a different thing than analyzing best/worst/average case! We can compute big O, big Omega and big Theta for all best, worst and average case, getting 9 different "complexities".

Now notice (also check by the formal definitions) that we simply don't care about additive and multiplicative constants and we also don't care about some initial oscillations of the complexity function -- it doesn't matter if the complexity function is $f(x) = x$ or $f(x) = 100000000 + 100000000 * x$, it still falls under linear complexity! If we have algorithm A and B and A has better complexity, A doesn't necessarily ALWAYS perform better, it's just that as we scale our data size to very high values, A will prevail in the end.

Another thing we have to clear up: **what does input size really mean?** I.e. what exactly is the N in $f(N)$? We've said that e.g. with array sorting we saw N as the length of the array to be sorted, but there are several things to additionally talk about. Firstly it usually doesn't matter if we measure the size of input in bits, bytes or number of items -- note that as we're now dealing with asymptotic complexity, i.e. only growth rate

towards infinity, we'll get the same complexity class no matter the units (e.g. a linear growth will always be linear, no matter if our x axis measures meters or centimeters or light years). SECONDLY however it sometimes DOES matter how we define the input size, take e.g. an algorithm that takes a square image with resolution $R * R$ on the input, iterates over all pixels and find the brightest one; now we can define the input size either as the total number of pixels of the image (i.e. $N = R * R$) OR the length of the image side (i.e. $N = R$) -- with the former definition we conclude the algorithm to have linear time complexity (for N input pixels the algorithm makes roughly N steps), with the latter definition we get QUADRATIC time complexity (for image with side length N the algorithm makes roughly $N * N$ steps). What now, how to solve this? Well, this isn't such an issue -- we can define the input size however we want, we just have to **stay consistent** so that we are able to compare different algorithms (i.e. it holds that if algorithm A has better complexity than algorithm B , it will stay so under whatever definition of input size we set), AND when mentioning complexity of some algorithm we should mention how we define the input size so as to prevent confusion.

With memory complexity we encounter a similar issue -- we may define memory consumption either as an EXTRA memory or TOTAL memory that includes the memory storing the input data. For example with array sorting if an algorithm works in situ (in place, needing no extra memory), considering the former we conclude memory complexity to be constant (extra memory needed doesn't depend on size of input array), considering the latter we conclude the memory complexity to be linear (total memory needed grows linearly with the size of input array). The same thing as above holds: whatever definition we choose, we should just mention which one we chose and stay consistent in using it.

Problem Complexity

See also [P vs NP](#).

As said, problem complexity is tied to algorithm complexity; a complexity of specific problem (e.g. sorting an array, factorization of a number, searching a sorted list etc.) is determined by the best possible algorithm that solves the problem (best in terms of analyzed complexity). Traditionally we use Turing machines and formal languages to analyze problem complexity. Here we'll stay a bit informal and only mention some ideas.

Similarly to algorithm complexity, with problems we again define **classes** that are only about "how quickly the resource usage grows as we increase input size". The main difference is we are examining problems, so the classes we get are classes of PROBLEMS (usually classes of formal languages, like e.g. in Chomsky's language hierarchy), not classes of functions (seen in algorithm complexity). Some common classes are:

- **DTime(f(x))**: Problems for whose solution a DETERMINISTIC Turing machine has an algorithm with time complexity $O(f(x))$.
- **NTime(f(x))**: Problems for whose solution a NON-DETERMINISTIC Turing machine has an algorithm with time complexity $O(f(x))$.
- **DSpace(f(x))**: Same as DTime but for space complexity.
- **NSpace(f(x))**: Same as NTime but for space complexity.
- **P**: Union of all classes $DTime(n^k)$, i.e. all problems that can be solved by a DETERMINISTIC Turing machine with polynomial time complexity.
- **NP**: Union of all classes $NTime(n^k)$, i.e. the same as above but for NON-DETERMINISTIC Turing machine. It is currently not known if classes P and NP are the same, though it's believed to not be the case -- in fact this is probably the most famous yet unsolved problem of computer science.
- **EXP**: Union of all classes $DTime(2^{n^k})$.
- ...

Examples

Practically analyzing time complexity of algorithms mostly involves looking at the loops in our algorithm as these are what makes number of steps in algorithm variable. Linear sequences of instructions (such as initializations) don't interest us, no matter how long, as these have no effect on asymptotic complexity. But remember that looping may also be achieved with recursion etc., so just look carefully.

Let's consider the simple bubble sort array sorting algorithm, with the simple optimization that ends as soon as the array is sorted; here it is written in C:

```

void bubbleSort(int *array)
{
    for (int i = 0; i < N - 1; ++i)
    {
        int end = 1;

        for (int j = 0; j < N - 1 - i; ++j)
            if (a[j] > a[j + 1])
            {
                swap(&a[j], &a[j + 1]);
                end = 0;
            }

        if (end) // no swap happened => sorted, end
            break;
    }
}

```

The size of input data is the length of input array (N), for memory we consider only the extra memory used. Let's see about different complexities now:

- **asymptotic TIME complexity for WORST case:** Worst case happens when the optimizing condition (if (end)) never triggers and so both loops (outer and inner one) in our algorithm run all their iterations; the outer loop is performed approximately N times (actually $N - 1$ times but remember that asymptotic complexity ignores -1 here as an additive constant) and for each of its iterations the inner loop runs approximately $N - i$ times. So e.g. for $N = 5$ we get approximately $5 + 4 + 3 + 2 + 1$ steps, so for given N we basically have to sum up numbers from 1 to N -- there is a formula for computing such sum and that is $(N(N + 1)) / 2 = N^2 / 2 + N / 2$. With asymptotic complexity we just take the biggest term and ignore any multiplication constant (division by two), so we just see N^2 here and conclude that the time complexity of worst case for bubble sort is quadratic, i.e. $O(N^2)$. Notice that technically we can also say the complexity belongs to any "worse" class, e.g. $O(N^3)$ (as $O(N^2)$ is its subclass), which is technically true but doesn't tell us as much. So here it's better to further more precisely say the complexity of worst case also belongs to $\Omega(N^2)$ (the lower bound) and therefore (by belonging to both $O(N^2)$ and $\Omega(N^2)$) also belongs to $\Theta(N^2)$, i.e. it "won't be slower BUT NOR faster than N^2 ".
- **asymptotic TIME complexity for BEST case:** Best case happens when the input array is already sorted -- here the algorithm enters the outer loop, then runs the inner loop -- approximately N iterations -- and then (since no swap happened) ends at the final condition. So the time complexity is linear -- we can say that the upper asymptotic bound on best case scenario is $O(N)$. Again we see the complexity is linearly bound from the bottom too so it's better to say the complexity of best case belongs to $\Theta(N)$.
- **space (memory) complexity:** Bubble sort works in place and though it uses extra variables, the size of those variables doesn't depend on the size of input array, so (as we only count extra memory requirements) we can say memory complexity is constant, i.e. $O(1)$, and also $\Theta(1)$.

TODO: also something simpler? problem complexity?

computer

Computer

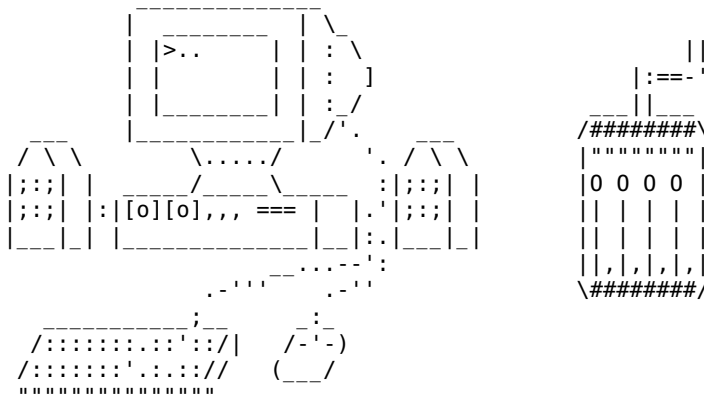
The word *computer* may be defined in countless ways and can also adopt many different meanings; a somewhat common definition may be this: computer is a machine that automatically performs mathematical computations. We can also see it as a machine for processing information, manipulating symbols or, very generally, as any tool that helps computation, in which case not just laptops, desktops and cellphones fit the definition, but also primitive computers like a sundial, one's fingers or even a mathematical formula. But nowadays the word of course implicitly implies an electronic digital computer.

The electronic digital computer turned out to be one of the greatest technological inventions in history for numerous reasons -- firstly computers allowed creation of many other things which previously required too complex calculations, such as highly complex planes, space rockets and undreamed of factories (and, of course, yet more powerful computers which is why we've seen the exponential growth in computer power),

they also allow us to crunch extreme volumes of data and accelerate science; secondly they offered extremely advanced work tools like robots, virtual 3D visualizations, artificial intelligence and physics simulators, and they also gave us high quality, cheap multimedia and entertainment like games -- with computers anyone can shoot video, record music, carry around hundreds of movies in his pocket or fly a virtual plane. Most important however is probably the fact that computers enabled the Internet -- by this they forever changed the world.

We can divide computers based on many attributes, e.g.:

- by continuous or discrete **representation of data**: digital vs analog
- by **hardware technology**: electronic ("lightning in sand"), mechanical, quantum, biological etc.
- by **purpose**: special purpose vs general purpose, personal, server, calculator, embedded, workstation, supercomputers, gaming computer etc.
- by **programmability**: non-programmable, partially or fully programmable
- by the theoretical **model of computation** it is based on: Turing machine, lambda calculus etc.
- by **computational power**: how difficult problems the computer is able to solve, i.e. where in the Chomsky hierarchy it stands (typically we want Turing complete computers)
- by **other criteria**: price, reliability, durability etc.



On the left typical personal computer, with case, monitor, keyboard, mouse and speakers; on the right a pocket mechanical calculator of the Curta type.

Computers are theoretically studied by computer science. The kind of computer we normally talk about consists of two main parts:

- **hardware**: physical parts
- **software**: programs executed by the hardware, made by programmers

The power of computers is mathematically limited, Alan Turing mathematically proved that there exist problems that can never be completely solved by any algorithm, i.e. there are problems a computer (including our brain) will never be able to solve (even if solution exists). This is related to the fact that the power of mathematics itself is limited in a similar way (see Godel's theorems). Turing also invented the theoretical model of a computer called the Turing machine. Besides the mentioned theoretical limitation, many solvable problems may take too long to compute, at least with computers we currently know (see computational complexity and P vs NP).

And let's also mention some curious statistics and facts about computers as of the year 2024. The first computer in modern sense of the word is frequently considered to have been the Analytical Engine designed in 1837 by an Englishman Charles Babbage, a general purpose mechanical computer which he however never constructed. After this the computers such as the Z1 (1938) and Z3 (1941) of a German inventor Konrad Zuse are considered to be the truly first "modern" computers. Shortly after the year 2000 the number of US households that had a computer surpassed 50%. The fastest supercomputer of today is Frontier (Tennessee, USA) which achieved computation speed of 1.102 exaFLOPS (that is over 10^{18} floating point operations per second) with power 22.7 MW, using the Linux kernel (like all top 500 supercomputers). Over time transistors have been getting much smaller -- there is the famous **Moore's law** which states that number of transistors in a chip doubles about every two years. Currently we are able to manufacture

transistors as small as a few nanometers and chips have billions of them. { There's some blurriness about exact size, apparently the new "X nanometers" labels are just marketing lies. ~drummyfish }

Typical Computer

Computers we ordinarily talk about in everyday conversations are electronic digital mostly personal computers such as desktops and laptops, possibly also cell phones, tablets etc.

Such a computer consists of some kind of case (chassis), internal hardware plus peripheral devices that serve for input and output -- these are for example a keyboard and mouse (input devices), a monitor (output device) or harddisk (input/output device). The internals of the computer normally include:

- **motherboard**: The main electronic circuit of the computer into which other components are plugged and which creates the network and interfaces that interconnect them (a chipset). It contains slots for expansion cards as well as connectors for external devices, e.g. USB. In a small memory on the board there is the most basic software (firmware), such as BIOS, to e.g. enable installation of other software. The board also carries the clock generator for synchronization of all hardware, heat sensors etc.
- **CPU** (central processing unit): Core of the computer, the chip plugged into motherboard that performs general calculations and which runs programs, i.e. software.
- **RAM/working memory/main memory**: Lower capacity volatile (temporary, erased when powered off) working memory of the computer, plugged into motherboard. It is used as a "pen and paper" by the CPU when performing calculations.
- **disk**: Non-volatile (persisting when powered off) large capacity memory for storing files and other data, connected to the motherboard via some kind of bus. Different types of disks exist, most commonly hard disks and SSDs.
- **expansion cards (GPU, sound card, network card, ...)**: Additional hardware cards plugged into motherboard for either enabling or accelerating specific functionality (e.g. GPU for graphics etc.).
- **PSU** (power supply unit): Converts the input electrical power from the plug to the electrical power needed by the computer.
- other things like fans for cooling, batteries in laptops etc.

Notable Computers

Here is a list of notable computers.

{ Some nice list of ancient computers is here: https://xnumber.com/xnumber/frame_malbum.htm. ~drummyfish }

name	year	specs (max, approx)	comment
<u>brain</u>	-500M	86+ billion neurons	biological computer, developed by nature
<u>abacus</u>	-2500		one of the simplest digital counting tools
Antikythera mechanism	-125	~30 gears, largest with 223 teeth	1st known analog comp., by Greeks (mechanical)
<u>slide rule</u>	1620		simple tool for multiplication and division
Shickard's calculating clock	1623	17 wheels	1st known calculator, could multiply, add and sub.
<u>Arithmometer</u>	1820	6 digit numbers	1st commercial calculator (add, sub., mult.)
Difference Engine	1822	8 digit numbers, 24 axles, 96 wheels	mech. digital comp. of polynomials, by Babbage
Analytical Engine design	1837	~16K RAM, 40 digit numbers	1st general purpose comp, not realized, by Babbage
<u>nomogram</u>	1884		graphical/geometrical tools aiding computation
<u>Z3</u>	1941		

name	year	specs (max, approx)	comment
		176B RAM, CPU 10Hz 22bit 2600 relays	1st fully programmable electronic digital computer
<u>ENIAC</u>	1945	~85B RAM, ~5KHz CPU, 18000 vaccum tubes	1st general purpose computer
<u>PDP 11</u>	1970	4M RAM, CPU 1.25Mhz 16bit	legendary <u>mini</u>
<u>Apple II</u>	1977	64K RAM, 1MHz CPU 8bit	popular TV-attached home computer by Apple
<u>Atari 800</u>	1979	8K RAM, CPU 1.7MHz 8bit	popular TV-attached home computer by Atari
<u>VIC 20</u>	1980	32K RAM, 1MHz CPU 8bit, 20K ROM	successful TV-connected home computer by Commodore
<u>IBM PC</u>	1981	256K RAM, CPU 4.7MHz 16bit, BASIC, DOS	1st personal computer as we know it now, modular
<u>Commodore 64</u>	1982	64K RAM, 20K ROM, CPU 1MHz 8bit	very popular TV-connected home computer
<u>ZX Spectrum</u>	1982	128K RAM, CPU 3.5MHz 8bit, 256x192 screen	successful UK TV-connected home comp. by Sinclair
<u>NES/Famicom</u>	1983	2K RAM, 2K VRAM, CPU 1.7MHz 8bit, PPU	TV-connected Nintendo game console
<u>Macintosh</u>	1984	128K RAM, CPU 7MHz 32bit, floppy, 512x342	very popular personal computer by Apple
<u>Amiga</u>	1985	256K RAM, 256K ROM, CPU 7MHz 16bit, AmigaOS	personal computer by Commodore, ahead of its time
<u>NeXT</u>	1988	8M RAM, 256M drive, CPU 25MHz 32bit, NeXTSTEP OS	famous workstation, used e.g. for Doom dev.
<u>SNES</u>	1990	128K RAM, 64K VRAM, CPU 21MHz 16bit	game console, NES successor
<u>PlayStation</u>	1994	2M RAM, 1M VRAM, CPU 33MHz 32bit, CD-ROM	popular TV-connected game console by Sony
<u>TI-80</u>	1995	7K RAM, CPU 980KHz, 48x64 1bit screen	famous programmable graphing calculator
<u>Deep Blue</u>	1995	30 128MHz CPUs, ~11 GFLOPS	1st computer to defeat world chess champion
<u>Nintendo 64</u>	1996	8M RAM, CPU 93MHz 64bit, 64M ROM cartr.	famous TV-connected game console
<u>GameBoy Color</u>	1998	32K RAM, 16K VRAM, CPU 2MHz 8bit, 160x144	handheld gaming console by Ninetendo
<u>GameBoy Advance</u>	2001	~256K RAM, 96K VRAM, CPU 16MHz 32bit ARM, 240x160	successor to GBC
<u>Xbox</u>	2001	64M RAM, CPU 733MHz Pentium III	TV-connected game console by Micro\$oft
<u>Nintendo DS</u>	2004	4M RAM, 256K ROM, CPU ARM 67MHz, touchscreen	famous handheld game console by Nintendo
<u>Nintendo Wii</u>	2006	24M RAM, 512M ROM, SD, CPU PPC 729M	famous family TV console with "stick" controllers
<u>iPhone</u> (aka spyphone)	2007	128M RAM, CPU ARM 620MHz, GPU, cam., Wifi, 480x320	1st of the harmful Apple "smartphones"
<u>ThinkPad X200</u>	2008	8G RAM, CPU 2.6GHz, Wifi	legendary laptop, great constr., freedom friendly
<u>ThinkPad T400</u>	2008	8G RAM, CPU 2.8GHz, Wifi	legendary laptop, great constr., freedom friendly
<u>Raspberry Pi 3</u>	2016	1G RAM, CPU 1.4GHz ARM, Wifi	very popular tiny inexpensive SBC

name	year	specs (max, approx)	comment
Arduboy	2016	2.5K RAM, CPU 16MHz AVR 8bit, 1b display	tiny Arduino open console
Pokitto	2017	36K RAM, 256K ROM, CPU 72MHz ARM	indie educational open console
Raspberry Pi 4	2019	8G RAM, CPU 1.5GHz ARM, Wifi	tiny inexpensive SBC, usable as desktop
Frontier	2021	9000+ 64 2GHz CPUs, 37000+ GPUs	fastest supercomputer to date, 1st with 1+ exaFLOPS
Deep Thought			fictional computer from Hitchhiker's Guide ...
HAL 9000			fictional AI computer (2001: A Space Odyssey)
PD computer			planned LRS computer
Turing machine			important theoretical computer by Alan Turing

TODO: mnt reform 2, pinephone, 3DO, ti-89, quantum?

comun

Comun

Comun is a beautiful, greatly minimalist programming language made by [drummyfish](#), based on his ideals of good, selfless technology known as less retarded software (LRS), of which it is now considered the official programming language, though still a greatly work in progress one. In the future it should gradually replace C as the preferred LRS language, however let's keep in mind the language is still highly experimental and work in progress, it may yet change more or less. The language has been inspired mainly by Forth but also C, brainfuck and other ones. Though already usable, it is still in quite early stages of development; currently there is a suckless implementation of comun in C and a number of supplemental materials such as a specification, tutorial and some example programs. The project repository is currently at <https://codeberg.org/drummyfish/comun>. The aim now is to make a self hosted implementation, i.e. write comun in comun. Though very young, **comun is probably already the best programming language ever conceived :-)**

{ NOTE: I found a language on esolang wiki called *Minim* that looks a bit similar to comun, however it looks a bit sucky. Anyway it should be researched more. ~drummyfish }

The language is intended to be the foundation of a completely new, non-capitalist computer technology built from the ground up, which should culminate in the creation of the LRS much desired public domain computer. This technology is derived from the model of an ideal society and as such will aim for completely different goals (such as helping all living beings as much as possible without enslaving them) and values; this makes comun astronomically different in philosophy and design of the shitty, toxic capitalist joke languages such as C++ and Rust which pursue fascism, enslavement of humans to the productivity cult etc.

A quick sum up is following: comun is minimalist, low level with minimum abstraction, portable, imperative and stack-based, using reverse Polish notation. It can be **both compiled and interpreted**. There are **only primitive integer data types** (native integer size by default with possibility to specify exact width where necessary, signed/unsigned interpretation is left to the programmer) and **optional pointers** that can be used as variables, for managing multiple stacks, creating arrays etc. Its **specification can fit on a sheet of paper** and is **completely public domain** under CC0 (as is its current implementation). It has **no standard library**. There are **no English keywords**; commands are rather very short (mostly 1 to three symbols) math-like symbols. Source code only allows ASCII symbols (no unicode). There is an **optional preprocessor that uses comun itself** (i.e. it doesn't use any extra language). **Functions and recursion** are supported. Many features of the language are optional and never burden the programmer if he doesn't use them. Simplified versions of the language (minicomun and microcomun) are also specified.

Examples

Here is a very short showcase of comun code, demonstrating some common functions:

```
max: <' ? >< . ^ .      # takes maximum of two values
max3: max max .         # takes maximum of three values

# recursive factorial
factR:
  '?'
  $0 -- factR *
  ;
  ^ 1
  .
  .

# iterative factorial
factI:
  $0 --

  '@'
  >< $1 * ><
  --
  .
  ^
  .
```

The following is a quine in comun:

```
0 46 32 34 S 34 32 58 83 S --> S: "0 46 32 34 S 34 32 58 83 S --> " .
```

The following code translates brainfuck to comun (proving comun really is Turing complete):

```
0 "$>0 " -->

@@
<? ?
<-

$0 "+" = $1 "-" = | ?
$0 -> $0 -> " " ->
.

$0 "<" = $1 ">" = | ?
"$" -> $0 -> "0" -> " " ->
.

$0 "." = ?
0 "->' " -->
.

$0 ", " = ?
0 "$<0 <- " -->
.

$0 91 = ? # left bracket
0 "@' " -->
.

$0 93 = ? # right bracker
0 ". " -->
.
^
;
!@
.
.
```


TODO: more, code examples, compare the above with C, ...

See Also

- [uxn](#)

consumerism

Consumerism

TODO: actual normal article possibly

Rant

{ Here I'll leave the rant I've written when I was kinda stressed. ~drummyfish }

CONSUME YOU FUCKING IDIOTIC BITCH YOU DON'T EVEN HAVE THE LATEST AI RAYTRACING ENCRYPTION GPUUUUUUUUU 1080K WIRELESS GAYMING MONITOR WITH BLOCKCHAIN BUY IT BUYYYYYY IT YOU IDIOT.
--[capitalism](#) { Unironically this is literally how ads of Alza.cz (the most successful tech store in Czech Republic) are. There's this unbelievably annoying green motherfucker that just yells over and over from the TVs and radios things like "BUY THIS BUY THIS BUUUUUUUUUUUUUUUUY ITTTTITTTTTT ITSSSS ON DISCOUUUUUUUNT DICSOUUUUUUUUNTTTTT", it makes me suicidal, everyone I ever met hates it to death. Also everyone I ever met buys stuff from them. ~drummyfish }

Consumerism (also consoomerism) is a built-in "feature" of [capitalism](#) that says EVERYTHING HAS TO BE CONSUMED on a regular short-term basis so as to keep CONSTANT [P.R.O.G.R.E.S.S.](#) ^TM(c)/PERSONAL DEVELOPMENT GROWTHP/PRODUCTIVITY^tm^tm^tm^tm, even things that in theory could last decades to generations such as houses, cars, computers, software, even just INFORMATION etc. Yes, we could make nice durable machines that wouldn't break and would serve a man for generations, we could write a [finished](#) operating system that would work and be useful, but that wouldn't be good for the seller if he only sold the thing once in a hundred years, would it? ALERT ALERT: BAD FOR CAPITAL. He wants to sell the thing and then PROFIT from it every day as he lies on the beach being fucked by 10 billion whore lolitas, so the thing has to break regularly and just demand to be replaced once in a year or so (see [planned obsolescence](#)) -- haha, actually you know what would be best? WHAT IF :D WHAT IF WE RAPE THE CUSTOMER EVERY DAI, LMAOOOOOO What IF THERE ARE NO PRODUCTS BUT PRODUCT ARE ACTUALLY JUST [SERVICES](#) :DDDDD LMAO THEN NO ONE CAN OWN ANYTHING, YOUR CAR AND YOUR TOOTHBRUSH IS JUST A SUBSCRIPTION LOOOOL, it just stops running if you stop paying. Why do this? Because in capitalism [economy](#) MUSTN'T STOP ULTRA EXPONENTIALLY EXPLODING EVERY TRILLISECOND and EVERYONE MUST HAVE 10000 BILLION [JOBS](#) ELSE THE POOR WORKER LOSES THE MEANING OF LIFE like the neanderthals who lacked the good capitalist overlords that assure the basic human need of ultraexponential personal growth and all killed themselves, also the same with stupid lazy animals. So capitalism has to constantly GROOOOOOOOOOOOOOOOOOOOOOOOOOWWWWWW FOR ANY COST JUST GROW GROW GROW GROW GROW GROOOOOOOOOOOOOOOOOOWWWWWWWWWWWW -- is it good? No, but it's called PWOGWEESSS so people LOVE IT, people will shit themselves and suffocate their mouths with their shit just to hear the word [PWOGWEEEEEES](#) (or alternatively UPDATE or ANTIPEDOPHILE PROTECTION), if a politician says PWEGWESS enough times in his speech the crowd will just start sucking his dick on the stage and he will win the elections by 130% majority. [LMAOOOOO](#) WHAT IF we make [updates a kind of consumerist product](#), LOL WHAT IF one group of people build houses one day and the other group destroys them the other day and so on so on, it's INFINIITEEEEEEEEEEEEEEEEEEE JOBS LOL :D I should fucking get into politics.

TODO

copyfree

Copyfree

Copyfree Initiative is yet another nonprofit group trying to create and maintain its own definition and standardization of "free as in freedom". Its website is <https://copyfree.org/>, the symbol they use is F in a circle. Similarly to e.g. FSF copyfree maintains a list of approved and rejected licenses; the main characteristics of the group are great opposition of copyleft in favor of permissive licenses, which is good, however the group justifies its operation by "helping business", i.e. it's most likely just another unethical capitalist organization that will ultimately stand against people and once/if becomes successful will sell its soul to the highest bidder. Copyfree was founded by Chad Perrin and has existed at least since 2007 (according to internet archive).

See Also

- FSF
 - OSI
 - suckless
 - GNG
 - Creative Commons
 - Bitreich
 - LRS
-

copyleft

Copyleft

Copyleft (also share-alike) is a concept of allowing sharing and modifications of intellectual works (such as pictures, music or computer programs) on the legal condition that others will share it under the same terms (i.e. that they will also allow the work's further free sharing and modification etc.); it was created by the critics of copyright as a "more sane" take on this concept. The symbol of copyleft is a mirrored copyright symbol, i.e. horizontally flipped C in circle (C looking "to the left", Unicode U+1F12F). Copyleft is widely utilized by some proponents of free (as in freedom) software and culture to legally (i.e. with a license) ensure this software/art and its modifications will always remain free (as in freedom), however other camps of freedom proponents argue that copyleft is still too restrictive and share their works under even more relaxed legal conditions. Copyleft kind of hacks copyright to de-facto remove copyright (the monopoly it creates) by its own power. Typical examples of copyleft licenses are the GPL (mostly used for software) and CC BY-SA (mostly used for non-software works).

Copyleft has been by its mechanisms likened to a virus because once it is applied to a certain work, it "infects" it and will force its conditions on any descendants of that work, i.e. it will spread itself -- the word virus here bears less of a negative connotation, at least to some who see it as a "good virus".

For free/open-source software the alternative to copyleft is so called **permissive** licensing which (same as with copyleft) grants all the necessary freedom rights, but, unlike copyleft, does NOT require further modified versions to grant these rights as well. This allows free software being forked and developed into proprietary software and is what copyleft proponents criticize. However, both copyleft and permissive licensing are free as in freedom.

In the FOSS world there is a huge **battle between the copyleft camp and permissive camp** (our LRS advocates permissive licenses with a preference for 100% public domain). These debates go beyond mere technology and law for the basic disagreement lies in whether freedom should be forced and if forced freedom really is freedom, thereby getting into questions of politics, ideologies, philosophy, morality and ethics. Some groups opposing copyleft include copyfree, GNG and LRS.

Issues With Copyleft

In the great debate of copyleft vs permissive free licenses we, as technological anarchists who oppose any "intellectual property" laws and their enforcement, stand on the permissive side. Here are some reasons for why we reject copyleft:

- It **burdens the reuser of the work by requiring him to do something extra** -- while a public domain and many permissive licensed works can simply be taken and used without taking any extra action, just as it should ideally be, a work under copyleft requires its user to take an action, for example copying the license file (and then forever making sure it doesn't get lost), giving credit etc. While one may think this is not such a big deal, it's a form of friction that can get in the way of creativity, especially when combining many works under possibly different copyleft licenses which suddenly becomes quite cumbersome to handle.
- By adopting copyleft one is **embracing and supporting the copyright laws and perpetuating the capitalist ways** ("marrying the lawyers") because copyleft relies on and uses copyright laws to function; to enforce copyleft (prevent "disallowed" use) one has to make a legal action (while with permissive license we simply basically give up the rights to make a legal action). Copyleft chooses to play along with the capitalist bullshit intellectual property game and threatens to fight and use force and bullying in order to enforce *correct* usage of information.
- In a way it is **bloat**. Copyleft introduces **legal complexity**, friction and takes programmers' head space (every programmer has to study a bit of copyright law nowadays due to such BS), especially considering that copyleft is also probably largely ineffective as **detecting its violation and actual legal enforcement is difficult, expensive and without a guaranteed positive outcome** (FSF encourages programmers to hand over their copyright to them so they can defend their programs which just confirms existence and relevance of this issue). The effort spent on dealing with this is a wasted human time. Sure, corporations can probably "abuse" permissive (non-copyleft) software easier, but we argue that this is a problem whose roots lie in the broken basic principles of our society (capitalism) and so the issue should be addressed by improving our socioeconomic system rather than by bullshit legal techniques that just imperfectly and many times completely ineffectively try to cure the symptoms while strengthening the system's mechanisms.
- **The scope of copyleft is highly debatable, introducing doubt/uncertainty** (which is why we have different kind of copyleft such as *strong*, *weak*, *network* etc.). I.e. it can't be objectively said what exactly should classify as violation of copyleft AND increasing copyleft scope leads to copylefted software being practically unusable. You may say "so what", but in law clarity is extremely important, it may also discourage people because they don't really know what they sign up for, commercial use may also be discouraged by this for the same reason which may have a similar effect to a non-free license that downright disallows commercial use. Consider this **example**: Linux is copylefted which means we can't create a proprietary version of Linux, nevertheless we can create a proprietary operating system of which Linux is part (e.g. Android in which its proprietary app store makes it de-facto owned by Google), and so Linux is effectively used as a part of proprietary software. **So copyleft can really be bypassed** (see e.g. bloat monopoly). One might try to increase the copyleft scope here by saying "*everything Linux ever touches has to be free software*" which would however render Linux unusable on practically any computer as most computers contain at least some small proprietary software and hardware. The restriction would be too great. You may of course try to combat the giants further until eternity, but then you are wasting your life being a shitty lawyer rather than doing useful programming.
- **Copyleft drags people into activism, leaving less place for actual creativity** -- one of the best examples is Richard Stallman and his GNU project, who were quite active in programming at their beginning but soon turned more or less just into a political activist group, spending time on petitions, propaganda, certifications (RYF, ...) and generally just the same kind of bullshit fights that capitalists like (often attacking even those who make free software, e.g. the *GNU boot* project for infringing on the name GNU without permission). Stallman himself said "he no longer programs because he has more important things to do". Maybe you say this has nothing to do with copyleft, but it's not a coincidence, copyleft is a mindset of constantly having to "protect" (as opposed to "letting go", the permissive mindset), for example once web applications appeared, the GNU people were suddenly all about having to make new licenses such as AGPL to update to the newest trends in technology and society. Any time a new technology or kind of legal abuse emerges, they have to update their licenses. Choosing copyleft really means choosing to be this kind of warrior and guard of right and wrong, which of course takes away some of your creative potential, with many people just giving in completely.
- **Copyleft licenses have to be complex and ugly** because they have to strictly describe the copyleft scope and include lots of legal boilerplate in order to make them well defendable in court (copyleft is really about preparing for a legal war) -- and as we know, complexity comes with bugs, vulnerabilities, it makes it incomprehensible to common people and imposes many additional burdens. Indeed, we see this in practice: the only practically used copyleft licenses are the various versions of GPL of which all are ugly and have historically shown many faults (which is again evident

from e.g. looking at GPL v1 vs v2 vs v3). This introduces great **license compatibility issues**, headaches for programmers who should rather be spending time programming and other similar bullshit. Permissive licenses on the other hand are simple, clear and well understandable, they aren't as much preparing for a court battle as trying to give other hackers a peace of mind and make them free of legal worries.

- **Copyleft prevents not only inclusion in proprietary software but also in permissive FREE software.** I.e. as a consequence of denying code to corporations collateral damage is done by also denying code to ethical free software that wishes to be distributed without copyleft conditions. Similarly to how proprietary software forces free software programmers to reinvent wheels by rewriting software as free, copyleft forces permissive free software programmers to reinvent wheels and rewrite copylefted code as permissive. In this way copyleft fights not only proprietary software, but also other kinds of free software.
- **There are currently no nice copyleft licenses** -- this of course isn't argument against copyleft itself but it's a practical argument nevertheless. Copyleft nowadays basically means GPL and GPL has a shitton of burdening stuff like requiring credit etc. If you want pure copyleft without anything on top, good luck looking for a license (keep in mind that making your own license or using some obscure, legally untested license is mostly a bad idea).

• ...

See Also

- [permissive](#)
- [copyfree](#)
- [copyright](#)

copyright

Copyright

"When copying is outlawed, only outlaws will have culture." --[Question Copyright website](#)

Copyright (better called *copyrestriction*, *copyrape* or *copywrong*) is one of many types of so called "intellectual property" (IP), a legal concept that allows "ownership", i.e. restriction, censorship and artificial monopoly on certain kinds of information, for example prohibition of sharing or viewing useful information or improving art works. Copyright specifically allows the copyright holder (not necessarily the author) a monopoly (practically absolute power) over art creations such as images, songs or texts, which also include source code of computer programs. Copyright is a capitalist mechanism for creating artificial scarcity, enabling censorship and elimination of the public domain (a pool of freely shared works that anyone can use and benefit from). Copyright is not to be confused with trademarks, patents and other kinds of "intellectual property", which are similarly harmful but legally different. Copyright is symbolized by C in a circle or in brackets: (C), which is often accompanied by the phrase "all rights reserved".

When someone creates something that can even remotely be considered artistic expression (even such things as e.g. a mere collection of already existing things), he automatically gains copyright on it, without having to register it, pay any tax, announce it or let it be known anywhere in any way. He then practically has a full control over the work and can successfully sue anyone who basically just touches the work in any way (even unknowingly and unintentionally). Therefore **any work (such as computer code) without a free license attached is implicitly fully "owned" by its creator** (so called "all rights reserved") and can't be used by anyone without permission. It is said that copyright can't apply to ideas (ideas are covered by patents), only to expressions of ideas, however that's bullshit, the line isn't clear and is arbitrarily drawn by judges; for example regarding stories in books it's been established that the story itself can be copyrighted, not just its expression (e.g. you can't rewrite the Harry Potter story in different words and start selling it).

As if copyright wasn't bad enough of a cancer, **there usually exist extra oppressive copyright-like restrictions called related rights or neighboring rights such as "moral rights", "personal rights" etc.** Such "rights" differ a lot by country and can be used to restrict and censor even copyright-free works. This is a stuff that makes you want to commit suicide. Waivers such as CC0 try to waive copyright as well as neighboring rights (to what extent neighboring rights can be waived is debatable though).

The current extreme form of copyright (as well as other types of IP such as software patents) has been highly criticized by many people, even those whom it's supposed to "protect" (small game creators, musicians etc.). Strong copyright laws basically benefit mainly corporations and "trolls" on the detriment of everyone else. It smothers creativity and efficiency by prohibiting people to reuse, remix and improve already existing works -- something that's crucial for art, science, education and generally just making any kind of progress. Most people are probably for *some* form of copyright but still oppose the current extreme form which is pretty crazy: **copyright applies to everything without any registration or notice and last usually 70 years (!!!) AFTER the author has died (!!!)** and is already rotting in the ground. This is 100 years in some countries. In some countries it is not even possible to waive copyright to own creations -- just think about what kind of twisted society we are living in when it PROHIBITS people from making a selfless donation of their own creations to others. Some people, including us, are against the very idea of copyright (those may either use waivers such as CC0 or unlicense or protest by not using any licenses and simply ignoring copyright which however will actually discourage other people from reusing their works). Though copyright was originally intended to ensure artists can make living with their works, it has now become the tool of states and corporations for universal censorship, control, bullying, surveillance, creating scarcity and bullshit jobs; states can use copyright to for example take down old politically inconvenient books shared on the Internet even if such takedowns do absolute not serve protection of anyone's living but purely political interests.

Prominent critics of copyright include Lawrence Lessig (who established free culture and Creative Commons as a response), Nina Paley and Richard Stallman. There are many movements and groups opposing copyright or its current form, most notably e.g. the free culture movement, free software movement, Creative Commons etc.

The book *Free Culture* by Lessig talks, besides others, about how copyright has started and how it's been shaped by corporations to becoming their tool for monopolizing art. The concept of copyright has appeared after the invention of printing press. The so called *Statute of Anne* of 1710 allowed the authors of books to control their copying for **14 years** and only after **registartion**. The term could be prolonged by another 14 years if the author survived. The laws started to get more and more strict as control of information became more valued and eventually the term grew to **life of author plus 70 years**, without any need for registration or deposit of the copy of the work. Furthermore with new technologies, the scope of copyright has also extended: if copyright originally only limited *copying* of books, in the Internet age it started to cover basically any use, as any manipulation of digital data in the computer age requires making local copies. Additionally the copyright laws were passing despite being unconstitutional as the US constitution says that copyright term has to be finite -- the corporations have found a way around this and simply regularly increased the copyright's term, trying to make it de-facto infinite (technically not infinite but ever increasing). Their reason, of course, was to firstly forever keep ownership of their own art but also, maybe more importantly, to **kill the public domain**, i.e. prevent old works from entering the public domain where they would become a completely free, unrestricted work for all people, competing with their proprietary art (who would pay for movies if there were thousands of movies available for free?). Nowadays, with corporations such as YouTube and Facebook de-facto controlling most of information sharing among common people, the situation worsens further: they can simply make their own laws that don't need to be passed by the government but simply implemented on the platform they control. This way they are already killing e.g. the right to fair use, they can simply remove any content on the basis of "copyright violation", even if such content would normally NOT violate copyright because it would fall under fair use. This would normally have to be decided by court, but a corporation here itself takes the role of the court. So in terms of copyright, corporations have now a greater say than governments, and of course they'll use this power against the people (e.g. to implement censorship and surveillance).

Copyright rules differ greatly by country, most notably the US measures copyright length from the publication of the work rather than from when the author died. It is possible for a work to be copyrighted in one country and not copyrighted in another. It is sometimes also very difficult to say whether a work is copyrighted because the rules have been greatly changing (e.g. a notice used to be required for some time), sometimes even retroactively copyrighting public domain works, and there also exists no official database of copyrighted works (you can't safely look up whether your creation is too similar to someone else's). All in all, copyright is a huge mess, which is why we choose free licenses and even public domain waivers.

Copyleft (also share-alike) is a concept standing against copyright, a kind of anti-copyright, invented by Richard Stallman in the context of free software. It's a license that grants people the rights to the author's work on the condition that they share its further modification under the same terms, which basically hacks

copyright to effectively spread free works like a "virus".

Copyright does **not** (or at least should not) apply to facts (including mathematical formulas) (even though the formulation of them may be copyrighted), ideas (though these may be covered by patents) and single words or short phrases (these may however still be trademarked) and similarly trivial works. As such copyright can't e.g. be applied to game mechanics of a computer game (it's an idea). It is also basically proven that copyright doesn't cover computer languages (Oracle vs Google). Also even though many try to claim so, copyright does NOT arise for the effort needed to create the work -- so called "sweat of the brow" -- some say that when it took a great effort to create something, the author should get a copyright on it, however this is NOT and must NOT be the case (otherwise it would be possible to copyright mere ideas, simple mathematical formulas, rules of games etc.). Depending on time and location there also exist various peculiar exceptions such as the freedom of panorama for photographs or uncopyrightable utilitarian design (e.g. no one can own the shape of a generic car). But it's never good to rely on these peculiarities as they are specific to time/location, they are often highly subjective, fuzzy and debatable and may even be retroactively changed by law. This constitutes a huge legal bloat and many time legal unsafety. Do not stay in the gray area, try to stay safely far away from the fuzzy copyright line.

A work which is not covered by copyright (and any other IP) -- which is nowadays pretty rare due to the extent and duration of copyright -- is in the **public domain**.

Free software (and free art etc.) is **not** automatically public domain, it is mostly still copyrighted, i.e. "owned" by someone, but the owner has given some key rights to everyone with a free software license and by doing so minimized or even eliminated the negative effects of full copyright. The owner may still keep the rights e.g. to being properly credited in all copies of the software, which he may enforce in court. Similarly software that is in public domain is **not** automatically free software -- this holds only if source code for this software is available (so that the rights to studying and modifying can be executed).

Copyright encourages murder. The sooner the author dies, the sooner his material will run out of copyright, so if you want some nice work to enter public domain soon, you are literally led by the law to try for him to die as soon as possible.

See Also

- bullshit
- free culture
- copyleft
- derivative work
- fair use
- creative commons
- license
- patent
- trademark
- public domain
- intellectual property

corporation

Corporation

Corporation is basically a huge company that doesn't have a single owner but is rather managed by many shareholders. Corporations are one of the most powerful, dangerous and unethical entities that ever came into existence -- their power is growing, sometimes even beyond the power of states and their sole goal is to make as much profit as possible without any sense of morality. Existence of corporations is enabled by capitalism. Examples of corporations are Micro\$oft, EA, Apple, Amazon, Walmart, Te\$la, McDonald\$, Facebook etc. Every startup is an aspiring corporation, so never support any startup.

The most basic fact to know about corporations is that **100% of everything a corporation ever does is done 100% solely for maximizing its own benefit for any cost, with no other reason, with 0 morality and without any consideration of consequences**. If a corporation could make 1 cent by raping

1000000000 children and get away with it, it would do so immediately without any hesitation and any regret. This is very important to keep in mind. Now try to not get depressed at realization that corporations are those to whom we gave power and who are in almost absolute control of the world.

Corporation is not a human, it has zero sense of morality and no emotion. The most basic error committed by retards is to reply to this argument with "but corporations are run by humans". This is an extremely dangerous argument because somehow 99.9999999999999999% people believe this could be true and accept it as a comforting argument so that they can continue their daily lives and do absolutely nothing about the disastrous state of society. The argument is of course completely false for a number of reasons: firstly corporations exclusively hire psychopaths for manager roles -- any corporation that doesn't do this will be eliminated by natural selection of the market environment because it will be weaker in a fight against other corporations, and its place will be taken by the next aspiring corporation waiting in line. Secondly corporations are highly sophisticated machines that have strong mechanisms preventing any ethical behavior -- for example division of labor in the "just doing my job"/"everyone does it" style allows for many people collaborating on something extremely harmful and unethical without any single one feeling responsibility for the whole, or sometimes without people even knowing what they are really collaborating on. This is taken to perfection by corporations not even having a single responsible owner -- there is a group of shareholders, none of whom has a sole responsibility, and there is the CEO who is just a tool and puppet with tied hands who is just supposed to implement the collective bidding of shareholders. Of course, most just don't care, and most don't even have a choice. Similar principles allowed for example the Holocaust to happen. Anyone who has ever worked anywhere knows that managers always pressure workers just to make money, not to behave more ethically -- of course, such a manager would be fired on spot -- and indeed, workers that try to behave ethically are replaced by those who make more money, just as companies that try to behave ethically in the market are replaced by those that rather make money, i.e. corporations. This is nothing surprising, the definition of capitalism implies existence of a system with Darwinian evolution that selects entities that are best at making money for any cost, and that is exactly what we are getting. To expect any other outcome in capitalism would be just trying to deny mathematics itself.

A corporation is made to exploit people just as a gun is made to kill people. When a corporation commits a crime, it is not punished like a human would be, the corporation is left to exist and continue doing what it has been doing -- a supposed "punishment" for a corporation that has been caught red handed committing a crime is usually just replacing whoever is ruled to be "responsible", for example the CEO, which is of course ridiculous, the guy is just replaced with someone else who will do exactly the same. This is like trying to fix the lethal nature of a weapon by putting all the blame on a screw in the weapon, then replacing the screw with another one and expecting the weapon to no longer serve killing people.

It is always better for a corporation to not exist than vice versa. The proof is following:

1. It is better to have no corporation than an evil corporation.
2. Corporation is always evil.
3. Therefore it is always better for a corporation to not exist. QED

There is probably nothing we can do to stop corporations from taking over the world and eventually eliminating humans, we have probably passed the capitalist singularity.

TODO

cos

Cosine

Cosine (shortened *cos*) is an important mathematical function; for more see the article about sine.

countercomplex

Countercomplex

"True progress is about deepness and compression instead of maximization and accumulation." -Viznut

Countercomplex is a [blog](http://countercomplex.blogspot.com/) (running since 2008) of a Finnish [hacker](#) and [demoscener](#) [Viznut](#), criticizing technological complexity/[bloat](#) and promoting [minimalism](#) as a basis of truly good technology. It is accessible at <http://countercomplex.blogspot.com/>.

c_pitfalls

C Pitfalls

C is a powerful language that offers almost absolute control and maximum performance which necessarily comes with responsibility and danger of shooting oneself in the foot. Without knowledge of the pitfalls you may well find yourself fallen into one of them.

This article will be focused on C specific/typical pitfalls, but of course C also comes with general [programming](#) pitfalls, such as those related to [floating point](#), [concurrency](#), bugs such as [off by one](#) and so on -- indeed, be aware of these ones too.

Unless specified otherwise, this article supposes the C99 standard of the C language.

Generally: be sure to check your programs with tools such as [valgrind](#), [splint](#), [cppcheck](#), UBSan or ASan, and turn on compiler auto checks (-Wall, -Wextra, -pedantic, ...), it's quick, simple and reveals many bugs!

Undefined/Unspecified Behavior

Undefined (completely unpredictable), unspecified (safe but potentially differing) and implementation-defined (consistent within implementation but potentially differing between them) behavior poses a kind of unpredictability and sometimes non-intuitive, tricky behavior of certain operations that may differ between compilers, platforms or runs because they are not exactly described by the language specification; this is mostly done on purpose so as to allow some implementation freedom which allows implementing the language in a way that is most efficient on given platform. One has to be very careful about not letting such behavior break the program on platforms different from the one the program is developed on. Note that tools such as [cppcheck](#) can help find undefined behavior in code. Description of some such behavior follows.

There are tools for detecting undefined behavior, see e.g. [clang's UBSan](#) (<https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>).

Data type sizes including int and char may not be the same on each platform. Even though we almost take it for granted that char is 8 bits wide, in theory it can be different (even though `sizeof(char)` is always 1). Int (and unsigned int) type width should reflect the architecture's native integer type, so nowadays it's mostly 32 or 64 bits. To deal with these differences we can use the standard library `limits.h` and `stdint.h` headers.

No specific [endianness](#) or even [encoding of numbers](#) is specified. Nowadays little endian and [two's complement](#) is what you'll encounter on most platforms, but e.g. [PowerPC](#) uses big endian ordering.

Order of evaluation of operands and function arguments is not specified. I.e. in an expression or function call it is not defined which operands or arguments will be evaluated first, the order may be completely random and the order may differ even when evaluating the same expression at another time. This is demonstrated by the following code:

```
#include <stdio.h>

int x = 0;

int a(void)
{
    x += 1;
    return x;
}

int main(void)
```



```
{
    printf("%d %d\n",x,a()); // may print 0 1 or 1 1
    return 0;
}
```

Overflow behavior of signed type operations is not specified. Sometimes we suppose that e.g. addition of two signed integers that are past the data type's limit will produce two's complement overflow (wrap around), but in fact this operation's behavior is undefined, C99 doesn't say what representation should be used for numbers. For portability, predictability and preventing bugs **it is safer to use unsigned types** (but safety may come at the cost of performance, i.e. you prevent compiler from performing some optimizations based on undefined behavior).

Bit shifts by type width or more are undefined. Also bit shifts by negative values are undefined. So e.g. `x >> 8` is undefined if width of the data type of `x` is 8 bits or fewer.

Char data type signedness is not defined. The signedness can be explicitly "forced" by specifying signed char or unsigned char.

Floating point results are not precisely specified, no representation (such as IEEE 754) is specified and there may appear small differences in float operations under different machines or e.g. compiler optimization settings -- this may lead to nondeterminism.

Memory Unsafety

Besides being extra careful about writing memory safe code, one needs to also know that **some functions of the standard library are memory unsafe**. This is regarding mainly string functions such as `strcpy` or `strlen` which do not check the string boundaries (i.e. they rely on not being passed a string that's not zero terminated and so can potentially touch memory anywhere beyond); safer alternatives are available, they have an `n` added in the name (`strncpy`, `strnlen`, ...) and allow specifying a length limit.

Be careful with pointers, pointers are hard and prone to errors, use them wisely and sparingly, assign NULLs to freed pointers and so on and so forth.

Watch out for memory leaks, try to avoid dynamic allocation (static/automatic allocation mostly suffices) and if you have to use it, simplify it as much as you can and additionally double and triple check everything (manually as well as with tools like valgrind).

Different Behavior Between C And C++ (And Different C Standards)

C is **not** a subset of C++, i.e. not every C program is a C++ program (for simple example imagine a C program in which we use the word `class` as an identifier: it is a valid C program but not a C++ program). Furthermore a C program that is at the same time also a C++ program may behave differently when compiled as C vs C++, i.e. there may be a semantic difference. Of course, all of this may also apply between different standards of C, not just between C and C++.

For portability sake it is good to try to write C code that will also compile as C++ (and behave the same). For this we should know some basic differences in behavior between C and C++.

One difference is e.g. in that type of character literals is `int` in C but `char` in C++, so `sizeof('x')` will likely yield different values.

Another difference lies for example in pointers to string literals. While in C it is possible to have non-const pointers such as

```
char *s = "abc";
```

C++ requires any such pointer to be `const`, i.e.:

```
const char *s = "abc";
```

C++ generally has stronger typing, e.g. C allows assigning a pointer to void to any other pointer while C++ requires explicit type cast, typically seen with malloc:

```
int *array1 = malloc(N * sizeof(int));           // valid only in C
int *array2 = (int *) malloc(N * sizeof(int));    // valid in both C and C++
```

C allows skipping initialization (variable declarations) e.g. gotos or switches, C++ prohibits it.

And so on.

{ A quite detailed list is at https://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B. ~drummyfish }

Compiler Optimizations

C compilers perform automatic optimizations and other transformations of the code, especially when you tell them to optimize aggressively (-O3) which is a standard practice to make programs run faster. However this makes compilers perform a lot of magic and may lead to unexpected and unintuitive undesired behavior such as bugs or even the "unoptimization of code". { I've seen a code I've written have bigger size when I set the -Os flag (optimize for smaller size). ~drummyfish }

Aggressive optimization may firstly lead to tiny bugs in your code manifesting in very weird ways, it may happen that a line of code somewhere which may somehow trigger some tricky undefined behavior may cause your program to crash in some completely different place. Compilers exploit undefined behavior to make all kinds of big brain reasoning and when they see code that MAY lead to undefined behavior a lot of chain reasoning may lead to very weird compiled results. Remember that undefined behavior, such as overflow when adding signed integers, doesn't mean the result is undefined, it means that ANYTHING CAN HAPPEN, the program may just start printing nonsensical stuff on its own or your computer may explode. So it may happen that the line with undefined behavior will behave as you expect but somewhere later on the program will just shit itself. For these reasons if you encounter a very weird bug, try to disable optimizations and see if it goes away -- if it does, you may be dealing with this kind of stuff. Also check your program with tools like cppcheck.

Automatic optimizations may also be dangerous when writing multithreaded or very low level code (e.g. a driver) in which the compiler may have wrong assumptions about the code such as that nothing outside your program can change your program's memory. Consider e.g. the following code:

```
while (x)
    puts("X is set!");
```

Normally the compiler could optimize this to:

```
if (x)
    while (1)
        puts("X is set!");
```

As in typical code this works the same and is faster. However if the variable x is part of shared memory and can be changed by an outside process during the execution of the loop, this optimization can no longer be done as it results in different behavior. This can be prevented with the `volatile` keyword which tells the compiler to not perform such optimizations.

Of course this applies to other languages as well, but C is especially known for having a lot of undefined behavior, so be careful.

Other

Watch out for **operator precedence**! Bracket expressions if unsure, or just to increase readability for others.

Preprocessor can give you headaches if you use it in overcomplicated ways -- ifdefs and macros are fine, but too many nesting can create real mess that's super hard to debug. It can also quite greatly slow down

compilation. Try to keep the preprocessing code simple and flat.

Also watch out for this one: `!=` is not `!=` :) I.e. `if (x != 4)` and `if (x =! 4)` are two different things, the first means *not equal* and is usually what you want, the latter is two operations, `=` and `!`, the tricky thing is it also compiles and may work as expected in some cases but fail in others, leading to a very nasty bug.

This is not really a pitfall, rather a headscratcher, but don't forget to link math library with `-lm` flag when using the `math.h` library.

cpp

C++

C++ (also crippled C) is an object-obsessed joke language based on C to which it adds only capitalist features and bloat, most notably object obsession. Most good programmers such as Richard Stallman and Linus Torvalds agree that C++ is hilariously messy and also tragic in that it actually succeeded to become mainstream. The language creator Bjarne Stroustrup himself infamously admitted the language sucks but laughs at its critics because it became successful anyway -- indeed, in a retarded society only shit can succeed. As someone once said, "C++ is not an increment, it is excrement". C++ specification has **over 2000 pages** :D

C++ source code files have the extensions `.cpp` or `.cc` (for "crippled C").

cpu

CPU

WORK IN PROGRESS

Central processing unit (CPU, often just *processor*) is the main, most central part of a computer, the one that performs the computation by following the instructions of the main program; CPU can be seen as the computer's brain. It stands at the center of the computer design -- other parts, such as the main memory, hard disk and input/output devices like keyboard and monitor are present to serve the CPU, CPU is at the top and issues commands to everyone else. A CPU is normally composed of ALU (arithmetic logic unit, the circuit performing calculations), CU (control unit, the circuit that directs the CPU's operation), a relatively small amount of memory (e.g. its registers, temporary buffers and cache, the main RAM memory is NOT part of a CPU!) and possibly some other parts. A specific model of CPU is characterized by its instruction set (ISA, e.g. x86 or Arm, which we mostly divide into CISC and RISC), which determines the machine code it will understand, then its transistor count (nowadays billions), operation frequency or **clock rate** (defining how many instructions per second it can execute, nowadays typically billions; the frequency can also be increased with overclocking), number of cores (determining how many programs it can run in parallel) and also other parameters and "features" such as amount of cache memory, possible operation modes etcetc. We also often associate the CPU with some **number of bits** (called e.g. word size) that's often connected to the data bus width and the CPU's native integer size, i.e. for example a 16 bit CPU will likely have 16 bit integer registers, it will see the memory as a sequence of 16 bit words etc. (note the CPU can still do higher bit operations but they'll typically have to be emulated so they'll be slower, will take more instructions etc.) -- nowadays most mainstream CPUs are 64 bit (to allow ungodly amounts of RAM), but 32 or even 16 and 8 bits is usually enough for good programs. CPU in form of a single small integrated circuit is called *microprocessor*. CPU is not to be confused with MCU, a small single board computer which is composed of a CPU and other parts.

CPU is meant for **general purpose computations**, i.e. it can execute anything reasonably fast but for some tasks, e.g. processing HD video, won't reach near optimum speed, which is why other specialized processing units such as GPUs (graphics processing unit) and sound cards exist. As a general algorithm executing unit CPU is made for executing **linear** programs, i.e. a series of instructions that go one after another; even though CPUs nowadays typically have multiple cores thanks to which they can run several linear programs in parallel, their level of parallelism is still low, not nearly as great as that of a GPU for example. However CPUs are good enough for most things and they are extremely fast nowadays, so a suckless/LRS program will likely

choose to only rely on CPU, knowing CPU will be present in any computer and so that our program will be portable.

Designs of CPUs differ, some may aim for very high performance while other ones may prefer low power consumption or low transistor count -- remember, a more complex CPU will require more transistors and will be more expensive! Of course it will also be harder to design, debug etc., so it may be better to keep it simple when designing a CPU. For this reason many CPUs, e.g. those in embedded microcontrollers, intentionally lack cache, microcode, multiple cores or even a complex instruction pipeline.

WATCH OUT: modern mainstream CPUs (i.e. basically the desktop ones, soon probably mobile ones too) are shit, they are hugely consumerist, bloated (they literally include shit like GPUs and whole operating systems, e.g. Intel's ME runs Minix) and have built-in antifeatures such as backdoors (post 2010 basically all Intel and AMD CPUs, see Intel Management Engine and AMD PSP) that can't be disabled and that allow remote infiltration of your computer by the CPU manufacturer (on hardware level, no matter what operating system you run). You are much better off using a simple CPU if you can (older, embedded etc.).

Details

TODO: diagrams, modes, transistor count history ...

Let's take a look at how a typical CPU works. Remember that anything may differ between CPUs, you can think of doing things differently and many real world CPUs do. Also we may simplify some things here, real world CPUs are complicated as hell.

What does a CPU really do? Basically it just reads instructions from the memory (depending on specific computer architecture this may be RAM or ROM) and does what they say -- these instructions are super simple, often things like "add two numbers", "write a number to memory" and so on. The instructions themselves are just binary data in memory and their format depends on each CPU, or its instruction set (basically a very low level language it understands) -- each CPU, or rather a CPU family, may generally have a different instruction set, so a program in one instruction set can't be executed by a CPU that doesn't understand this instruction set. The whole binary program for the CPU is called machine code and machine code corresponds to assembly language (basically a textual representation of the machine code, for better readability by humans) of the CPU (or better said its instruction set). So a CPU can be seen as a hardware interpreter of specific machine code, machine code depends on the instruction set and programmer can create machine code by writing a program in assembly language (which is different for each instruction set) and then using an assembler to translate the program to machine code. Nowadays mostly two instruction sets are used: x86 and Arm, but there are also other ones, AND it's still not so simple because each instruction set gets some kind of updates and/or has some extensions that may or may not be supported by a specific CPU, so it's a bit messy. For example IA-32 and x86_64 are two different versions of the x86 ISA, one 32 bit and one 64 bit.

The CPU has some internal state (we can see it as a state machine), i.e. it has a few internal variables, called registers; these are NOT variables in RAM but rather in the CPU itself, there is only a few of them (there may be let's say 32) but they are extremely fast. What exactly these registers are, what they are called, how many bits they can hold and what their purpose is depends again on the instruction set architecture. However there are usually a few special registers, notably the **program counter** which holds the address of the currently executed instruction. After executing an instruction program counter is incremented so that in the next step the next instruction will be executed, AND we can also modify program counter (sometimes directly, sometimes by specialized instructions) to jump between instruction to implement branching, loops, function calls etc.

So at the beginning (when powered on) the CPU is set to some initial state, most notably it sets its program counter to some initial value (depending on each CPU, it may be e.g. 0) so that it points to the first instruction of the program. Then it performs so called **fetch, decode, execute** cycle, i.e. it reads the instruction, decodes what it means and does what it says. In simpler CPUs this functionality is hard wired, however more complex CPUs (especially CISC) are programmed in so called microcode, a code yet at the lower level than machine code, machine code execution is programmed in microcode -- microcode is something like "firmware for the CPU" (or a "CPU shader"?), it basically allows later updates and reprogramming of how the CPU internally works. However this is pretty overcomplicated and you shouldn't make CPUs like this.

A CPU works in **clock cycles**, i.e. it is a sequential circuit which has so called *clock* input; on this input voltage periodically switches between high and low (1 and 0) and each change makes the CPU perform another operation cycle. How fast the clock changes is determined by the clock **frequency** (nowadays usually around 3 GHz) -- the faster the frequency, the faster the CPU will compute, but the more it will also heat up (so we can't just set it up arbitrarily high, but we can overclock it a bit if we are cooling it down). **WATCH OUT: one clock cycle doesn't necessarily equal one executed instruction**, i.e. frequency of 1 Hz doesn't have to mean the CPU will execute 1 instruction per second because executing an instruction may take several cycles (how many depends on each instruction and also other factors). The number saying how many cycles an instruction takes is called CPI (cycles per instruction) -- CPUs try to aim for CPI 1, i.e. they try to execute 1 instruction per cycle, but they can't always do it.

One way to try to achieve CPI 1 is by optimizing the *fetch, decode, execute* cycle in hardware so that it's performed as fast as possible. This is typically done by utilizing an instruction **pipeline** -- a pipeline has several stages that work in parallel so that when one instruction is entering e.g. the *decode* stage, another one is already entering the *fetch* stage (and the previous instruction is in *execute* stage), i.e. we don't have to wait for an instruction to be fully processed before starting to process the next one. This is practically the same principle as that of manufacturing lines in factories; if you have a long car manufacturing pipeline, you can make a factory produce let's say one car each hour, though it is impossible to make a single car from scratch in one hour (or imagine e.g. a university producing new PhDs each year despite no one being able to actually earn PhD in a year). This is also why branching (jumps between instructions) are considered bad for program performance -- a jump to different instruction makes the CPU have to throw away its currently preprocessed instruction because that will not be executed (though CPUs again try to deal with this with so called *branch prediction*, but it can't work 100%). Some CPUs even have multiple pipelines, allowing for execution of multiple instructions at the same time -- however this can only be done sometimes (the latter instruction must be independent of the former, also the other pipelines may be simpler and able to only handle simple instructions).

In order for a CPU to be useful it has to be able to perform some **input/output**, i.e. it has to be able to retrieve data from the outside and present what it has computed. Notable ways of performing I/O are:

- Through **memory**: here some parts of memory serve to pass data to the CPU and to retrieve computed results back. For example a keyboard may be mapped to memory so that when certain keys are pressed, the memory bits are set to 1 -- this way a CPU can simply read from memory and know if a key is pressed. Similarly a display may be mapped to memory so that when a CPU writes a value to this address, a pixel appears on the display. Note that this doesn't always have to PHYSICALLY pass through memory, there may be a special circuit that translates e.g. memory access in some address range to signals to hardware etc., but the CPU is using the same instructions it would use for interacting with memory.
- Through **GPIO pins**: CPUs typically have pins that are reserved for general purpose input/output, i.e. we can electronically communicate through them with whatever device we physically connect to those pins. A CPU can set and read voltage to/from those pins e.g. with some special instructions. This may be convenient if we just want to e.g. light up some LED without having to somehow hook it to the main memory.
- **Interrupts**: a CPU can be informed about an external event with an interrupt (see further on).

CPUs often also have a **cache** memory that speeds up communication with the main memory (RAM, ROM, ...), though simpler CPUs may live even without cache of course. Mainstream CPUs even have several levels of cache, called L1, L2 etc. Caches are basically transparent for the programmer, they don't have to deal with them, it's just something that makes memory access faster, however a programmer knowing how a cache works can write code so as to be friendlier to the cache and utilize it better.

Mainstream consumer CPUs nowadays have multiple **cores** so that each core can basically run a separate computation in parallel. The separate cores can be seen kind of like duplicate copies of the single core CPU with some connections between them (details again depend on each model), for example cores may share the cache memory, they will be able to communicate with each other etc. Of course this doesn't just magically make the whole CPU faster, it can now only run multiple computations at once, but someone has to make programs so as to make use of this -- typical use cases are e.g. multitasking operating systems which can run different programs (or rather processes) on each core (note that multitasking can be done even with a single core by rapidly switching between the processes, but that's slower), or multithreading programming languages which may run each thread on a separate core.

Interrupts are an important concept for the CPU and for low level programming, they play a role e.g. in saving power -- high level programmers often don't know what interrupts are, to those interrupts can be likened to "event callbacks". An interrupt happens on some kind of even, for example when a key is pressed, when timer ticks, when error occurred etc. (An interrupt can also be raised by the CPU itself, this is how operating system syscalls are often implemented). What kinds of interrupts there are depends on each CPU architecture (consult your datasheet) and one can usually configure which interrupts to enable and which "callbacks" to use for them -- this is often done through so called **vector table**, a special area in memory that records addresses ("vectors") of routines (functions/subprograms) to be called on specified interrupts. When interrupt happens, the current program execution is paused and the CPU automatically jumps to the subroutine for handling the interrupt -- after returning from the subroutine the main program execution continues. Interrupts are contrasted with **polling**, i.e. manually checking some state and handling things as part of the main program, e.g. executing an infinite loop in which we repeatedly check keyboard state until some key is pressed. However polling is inefficient, it wastes power by constantly performing computation just by waiting -- interrupts on the other hand are a hard wired functionality that just performs a task when it happens without any overhead of polling. Furthermore interrupts can make programming easier (you save many condition checks and memory reads) and mainly **interrupts allow CPU to go into sleep mode** and so save a lot of power. When a CPU doesn't have any computation to do, it can stop itself and go into waiting state, not executing any instructions -- however interrupts still work and when something happens, the CPU jumps back in to work. This is typically what the `sleep/wait` function in your programming language does -- it puts the CPU to sleep and sets a timer interrupt to wake up after given amount of time. As a programmer you should know that you should call this `sleep/wait` function in your main program loop to relieve the CPU -- if you don't, you will notice the **CPU utilization** (amount of time it is performing computations) will go to 100%, it will heat up, your computer starts spinning the fans and be noisy because you don't let it rest.

Frequently there are several **modes** of operation in a CPU which is typically meant for operating systems -- there will usually be some kind of privileged mode in which the CPU can do whatever it wants (this is the mode for the OS kernel) and a restricted mode in which there are restrictions, e.g. on which areas of memory can be accessed or which instructions can be used (this will be used for user program). Thanks to this a user program won't be able to crash the operating system, it will at worst crash itself. Most notably x86 CPUs have the *real mode* (addresses correspond to real, physical addresses) and *protected mode* (memory is virtualized, protected, addresses don't generally correspond to physical addresses).

A CPU may also have integrated some **coprocessors**, though sometimes coprocessors are really a separate chip. Coprocessors that may be inside the CPU include e.g. the FPU (floating point unit) or encryption coprocessor. Again, this will make the CPU a lot more expensive.

TODOOOOOOO: ALU, virtual memory, IP cores, architectures (register, ...), ...

Notable CPUs

UNDER CONSTRUCTION

Here are listed some notable CPUs (or sometimes CPU families or cores).

{ I'm not so great with HW, suggest me improvements for this section please, thanks <3 ~drummyfish }

{ WTF, allthetropes has quite a big list of famous CPUs, isn't it a site about movies?
https://allthetropes.org/wiki/Central_Processing_Unit. ~drummyfish }

TODO: add more, mark CPUs with ME, add features like MMX, FPU, ...

CPU	year	bits (/a)	ISA	~tr. c.	tr. size	freq.	pins	cores	other	notes
Intel 4004	1971	4 / 12	own	2.3 K	10 um	750 K	16	1		1st commercial microproc.
Intel 8008	1972	8 / 14	own	3.5 K	10 um	800 K	18	1		
Intel 8080	1974	8 / 16	own	6 K	6 um	3 M	40	1		

CPU	year	bits (/a)	ISA	~tr. c.	tr. size	freq.	pins	cores	other	notes
AMD Am9080	1975	8 / 16	own	6 K	6 um	4 M	40	1		reverse-eng. clone of i8080
MOS Technology 6502	1975	8 / 16	own	3.5 K	8 um	3 M	40	1		popular, cheap, Atari 2600, C64, ...
Zilog Z80	1976	8 / 16	own	8.5 K	4 um	10 M	40	1		popular
Intel 8086	1978	16 / 20	x86 (x86-16)	29 K	3 um	10 M	40	1		started x86 ISA
Motorola 68000	1979	32 / 24	own (CISC)	68 K			64	1		popular, e.g. Amiga, Mega Drive, ...
Intel (80)286	1982	16 / 24	x86 (x86-16)	130 K	1.5 um	25 M	68	1		
Intel (80)386	1985	32	x86 (IA-32)	275 K	1 um	40 M	132	1		
Intel (80)486	1989	32	x86 (IA-32)	1.6 M	600 nm	100 M	196	1	16 K cache, FPU	1st intel with cache and FPU
AMD Am386	1991	32	x86 (IA-32)	275 K	800 nm	40 M	132	1		clone of i386, lawsuit
Intel Pentium P5	1993	32	x86 (IA-32)	3 M	800 nm	60 M	273	1	16 K cache	starts Pentium line with many to follow
AMD K5	1996	32	x86 (IA-32)	4.3 M	500 nm	133 M	296	1	24 K cache	1st in-house AMD CPU, compet. of Pentium
Intel Pentium II	1997	32	x86 (IA-32)	7 M	180 nm	450 M	240	1	512 K L2 cache, MMX	
ARM7TDMI	1994	32	ARM			100 M		1		ARM core, e.g. GBA, PS2, Nokia 6110 ...
AMD Athlon 1000 Thunderbird	2000	32	x86 (IA-32)	37 M	180 nm	1 G	453	1	~300 K cache	1st 1GHz+ CPU
RAD750	2001	32	PowerPC	10 M	150 nm	200 M	360	1	64 K cache	radiation hard., space (Curiosity, ...)
AMD Opteron	2003	64	x86 (x86-64)	105 M	130 nm	1.6 G	940	1	~1 M cache	1st 64 bit x86 CPU
Intel Pentium D 820	2005	64	x86 (x86-64)	230 M	90 nm	2.8 G	775	2	~2 M cache	1st desktop multi core CPU
Intel Core i5-2500K	2011	64	x86 (x86-64)	1 B	32 nm	3.3 G		4	~6 M cache, ME	
PicoRV32	2015?	32	RISC-V (RV32IMC)			~700 M				simple, free hardware RISV-V core
Apple A9	2015	64	ARM (ARMv8)	2 B	14 nm	1.8 G		2	~7 M cache	iPhones
AMD Ryzen Threadrip. PRO 5995WX	2022	64	x86 (x86-64)	33 B	7 nm	4.5 G	4094	64	~300 M cache, PSP	high end bloat
<u>Talos ES</u>	2023	8	own (RISC)							simple but usable DIY free hardware

CPU	year	bits (/a)	ISA	~tr. c.	tr. size	freq.	pins	cores	other	notes
CPU										

See Also

- GPU
- MCU
- WPU (weird processing unit)

cracker

Cracker

Crackers are either "bad hackers" that break into computer systems or the good people who with the power of hacking remove artificial barriers to obtaining and sharing infomration; for example they help remove DRM from games or leak data from secret databases. This is normally illegal which makes the effort even more admirable.

Cracker is also food.

Cracker is also the equivalent of nigger-word for the white people.

cracking

Cracking

See cracker.

creative_commons

Creative Commons

Creative Commons (CC) is the forefront non-profit organization promoting free culture, i.e. basically relaxation of "intellectual property" (such as copyright) in art. One of the most important contributions of the organization are the widely used Creative Commons licenses which artists may use to make their works more legally free and even put them to the public domain.

Generally speaking Creative Commons brought a lot of good -- not only did it bring attention to the issues of "intellectual property", it made a huge number of people and organizations actually relax or completely waive their rights on works they create. We, LRS, especially appreciate the CC0 public domain waiver that we prefer for our own works, like did many others, and other licenses such as CC BY-SA are still popular and better than "all rights reserved". However Creative Commons is still a big, centralized organization prone to corruption, it will most definitely suffer the same degeneration as any other organization in history, so don't get attached to it.

TODO

Licenses

Creative commons licenses/waivers form a spectrum spanning from complete freedom (CC0, public domain, no conditions on use) to complete fascism (prohibiting basically everything except for non-commercial sharing). This means that **NOT all Creative Commons licenses are free cultural licenses** -- this is acknowledged by Creative Commons and has been intended. Keep in mind that as a good human you mustn't ever use licenses with NC (non-commercial use only) or ND (no derivatives allowed) clauses, these make your work non-free and therefore unusable.

Here is a comparison of the Creative Commons licenses/waivers, from most free (best) to least free (worst):

name	abbreviation	free culture	use	share	remix	copyleft	attribution	non-commercial	com
<u>Creative Commons Zero</u>	CC0	yes :)	yes :)	yes :)	yes :)	no :)	no need :)	no :)	public domain, copyright, no restrictions, most free, best, sad, waive patents, trademark, requires a license to authorize some use (e.g. DRM), <u>copyfree</u> , don't use retired, some license, not recommended CC, pure copyleft/s without formal attribution, requires a license to authorize copyleft (under same <u>proprietary</u> license for commercial use, DO NOT USE <u>proprietary</u> license for modification, NOT USE <u>proprietary</u> license for commercial and even modification, NOT USE <u>joke</u> license, Question :)
Creative Commons <u>Attribution</u>	CC BY	yes?*	yes?*	yes :)	yes :)	no :)	forced :(no :)	
Creative Commons Sharealike	CC SA	yes :)	yes :)	yes :)	yes :)	yes :/	no need :)	no :)	
Creative Commons Attribution Sharealike	CC BY-SA	yes :)	yes :)	yes :)	yes :)	yes :/	forced :(no :)	
Creative Commons Attribution NonCommercial	CC BY-NC	NO! :(((yes but	yes but	yes but	yes :/	forced :(yes :(
Creative Commons Attribution NoDerivs	CC BY-ND	NO! :(((yes but	yes but	NO! :(yes :/	forced :(no but	
Creative Commons Attribution NonCommercial NoDerivs	CC BY-NC-ND	NO! :(((yes but	yes but	NO! :(yes :/	forced :(yes :(
Creative Commons Attribution NoValue	CC BY NV	no	yes	yes	no	no	forced	yes	
none (all rights reserved)		NO! :(((NO! :(NO! :(NO! :(FUCK YOU	FUCK YOU	FUCK YOU	<u>proprietary</u> option, pretty much everything, NOT USE

Out of Creative Commons licenses/waivers **always use CC0**, that's the only one aligned with our goals, it's the one that's closest to completely rejecting any control over the work. Even though legally and practically there probably won't be such a large difference between CC0 and let's say CC BY, the mental jump to absolute public domain is important (small step for lawyer, huge leap for freedom) -- it's known that people who use the imperfect licenses such as CC BY SA still feel a small grip and authority over their work, they still have to overlook that the license "isn't violated" and sometimes even start making trouble (see e.g. the infamous meltdown of David Revoxy over his "moral rights being violated with NFTs" despite his work being

CC BY SA { Thanks to a friend for finding this. ~drummyfish }). Don't do this, just let go. If you love it, let it go.

There **Creative Commons license paradox**: there seems to be a curious pattern noticeable in the world of Creative Commons licensed works (and possibly free culture and free software in general) -- the phenomenon is that **the shittier the art, the more restrictive license it will have**. { I noticed this on opengameart but then found it basically applies everywhere. ~drummyfish } Upon closer inspection it doesn't look so surprising after all: more restrictive licenses are used as a slow and careful transition from "all right reserved" world, i.e. they are used by newcomers and noobs who fear that if they don't enforce attribution people will immediately exploit it. More skilled people who have spent some time in the world of free art and published more things already know this doesn't happen and they know that less restrictive licenses are just better in all aspects.

See Also

- free culture
- free software

crime_against_economy

Crime Against Economy

Crime against economy refers to any bullshit "crime" invented by capitalism that is deemed to "hurt economy", the new God of society. In the current dystopian society where money has replaced God, worshiping economy is the new religion; to satisfy economy human and animal lives are sacrificed just as such sacrifices used to be made to please the gods of ancient times.

Examples of crimes against economy include:

- Fixing purposefully broken technology, e.g. removing DRM.
- Shielding oneself from marketing torture and refusing to consume, e.g. by using adblocks.
- Burning money.
- Doing literally anything that could destabilize economy such as simply giving away too many things for free or for very low prices.
- Sharing useful information with other people which is called a "theft" of intellectual property, "piracy" etc.
- Taking basic natural resources from monstrously rich corporations who declare to own natural resources and deny access to them. E.g. printing money or physically taking goods from corporations without paying is declared a crime.
- Destroying ads and so sparing other of suffering.
- Encouraging others to commit crimes against economy.
- Revealing certain truths about rich people, corporations, their products or states, so called whistleblowing, antivaxxing etc.
- ...

crow_funding

Crow Funding

Crow funding is when a crow pays for your program.

You probably misspelled crowd funding.

crypto

Cryptocurrency

Cryptocurrency, or just *crypto*, is a digital virtual (non-physical) currency used on the Internet which uses cryptographic methods (electronic signatures etc.) to implement a decentralized system in which there is no authority to control the currency (unlike e.g. with traditional currencies that are controlled by the state or systems of digital payments controlled by the banks that run these systems). Cryptocurrencies traditionally use so called **blockchain** as the underlying technology and are practically always implemented as FOSS. Example of cryptocurrencies are Bitcoin, Monero or Dogecoin.

The word *crypto* in *cryptocurrency* **doesn't imply that the currency provides or protects "privacy"** -- it rather refers to the cryptographic algorithms used to make the currency work -- even though thanks to the decentralization, anonymity and openness cryptocurrencies actually are mostly "privacy friendly" (up to the points of being considered the currency of criminals).

LRS sees cryptocurrencies as not only unnecessary bullshit, but downright as an **unethical** technology because money itself is unethical, plus the currencies based on proof of work waste not only human effort but also enormous amount of electricity and computing power that could be spent in a better way. Keep in mind that cryptocurrencies are a way of digitizing harmful concepts existing in society. Crypto is just an immensely expensive game in which people try to fuck each other over money that have been stolen from the people.

History

TODO

How It Works

Cryptocurrency is build on top of so called blockchain -- a kind structure that holds records of transactions (exchanges of money or "coins", as called in the crypto world). Blockchain is a data structure serving as a database of the system. As its name suggests, it consists of **blocks**. Each block contains various data, most important of which are performed transactions (e.g. "A sent 1 coin to B"), and each block points to a previous one (forming a linked list). As new transactions are made, new blocks are created and appended at the end of the blockchain.

But where is the blockchain stored? It is not on a single computer; many computers participating in the system have their own copy of the blockchain and they share it together (similarly to how people share files via torrents).

But how do we know which one is the "official" blockchain? Can't just people start forging information in the blockchain and then distribute the fake blockchains? Isn't there a chaos if there are so many copies? Well yes, it would be messy -- that's why we need a **consensus** of the participants on which blockchain is the *real* one. And there are a few algorithms to ensure the consensus. Basically people can't just spam add new blocks, a new block to be added needs to be validated via some process (which depends on the specific algorithm) in order to be accepted by others. Two main algorithms for this are:

- proof of work: For a block to be confirmed it has to have a specific cryptographic puzzle solved, e.g. it may need to have appended some string that makes the block's hash some predetermined value. Participants try to solve this puzzle: finding the string is difficult and has to be done by brute force (which wastes electricity and makes this method controversial). Once someone finds a solution, the block is confirmed and the solver gets a reward in coin -- this is therefore called **mining**.
- proof of stake: This methods tries to waste less energy by not solving cryptographics puzzles but rather having some chosen participants validate/confirm the blocks. Basically participants can give some of their money at stake which then gives them a chance (proportional to the amount of money put at stake) to be chosen as validators. A validator is then chosen at random who will check the transactions and sign the block. For this they will get a small reward in coins. If they try to confirm fraudulent transactions (e.g. money sent from people without any money), the network will punish them by taking away the money they put at stake (so there is a financial motivation to not "cheat").

Can't people just forge transactions, e.g. by sending out a record that says someone else sent them money? This can be easily prevented by digitally signing the transactions, i.e. if there is e.g. a transaction "A sends 1 coin to B", it has to be signed by A to confirm that A really intended to send the money. But can't someone just copy-paste someone else's already signed transactions and try to perform them multiple times? This can also be prevented by e.g. numbering the transactions, i.e. recording something like "A sent 1 coin to B as his 1st transaction".

But where are one's coins actually stored? They're not explicitly stored anywhere; the amount of coins any participant has is deduced from the list of transactions, i.e. if it is known someone joined the network with 0 coins and there is a record of someone else sending him 1 coin, it is clear he now has 1 coin. For end users there are so called **wallets** which to them appear to store their coins, but a wallet is in fact just the set of cryptographic keys needed to perform transactions.

But why is blockchain even needed? Can't we just have a list of signed transactions without any blocks? Well, blockchain is designed to ensure coherency and the above mentioned consensus.

c_sharp

C Sharp

C# is supposed to be a "programming language" but it's just some capitalist shit by Micro\$oft that's supposed to give it some kind of monopoly. Really it's not even worth writing about. It's like java but worse. I'm tired, DO NOT USE THIS PSEUDOSHIT. Learn C.

CSS

CSS

{ Check out our cool CSS styles in the wiki consommer edition. ~drummyfish }

Cascading Style Sheets (CSS, *cascading* because of the possible style hierarchy) is a computer language for styling documents (i.e. defining their visual appearance), used mainly on the web for giving websites (HTML documents) their look. The language is standardized by W3C. CSS is NOT a programming language, it's merely a language that says things about visual presentation such as "headings should use this font" or "background should have this color"; it is one of the three main languages a website is written in: HTML (for writing the document), CSS (for giving the document a specific look) and JavaScript (programming language for the website's scripts). As of 2024 the latest CSS specification is version 2.1 from 2016, version 3 is being worked on.

A website is not required to have a CSS style, without it it will just use the plain default look (which is mostly good enough for communicating any information, but won't impress normies), though only boomers and hardcore minimalists nowadays have websites without any CSS at all. Similarly a single HTML website may use several styles or allow switching between them -- this is thanks to the fact that the style is completely separate from the underlying document (you can take any document's style and apply it to any other document) AND thanks to the rules that say which style will take precedence over which (based on which one is more specific etc.), allowing usage of multiple styles at once (creating the "cascades" the name refers to). In theory a web browser may even allow the user to e.g. apply his own CSS style to given website (e.g. a half blind guy may apply style with big font, someone reading in dark will apply "dark mode" style and so on), though for some reason browsers don't really do this.

Back in the boomer web days -- basically before the glorious year 2000 -- there was no CSS. Well, it was around, but support was poor and no one used it (or needed it for that matter). People cared more for sharing information than pimping muh graphics. Sometimes people needed to control the look of their website to some degree though, for example in an image gallery it's good to have thumbnail sizes the same, so HTML itself included some simple things to manipulate the looks (e.g. the width property in the img tag). People also did hacks such as raping tables or spamming the
 tags or using ASCII art to somehow force displaying something how they wanted it. However as corporations started to invade the web, they naturally wanted more consumerism, flashing lights and brainwas... ummm... marketing. They wanted to

redefine the web from "collection of interlinked documents" or a "global database" to something more like "virtual billboard space" or maybe "gigantic electronic shopping center", which indeed they did. So they supported more work on CSS, more browsers started to support it and normies with blogs jumped on the train too, so CSS just became standard. On one hand CSS allows nice things, you can restyle your whole website with a single line change, but it is still bloat, so beware, use it wisely (or rather don't use it -- you can never go wrong with that).

Correct, LRS approved attitude towards this piece of bloat: as a minimalist should you avoid CSS like the devil and never use it? Usual LRS recommendations apply but, just in case, let's reiterate. Use your brain, maximize good, minimize damage, just make it so that no one can ever say "oh no, I wish this site didn't have CSS". You CAN use CSS on your site, but it mustn't become any burden, only something optional that will make life better for those using a browser supporting CSS, i.e. **your site MUSTN'T RELY on CSS**, CSS mustn't be its dependency, the site has to work perfectly fine without it (remember that many browsers, especially the minimalist ones not under any corporation's control, don't even support CSS), the site must not be crippled without a style, i.e. firstly design your site without CSS and only add CSS as an optional improvement. Do not make your HTML bow to CSS, i.e. don't let CSS make you add tons of divs and classes, make HTML first and then make CSS bow to the HTML. Light CSS is better than heavy one. If you have a single page, embed CSS right into it (KISS, site is self contained and browser doesn't have to download extra files for your site) and make it short to save bandwidth on downloading your site. Don't use heavy CSS features like animation, blurs, color transitions or wild repositioning, save the CPU, save the planet (:D). Etcetc.

TODO: more more more

How It Works

The CSS style can be put into three places:

- **separate file** (external CSS): Here the style is written in its own file with .css extension and any HTML file wanting to use the style links to this file (with *link* tag inside *head*). This is good if you have multiple HTML files (i.e. a whole website) that use the same style.
- **inside the HTML file itself** (internal CSS): The style is written right inside the same file as the HTML document, between *style* tags in *head*, so it's all nicely self contained. This is good if the style is used only by this one HTML document (e.g. you have a single webpage or some special page that just has its own style).
- **inside HTML tags** (inline CSS): Style can be specified also as HTML tag attributes (the *style* attribute), but this is discouraged as it intermixes HTML and CSS, something CSS wants to avoid.

The style itself is quite simple, it's just a list of styling rules, each one in format:

```
selectors
{
  style
}
```

Here *selectors* says which elements the rule applies to and *style* defines the visual attributes we want to define. For example

```
p
{
  color: blue;
  font-size: 20px;
}

h1, h2, h3
{
  color: red;
}
```

Specifies two rules. One says that all *p* tags (paragraphs) should have blue text color and font that's 20 pixels tall. The second rule says that *h1*, *h2* and *h3* tags (headings) should have red text color. Pretty simple, no?

Tho it can get more complex, especially when you start positioning and aligning things -- it takes a while to learn how this really works, but in the end it's no rocket science.

TODO: moar

CSS Gore And Pissing People Off

A user running bloatbrowser that implements CSS and has it turned on by default is openly inviting you to freely remotely manipulate what his computer is showing to him, so let's take this opportunity to do some lighthearted trolling :) Here are some possible approaches:

- cursor: Change or even hide the mouse cursor :D You can set it to none (hide), progress (make the user think something's loading indefinitely, see how long it takes for him to realize), wait, col-resize or even specific image with `url(...)`.
- Make the site work only without CSS :D For example: `body * { display: none; } body:before { content: "This site only works without CSS." }.`
- CSS can do animation! This can be used to induce seizures. E.g.: `@keyframes lul { 0% { background-color: red; } 100% { background-color: green; } } body { animation-name: lul; animation-duration: 0.1s; animation-iteration-count: infinite; }.`
- Animate `<body>` size so that the scroll bars keep resizing.
- `a:hover { display: none; }:` Makes links disappear when they're pointed at with the cursor :D Can also be used for buttons etc.
- Make some huge clusterfuck of divs that get arranged in some intricate way, then make each div change its size with `:hover`, or better yet use `transform` to rotate or skew it, triggering a spectacular domino effect. You have to make it so that if one div reshapes on mouse over, another one gets under the cursor, triggering reshape of that one, which pushes another one under the cursor etc.
- Alternative to the previous: make one huge ass div covering the whole screen and make it resize to 1x1 pixels on `:hover`, this will cause some vomit inducing blinking whenever mouse is moved.
- Use animation to very slowly alter the site, e.g. keep making text more and more transparent, so that it can't be noticed immediately but will become apparent after having the site open for 15 minutes, or maybe just have the site normal but after 10 minutes just immediately rotate it 180 degrees, the user will be like WTF :D or maybe instead of this after 10 minutes just replace the site with some porn image -- there is a chance someone will open the site, then leave the computer for a while, leaving the innocent site open but in the meanwhile it will change to porn and suddenly he will look like the biggest pervert :D
- TODO: some shit that makes CPU burn aka bitcoin miner without bitcoin
- TODO: make the page 1 light year long or something
- TODO: more

c_tutorial

C Tutorial

{ Still a work in progress, but 99% complete. ~drummyfish }

This is a relatively quick C tutorial.

You should probably know at least the completely basic ideas of programming before reading this (what's a programming language, source code, command line etc.). If you're as far as already knowing another language, this should be pretty easy to understand.

About C And Programming

C is

- A **programming language**, i.e. a language that lets you express algorithms.
- Compiled language (as opposed to interpreted), i.e. you have to compile the code you write (with compiler) in order to obtain a native executable program (a binary file that you can run directly).

- Extremely **fast and efficient**.
- Very **widely supported and portable** to almost anything.
- **Low level**, i.e. there is relatively little abstraction and not many comfortable built-in functionality such as garbage collection, you have to write many things yourself, you will deal with pointers, endianness etc.
- Imperative (based on sequences of commands), without object oriented programming.
- Considered **hard**, but in certain ways it's simple, it lacks bloat and bullshit of "modern" languages which is an essential thing. It will take long to learn (don't worry, not nearly as long as learning a foreign language) but it's the most basic thing you should know if you want to create good software. You won't regret.
- **Not holding your hand**, i.e. you may very easily "shoot yourself in your foot" and crash your program. This is the price for the language's power.
- Very old, well established and tested by time.
- Recommended by us for serious programs.

If you come from a language like Python or JavaScript, you may be shocked that C doesn't come with its own package manager, debugger or build system, it doesn't have modules, generics, garbage collection, OOP, hashmaps, dynamic lists, type inference and similar "modern" features. When you truly get into C, you'll find it's a good thing.

Programming in C works like this:

1. You write a C source code into a file.
2. You compile the file with a C compiler such as gcc (which is just a program that turns source code into a runnable program). This gives you the executable program.
3. You run the program, test it, see how it works and potentially get back to modifying the source code (step 1).

So, for writing the source code you'll need a text editor; any plain text editor will do but you should use some that can highlight C syntax -- this helps very much when programming and is practically a necessity. Ideal editor is vim but it's a bit difficult to learn so you can use something as simple as Gedit or Geany. We do NOT recommend using huge programming IDEs such as "VS Code" and whatnot. You definitely can NOT use an advanced document editor that works with rich text such as LibreOffice or that shit from Micro\$oft, this won't work because it's not plain text.

Next you'll need a C compiler, the program that will turn your source code into a runnable program. We'll use the most commonly used one called gcc (you can try different ones such as clang or tcc if you want). If you're on a Unix-like system such as GNU/Linux (which you probably should), gcc is probably already installed. Open up a terminal and write gcc to see if it's installed -- if not, then install it (e.g. with `sudo apt install build-essential` if you're on a Debian-based system).

If you're extremely lazy, there are online web C compilers that work in a web browser (find them with a search engine). You can use these for quick experiments but note there are some limitations (e.g. not being able to work with files), and you should definitely know how to compile programs yourself.

Last thing: there are multiple standards of C. Here we will be covering C99, but this likely doesn't have to bother you at this point.

First Program

Let's quickly try to compile a tiny program to test everything and see how everything works in practice.

Open your text editor and paste this code:

```
/* simple C program! */

#include <stdio.h> // include IO library

int main(void)
{
    puts("It works.");
}
```

```
    return 0;
}
```

Save this file and name it `program.c`. Then open a terminal emulator (or an equivalent command line interface), locate yourself into the directory where you saved the file (e.g. `cd somedirectory`) and compile the program with the following command:

```
gcc -o program program.c
```

The program should compile and the executable program should appear in the directory. You can run it with

```
./program
```

And you should see

```
It works.
```

written in the command line.

Now let's see what the source code means:

- `/* simple C program! */` is so called *block comment*, it does nothing, it's here only for the humans that will read the source code. Such comments can be almost anywhere in the code. The comment starts at `/*` and ends with `*/`.
- `// include IO library` is another comment, but this is a *line comment*, it starts with `//` and ends with the end of line.
- `#include <stdio.h>` tells the compiler we want to include a library named *stdio* (the weird syntax will be explained in the future). This is a standard library with input output functions, we need it to be able to use the function `puts` later on. We can include more libraries if we want to. These includes are almost always at the very top of the source code.
- `int main(void)` is the start of the main program. What exactly this means will be explained later, for now just remember there has to be this function named `main` in basically every program -- inside it there are commands that will be executed when the program is run. Note that the curly brackets that follow (`{` and `}`) denote the block of code that belongs to this function, so we need to write our commands between these brackets.
- `puts("It works.");` is a "command" for printing text strings to the command line (it's a command from the `stdio` library included above). Why exactly this is written like this will be explained later, but for now notice the following. The command starts with its name (`puts`, for *put string*), then there are left and right brackets (`(` and `)`) between which there are arguments to the command, in our case there is one, the text string `"It works."`. Text strings have to be put between quotes (`"`), otherwise the compiler would think the words are other commands (the quotes are not part of the string itself, they won't be printed out). The command is terminated by `;` -- all "normal" commands in C have to end with a semicolon.
- `return 0;` is another "command", it basically tells the operating system that everything was terminated successfully (`0` is a code for success). This command is an exception in that it doesn't have to have brackets (`(` and `)`). This doesn't have to bother us too much now, let's just remember this will always be the last command in our program.

Also notice how the source code is formatted, e.g. the indentation of code within the `{` and `}` brackets. White characters (spaces, new lines, tabs) are ignored by the compiler so we can theoretically write our program on a single line, but that would be unreadable. We use indentation, spaces and empty lines to format the code to be well readable.

To sum up let's see a general structure of a typical C program. You can just copy paste this for any new program and then just start writing commands in the main function.

```
#include <stdio.h> // include the I/O library
// more libraries can be included here

int main(void)
{
```



```
// write commands here

return 0; // always the last command
}
```

Variables, Arithmetic, Data Types

Programming is a lot like mathematics, we compute equations and transform numerical values into other values. You probably know in mathematics we use *variables* such as x or y to denote numerical values that can change (hence variables). In programming we also use variables -- here **variable is a place in memory which has a name** (and in this place there will be stored a value that can change over time).

We can create variables named x , y , `myVariable` or `score` and then store specific values (for now let's only consider numbers) into them. We can read from and write to these variables at any time. These variables physically reside in RAM, but we don't really care where exactly (at which address) they are located -- this is e.g. similar to houses, in common talk we normally say things like *John's house* or *the pet store* instead of *house with address 3225*.

Variable names can't start with a digit (and they can't be any of the keywords reserved by C). By convention they also shouldn't be all uppercase or start with uppercase (these are normally used for other things). Normally we name variables like this: `myVariable` or `my_variable` (pick one style, don't mix them).

In C as in other languages each variable has a certain **data type**; that is each variable has associated an information of what kind of data is stored in it. This can be e.g. a *whole number*, *fraction*, a *text character*, *text string* etc. Data types are a more complex topic that will be discussed later, for now we'll start with the most basic one, the **integer type**, in C called `int`. An `int` variable can store whole numbers in the range of at least -32768 to 32767 (but usually much more).

Let's see an example.

```
#include <stdio.h>

int main(void)
{
    int myVariable;

    myVariable = 5;

    printf("%d\n",myVariable);

    myVariable = 8;

    printf("%d\n",myVariable);
}
```

- `int myVariable;` is so called **variable declaration**, it tells the compiler we are creating a new variable with the name `myVariable` and data type `int`. Variables can be created almost anywhere in the code (even outside the `main` function) but that's a topic for later.
- `myVariable = 5;` is so called **variable assignment**, it stores a value 5 into variable named `myVariable`. IMPORTANT NOTE: the `=` does NOT signify mathematical equality but an assignment (equality in C is written as `==`); when compiler encounters `=`, it simply takes the value on the right of it and writes it to the variable on the left side of it. Sometimes people confuse assignment with an equation that the compiler solves -- this is NOT the case, assignment is much more simple, it simply writes a value into variable. So `x = x + 1;` is a valid command even though mathematically it would be an equation without a solution.
- `printf("%d\n",myVariable);` prints out the value currently stored in `myVariable`. Don't get scared by this complicated command, it will be explained later (once we learn about pointers). For now only know this prints the variable content.
- `myVariable = 8;` assigns a new value to `myVariable`, overwriting the old.
- `printf("%d\n",myVariable);` again prints the value in `myVariable`.

After compiling and running of the program you should see:

Last thing to learn is **arithmetic operators**. They're just normal math operators such as +, - and /. You can use these along with brackets ((and)) to create **expressions**. Expressions can contain variables and can themselves be used in many places where variables can be used (but not everywhere, e.g. on the left side of variable assignment, that would make no sense). E.g.:

```
#include <stdio.h>

int main(void)
{
    int heightCm = 175;
    int weightKg = 75;
    int bmi = (weightKg * 10000) / (heightCm * heightCm);

    printf("%d\n",bmi);
}
```

calculates and prints your BMI (body mass index).

Let's quickly mention how you can read and write values in C so that you can begin to experiment with your own small programs. You don't have to understand the following syntax as of yet, it will be explained later, now simply copy-paste the commands:

- `puts("hello");` Prints a text string with newline.
- `printf("hello");` Same as above but without newline.
- `printf("%d\n",x);` Prints the value of variable x with newline.
- `printf("%d ");` Same as above but without a newline.
- `scanf("%d",&x);` Read a number from input to the variable x. Note there has to be & in front of x.

Branches And Loops (If, While, For)

When creating algorithms, it's not enough to just write linear sequences of commands. Two things (called control structures) are very important to have in addition:

- **branches**: Conditionally executing or skipping certain commands (e.g. if a user enters password we want to either log him in if the password was correct or write error if the password was incorrect). This is informally known as **"if-then-else"**.
- **loops** (also called **iteration**): Repeating certain commands given number of times or as long as some condition holds (e.g. when searching a text we repeatedly compare words one by one to the searched word until a match is found or end of text is reached).

Let's start with **branches**. In C the command for a branch is `if`. E.g.:

```
if (x > 10)
    puts("X is greater than 10.");
```

The syntax is given, we start with `if`, then brackets ((and)) follow inside which there is a condition, then a command or a block of multiple commands (inside { and }) follow. If the condition in brackets holds, the command (or block of commands) gets executed, otherwise it is skipped.

Optionally there may be an `else` branch which is gets executed only if the condition does NOT hold. It is denoted with the `else` keyword which is again followed by a command or a block of multiple commands. Branching may also be nested, i.e. branches may be inside other branches. For example:

```
if (x > 10)
    puts("X is greater than 10.");
else
{
    puts("X is not greater than 10.");

    if (x < 5)
```

```
    puts("And it is also smaller than 5.");
}
```

So if x is equal e.g. 3, the output will be:

```
X is not greater than 10.
And it is also smaller than 5.
```

About **conditions** in C: a condition is just an expression (variables/functions along with arithmetic operators). The expression is evaluated (computed) and the number that is obtained is interpreted as *true* or *false* like this: **in C 0 (zero) means false, 1 (and everything else) means true**. Even comparison operators like < and > are technically arithmetic, they compare numbers and yield either 1 or 0. Some operators commonly used in conditions are:

- == (equals): yields 1 if the operands are equal, otherwise 0.
- != (not equal): yields 1 if the operands are NOT equal, otherwise 0.
- < (less than): yields 1 if the first operand is smaller than the second, otherwise 0.
- <=: yields 1 if the first operand is smaller or equal to the second, otherwise 0.
- && (logical AND): yields 1 if both operands are non-0, otherwise 0.
- || (logical OR): yields 1 if at least one operand is non-0, otherwise 0.
- ! (logical NOT): yields 1 if the operand is 0, otherwise 0.

E.g. an if statement starting as `if (x == 5 || x == 10)` will be true if x is either 5 or 10.

Next we have **loops**. There are multiple kinds of loops even though in theory it is enough to only have one kind of loop (there are multiple types out of convenience). The loops in C are:

- **while**: Loop with condition at the beginning.
- **do while**: Loop with condition at the end, not used so often so we'll ignore this one.
- **for**: Loop executed a fixed number of times. This is a very common case, that's why there is a special loop for it.

The **while** loop is used when we want to repeat something without knowing in advance how many times we'll repeat it (e.g. searching a word in text). It starts with the while keyword, is followed by brackets with a condition inside (same as with branches) and finally a command or a block of commands to be looped. For instance:

```
while (x > y) // as long as x is greater than y
{
    printf("%d %d\n",x,y); // prints x and y

    x = x - 1; // decrease x by 1
    y = y * 2; // double y
}

puts("The loop ended.");
```

If x and y were to be equal 100 and 20 (respectively) before the loop is encountered, the output would be:

```
100 20
99 40
98 60
97 80
The loop ended.
```

The **for** loop is executed a fixed number of time, i.e. we use it when we know in advance how many time we want to repeat our commands. The syntax is a bit more complicated: it starts with the keywords for, then brackets ((and)) follow and then the command or a block of commands to be looped. The inside of the brackets consists of an initialization, condition and action separated by semicolon (;) -- don't worry, it is enough to just remember the structure. A for loop may look like this:

```
puts("Counting until 5...");

for (int i = 0; i < 5; ++i)
```

```
printf("%d\n",i); // prints i
```

`int i = 0` creates a new temporary variable named `i` (name normally used by convention) which is used as a **counter**, i.e. this variable starts at 0 and increases with each iteration (cycle), and it can be used inside the loop body (the repeated commands). `i < 5` says the loop continues to repeat as long as `i` is smaller than 5 and `++i` says that `i` is to be increased by 1 after each iteration (`++i` is basically just a shorthand for `i = i + 1`). The above code outputs:

```
Counting until 5...
0
1
2
3
4
```

IMPORTANT NOTE: in programming we **count from 0**, not from 1 (this is convenient e.g. in regards to pointers). So if we count to 5, we get 0, 1, 2, 3, 4. This is why `i` starts with value 0 and the end condition is `i < 5` (not `i <= 5`).

Generally if we want to repeat the for loop *N* times, the format is `for (int i = 0; i < N; ++i)`.

Any loop can be exited at any time with a special command called `break`. This is often used with so called infinite loop, a *while* loop that has 1 as a condition; recall that 1 means true, i.e. the loop condition always holds and the loop never ends. `break` allows us to place conditions in the middle of the loop and into multiple places. E.g.:

```
while (1) // infinite loop
{
    x = x - 1;

    if (x == 0)
        break; // this exits the loop!

    y = y / x;
}
```

The code above places a condition in the middle of an infinite loop to prevent division by zero in `y = y / x`.

Again, loops can be nested (we may have loops inside loops) and also loops can contain branches and vice versa.

Simple Game: Guess A Number

With what we've learned so far we can already make a simple game: guess a number. The computer thinks a random number in range 0 to 9 and the user has to guess it. The source code is following.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    srand(clock()); // random seed

    while (1) // infinite loop
    {
        int randomNumber = rand() % 10;

        puts("I think a number. What is it?");

        int guess;

        scanf("%d",&guess); // read the guess

        getchar();
```

```

    if (guess == randomNumber)
        puts("You guessed it!");
    else
        printf("Wrong. The number was %d.\n",randomNumber);

    puts("Play on? [y/n]");

    char answer;

    scanf("%c",&answer); // read the answer

    if (answer == 'n')
        break;
}

puts("Bye.");

return 0; // return success, always here
}

```

- `#include <stdlib.h>`, `#include <time.h>`: we're including additional libraries because we need some specific functions from them (rand, srand, clock).
- `srand(clock());`: don't mind this line too much, its purpose is to seed a pseudorandom number generator. Without doing this the game would always generate the same sequence of random numbers when run again.
- `while (1)` is an infinite game loop -- it runs over and over, in each cycle we perform one game round. The loop can be exited with the break statement later on (if the user answers he doesn't want to continue playing).
- `int randomNumber = rand() % 10;` this line declares a variable named `randomNumber` and immediately assigns a value to it. The value is a random number from 0 to 9. This is achieved with a function `rand` (from the above included `stdlib` library) which returns a random number, and with the modulo (remainder after division) arithmetic operator (`%`) which ensures the number is in the correct range (less than 10).
- `int guess;` creates another variable in which we'll store the user's guessed number.
- `scanf("%d",&guess);` reads a number from the input to the variable named `guess`. Again, don't be bothered by the complicated structure of this command, for now just accept that this is how it's done.
- `getchar();` don't mind this line, it just discards a newline character read from the input.
- `if (guess == randomNumber) ...` this is a branch which checks if the user guess is equal to the generated random number. If so, a success message is printed out. If not, a fail message is printed out along with the secret number. Note we use the `puts` function for the first message as it only prints a text sting, while for the latter message we have to use `printf`, a more complex function, because that requires inserting a number into the printed string. More on these functions later.
- `char answer;` declares a variable to store user's answer to a question of whether to play on. It is of `char` data type which can store a single text character.
- `scanf("%c",&answer);` reads a single character from input to the `answer` variable.
- `if (answer == 'n') break;` is a branch that exits the infinite loop with `break` statement if the answer entered was `n` (*no*).

Functions (Subprograms)

Functions are extremely important, no program besides the most primitive ones can be made without them (well, in theory any program can be created without functions, but in practice such programs would be extremely complicated and unreadable).

Function is a subprogram (in other languages functions are also called procedures or subroutines), i.e. it is code that solves some smaller subproblem that you can repeatedly invoke, for instance you may have a function for computing a square root, for encrypting data or for playing a sound from speakers. We have already met functions such as `puts`, `printf` or `rand`.

Functions are similar to but **NOT the same as mathematical functions**. Mathematical function (simply put) takes a number as input and outputs another number computed from the input number, and this output number depends only on the input number and nothing else. C functions can do this too but they can also do additional things such as modify variables in other parts of the program or make the computer do something

(such as play a sound or display something on the screen) -- these are called **side effects**; things done besides computing an output number from an input number. For distinction mathematical functions are called *pure* functions and functions with side effects are called non-pure.

Why are function so important? Firstly they help us divide a big problem into small subproblems and make the code better organized and readable, but mainly they help us respect the DRY (*Don't Repeat Yourself*) principle -- this is extremely important in programming. Imagine you need to solve a quadratic equation in several parts of your program; you do NOT want to solve it in each place separately, you want to make a function that solves a quadratic equation and then only invoke (call) that function anywhere you need to solve your quadratic equation. This firstly saves space (source code will be shorter and compiled program will be smaller), but it also makes your program manageable and eliminates bugs -- imagine you find a better (e.g. faster) way to solving quadratic equations; without functions you'd have to go through the whole code and change the algorithm in each place separately which is impractical and increases the chance of making errors. With functions you only change the code in one place (in the function) and in any place where your code invokes (calls) this function the new better and updated version of the function will be used.

Besides writing programs that can be directly executed programmers write **libraries** -- collections of functions that can be used in other projects. We have already seen libraries such as *stdio*, *standard input/output library*, a standard (official, bundled with every C compiler) library for input/output (reading and printing values); *stdio* contains functions such as *puts* which is used to printing out text strings. Examples of other libraries are the standard *math* library containing function for e.g. computing sine, or SDL, a 3rd party multimedia library for such things as drawing to screen, playing sounds and handling keyboard and mouse input.

Let's see a simple example of a function that writes out a temperature in degrees of Celsius as well as in Kelvin:

```
#include <stdio.h>

void writeTemperature(int celsius)
{
    int kelvin = celsius + 273;
    printf("%d C (%d K)\n",celsius,kelvin);
}

int main(void)
{
    writeTemperature(-50);
    writeTemperature(0);
    writeTemperature(100);

    return 0;
}
```

The output is

```
-50 C (223 K)
0 C (273 K)
100 C (373 K)
```

Now imagine we decide we also want our temperatures in Fahrenheit. We can simply edit the code in *writeTemperature* function and the program will automatically be writing temperatures in the new way.

Let's see how to create and invoke functions. Creating a function in code is done between inclusion of libraries and the *main* function, and we formally call this **defining a function**. The function definition format is following:

```
RETURN_TYPE FUNCTION_NAME(FUNCTION_PARAMETERS)
{
    FUNCTION_BODY
}
```

- **RETURN_TYPE** is the data type the function returns. A function may or may not return a certain value, just as the pure mathematical function do. This may for example be *int*, if the function returns an

integer number. If the function doesn't return anything, this type is void.

- **FUNCTION_NAME** is the name of the function, it follows the same rules as the names for variables.
- **FUNCTION_PARAMETERS** specifies the input values of the function. The function can take any number of parameters (e.g. a function `playBeep` may take 0 arguments, `sine` function takes 1, `logarithm` may take two etc.). This list is comma-separated and each item consists of the parameter data type and name. If there are 0 parameters, there should be the word `void` inside the brackets, but compilers tolerate just having empty brackets.
- **FUNCTION_BODY** are the commands executed by the function, just as we know them from the *main* function.

Let's see another function:

```
#include <stdio.h>

int power(int x, int n)
{
    int result = 1;

    for (int i = 0; i < n; ++i) // repeat n times
        result = result * x;

    return result;
}

int main(void)
{
    for (int i = 0; i < 5; ++i)
    {
        int powerOfTwo = power(2,i);
        printf("%d\n",powerOfTwo);
    }

    return 0;
}
```

The output is:

```
2
4
8
16
```

The function `power` takes two parameters: `x` and `n`, and returns `x` raised to the `n`s power. Note that unlike the first function we saw here the return type is `int` because this function does return a value. **Notice the command `return`** -- it is a special command that causes the function to terminate and return a specific value. In functions that return a value (their return type is not `void`) there has to be a `return` command. In function that return nothing there may or may not be one, and if there is, it has no value after it (`return;`);

Let's focus on how we invoke the function -- in programming we say we **call the function**. The function call in our code is `power(2,i)`. If a function returns a value (return type is not `void`), its function call can be used in any expression, i.e. almost anywhere where we can use a variable or a numerical value -- just imagine the function computes a return value and this value is **substituted to the place where we call the function**. For example we can imagine the expression `power(3,1) + power(3,0)` as simply `3 + 1`.

If a function returns nothing (return type is `void`), it can't be used in expressions, it is used "by itself"; e.g. `playBeep()`; . (Function that do return a value can also be used like this -- their return value is in this case simply ignored.)

We call a function by writing its name (`power`), then adding brackets (`(` and `)`) and inside them we put **arguments** -- specific values that will substitute the corresponding parameters inside the function (here `x` will take the value 2 and `n` will take the current value of `i`). If the function takes no parameters (the function parameter list is `void`), we simply put nothing inside the brackets (e.g. `playBeep()`);

Here comes the nice thing: **we can nest function calls**. For example we can write `x = power(3,power(2,1))`; which will result in assigning the variable `x` the value of 9. **Functions can also call other functions** (even themselves, see [recursion](#)), but only those that have been defined before them in the source code (this can be fixed with so called [forward declarations](#)).

Notice that the main function we always have in our programs is also a function definition. The definition of this function is required for runnable programs, its name has to be `main` and it has to return `int` (an error code where 0 means no error). It can also take parameters but more on that later.

These is the most basic knowledge to have about C functions. Let's see one more example with some peculiarities that aren't so important now, but will be later.

```
#include <stdio.h>

void writeFactors(int x) // writes divisors of x
{
    printf("factors of %d:\n",x);

    while (x > 1) // keep dividing x by its factors
    {
        for (int i = 2; i <= x; ++i) // search for a factor
            if (x % i == 0) // i divides x without remainder?
            {
                printf("  %d\n",i); // i is a factor, write it
                x = x / i; // divide x by i
                break; // exit the for loop
            }
    }
}

int readNumber(void)
{
    int number;

    puts("Please enter a number to factor (0 to quit).");
    scanf("%d",&number);

    return number;
}

int main(void)
{
    while (1) // infinite loop
    {
        int number = readNumber(); // <- function call

        if (number == 0) // 0 means quit
            break;

        writeFactors(number); // <- function call
    }

    return 0;
}
```

We have defined two functions: `writeFactors` and `readNumber`. `writeFactors` return no values but it has side effects (print text to the command line). `readNumber` takes no parameters but return a value; it prompts the user to enter a value and returns the read value.

Notice that inside `writeFactors` we modify its parameter `x` inside the function body -- this is okay, it won't affect the argument that was passed to this function (the `number` variable inside the `main` function won't change after this function call). `x` can be seen as a **local variable** of the function, i.e. a variable that's created inside this function and can only be used inside it -- when `writeFactors` is called inside `main`, a new local variable `x` is created inside `writeFactors` and the value of `number` is copied to it.

Another local variable is `number` -- it is a local variable both in `main` and in `readNumber`. Even though the names are the same, these are two different variables, each one is local to its respective function (modifying

number inside `readNumber` won't affect number inside `main` and vice versa).

And a last thing: keep in mind that not every command you write in C program is a function call. E.g. control structures (`if`, `while`, ...) and special commands (`return`, `break`, ...) are not function calls.

More Details (Globals, Switch, Float, Forward Decls, ...)

We've skipped a lot of details and small tricks for simplicity. Let's go over some of them. Many of the following things are so called syntactic sugar: convenient syntax shorthands for common operations.

Multiple variables can be defined and assigned like this:

```
int x = 1, y = 2, z;
```

The meaning should be clear, but let's mention that `z` doesn't generally have a defined value here -- it will have a value but you don't know what it is (this may differ between different computers and platforms). See undefined behavior.

The following is a shorthand for using operators:

```
x += 1;      // same as: x = x + 1;
x -= 10;     // same as: x = x - 1;
x *= x + 1;  // same as: x = x * (x + 1);
x++;        // same as: x = x + 1;
x--;        // same as: x = x - 1;
// etc.
```

The last two constructs are called **incrementing** and **decrementing**. This just means adding/subtracting 1.

In C there is a pretty unique operator called the **ternary operator** (ternary for having three operands). It can be used in expressions just as any other operators such as `+` or `-`. Its format is:

```
CONDITION ? VALUE1 : VALUE2
```

It evaluates the `CONDITION` and if it's true (non-0), this whole expression will have the value of `VALUE1`, otherwise its value will be `VALUE2`. It allows for not using so many `ifs`. For example instead of

```
if (x >= 10)
    x -= 10;
else
    x = 10;
```

we can write

```
x = x >= 10 ? x - 10 : 10;
```

Global variables: we can create variables even outside function bodies. Recall that variables inside functions are called *local*; variables outside functions are called *global* -- they can basically be accessed from anywhere and can sometimes be useful. For example:

```
#include <stdio.h>
#include <stdlib.h> // for rand()

int money = 0; // total money, global variable

void printMoney(void)
{
    printf("I currently have %d.\n", money);
}

void playLottery(void)
{
    puts("I'm playing lottery.");
}
```

```

    money -= 10; // price of lottery ticket

    if (rand() % 5) // 1 in 5 chance
    {
        money += 100;
        puts("I've won!");
    }
    else
        puts("I've lost!");

    printMoney();
}

void work(void)
{
    puts("I'm going to work :(");

    money += 200; // salary

    printMoney();
}

int main()
{
    work();
    playLottery();
    work();
    playLottery();

    return 0;
}

```

In C programs you may encounter a **switch** statement -- it is a control structure similar to a branch `if` which can have more than two branches. It looks like this:

```

switch (x)
{
    case 0: puts("X is zero. Don't divide by it."); break;
    case 69: puts("X is 69, haha."); break;
    case 42: puts("X is 42, the answer to everything."); break;
    default: printf("I don't know anything about X."); break;
}

```

Switch can only compare exact values, it can't e.g. check if a value is greater than something. Each branch starts with the keyword `case`, then the match value follows, then there is a colon (`:`) and the branch commands follow. **IMPORTANT:** there has to be the `break;` statement at the end of each case branch (we won't go into details). A special branch is the one starting with the word `default` that is executed if no case label was matched.

Let's also mention some additional data types we can use in programs:

- **char:** A single text character such as `'a'`, `'G'` or `'_'`. We can assign characters as `char c = 'a';` (single characters are enclosed in apostrophes similarly to how text strings are inside quotes). We can read a character as `c = getchar();` and print it as `putchar(c);`. Special characters that can be used are `\n` (newline) or `\t` (tab). Characters are in fact small numbers (usually with 256 possible values) and can be used basically anywhere a number can be used (for example we can compare characters, e.g. `if (c < 'b') ...`). Later we'll see characters are basic building blocks of text strings.
- **unsigned int:** Integer that can only take positive values or 0 (i.e. no negative values). It can store higher positive values than normal `int` (which is called a *signed int*).
- **long:** Big integer, takes more memory but can store number in the range of at least a few billion.
- **float and double:** Floating point number (double is bigger and more precise than float) -- an approximation of real numbers, i.e. numbers with a fractional part such as 2.5 or 0.0001. You can print these numbers as `printf("%lf\n", x);` and read them as `scanf("%f", &x);`.

Here is a short example with the new data types:

```

#include <stdio.h>

```

```

int main(void)
{
    char c;
    float f;

    puts("Enter character.");
    c = getchar(); // read character

    puts("Enter float.");
    scanf("%f",&f);

    printf("Your character is :%c.\n",c);
    printf("Your float is %lf\n",f);

    float fSquared = f * f;
    int wholePart = f; // this can be done

    printf("It's square is %lf.\n",fSquared);
    printf("It's whole part is %d.\n",wholePart);

    return 0;
}

```

Notice mainly how we can assign a float value into the variable of int type (`int wholePart = f;`). This can be done even the other way around and with many other types. C can do automatic **type conversions** (casting), but of course, some information may be lost in this process (e.g. the fractional part).

In the section about functions we said a function can only call a function that has been defined before it in the source code -- this is because the compiler read the file from start to finish and if you call a function that hasn't been defined yet, it simply doesn't know what to call. But sometimes we need to call a function that will be defined later, e.g. in cases where two functions call each other (function *A* calls function *B* in its code but function *B* also calls function *A*). For this there exist so called **forward declarations** -- a forward declaration is informing that a function of certain name (and with certain parameters etc.) will be defined later in the code. Forward declaration look the same as a function definition, but it doesn't have a body (the part between { and }), instead it is terminated with a semicolon (;). Here is an example:

```

#include <stdio.h>

void printDecorated2(int x, int fancy); // forward declaration

void printDecorated1(int x, int fancy)
{
    putchar('~');

    if (fancy)
        printDecorated2(x,0); // would be error without f. decl.
    else
        printf("%d",x);

    putchar('~');
}

void printDecorated2(int x, int fancy)
{
    putchar('>');

    if (fancy)
        printDecorated1(x,0);
    else
        printf("%d",x);

    putchar('<');
}

int main()
{
    printDecorated1(10,1);
    putchar('\n'); // newline
}

```

```
    printDecorated2(20,1);
}
```

which prints

```
~>10<~
>~20~<
```

The functions `printDecorated1` and `printDecorated2` call each other, so this is the case when we have to use a forward declaration of `printDecorated2`. Also note the condition `if (fancy)` which is the same thing as `if (fancy != 0)` (imagine `fancy` being 1 and 0 and about what the condition evaluates to in each case).

Header Files, Libraries, Compilation/Building

So far we've only been writing programs into a single source code file (such as `program.c`). More complicated programs consist of multiple files and libraries -- we'll take a look at this now.

In C we normally deal with two types of source code files:

- *.c files*: These files contain so called **implementation** of algorithms, i.e. code that translates into actual program instructions. These files are what's handed to the compiler.
- *.h files*, or **header files**: These files typically contain **declarations** such as constants and function headers (but not their bodies, i.e. implementations).

When we have multiple source code files, we typically have pairs of *.c* and *.h* files. E.g. if there is a library called *mathfunctions*, it will consist of files *mathfunctions.c* and *mathfunctions.h*. The *.h* file will contain the function headers (in the same manner as with forward declarations) and constants such as `pi`. The *.c* file will then contain the implementations of all the functions declared in the *.h* file. But why do we do this?

Firstly *.h* files may serve as a nice documentation of the library for programmers: you can simply open the *.h* file and see all the functions the library offers without having to skim over thousands of lines of code. Secondly this is for how multiple source code files are compiled into a single executable program.

Suppose now we're compiling a single file named *program.c* as we've been doing until now. The compilation consists of several steps:

1. The compiler reads the file *program.c* and makes sense of it.
2. It then creates an intermediate file called *program.o*. This is called an **object file** and is a binary compiled file which however cannot yet be run because it is not *linked* -- in this code all memory addresses are relative and it doesn't yet contain the code from external libraries (e.g. the code of `printf`).
3. The compiler then runs a **linker** which takes the file *program.o* and the object files of libraries (such as the *stdio* library) and it puts them all together into the final executable file called *program*. This is called **linking**; the code from the libraries is copied to complete the code of our program and the memory addresses are settled to some specific values.

So realize that when the compiler is compiling our program (*program.c*), which contains function such as `printf` from a separate library, it doesn't have the code of these functions available -- this code is not in our file. Recall that if we want to call a function, it must have been defined before and so in order for us to be able to call `printf`, the compiler must know about it. This is why we include the *stdio* library at the top of our source code with `#include <stdio.h>` -- this basically copy-pastes the content of the header file of the *stdio* library to the top of our source code file. In this header there are forward declarations of functions such as `printf`, so the compiler now knows about them (it knows their name, what they return and what parameters they take) and we can call them.

Let's see a small example. We'll have the following files (all in the same directory).

library.h (the header file):

```
// Returns the square of n.
int square(int n);
```

library.c (the implementation file):

```
int square(int x)
{
    // function implementation
    return x * x;
}
```

program.c (main program):

```
#include <stdio.h>
#include "library.h"

int main(void)
{
    int n = square(5);

    printf("%d\n",n);

    return 0;
}
```

NOTE: "library.h" here is between double quotes, unlike <stdio.h>. This just says we specify an absolute path to the file as it's not in the directory where installed libraries go.

Now we will manually compile the library and the final program. First let's compile the library, in command line run:

```
gcc -c -o library.o library.c
```

The -c flag tells the compiler to only compile the file, i.e. only generate the object (.o) file without trying to link it. After this command a file *library.o* should appear. Next we compile the main program in the same way:

```
gcc -c -o program.o program.c
```

This will generate the file *program.o*. Note that during this process the compiler is working only with the *program.c* file, it doesn't know the code of the function square, but it knows this function exists, what it returns and what parameter it has thanks to us including the library header *library.h* with `#include "library.h"` (quotes are used instead of < and > to tell the compiler to look for the files in the current directory).

Now we have the file *program.o* in which the compiled main function resides and file *library.o* in which the compiled function square resides. We need to link them together. This is done like this:

```
gcc -o program program.o library.o
```

For linking we don't need to use any special flag, the compiler knows that if we give it several .o files, it is supposed to link them. The file *program* should appear that we can already run and it should print

25

This is the principle of compiling multiple C files (and it also allows for combining C with other languages). This process is normally automated, but you should know how it works. The systems that automate this action are called **build systems**, they are for example [Make](#) and [Cmake](#). When using e.g. the Make system, the whole codebase can be built with a single command `make` in the command line.

Some programmers simplify this whole process further so that they don't even need a build system, e.g. with so called [header-only libraries](#), but this is outside the scope of this tutorial.

As a bonus, let's see a few useful compiler flags:

- -O1, -O2, -O3: Optimize for speed (higher number means better optimization). Adding -O3 normally instantly speeds up your program. This is recommended.
- -Os: Optimize for size, the same as above but the compiler will try to make as small executable as possible.
- -Wall -Wextra -pedantic: The compiler will write more warnings and will be more strict. This can help spot many bugs.
- -c: Compile only (generate object files, do not link).
- -g: Include debug symbols, this will be important for debugging.

Advanced Data Types And Variables (Structs, Arrays, Strings)

Until now we've encountered simple data types such as `int`, `char` or `float`. These identify values which can take single atomic values (e.g. numbers or text characters). Such data types are called **primitive types**.

Above these there exist **compound data types** (also *complex* or *structured*) which are composed of multiple primitive types. They are necessary for any advanced program.

The first compound type is a structure, or **struct**. It is a collection of several values of potentially different data types (primitive or compound). The following code shows how a struct can be created and used.

```
#include <stdio.h>

typedef struct
{
    char initial; // initial of name
    int weightKg;
    int heightCm;
} Human;

int bmi(Human human)
{
    return (human.weightKg * 10000) / (human.heightCm * human.heightCm);
}

int main(void)
{
    Human carl;

    carl.initial = 'C';
    carl.weightKg = 100;
    carl.heightCm = 180;

    if (bmi(carl) > 25)
        puts("Carl is fat.");

    return 0;
}
```

The part of the code starting with `typedef struct` creates a new data type that we call `Human` (one convention for data type names is to start them with an uppercase character). This data type is a structure consisting of three members, one of type `char` and two of type `int`. Inside the `main` function we create a variable `carl` which is of `Human` data type. Then we set the specific values -- we see that each member of the struct can be accessed using the dot character (`.`), e.g. `carl.weightKg`; this can be used just as any other variable. Then we see the type `Human` being used in the parameter list of the function `bmi`, just as any other type would be used.

What is this good for? Why don't we just create global variables such as `carl_initial`, `carl_weightKg` and `carl_heightCm`? In this simple case it might work just as well, but in a more complex code this would be burdening -- imagine we wanted to create 10 variables of type `Human` (`john`, `becky`, `arnold`, ...). We would have to painstakingly create 30 variables (3 for each person), the function `bmi` would have to take two parameters (height and weight) instead of one (`human`) and if we wanted to e.g. add more information about every human (such as `hairLength`), we would have to manually create another 10 variables and add one parameter to the function `bmi`, while with a struct we only add one member to the struct definition and create more variables of type `Human`.

Structs can be nested. So you may see things such as `myHouse.groundFloor.livingRoom.ceilingHeight` in C code.

Another extremely important compound type is **array** -- a sequence of items, all of which are of the same data type. Each array is specified with its length (number of items) and the data type of the items. We can have, for instance, an array of 10 ints, or an array of 235 Humans. The important thing is that we can **index** the array, i.e. we access the individual items of the array by their position, and this position can be specified with a variable. This allows for **looping over array items** and performing certain operations on each item. Demonstration code follows:

```
#include <stdio.h>
#include <math.h> // for sqrt()

int main(void)
{
    float vector[5];

    vector[0] = 1;
    vector[1] = 2.5;
    vector[2] = 0;
    vector[3] = 1.1;
    vector[4] = -405.054;

    puts("The vector is:");

    for (int i = 0; i < 5; ++i)
        printf("%lf ", vector[i]);

    putchar('\n'); // newline

    /* compute vector length with
       pythagoren theorem: */

    float sum = 0;

    for (int i = 0; i < 5; ++i)
        sum += vector[i] * vector[i];

    printf("Vector length is: %lf\n", sqrt(sum));

    return 0;
}
```

We've included a new library called `math.h` so that we can use a function for square root (`sqrt`). (If you have trouble compiling the code, add `-lm` flag to the compile command.)

`float vector[5];` is a declaration of an array of length 5 whose items are of type `float`. When compiler sees this, it creates a continuous area in memory long enough to store 5 numbers of `float` type, the numbers will reside here one after another.

After doing this, we can **index** the array with square brackets (`[` and `]`) like this: `ARRAY_NAME[INDEX]` where `ARRAY_NAME` is the name of the array (here `vector`) and `INDEX` is an expression that evaluates to integer, **starting with 0** and going up to the vector length minus one (remember that **programmers count from zero**). So the first item of the array is at index 0, the second at index 1 etc. The index can be a numeric constant like 3, but also a variable or a whole expression such as `x + 3 * myFunction()`. Indexed array can be used just like any other variable, you can assign to it, you can use it in expressions etc. This is seen in the example. Trying to access an item beyond the array's bounds (e.g. `vector[100]`) will likely crash your program.

Especially important are the parts of code starting with `for (int i = 0; i < 5; ++i)`: this is an iteration over the array. It's a very common pattern that we use whenever we need to perform some action with every item of the array.

Arrays can also be multidimensional, but we won't bother with that right now.

Why are arrays so important? They allow us to work with great number of data, not just a handful of numeric variables. We can create an array of million structs and easily work with all of them thanks to indexing and loops, this would be practically impossible without arrays. Imagine e.g. a game of chess; it would be very silly to have 64 plain variables for each square of the board (squareA1, squareA2, ..., squareH8), it would be extremely difficult to work with such code. With an array we can represent the square as a single array, we can iterate over all the squares easily etc.

One more thing to mention about arrays is how they can be passed to functions. A function can have as a parameter an array of fixed or unknown length. There is also one exception with arrays as opposed to other types: **if a function has an array as parameter and the function modifies this array, the array passed to the function (the argument) will be modified as well** (we say that arrays are *passed by reference* while other types are *passed by value*). We know this wasn't the case with other parameters such as `int` -- for these the function makes a local copy that doesn't affect the argument passed to the function. The following example shows what's been said:

```
#include <stdio.h>

// prints an int array of length 10
void printArray10(int array[10])
{
    for (int i = 0; i < 10; ++i)
        printf("%d ",array[i]);
}

// prints an int array of arbitrary length
void printArrayN(int array[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ",array[i]);
}

// fills an array with numbers 0, 1, 2, ...
void fillArrayN(int array[], int n)
{
    for (int i = 0; i < n; ++i)
        array[i] = i;
}

int main(void)
{
    int array10[10];
    int array20[20];

    fillArrayN(array10,10);
    fillArrayN(array20,20);

    printArray10(array10);
    putchar('\n');
    printArrayN(array20,20);

    return 0;
}
```

The function `printArray10` has a fixed length array as a parameter (`int array[10]`) while `printArrayN` takes as a parameter an array of unknown length (`int array[]`) plus one additional parameter to specify this length (so that the function knows how many items of the array it should print). The function `printArray10` is important because it shows how a function can modify an array: when we call `fillArrayN(array10,10)`; in the main function, the array `array10` will be actually modified after when the function finishes (it will be filled with numbers 0, 1, 2, ...). This can't be done with other data types (though there is a trick involving pointers which we will learn later).

Now let's finally talk about **text strings**. We've already seen strings (such as "hello"), we know we can print them, but what are they really? A string is a data type, and from C's point of view strings are nothing but **arrays of chars** (text characters), i.e. sequences of chars in memory. **In C every string has to end with a 0 char** -- this is NOT '0' (whose ASCII value is 48) but the direct value 0 (remember that chars are really just numbers). The 0 char cannot be printed out, it is just a helper value to terminate strings. So to

store a string "hello" in memory we need an array of length at least 6 -- one for each character plus one for the terminating 0. These types of string are called **zero terminated strings** (or *C strings*).

When we write a string such as "hello" in our source, the C compiler creates an array in memory for us and fills it with characters 'h', 'e', 'l', 'l', 'o', 0. In memory this may look like a sequence of numbers 104, 101, 108, 108 111, 0.

Why do we terminate strings with 0? Because functions that work with strings (such as puts or printf) don't know what length the string is. We can call puts("abc"); or puts("abcdefghijk"); -- the string passed to puts has different length in each case, and the function doesn't know this length. But thanks to these strings ending with 0, the function can compute the length, simply by counting characters from the beginning until it finds 0 (or more efficiently it simply prints characters until it finds 0).

The syntax that allows us to create strings with double quotes (") is just a helper (*syntactic sugar*); we can create strings just as any other array, and we can work with them the same. Let's see an example:

```
#include <stdio.h>

int main(void)
{
    char alphabet[27]; // 26 places for letters + 1 for terminating 0

    for (int i = 0; i < 26; ++i)
        alphabet[i] = 'A' + i;

    alphabet[26] = 0; // terminate the string

    puts(alphabet);

    return 0;
}
```

alphabet is an array of chars, i.e. a string. Its length is 27 because we need 26 places for letters and one extra space for the terminating 0. Here it's important to remind ourselves that we count from 0, so the alphabet can be indexed from 0 to 26, i.e. 26 is the last index we can use, doing alphabet[27] would be an error! Next we fill the array with letters (see how we can treat chars as numbers and do 'A' + i). We iterate while i < 26, i.e. we will fill all the places in the array up to the index 25 (including) and leave the last place (with index 26) empty for the terminating 0. That we subsequently assign. And finally we print the string with puts(alphabet) -- here note that there are no double quotes around alphabet because it's a variable name. Doing puts("alphabet") would cause the program to literally print out alphabet. Now the program outputs:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

In C there is a standard library for working with strings called *string* (#include <string.h>), it contains such function as strlen for computing string length or strcmp for comparing strings.

One final example -- a creature generator -- will show all the three new data types in action:

```
#include <stdio.h>
#include <stdlib.h> // for rand()

typedef struct
{
    char name[4]; // 3 letter name + 1 place for 0
    int weightKg;
    int legCount;
} Creature; // some weird creature

Creature creatures[100]; // global array of Creatures

void printCreature(Creature c)
{
    printf("Creature named %s ", c.name); // %s prints a string
    printf("(%d kg, ", c.weightKg);
    printf("%d legs)\n", c.legCount);
}
```

```

int main(void)
{
    // generate random creatures:

    for (int i = 0; i < 100; ++i)
    {
        Creature c;

        c.name[0] = 'A' + (rand() % 26);
        c.name[1] = 'a' + (rand() % 26);
        c.name[2] = 'a' + (rand() % 26);
        c.name[3] = 0; // terminate the string

        c.weightKg = 1 + (rand() % 1000);
        c.legCount = 1 + (rand() % 10); // 1 to 10 legs

        creatures[i] = c;
    }

    // print the creatures:

    for (int i = 0; i < 100; ++i)
        printCreature(creatures[i]);

    return 0;
}

```

When run you will see a list of 100 randomly generated creatures which may start e.g. as:

```

Creature named Nwl (916 kg, 4 legs)
Creature named Bmq (650 kg, 2 legs)
Creature named Cda (60 kg, 4 legs)
Creature named Owk (173 kg, 7 legs)
Creature named Hid (430 kg, 3 legs)
...

```

Macros/Preprocessor

The C language comes with a feature called *preprocessor* which is necessary for some advanced things. It allows automatized modification of the source code before it is compiled.

Remember how we said that compiler compiles C programs in several steps such as generating object files and linking? There is one more step we didn't mention: **preprocessing**. It is the very first step -- the source code you give to the compiler first goes to the preprocessor which modifies it according to special commands in the source code called **preprocessor directives**. The result of preprocessing is a pure C code without any more preprocessing directives, and this is handed over to the actual compilation.

The preprocessor is like a **mini language on top of the C language**, it has its own commands and rules, but it's much more simple than C itself, for example it has no data types or loops.

Each directive begins with #, is followed by the directive name and continues until the end of the line (\ can be used to extend the directive to the next line).

We have already encountered one preprocessor directive: the `#include` directive when we included library header files. This directive pastes a text of the file whose name it is handed to the place of the directive.

Another directive is `#define` which creates so called macro -- in its basic form a macro is nothing else than an alias, a nickname for some text. This is used to create constants. Consider the following code:

```

#include <stdio.h>

#define ARRAY_SIZE 10

int array[ARRAY_SIZE];

```

```

void fillArray(void)
{
    for (int i = 0; i < ARRAY_SIZE; ++i)
        array[i] = i;
}

void printArray(void)
{
    for (int i = 0; i < ARRAY_SIZE; ++i)
        printf("%d ",array[i]);
}

int main()
{
    fillArray();
    printArray();
    return 0;
}

```

`#define ARRAY_SIZE 10` creates a macro that can be seen as a constant named `ARRAY_SIZE` which stands for 10. From this line on any occurrence of `ARRAY_SIZE` that the preprocessor encounters in the code will be replaced with 10. The reason for doing this is obvious -- we respect the DRY (don't repeat yourself) principle, if we didn't use a constant for the array size and used the direct numeric value 10 in different parts of the code, it would be difficult to change them all later, especially in a very long code, there's a danger we'd miss some. With a constant it is enough to change one line in the code (e.g. `#define ARRAY_SIZE 10` to `#define ARRAY_SIZE 20`).

The macro substitution is literally a copy-paste text replacement, there is nothing very complex going on. This means you can create a nickname for almost anything (for example you could do `#define when if` and then also use `when` in place of `if` -- but it's probably not a very good idea). By convention macro names are to be `ALL_UPPER_CASE` (so that whenever you see an all upper case word in the source code, you know it's a macro).

Macros can optionally take parameters similarly to functions. There are no data types, just parameter names. The usage is demonstrated by the following code:

```

#include <stdio.h>

#define MEAN3(a,b,c) (((a) + (b) + (c)) / 3)

int main()
{
    int n = MEAN3(10,20,25);

    printf("%d\n",n);

    return 0;
}

```

`MEAN3` computes the mean of 3 values. Again, it's just text replacement, so the line `int n = MEAN3(10,20,25);` becomes `int n = (((10) + (20) + (25)) / 3);` before code compilation. Why are there so many brackets in the macro? It's always good to put brackets over a macro and all its parameters because the parameters are again a simple text replacement; consider e.g. a macro `#define HALF(x) x / 2` -- if it was invoked as `HALF(5 + 1)`, the substitution would result in the final text `5 + 1 / 2`, which gives 5 (instead of the intended value 3).

You may be asking why would we use a macro when we can use a function for computing the mean? Firstly macros don't just have to work with numbers, they can be used to generate parts of the source code in ways that functions can't. Secondly using a macro may sometimes be simpler, it's shorter and will be faster to execute because there is no function call (which has a slight overhead) and because the macro expansion may lead to the compiler precomputing expressions at compile time. But beware: macros are usually worse than functions and should only be used in very justified cases. For example macros don't know about data types and cannot check them, and they also result in a bigger compiled executable (function code is in the executable only once whereas the macro is expanded in each place where it is used and so the code it generates multiplies).

Another very useful directive is `#if` for conditional inclusion or exclusion of parts of the source code. It is similar to the C `if` command. The following example shows its use:

```
#include <stdio.h>

#define RUDE 0

void printNumber(int x)
{
    puts(
#ifdef RUDE
        "You idiot, the number is:"
#else
        "The number is:"
#endif
    );

    printf("%d\n",x);
}

int main()
{
    printNumber(3);
    printNumber(100);

#ifdef RUDE
    puts("Bye bitch.");
#endif

    return 0;
}
```

When run, we get the output:

```
The number is:
3
The number is:
100
```

And if we change `#define RUDE 0` to `#define RUDE 1`, we get:

```
You idiot, the number is:
3
You idiot, the number is:
100
Bye bitch.
```

We see the `#if` directive has to have a corresponding `#endif` directive that terminates it, and there can be an optional `#else` directive for an *else* branch. The condition after `#if` can use similar operators as those in C itself (`+`, `==`, `&&`, `||` etc.). There also exists an `#ifdef` directive which is used the same and checks if a macro of given name has been defined.

`#if` directives are very useful for conditional compilation, they allow for creation of various "settings" and parameters that can fine-tune a program -- you may turn specific features on and off with this directive. It is also helpful for portability; compilers may automatically define specific macros depending on the platform (e.g. `_WIN64`, `__APPLE__`, ...) based on which you can trigger different code. E.g.:

```
#ifdef _WIN64
    puts("Your OS sucks.");
#endif
```

Let us talk about one more thing that doesn't fall under the preprocessor language but is related to constants: **enumerations**. Enumeration is a data type that can have values that we specify individually, for example:

```
typedef enum
{
```

```

APPLE,
PEAR,
TOMATO
} Fruit;

```

This creates a new data type `Fruit`. Variables of this type may have values `APPLE`, `PEAR` or `TOMATO`, so we may for example do `Fruit myFruit = APPLE;`. These values are in fact integers and the names we give them are just nicknames, so here `APPLE` is equal to 0, `PEAR` to 1 and `TOMATO` to 2.

Pointers

Pointers are an advanced topic that many people fear -- many complain they're hard to learn, others complain about memory unsafety and potential dangers of using pointers. These people are stupid, pointers are great.

But beware, there may be too much new information in the first read. Don't get scared, give it some time.

Pointers allow us to do certain advanced things such as allocate dynamic memory, return multiple values from functions, inspect content of memory or use functions in similar ways in which we use variables.

A **pointer** is essentially nothing complicated: it is a **data type that can hold a memory address** (plus an information about what data type should be stored at that address). An address is simply a number. Why can't we just use an `int` to store a memory address? Because the size of `int` and a pointer may differ, the size of pointer depends on each platform's address width. Besides this, as said, a pointer actually holds not only an address but also the information about the type it points to, which is a safety mechanism that will become clear later. It is also good when the compiler knows a certain variable is supposed to point to a memory rather than to hold a generic number -- this can all prevent bugs. I.e. pointers and generic integers are distinguished for the same reason other data types are distinguished -- in theory they don't have to be distinguished, but it's safer.

It is important to stress again that a pointer is not a pure address but it also knows about the data type it is pointing to, so there are many kinds of pointers: a pointer to `int`, a pointer to `char`, a pointer to a specific struct type etc.

A variable of pointer type is created similarly to a normal variable, we just add `*` after the data type, for example `int *x;` creates a variable named `x` that is a pointer to `int` (some people would write this as `int* x;`).

But how do we assign a value to the pointer? To do this, we need an address of something, e.g. of some variable. To get an address of a variable we use the `&` character, i.e. `&a` is the address of a variable `a`.

The last basic thing we need to know is how to **dereference** a pointer. Dereferencing means accessing the value at the address that's stored in the pointer, i.e. working with the pointed to value. This is again done (maybe a bit confusingly) with `*` character in front of a pointer, e.g. if `x` is a pointer to `int`, `*x` is the `int` value to which the pointer is pointing. An example can perhaps make it clearer.

```

#include <stdio.h>

int main(void)
{
    int normalVariable = 10;
    int *pointer;

    pointer = &normalVariable;

    printf("address in pointer: %p\n",pointer);
    printf("value at this address: %d\n",*pointer);

    *pointer = *pointer + 10;

    printf("normalVariable: %d\n",normalVariable);

    return 0;
}

```

```
}
```

This may print e.g.:

```
address in pointer: 0x7fff226fe2ec
value at this address: 10
normalVariable: 20
```

`int *pointer;` creates a pointer to `int` with name `pointer`. Next we make the pointer point to the variable `normalVariable`, i.e. we get the address of the variable with `&normalVariable` and assign it normally to `pointer`. Next we print firstly the address in the pointer (accessed with `pointer`) and the value at this address, for which we use dereference as `*pointer`. At the next line we see that we can also use dereference for writing to the pointed address, i.e. doing `*pointer = *pointer + 10`; here is the same as doing `normalVariable = normalVariable + 10`; The last line shows that the value in `normalVariable` has indeed changed.

IMPORTANT NOTE: You generally cannot read and write from/to random addresses! This will crash your program. To be able to write to a certain address it must be allocated, i.e. reserved for use. Addresses of variables are allocated by the compiler and can be safely operated with.

There's a special value called `NULL` (a macro defined in the standard library) that is meant to be assigned to pointer that points to "nothing". So when we have a pointer `p` that's currently not supposed to point to anything, we do `p = NULL`; In a safe code we should always check (with `if`) whether a pointer is not `NULL` before dereferencing it, and if it is, then NOT dereference it. This isn't required but is considered a "good practice" in safe code, storing `NULL` in pointers that point nowhere prevents dereferencing random or unallocated addresses which would crash the program.

But what can pointers be good for? Many things, for example we can kind of "store variables in variables", i.e. a pointer is a variable which says which variable we are now using, and we can switch between variable any time. E.g.:

```
#include <stdio.h>

int bankAccountMonica = 1000;
int bankAccountBob = -550;
int bankAccountJose = 700;

int *payingAccount; // pointer to who's currently paying

void payBills(void)
{
    *payingAccount -= 200;
}

void buyFood(void)
{
    *payingAccount -= 50;
}

void buyGas(void)
{
    *payingAccount -= 20;
}

int main(void)
{
    // let Jose pay first

    payingAccount = &bankAccountJose;

    payBills();
    buyFood();
    buyGas();

    // that's enough, now let Monica pay

    payingAccount = &bankAccountMonica;
```

```

    buyFood();
    buyGas();
    buyFood();
    buyFood();

    // now it's Bob's turn

    payingAccount = &bankAccountBob;

    payBills();
    buyFood();
    buyFood();
    buyGas();

    printf("Monika has %d left.\n",bankAccountMonica);
    printf("Jose has %d left.\n",bankAccountJose);
    printf("Bob has %d left.\n",bankAccountBob);

    return 0;
}

```

Well, this could be similarly achieved with arrays, but pointers have more uses. For example they allow us to **return multiple values by a function**. Again, remember that we said that (with the exception of arrays) a function cannot modify a variable passed to it because it always makes its own local copy of it? We can bypass this by, instead of giving the function the value of the variable, giving it the address of the variable. The function can read the value of that variable (with dereference) but it can also **CHANGE** the value, it simply writes a new value to that address (again, using dereference). This example shows it:

```

#include <stdio.h>
#include <math.h>

#define PI 3.141592

// returns 2D coordinates of a point on a unit circle
void getUnitCirclePoint(float angle, float *x, float *y)
{
    *x = sin(angle);
    *y = cos(angle);
}

int main(void)
{
    for (int i = 0; i < 8; ++i)
    {
        float pointX, pointY;

        getUnitCirclePoint(i * 0.125 * 2 * PI,&pointX,&pointY);

        printf("%lf %lf\n",pointX,pointY);
    }

    return 0;
}

```

Function `getUnitCirclePoint` doesn't return any value in the strict sense, but thank to pointers it effectively returns two float values via its parameters `x` and `y`. These parameters are of the data type pointer to float (as there's `*` in front of them). When we call the function with `getUnitCirclePoint(i * 0.125 * 2 * PI,&pointX,&pointY);`, we hand over the addresses of the variables `pointX` and `pointY` (which belong to the main function and couldn't normally be accessed in `getUnitCirclePoint`). The function can then compute values and write them to these addresses (with dereference, `*x` and `*y`), changing the values in `pointX` and `pointY`, effectively returning two values.

Now let's take a look at pointers to structs. Everything basically works the same here, but there's one thing to know about, a **syntactic sugar** known as an arrow (`->`). Example:

```

#include <stdio.h>

```

```

typedef struct
{
    int a;
    int b;
} SomeStruct;

SomeStruct s;
SomeStruct *sPointer;

int main(void)
{
    sPointer = &s;

    (*sPointer).a = 10; // without arrow
    sPointer->b = 20;    // same as (*sPointer).b = 20

    printf("%d\n", s.a);
    printf("%d\n", s.b);

    return 0;
}

```

Here we are trying to write values to a struct through pointers. Without using the arrow we can simply dereference the pointer with `*`, put brackets around and access the member of the struct normally. This shows the line `(*sPointer).a = 10;`. Using an arrow achieves the same thing but is perhaps a bit more readable, as seen in the line `sPointer->b = 20;`. The arrow is simply a special shorthand and doesn't need any brackets.

Now let's talk about arrays -- these are a bit special. The important thing is that **an array is itself basically a pointer**. What does this mean? If we create an array, let's say `int myArray[10];`, then `myArray` is basically a pointer to `int` in which the address of the first array item is stored. When we index the array, e.g. like `myArray[3] = 1;`, behind the scenes there is basically a dereference because the index 3 means: 3 places after the address pointed to by `myArray`. So when we index an array, the compiler takes the address stored in `myArray` (the address of the array start) and adds 3 to it (well, kind of) by which it gets the address of the item we want to access, and then dereferences this address.

Arrays and pointer are kind of a duality -- we can also use array indexing with pointers. For example if we have a pointer declared as `int *x;`, we can access the value `x` points to with a dereference (`*x`), but ALSO with indexing like this: `x[0]`. Accessing index 0 simply means: take the value stored in the variable and add 0 to it, then dereference it. So it achieves the same thing. We can also use higher indices (e.g. `x[10]`), BUT ONLY if `x` actually points to a memory that has at least 11 allocated places.

This leads to a concept called **pointer arithmetic**. Pointer arithmetic simply means we can add or subtract numbers to pointer values. If we continue with the same pointer as above (`int *x;`), we can actually add numbers to it like `*(x + 1) = 10;`. What does this mean?! It means exactly the same thing as `x[1]`. Adding a number to a pointer shifts that pointer given number of *places* forward. We use the word *places* because each data type takes a different space in memory, for example `char` takes one byte of memory while `int` takes usually 4 (but not always), so shifting a pointer by *N* places means adding *N* times the size of the pointed to data type to the address stored in the pointer.

This may be a lot information to digest. Let's provide an example to show all this in practice:

```

#include <stdio.h>

// our own string print function
void printString(char *s)
{
    int position = 0;

    while (s[position] != 0)
    {
        putchar(s[position]);
        position += 1;
    }
}

```



```
// returns the length of string s
int stringLength(char *s)
{
    int length = 0;

    while (*s != 0) // count until terminating 0
    {
        length += 1;
        s += 1; // shift the pointer one character to right
    }

    return length;
}

int main(void)
{
    char testString[] = "catdog";

    printString("The string '");
    printString(testString);
    printString("' has length ");

    int l = stringLength(testString);

    printf("%d.", l);

    return 0;
}
```

The output is:

The string 'catdog' has length 6.

We've created a function for printing strings (`printString`) similar to `puts` and a function for computing the length of a string (`stringLength`). They both take as an argument a pointer to `char`, i.e. a string. In `printString` we use indexing (`[` and `]`) just as if `s` was an array, and indeed we see it works! In `stringLength` we similarly iterate over all characters in the string but we use dereference (`*s`) and pointer arithmetic (`s += 1`). It doesn't matter which of the two styles we choose -- here we've shown both, for educational purposes. Finally notice that the string we actually work with is created in `main` as an array with `char testString[] = "catdog";` -- here we don't need to specify the array size between `[` and `]` because we immediately assign a string literal to it (`"catdog"`) and in such a case the compiler knows how big the array needs to be and automatically fills in the correct size.

Now that know about pointers, we can finally completely explain the functions from `stdio` we've been using:

- `int puts(char *s)`: A simple and fast function for printing a string (adds the newline character `\n` at the end).
- `int printf(char *format, ...)`: A little bit more complex function that can print not only strings but also other data types. It takes a variable number of parameters. The first one is always a string that specifies the print format -- this string can contain special sequences that will be replaced by textual representations of values we additionally provide as extra parameters after `format`. E.g. the sequence `"%d"` is replaced with a number obtained from the value of a corresponding `int` parameter. Similarly `%c` is for `char`, `%s` for strings, `%p` for pointers. Example: `printf("MyInt = %d, myChar = %c, MyStr = %s\n", myInt, myChar, myStr);`.
- `int getchar(void)`: Reads a single text character from the input and returns it. Why does the function return `int` and not `char`? Because the function can return additional special values such as EOF (end of file) which couldn't be stored in plain `char`.
- `int scanf(char *format, ...)`: Function for reading various data types from the input. Like `printf` it takes a variable number of parameters. The first one is a string that specifies which data type(s) to read -- this is a bit complicated but `"%d"` reads an `int`, `"%f"` float, `"%c"` `char` and `"%s"` string. The following arguments are **pointers** to expected data types, so e.g. if we've provided the format string `"%d"`, a pointer to `int` has to follow. Through this parameter the value that's been read will be returned (in the same way we've seen in one example above).

Files

Now we'll take a look at how we can read and write from/to files on the computer disk which enables us to store information permanently or potentially process data such as images or audio. Files aren't so difficult.

We work with files through functions provided in the *stdio* library (so it has to be included). We distinguish two types of files:

- **text files:** Contain text, are human readable.
- **binary files:** Contain binary data, aren't human readable, are more efficient but also more prone to corruption.

From programmer's point of view there's actually not a huge difference between the two, they're both just sequences of characters or bytes (which are kind of almost the same). Text files are a little more abstract, they handle potentially different format of newlines etc. The main thing for us is that we'll use slightly different functions for each type.

There is a special data type for file called `FILE` (we'll be using a pointer to it). Whatever file we work with, we need to firstly open it with the function `fopen` and when we're done with it, we need to close it with a function `fclose`.

First we'll write something to a text file:

```
#include <stdio.h>

int main(void)
{
    FILE *textFile = fopen("test.txt","w"); // "w" for write

    if (textFile != NULL) // if opened successfully
        fprintf(textFile,"Hello file.");
    else
        puts("ERROR: Couldn't open file.");

    fclose(textFile);

    return 0;
}
```

When run, the program should create a new file named *test.txt* in the same directory we're in and in it you should find the text *Hello file..* `FILE *textFile` creates a new variable `textFile` which is a pointer to the `FILE` data type. We are using a pointer simply because the standard library is designed this way, its functions work with pointers (it can be more efficient). `fopen("test.txt","w");` attempts to open the file *test.txt* in text mode for writing -- it returns a pointer that represents the opened file. The mode, i.e. text/binary, read/write etc., is specified by the second argument: `"w"`; *w* simply specifies *write* and the text mode is implicit (it doesn't have to be specified). `if (textFile != NULL)` checks if the file has been successfully opened; the function `fopen` returns `NULL` (the value of "point to nothing" pointers) if there was an error with opening the file (such as that the file doesn't exist). On success we write text to the file with a function `fprintf` -- it's basically the same as `printf` but works on files, so it's first parameter is always a pointer to a file to which it should write. You can of course also print numbers and anything that `printf` can with this function. Finally we mustn't forget to close the file at the end with `fclose`!

Now let's write another program that reads the file we've just created and writes its content out in the command line:

```
#include <stdio.h>

int main(void)
{
    FILE *textFile = fopen("test.txt","r"); // "r" for read

    if (textFile != NULL) // if opened successfully
    {
        char c;
```

```

    while (fscanf(textFile,"%c",&c) != EOF) // while not end of file
        putchar(c);
}
else
    puts("ERROR: Couldn't open file.");

fclose(textFile);

return 0;
}

```

Notice that in `fopen` we now specify `"r"` (read) as a mode. Again, we check if the file has been opened successfully (if `(textFile != NULL)`). If so, we use a while loop to read and print all characters from the file until we encounter the end of file. The reading of file characters is done with the `fscanf` function inside the loop's condition -- there's nothing preventing us from doing this. `fscanf` again works the same as `scanf` (so it can read other types than only chars), just on files (its first argument is the file to read from). On encountering end of file `fscanf` returns a special value `EOF` (which is macro constant defined in the standard library). Again, we must close the file at the end with `fclose`.

We will now write to a binary file:

```

#include <stdio.h>

int main(void)
{
    unsigned char image[] = // image in ppm format
    {
        80, 54, 32, 53, 32, 53, 32, 50, 53, 53, 32,
        255,255,255, 255,255,255, 255,255,255, 255,255,255, 255,255,255,
        255,255,255, 0, 0, 0, 255,255,255, 0, 0, 0, 255,255,255,
        255,255,255, 255,255,255, 255,255,255, 255,255,255, 255,255,255,
        0, 0, 0, 255,255,255, 255,255,255, 255,255,255, 0, 0, 0,
        255,255,255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255,255,255
    };

    FILE *binFile = fopen("image.ppm","wb");

    if (binFile != NULL) // if opened successfully
        fwrite(image,1,sizeof(image),binFile);
    else
        puts("ERROR: Couldn't open file.");

    fclose(binFile);

    return 0;
}

```

Okay, don't get scared, this example looks complex because it is trying to do a cool thing: it creates an image file! When run, it should produce a file named *image.ppm* which is a tiny 5x5 smiley face image in ppm format. You should be able to open the image in any good viewer (I wouldn't bet on Windows programs though). The image data was made manually and are stored in the `image` array. We don't need to understand the data, we just know we have some data we want to write to a file. Notice how we can manually initialize the array with values using `{` and `}` brackets. We open the file for writing and in binary mode, i.e. with a mode `"wb"`, we check the success of the action and then write the whole array into the file with one function call. The function is name `fwrite` and is used for writing to binary files (as opposed to `fprintf` for text files). `fwrite` takes these parameters: pointer to the data to be written to the file, size of one data element (in bytes), number of data elements and a pointer to the file to write to. Our data is the `image` array and since "arrays are basically pointers", we provide it as the first argument. Next argument is `1` (unsigned char always takes 1 byte), then length of our array (`sizeof` is a special operator that substitutes the size of a variable in bytes -- since each item in our array takes 1 byte, `sizeof(image)` provides the number of items in the array), and the file pointer. At the end we close the file.

And finally we'll finish with reading this binary file back:

```

#include <stdio.h>

```

```

int main(void)
{
    FILE *binFile = fopen("image.ppm","rb");

    if (binFile != NULL) // if opened successfully
    {
        unsigned char byte;

        while (fread(&byte,1,1,binFile))
            printf("%d ",byte);

        putchar('\n');
    }
    else
        puts("ERROR: Couldn't open file.");

    fclose(binFile);

    return 0;
}

```

The file mode is now "rb" (read binary). For reading from binary files we use the `fread` function, similarly to how we used `fscanf` for reading from a text file. `fread` has these parameters: pointer where to store the read data (the memory must have sufficient space allocated!), size of one data item, number of items to read and the pointer to the file which to read from. As the first argument we pass `&byte`, i.e. the address of the variable `byte`, next `1` (we want to read a single byte whose size in bytes is 1), `1` (we want to read one byte) and the file pointer. `fread` returns the number of items read, so the `while` condition holds as long as `fread` reads bytes; once we reach end of file, `fread` can no longer read anything and returns `0` (which in C is interpreted as a false value) and the loop ends. Again, we must close the file at the end.

More On Functions (Recursion, Function Pointers)

There's more to be known about functions.

An important concept in programming is recursion -- the situation in which a function calls itself. Yes, it is possible, but some rules have to be followed.

When a function calls itself, we have to ensure that we won't end up in infinite recursion (i.e. the function calls itself which subsequently calls itself and so on until infinity). This crashes our program. There always has to be a **terminating condition** in a recursive function, i.e. an `if` branch that will eventually stop the function from calling itself again.

But what is this even good for? Recursion is actually very common in math and programming, many problems are recursive in nature. Many things are beautifully described with recursion (e.g. fractals). But remember: anything a recursion can achieve can also be achieved by iteration (loop) and vice versa. It's just that sometimes one is more elegant or more computationally efficient.

Let's see this on a typical example of the mathematical function called factorial. Factorial of N is defined as $N \times (N - 1) \times (N - 2) \times \dots \times 1$. It can also be defined recursively as: factorial of N is 1 if N is 0, otherwise N times factorial of $N - 1$. Here is some code:

```

#include <stdio.h>

unsigned int factorialRecursive(unsigned int x)
{
    if (x == 0) // terminating condition
        return 1;
    else
        return x * factorialRecursive(x - 1);
}

unsigned int factorialIterative(unsigned int x)
{
    unsigned int result = 1;

```

```

    while (x > 1)
    {
        result *= x;
        x--;
    }

    return result;
}

int main(void)
{
    printf("%d %d\n", factorialRecursive(5), factorialIterative(5));
    return 0;
}

```

factorialIterative computes the factorial by iteration. factorialRecursive uses recursion -- it calls itself. The important thing is the recursion is guaranteed to end because every time the function calls itself, it passes a decremented argument so at one point the function will receive 0 in which case the terminating condition (if (x == 0)) will be triggered which will avoid the further recursive call.

It should be mentioned that performance-wise recursion is almost always worse than iteration (function calls have certain overhead), so in practice it is used sparingly. But in some cases it is very well justified (e.g. when it makes code much simpler while creating unnoticeable performance loss).

Another thing to mention is that we can have **pointers to functions**; this is an advanced topic so we'll stay at it just briefly. Function pointers are pretty powerful, they allow us to create so called callbacks: imagine we are using some GUI framework and we want to tell it what should happen when a user clicks on a specific button -- this is usually done by giving the framework a pointer to our custom function that it will be called by the framework whenever the button is clicked.

Dynamic Allocation (Malloc)

Dynamic memory allocation means the possibility of reserving additional memory (RAM) for our program at run time, whenever we need it. This is opposed to static memory allocation, i.e. reserving memory for use at compile time (when compiling, before the program runs). We've already been doing static allocation whenever we created a variable -- compiler automatically reserves as much memory for our variables as is needed. But what if we're writing a program but don't yet know how much memory it will need? Maybe the program will be reading a file but we don't know how big that file is going to be -- how much memory should we reserve? Dynamic allocation allows us to reserve this memory with functions when the program is actually running and already knows how much of it should be reserved.

It must be known that dynamic allocation comes with a new kind of bug known as a **memory leak**. It happens when we reserve a memory and forget to free it after we no longer need it. If this happens e.g. in a loop, the program will continue to "grow", eat more and more RAM until operating system has no more to give. For this reason, as well as others such as simplicity, it may sometimes be better to go with only static allocation.

Anyway, let's see how we can allocate memory if we need to. We use mostly just two functions that are provided by the *stdlib* library. One is `malloc` which takes as an argument size of the memory we want to allocate (reserve) in bytes and returns a pointer to this allocated memory if successful or `NULL` if the memory couldn't be allocated (which in serious programs we should always check). The other function is `free` which frees the memory when we no longer need it (every allocated memory should be freed at some point) -- it takes as the only parameter a pointer to the memory we've previously allocated. There is also another function called `realloc` which serves to change the size of an already allocated memory: it takes a pointer to the allocated memory and the new size in byte, and returns the pointer to the resized memory.

Here is an example:

```

#include <stdio.h>
#include <stdlib.h>

#define ALLOCATION_CHUNK 32 // by how many bytes to resize

```

```

int main(void)
{
    int charsRead = 0;
    int resized = 0; // how many times we called realloc
    char *inputChars = malloc(ALLOCATION_CHUNK * sizeof(char)); // first allocation

    while (1) // read input characters
    {
        char c = getchar();

        charsRead++;

        if ((charsRead % ALLOCATION_CHUNK) == 0)
        {
            inputChars = // we need more space, resize the array
                realloc(inputChars, (charsRead / ALLOCATION_CHUNK + 1) * ALLOCATION_CHUNK * sizeof(char));

            resized++;
        }

        inputChars[charsRead - 1] = c;

        if (c == '\n')
        {
            charsRead--; // don't count the last newline character
            break;
        }
    }

    puts("The string you entered backwards:");

    while (charsRead > 0)
    {
        putchar(inputChars[charsRead - 1]);
        charsRead--;
    }

    free(inputChars); // important!

    putchar('\n');
    printf("I had to resize the input buffer %d times.",resized);

    return 0;
}

```

This code reads characters from the input and stores them in an array (inputChars) -- the array is dynamically resized if more characters are needed. (We restrain from calling the array inputChars a string because we never terminate it with 0, we couldn't print it with standard functions like puts.) At the end the entered characters are printed backwards (to prove we really stored all of them), and we print out how many times we needed to resize the array.

We define a constant (macro) ALLOCATION_CHUNK that says by how many characters we'll be resizing our character buffer. I.e. at the beginning we create a character buffer of size ALLOCATION_CHUNK and start reading input character into it. Once it fills up we resize the buffer by another ALLOCATION_CHUNK characters and so on. We could be resizing the buffer by single characters but that's usually inefficient (the function malloc may be quite complex and take some time to execute).

The line starting with char *inputChars = malloc(...) creates a pointer to char -- our character buffer -- to which we assign a chunk of memory allocated with malloc. Its size is ALLOCATION_CHUNK * sizeof(char). Note that for simplicity we don't check if inputChars is not NULL, i.e. whether the allocation succeeded -- but in your program you should do it :) Then we enter the character reading loop inside which we check if the buffer has filled up (if ((charsRead % ALLOCATION_CHUNK) == 0)). If so, we used the realloc function to increase the size of the character buffer. The important thing is that once we exit the loop and print the characters stored in the buffer, we free the memory with free(inputChars); as we no longer need it.

Debugging, Optimization

Debugging means localizing and fixing bugs (errors) in your program. In practice there are always bugs, even in very short programs (you've probably already figured that out yourself), some small and insignificant and some pretty bad ones that make your program unusable, vulnerable or even dangerous.

There are two kinds of bugs: **syntactic errors** and **semantic errors**. A syntactic error is when you write something not obeying the C grammar, it's like a typo or grammatical error in a normal language -- these errors are very easy to detect and fix, a compiler won't be able to understand your program and will point you to the exact place where the error occurs. A semantic error can be much worse -- it's a logical error in the program; the program will compile and run but the program will behave differently than intended. The program may crash, leak memory, give wrong results, run slowly, corrupt files etc. These errors may be hard to spot and fix, especially when they happen in rare situations. We'll be only considering semantic errors from now on.

If we spot a bug, how do we fix it? The first thing is to find a way to **replicate** it, i.e. find the exact steps we need to make with the program to make the bug appear (e.g. "in the menu press keys A and B simultaneously", ...). Next we need to trace and locate which exact line or piece of code is causing the bug. This can either be done with the help of specialized debuggers such as gdb or valgrind, but there's usually a much easier way of using printing functions such as `printf`. (Still do check out the above mentioned debuggers, they're very helpful.)

Let's say your program crashes and you don't know at which line. You simply put prints such as `printf("A\n");` and `printf("B\n");` at the beginning and end of a code you suspect might be causing the crash. Then you run the program: if A is printed but B isn't, you know the crash happened somewhere between the two prints, so you shift the B print a little bit up and so on until you find exactly after which line B stops printing -- this is the line that crashes the program. IMPORTANT NOTE: the prints have to have newline (`\n`) at the end, otherwise this method may not work because of output buffering.

Of course, you may use the prints in other ways, for example to detect at which place a value of variable changes to a wrong value. (Asserts are also good for keeping an eye on correct values of variables.)

What if the program isn't exactly crashing but is giving wrong results? Then you need to trace the program step by step (not exactly line by line, but maybe function by function) and check which step has a problem in it. If for example your game AI is behaving stupid, you firstly check (with prints) if it correctly detects its circumstances, then you check whether it makes the correct decision based on the circumstances, then you check whether the pathfinding algorithm finds the correct path etc. At each step you need to know what the correct behavior should be and you try to find where the behavior is broken.

Knowing how to fix a bug isn't everything, we also need to find the bugs in the first place. **Testing** is the process of trying to find bugs by simply running and using the program. Remember, testing can't prove there are no bugs in the program, it can only prove bugs exist. You can do testing manually or automate the tests. Automated tests are very important for preventing so called **regressions** (so the tests are called regression tests). Regression happens when during further development you break some of its already working features (it is very common, don't think it won't be happening to you). Regression test (which can simply be just a normal C program) simply automatically checks whether the already implemented functions still give the same results as before (e.g. if $\sin(0) = 0$ etc.). These tests should be run and pass before releasing any new version of the program (or even before any commit of new code).

Optimization is also a process of improving an already working program, but here we try to make the program more efficient -- the most common goal is to make the program faster, smaller or consume less RAM. This can be a very complex task, so we'll only mention it briefly.

The very basic thing we can do is to turn on automatic optimization with a compiler flag: `-O3` for speed, `-Os` for program size (`-O2` and `-O1` are less aggressive speed optimizations). Yes, it's that simple, you simply add `-O3` and your program gets magically faster. Remember that **optimizations against different resources are often antagonistic**, i.e. speeding up your program typically makes it consume more memory and vice versa. You need to choose. Optimizing manually is a great art. Let's suppose you are optimizing for speed -- the first, most important thing is to locate the part of code that's slowing down your program the most, so called **bottleneck**. That is the code you want to make faster. Trying to optimize non-bottlenecks doesn't

speed up your program as a whole much; imagine you optimize a part of the code that takes 1% of total execution time by 200% -- your program will only get 0.5% faster. Bottlenecks can be found using profiling -- measuring the execution time of different parts of the program (e.g. each function). This can be done manually or with tools such as gprof. Once you know where to optimize, you try to apply different techniques: using algorithms with better time complexity, using look up tables, optimizing cache behavior and so on. This is beyond the scope of this tutorial.

Final Program

TODO

Where To Go Next

We haven't nearly covered the whole of C, but you should have pretty solid basics now. Now you just have to go and write a lot of C programs, that's the only way to truly master C. WARNING: Do not start with an ambitious project such as a 3D game. You won't make it and you'll get demotivated. Start very simple (a Tetris clone perhaps?). Try to develop some consistent programming style/formatting -- see our LRS programming style you may adopt (it's better than trying to make your own really).

You should definitely learn about common data structures (linked lists, binary trees, hash tables, ...) and algorithms (sorting, searching, ...). As an advanced programmer you should definitely know a bit about memory management. Also take a look at basic licensing. Another thing to learn is some version control system, preferably git, because this is how we manage bigger programs and how we collaborate on them. To start making graphical programs you should get familiar with some library such as SDL.

A great amount of experience can be gained by contributing to some existing project, collaboration really boosts your skill and knowledge of the language. This should only be done when you're at least intermediate. Firstly look up a nice project on some git hosting site, then take a look at the bug tracker and pick a bug or feature that's easy to fix or implement (low hanging fruit).

culture

Culture

Culture is an abstract term that includes behavioral norms, common beliefs, moral values, habits and similar concept within certain society or some smaller group within it. It's the unwritten rules, what people generally do, what they like and appreciate, what's considered rude, what they strive for, what they dislike, how they react to things and so on -- culture is therefore connected to other attributes of society such as language, art and law -- they all influence each other.

Culture is more important than laws as culture is the strongest force defining how we live the majority of our lives, what actions we take and how they are judged by others; law may try to capture some cultural demands, albeit in quite simplified and limited way, and enforce them, however courts, police and prison only step in in absolute extreme cases. You do many more things (such as eating meat, cutting your hair, watching TV or living with a single partner) because of culture, not because you are obliged by law. There aren't even enough policemen to guarantee law enforcement in all cases and all states rely (by basically not even having any other choice) on culture doing most of the job in keeping society working (which is also exploited by states and corporations when they try to manipulate culture with propaganda rather than changing laws). Consider for example that you download a random photo from the internet and set it as a wallpaper on your computer -- officially you have committed a crime of piracy as you had no rights for downloading the image, however culturally no one sees this as harmful, no one is going to bully you, sue you and even if someone tried to sue you, no judge would actually punish such a laughable "crime". On the other hand if you do a legal but culturally unacceptable thing, such as making a public art exhibition of non-sexual photos of naked children (also notice this might have been culturally OK to do in the previous century, but not now), you will be culturally punished by everyone distancing themselves from you and someone perhaps even illegally physically attacking you. A sentence, such as "black people aren't as intelligent as white people", spoken half a century ago may nowadays be judged by a court in a much different way just by the context of today's culture and even under the same set of laws in the past you would not have been

convicted of a crime while nowadays you would, as legal terms are eventually at some level defined in a plain language, which is permeated by culture. Therefore in trying to change society we should remember two things:

1. Focus on laws is a short term necessary evil.
2. Focus on culture (and eventual elimination of law as such) is our long term focus.

Types Of Culture

Here are some notable, named cultural patterns:

- **cancel culture**
 - **fear culture**
 - **fight culture**
 - **free culture**
 - **hacker culture**
 - **hero culture**
 - **offended culture**
 - **permission culture**
 - **remix culture**
 - **rights culture**
 - **update culture**
 - ...
-

cyber

Cyber

Cyber (taken from cybernetics, the theory of communication and control, coming from Greek *kybernetes*, *steersman*) is a word prefix that signifies relatedness to computers, especially computer networks and Internet. It is nowadays used mainly for relating old concept to the modern world dominated by computers. By itself or as a verb "cyber" often signifies *cybersex*. Some terms using the prefix include mentioned cybernetics, cyberpunk, cybersex, cyberspace, cybercrime, cyberbullying, cyberculture, cyborg, cybersecurity etc.

data_hoarding

Data Hoarding

TODO: is it based or is it a disease? Hoarding data on paper (books, encyclopedias, ...) is probably good.

data_structure

Data Structure

Not to be confused with data type.

Data structure refers to a any specific way in which data is organized in computer memory, which often comes with associated efficient operations on such data. A specific data structure describes such things as order, relationships (interconnection, hierarchy, ...), helper values (checksum, indices, ...), formats and types of parts of the data. Programming is sometimes seen as consisting mainly of two things: design of algorithms and data structures these algorithm work with.

As a programmer dealing with a specific problem you oftentimes have a choice of multiple data structures -- choosing the right one is essential for performance and efficiency of your program. As with everything, each data structure has advantages and also its downsides; some are faster, some take less memory etc. For example for a searchable database of text string we can be choosing between a binary tree and a hash table;

hash table offers theoretically much faster search, but binary trees may be more memory efficient and offer many other efficient operations like range search and sorting (which hash tables can do but very inefficiently).

What's the difference between data structure and (a potentially structured/complex) data type?

This can be tricky, in some specific cases the terms may even be interchanged without committing an error, but there is an important difference -- data structure is a PHYSICAL ORGANIZATION of data and though it's often associated with operations and algorithms (e.g. a binary tree comes with a natural search algorithm), the stress is on the layout of data in memory; on the other hand data type can be seen as a more abstract term defined by a SET OF ALLOWED VALUES and OPERATIONS on those values, usually without paying much attention to how those values and operations internally work, although in practice of course we rarely ignore this and often talk about a data type as being connected to specific data structure, which may be where the confusion comes from (also `struct` is a name of a data type in some languages, something potentially confusing as well). For example an ASCII text string is a data type, its set of values are all possible sequences of ASCII symbols and operations it allows are e.g. concatenation, substring search, substring replacement etc. This specific data type can be internally implemented differently, though one of the most natural ways is a "zero terminated string", i.e. array of values that always ends with value zero -- this is A DATA STRUCTURE. Because string, a data type, and zero terminated string (an array of values) are so closely connected, we may sometimes hear a *string* being called both a data type and data structure. However consider another example: a dictionary -- this is a DATA TYPE, very frequently used e.g. in Python, which allows storage of pairs of values; again dictionary itself is a data type defining only "how it behaves on the outside", but it can be implemented in several ways, for example with trees, hash tables or arrays, i.e. different DATA STRUCTURES. Different Python implementations will all offer the same dictionary data type but may use a different underlying data structure for it.

Specific Data Structures

These are just some common ones:

- array
- binary tree
- bitfield
- blockchain
- B+ tree
- circular buffer
- directed acyclic graph
- graph
- hash table
- heap
- linked list
- N-ary tree
- pascal string
- record
- stack
- zero terminated string
- tree
- tuple
- queue
- ...

See Also

- data
- data type

debugging

Debugging

WORK IN PROGRESS

Debugging is a term usually related to computer technology (but sometimes also extended beyond it) where it stands for the practice of actively searching for bugs (errors, design flaws, defects, ...) and fixing them; most typically it happens in software programming, but we may also talk about debugging hardware etc. Debugging is notoriously tedious and stressful, it can even take majority of the programmer's time and some bugs are extremely hard to track down, however systematic approaches can be applied to basically always succeed in fixing any bug. Debugging is sometimes humorously defined as "replacing old bugs with new ones".

Fun fact: the term *debugging* allegedly comes from the old times when it meant LITERALLY getting rid of bugs that broke computers by getting stuck in the relays.

Spare yourself debugging by testing as you go -- while programming it's best to at least quickly test the program is working after each small step change you make. Actually you should be writing **automatic tests** along with your main program that quickly tests that all you've programmed so far still works (see also regression). This way you discover a bug early and you know it's in the part you just changed so you find it and fix it quickly. If you don't do this and just write the whole program before even running it, your program will just crash and you won't have a clue why -- at this point you most likely have SEVERAL bugs working together and so even finding one or two of them will still leave your program crashing -- this situation is so shitty that the time you saved earlier won't nearly be worth it.

Debugging Software

Debugging programs mostly happens in these steps:

1. **Discovering bug:** you notice a bug, this usually happens during testing but of course can also just happen during normal use etc.
2. **Reproducing it:** reproducing the bug is extremely important -- actually you probably can't move on without it. Reproducing means finding an exact way to make the bug happen, e.g. "click this button while holding down this key" etc.
3. **Locating it:** now as you have a crashing program, you examine it and find WHY exactly it crashes, which line of code causes the crash etc.
4. **Fixing it:** now as you know why and where the bug exists, you just make it go away. Sometimes a hotfix is quickly applied before implementing a proper fix.

For debugging your program it may greatly help to make **debugging builds** -- you may pass flags to your compiler that make the program better for debugging, e.g. with gcc you will likely want to use -g (generate debug symbols) and -Og (optimize for debugging) flags. Without this debuggers will still work but may be e.g. unable to tell you the name of function inside which the program crashed etc.

Also as with everything you get better at debugging with practice, especially when you learn about common types of bugs and how they manifest -- for example you'll learn to just quickly scan for the famous off by one bugs near any loop, you'll learn that when a value grows and then jumps to zero it's an overflow, that your program getting stuck and crashing after a while could mean infinite recursion etc.

The following are some of the most common methods used to debug software, roughly in order by which one typically applies them in practice.

Testing

Testing is an area of itself, it's the main method of finding bugs. There are many kind of testing like manual testing (just playing around with the program), automatic testing (automatized testing by a program), security/penetration testing, stress testing, whitebox/blackbox testing, unit testing, code reviews and whatnot. **Formal verification** is similar to testing that can reveal further bugs, but it's more difficult to do.

Eyeballing

Quick way to spot small bugs is obviously to just look at the code, however this really works for the small, extremely obvious bugs like writing `=` instead of `==` etc.

Manual Execution

In this method you try to go through the program yourself step by step, just as the computer would. By this you will find out just WHY and WHERE your program gets to a wrong result or to a line that makes it crash.

Printing

Using print statements is extremely popular and efficient method of locating bugs; the idea is to use the language's print functions to log what's happening. By this you can e.g. find where exactly (which line of source code) your program crashes, you simply insert `printf("asdf\n");` somewhere and keep moving this print statement and re running the program until the text stops showing up on the screen - then you know the program crashes before it reached the print. Note that you can use the principle of binary search (also known as *wolf fence algorithm*) to move the print in the code so that you find the crash place relatively quickly. Besides this prints can of course also show you e.g. values in variables so you can e.g. check WHERE EXACTLY the value changes to a wrong value and so on.

The advantage of this is that you don't need any extra debugger, the method works basically everywhere and is actually very effective, it may be all you will need in 99% of cases. { TBH I don't even regularly use debugger, debugging with prints just works for me. ~drummyfish }

IMPORTANT NOTE especially for C programmers: output is usually line buffered, so in each print you HAVE TO add a newline (`\n`) at the end to make it print immediately. If you don't do this, it may happen that the print will be executed but the output will stay waiting in the output buffer as the program crashes so it won't show up on your screen. Similarly in other languages you may want to call some flush function etc.

Sometimes a bug can be super nasty and make the program crash always in random places, even depending e.g. on where you put the print statement, even if the print statement shouldn't really have an effect on the rest of the program. When you spot something like this, you probably have some super nasty bug related to undefined behavior or optimization, try to mess with optimization flags, use static analyzers, reduce your program to a minimum program that still bugs etc. This may take some time.

Rubber Duck

Rubber duck debugging works like this: you try to explain your code to someone -- even someone who doesn't understand programming, for example rubber duck -- and in doing this you often spot some error in reasoning. Explaining code to a programmer may have a further advantage as he may ask you clever questions.

Reducing Your Program To Minimum

When dealing with a super nasty bug in a complex program that's dodging solutions by the simpler methods, it is useful to just copy your program elsewhere and there strip down everything off of it while still keeping the bug in place. I.e. you just keep deleting functions and all the program does while making sure the bug you're after is still happening. This will firstly eliminate places where you have to look for the bug but mainly will usually lead you to reducing the program to just a few lines of code that behave extremely weirdly, like a function whose behavior depends on where you put a print statement or if you use a wider data type etc. Then you usually find the problematic line or whatever it is that's causing the bug and once you know the line, you can look at it really carefully, google the behavior of each operator etc. to really find the bug.

Debuggers And Other Debugging Tools Like Profilers

There exist many software tools specialized for just helping with debugging (there are even physical hardware debuggers etc.), they are either separate software (good) or integrated as part of some development environment (bad by Unix philosophy) such as an IDE, web browser etc. Nowadays a compiler

itself will typically do some basic checks and give you warning about many things, but oftentimes you have to turn them on (check man pages of your compiler).

The most typical debugging tool is a **debugger**, a program that lets you to play around with the program as it's running, it typically allows doing things like like:

- Step through the program line-by-line (typically there is are two options: step by lines and step by functions), sometimes even backwards in time.
- Run the program and pause it exactly where you need (breakpoints).
- Inspect values in RAM, CPU registers etc.
- Modify values in RAM, registers etc.
- Modify code on-the-fly.
- Assert if certain conditions hold.
- Link lines of assembly to lines in original source code.
- Warn about suspicious things.
- ...

Furthermore there many are other useful tools such as:

- **dynamic program analyzer**: A tool that will watch your program running and check for things like memory leaks, access of unallocated memory, suspicious behavior, unsafe behavior, call of obsolete functions and many others. The most famous such tool is probably **valgrind**, it's a good habit to just use valgrind from time to time to check our program.
- **profiler**: A kind of dynamic analyzer that focuses on statistical measuring of resource usage of your program, typically execution times of different functions, memory consumption or network usage. Basically it will watch your program run and then draw you graphs of which parts of your programs consume most resources. This usually helps optimization but can also serve to find bugs (you can spot where your memory starts leaking etc.). Some basic profiling can be done even without profiler, just inside your own program, but it can get tricky. One famous profiler is e.g. gprof.
- **static source code analyzer**: Static analyzers look at the source code (or potentially even compiled binary) and try to find bugs/inefficiencies in it without running it, just by reasoning. Static analyzers often tell you about undefined behavior, potential overflows, unused code, unreachable branches, unsatisfiable conditions, confusing formatting and so on. This complement the dynamic analysis. Some tools to do this are e.g. cppcheck and splint, though thanks to compilers performing a lot of static analysis themselves these seem not as widely used as dynamic analyzers nowadays.
- **hex editor**: Tool allowing you to mess with binary files, useful for any program that works with binary files. A simple hex viewer is e.g. hexdump.
- **emulators, virtual machines, ...**: Running your program on different platform often reveals bugs -- while your program may work perfectly fine on your computer, it may start crashing on another because that computer may have different integer size, endianness, amount of RAM, timings, file system etcetc. Emulators and VMs allow you to test exactly this AND furthermore often allow easy inspection of the emulated machine's memory and so on.
- ...

Shotgun Debugging

This is kind of an improper YOLO way of trying to fix bugs, you just change a lot of stuff in your program in hopes a bug will go away, it rarely works, doesn't really get rid of the bug (just of its manifestation at best) and can at best perhaps be a simple hotfix. Remember **if you don't understand how you fixed a bug, you didn't actually fix it.**

TODO: mini gdb tutorial

deep_blue

Deep Blue

Deep Blue was a legendary chess playing IBM supercomputer, which in 1997 made history by being the first ever computer to beat the human world chess champion at the time (Garry Kasparov), marking a moment

which many consider that at which "computers finally outsmarted humans". Since then computers really did continue to surpass humans at chess by much greater margins; nowadays a mere cellphone running stockfish can easily rape the world chess champion.

History: it all started around 1985 as a program called ChipTest by some Taiwanese guy with unpronounceable name. It went on to win some computer chess tournaments and when multiple people were already working on it as a part of IBM research, it was renamed to Deep Thought after the computer in Hitchhiker's Guide to the Galaxy, however **it had to be later renamed to Deep Blue because the name too much resembled deep throat** :D By 1990 it has already played the world champion, Kasparov, but lost. In 1996 Deep Blue played him again, this time losing the match again but already having won a game, showing the potential was there. In May 1997, after upgrade both in hardware and software, it finally beat Kasparov with 3 wins, 2 losses and 1 draw.

{ Lol, according to Wikipedoa it trolled Kasparov in the first game by making a completely random move due to a bug once, it scared him because he thought it was some deeply calculated threat while it was just some completely dumb move. ~drummyfish }

It's important to see that Deep Blue wasn't really a general chess engine like stockfish, it was a single purpose supercomputer, a combination of hardware and software engineered from the ground up with the single purpose: win the match against Garry Kasparov. It was being fine tuned in between the games with assistance of grandmasters. A team of experts on computers and chess focused their efforts on this single opponent at the specific time controls and match set up, rather than trying to make a generally usable chess computer. They studied Kasparov's play and made Deep Blue ready for it; they even employed psychological tricks -- for example it had preprogrammed instant responses to some Kasparov's expected moves, so as to make him more nervous.

Technical details: Deep Blue was mainly relying on massively parallel brute force, i.e. looking many moves ahead and consulting stored databases of games; in 1997 it had some 11 GFLOPS. The base computer was IBM RS/6000 SP (taking two cabinets) with IBM AIX operating system, using 32 PowerPC 200 MHz processors and 480 specialized "chess chips". It had evaluation function implemented in hardware. All in all the whole system could search hundreds of millions positions per second. Non-extended search was performed to a depth of about 12 plies, extended search went even over 40 plies deep. It had an opening book with about 4000 positions and endgame tablebases for up to 6 pieces. It was programmed in C. { Sources seems to sometimes give different numbers and specs, so not exactly sure about this. ~drummyfish }

See Also

- stockfish

de_facto

De Facto

De facto is Latin for "in fact" or "by facts", it means that something holds in practice; it is contrasted with de jure ("by law"). We use the term to say whether something is actually true in reality as opposed to "just on paper".

For example in technology a so called de facto standard is something that, without it being officially formalized or forced by law in prior, most developers naturally come to adopt so as to keep compatibility; for example the Markdown format has become the de facto standard for READMEs in FOSS development. Of course it happens often that de facto standards are later made into official standards. On the other hand there may be standards that are created by official standardizing authorities, such as the state, which however fail to gain wide adoption in practice -- these are official standards but not de facto one. TODO: example? :)

Regarding politics and society, we often talk about **de facto freedom** vs **de jure freedom**. For example in the context of free (as in freedom) software it is stressed that software ought to bear a free license -- this is to ensure de jure freedom, i.e. legal rights to being able to use, study, modify and share such software. However in these talks the **de facto freedom of software is often forgotten**; the legal (de jure) freedom

is worth nothing if it doesn't imply real and practical (de facto) freedom to exercise the rights given by the license; for example if a piece of "free" (having a free license) software is extremely bloated, our practical ability to study and modify it gets limited because doing so gets considerably expensive and therefore limits the number of people who can truly exercise those rights in practice. This issue of diminishing de facto freedom of free software is addressed e.g. by the suckless movement, and of course our LRS movement.

There is also a similar situation regarding free speech: if speech is free only de jure, i.e. we can "in theory" legally speak relatively freely, BUT if then in reality we also CANNOT speak freely because e.g. of fear of being cancelled, **our speech is de facto not free**.

deferred_shading

Deferred Shading

In computer graphics programming deferred shading is a technique for speeding up the rendering of (mainly) shaded 3D graphics (i.e. graphics with textures, materials, normal maps etc.). It is nowadays used in many advanced 3D engines. In principle of course the idea may also be used in 2D graphics and outside graphics.

The principle is following: in normal forward shading (non-deferred) the shading computation is applied immediately to any rendered pixel (fragment) as they are rendered. However, as objects can overlap, many of these expensively computed pixels may be overwritten by pixels of other objects, so many pixels end up being expensively computed but invisible. This is of course wasted computation. Deferred shading only computes shading of the pixels that will end up actually being visible -- this is achieved by **two rendering passes**:

1. At first geometry is rendered without shading, only with information that is needed for shading (for example normals, material IDs, texture IDs etc.). The rendered image is stored in so called G-buffer which is basically an image in which every pixel stores the above mentioned shading information.
2. The second pass applies the shading effects by applying the pixel/fragment shader on each pixel of the G-buffer.

This is especially effective when we're using very expensive/complex pixel/fragment shaders AND we have many overlapping objects. **Sometimes deferred shading may be replaced by simply ordering the rendered models**, i.e. rendering front-to-back, which may achieve practically the same speed up. In simple cases deferred shading may not even be worth it -- in LRS programs we may use it only rarely.

Deferred shading also comes with complications, for example **rasterization anti aliasing can't be used** because, of course, anti-aliasing in G-buffer doesn't really make sense. This is usually solved by some screen-space antialiasing technique such as FXAA, but of course that may be a bit inferior. **Transparency also poses an issue**.

democracy

Democracy

Democracy (also *democracy*) stands for *rule of the people*, it is a form of government that somehow lets all citizens collectively make political decisions, which is usually implemented by voting but possibly also by other means. The opposite of democracy is autocracy (for example dictatorship), the absolute rule of a single individual; possible yet greater opposite of democracy is final stage capitalism, rule of no people at all, with money enslaving everyone. It can also be contrasted with oligarchy, the rule of a few (e.g. plutocracy, the rule of the rich, which we see under advanced capitalism). Democracy may take different forms, e.g. direct (people directly vote on specific questions) or representative (people vote for officials who then make decisions on their behalf).

Democracy does NOT equal voting, even though this simplification is too often made. Voting doesn't imply democracy and democracy doesn't require voting, an alternative to voting may be for example a scientifically made decision. Democracy in the wide sense doesn't even require a state or legislation -- true democracy simply means that rules and actions of a society are controlled by all the people and in a way

that benefits all the people. Even though we are led to believe we live in democratic society, the truth is that a large scale largely working democracy has never been established and that nowadays most of so called democracy is just an illusion as society clearly works for the benefit of the few richest and most powerful people while greatly abusing everyone else, especially the poorest majority of people. **We do NOT live in true democracy.** A true democracy would be achieved by ideal models of society such as those advocated by (true) anarchism or LRS, however some anarchists may be avoiding the use the term democracy as that in many narrower contexts implies an existence of government.

Nowadays the politics of most first world countries is based on elections and voting by people, but despite this being called democracy by the propaganda the reality is de facto not a democracy but rather an oligarchy, the rule THROUGH the people, creating an illusion of democracy which however lacks a real choice (e.g. the US two party system in which people can either vote for capitalists or capitalists) or pushes the voters towards a certain choice by huge propaganda, misinformation and manipulation.

Also nowadays democracy has mostly degenerated to "let's bully those who disagree with majority", i.e. "rule of the mainstream" (and of course, the mainstream is fully controlled by handful of rich etcetc.).

Small brain simplification of democracy to mere "voting" may be highly ineffective and even dangerous. Democracy was actually considered to be very weak or even downright bad by many Greek philosophers such as Plato and Aristotle. We have to realize that **sometimes voting is awesome, but sometimes it's an extremely awful idea.** Why? Consider the two following scenarios:

- **On simple issues wisdom of the crowd work very well**, as demonstrated by the famous experiment in which averaging guesses of many people on a number of beans in a jar resulted in an extremely precise estimate, a much more precise than any man alone could give. This is an example of when voting is the superior solution to making a decision.
- **Non-experts voting on complex issues and voting on issues requiring large vision is a disaster** (which is why we mostly don't have direct democracy but rather representative one). Many retarded rightists believe direct democracy would somehow be "better" -- no, it would indeed be infinitely worse to let braindead rednecks vote on complex issues. When a chess grandmaster plays against thousands of people who make moves by voting, the master easily wins, as demonstrated e.g. by the Karpov vs the World (or Twitch plays PokÃ©mon lol) experiment (later projects such as Kasparov vs the World had to somehow moderate and filter the move votes to give the world a chance). The reason is that the majority of weak moves voted by non-experts outweigh the few good votes of experts, but also ADDITIONALLY even if only expert votes are takes, the result may be inferior because different long-term plans and visions will collide with the long term plans of others, which is probably the reason why e.g. Romans used to elect a single dictator in times of a crisis rather than relying on a council of experts. In such cases democracy may be similar to wanting to create a nice picture by averaging all pictures ever made by all people, the result will probably be just an ugly gray noisy blob (imagine e.g. creating a picture by having many pictures "vote" on color of every pixel simply by voting for the color they have on the same pixel position { Actually I've tried this now and yes, it looks just like a noisy gray blob. ~drummyfish }). This is why it's a very bad idea to have people vote directly e.g. on complex economic or diplomatic issues. We have to say we do NOT advocate for dictators (we are anarchists) -- we rather believe in implementing a decentralized, self-regulating society in which we avoid the need for any dictators or governments.

The democracy **paradox**: what happens when it is democratically decided that democracy is not a good tool for decision making? I.e. what if democracy denies its own validity? If we believe democracy is valid, then we have to accept its decision and stop believing in democracy, but then if we stop believing in democracy we can just reject the original decision because it was made by something that's not to be trusted, but then...

demo

Demo

Demo may stand for:

- A special kind of computer program that's made as a non-interactive audiovisual art. See demoscene.

- A non-video recording of game play (sometimes also called a *replay*), typically done by recording only user inputs in a deterministic game, or by only recording game state information (such as player positions at each frame etc.). Demos are used e.g. in Doom or Quake. The advantage of a demo is its much smaller size compared to a video, as well as the possibility to replay it with different graphic settings, ability to detect cheating (i.e. verifying the game inputs are correct and humanly possible) etc.
- A significantly limited gratis trial version of an otherwise paid program. This term used to be used mainly in the past, when demo versions of programs (such as games) were distributed e.g. on CDs in magazines for user to try out and potentially buy the full version.
- ...

demoscene

Demoscene

Demoscene is a hacker art subculture revolving around making so called demos, programs that produce rich and highly curious and intriguing audiovisual effects which are sometimes limited by strict size constraints (so called intros). The scene originated in northern Europe sometime in 1980s (although things like screen hacks existed long before) among groups of crackers who were adding small signature effect screens into their cracked software (popularly likened to "digital graffiti"); programming of these cool effects later became an art of its own and started to have their own competitions (sometimes with high financial prizes), so called compos, at dedicated real life events called demoparties (which themselves evolved from copyparties, real life events focused on piracy). The community is still centered mostly in the Europe (primarily Finland, in some countries demoscene was even officially added to the cultural heritage), it is underground, out of the mainstream; Wikipedia says that by 2010 its size was estimated to 10000 people (such people are called *demosceners*).

Demoscene is a bit of a bittersweet topic: on one side it's awesome, full of beautiful hacking, great ideas and minimalism, on the other side there are secretive people who don't share their source code (most demos are proprietary) and ugly unportable programs that exploit quirks of specific platforms. Common platforms are DOS, Commodore 64, Amiga or Windows. These guys simply try to make the coolest visuals and smallest programs, with all good and bad that comes with it. Please strive to take only the good of it.

Besides "digital graffiti" the scene is also perhaps a bit remotely similar to the culture of street rap in its underground and competitive nature, but of course it differs by lack of improvisation and in centering on groups rather than individuals. Nevertheless the focus is on competition, originality, style etc. But demos should show off technological skills as the highest priority -- trying to "win by content" rather than programming skills is sometimes frowned upon. Individuals within a demogroup have roles such as a programmer, visual artist, music artist, director, even PR etc. The whole mindset and relationship to technology within demoscene is much different from the mainstream; for example it's been stated that while mainstream sees computers just as a tool that should just make happen what we imagine, a demoscener puts technology first, he doesn't see computing platforms in terms of better or worse e.g. for its raw computational power, he rather sees a rich world of unique computing platforms, each one with specific personality and feel, kind of like a visual artist sees different painting styles.

A demo isn't a video, it is a non-interactive real time executable that produces the same output on every run (even though categories outside of this may also appear). Viznut has noted that this "static nature" of demos may be due to the established culture in which demos are made for a single show to the audience. Demos themselves aren't really limited by resource constraints (well, sometimes a limit such as 4 MB is imposed), it's where the programmers can show off all they have. However compos are often organized for **intros**, demos whose executable size is limited (i.e. NOT the size of the source code, like in code golfing, but the size of the compiled binary). The main categories are 4Kib intros and 64Kib intros, rarely also 256Kib intros (all sizes are in kibibytes). Apparently even such categories as 256 byte intro appear. Sometimes also platform may be specified (e.g. Commodore 64, PC etc.). The winner of a compo is decided by voting.

Some of the biggest demoparties are or were Assembly (Finland), The Party (Denmark), The Gathering (Norway), Kindergarden (Norway) and Revision (Germany). A guy on <https://mlab.taik.fi/~eye/demos/> says that he has never seen a demo female programmer and that females often have free entry to demoparties while men have to pay because there are almost no women anyway xD Some famous demogroups include

Farbrausch (Germany, also created a tiny 3D shooter game .kkrieger), Future Crew (Finland), Pulse (international), Haujobb (international), Conspiracy (Hungary) and Razor 1911 (Norway). { Personally I liked best the name of a group that called themselves *Byterapers*. ~drummyfish } There is an online community of demosceners at <https://www.pouet.net>.

On technological side of demos: great amount of hacking, exploitation of bugs and errors and usage of techniques going against "good programming practices" are made use of in making of demos. Demosceners make use and invent many kinds of effects, such as the *plasma* (cycling color palette on a 2D noise pattern), *copper bars*, moire patterns, waving, lens distortion etc. Demos are usually written in C, C++ or assembly (though some retards even make demos in Java lmao). In intros it is extremely important to save space wherever possible, so things such as procedural generation and compression are heavily used. Manual assembly optimization for size can take place. Tracker music, chiptune, fractals and ASCII art are very popular. New techniques are still being discovered, e.g. bytebeat. GLSL shader source code that's to be embedded in the executable has to be minified or compressed. Compiler flags are chosen so as to minimize size, e.g. small size optimization (-Os), turning off buffer security checks or turning on fast float operations. The final executable is also additionally compressed with specialized executable compression.

See Also

- hacker culture
 - code golf
 - kkrieger
 - LAN party
 - MUD
-

dependency

Dependency

Dependency of a piece of technology is another piece of technology that's required for the former to work (typically e.g. a software library that's required by given computer program). Dependencies are bad! Among programmers the term **dependency hell** refers to a very common situation of having to deal with the headaches of managing dependencies (and recursively dependencies of those dependencies). Unfortunately dependencies are also unavoidable. However we at least try to minimize dependencies as much as possible while keeping our program functioning as intended, and those we can't avoid we try to abstract (see portability) in order to be able to quickly drop-in replace them with alternatives. It turns out with good approach we can minimize dependencies very close to zero.

Having many dependencies is a sign of **bloat and bad design**. Unfortunately this is the reality of mainstream "modern" programming. For example at the time of writing this Chromium in Debian requires (recursively) 395 packages LMAO xD And these are just runtime dependency packages, we aren't even counting all the hardware features each of this package relies on etc...

Though dependencies are primarily bad because they endanger whole functionality of a program as such, i.e. "it simply won't run without it", they are also bad for another reason: you have no control over how a dependency will behave, if it will be implemented well and if it will behave consistently. OpenGL for example caused a lot of trouble by this because even though the API is the same, different OpenGL implementations performed differently under different situations and made one game run fast with certain combinations of GPUs and drivers and slow with others, which is why Vulkan was created. It is also why some programmers write their own memory allocation functions even though they are available in the standard library etc. -- they know they can write one that's fast and will be fast where they want it to be.

In software development context we usually talk about software dependencies, typically libraries and other software packages such as various frameworks. However, there are many other types of dependencies we need to consider when striving for the best programs. Let us list just some of the possible types:

- software
 - ♦ libraries
 - ♦ compiler supporting specific language standard, extensions etc.

- ♦ build system
- ♦ GUI capabilities
- ♦ operating system and its services such as support of multitasking, presence of a window manager, desktop environment, presence of a file system etc.
- ♦ ...
- hardware
 - ♦ sufficient computing resources (enough RAM, CPU frequency and cores, ...)
 - ♦ graphics card with supported features
 - ♦ floating point unit and other coprocessors
 - ♦ CPU features such as special instructions, modes, ...
 - ♦ network/Internet connection
 - ♦ mouse, speakers and other I/O devices
 - ♦ ...
- other:
 - ♦ know-how/education: Your program may require specific knowledge, e.g. knowledge of advanced math to be able to meaningfully modify the program, or nonnegligible amount of time spent studying your codebase.
 - ♦ running cost: e.g. electricity, Internet connection cost
 - ♦ culture: Your program may require the culture to allow what it is presenting or dealing with.
 - ♦ ...

Good program will take into account ALL kinds of these dependencies and try to minimize them to offer freedom, stability and safety while keeping its functionality or reducing it only very little.

Why are dependencies so bad? Because your program is for example:

- **more buggy** (more fuck up surface)
- **less portable** (to port the program you also need to port or replace all the dependencies)
- **more expensive to maintain (and create)** (requires someone's constant attention to just keep the dependencies up to date and keeping up with their changing API)
- **less future proof and more fragile** (your program dies as soon as one of its dependencies, or any dependency of these dependencies...)
- **more bloated and so probably less efficient**, i.e. slower, eating up more RAM than necessary etc.
- **less under your control** (in practice it's extremely difficult to fork, modify and maintain a library you depend on even if it's free as you just take up another project to learn and maintain, so you're typically doomed to just accept whatever is offered to you)
- **less "secure"** (more attack surface, i.e. potential for vulnerabilities which may arise in the dependencies) -- though we don't fancy the privacy/security hysteria, it is something that matters to many
- **more dangerous legally** (reusing work of other people requires dealing with several to many different licenses with possibly wild conditions and there's always a chance of someone starting to make trouble such as threatening to withdraw a license)
- **more complicated to develop and work with**, customize, modify etc. -- a nice program without dependencies is built very simply: you just download it and compile it. A program with tons of dependencies will require a complex set up of all the dependencies first, making sure they're of required versions, and then you have to build EVERYTHING of course, usually adding the need for some complex build system to even make recompiling bearable.
- ...

Really it can't be stressed enough that **ALL dependencies have to be considered**, even things such as the standard library of a programming language or built-in features of a language that "should always" come with the language. It is e.g. common to hear C programmers say "I can just use float because it's part of C specification and so it has to be there" -- well technically yes, but in practice many C implementations for some obscure platforms will end up being unfinished, incomplete or even intentionally non-compliant with the standard, no standard can really physically force people to follow it, OR the compiler's floating point implementation may simply suck (or it HAS TO suck because there's no floating point hardware on the platform) so much that it will technically be present but practically unusable. This will mean that your program COULD work on the platform but DOESN'T, even if some piece of paper somewhere says it SHOULD. So REALLY REALLY do not use non-trivial features that you don't really need, it really does help. If you really want to make your program truly dependency light, always ask something like this: "If our civilization and all

its computers disappear and only the literal text of my program survives, how hard will it be for future civilizations to make it actually run?".

How to Avoid Them

TODO

determinism

Determinism

"God doesn't play dice." --some German dude

Deterministic system (such as a computer program or an equation) is one which over time evolves without any involvement of randomness; i.e. its current state along with the rules according to which it behaves unambiguously and precisely determine its following states. This means that a deterministic algorithm will always give the same result if run multiple times with the same input values. Determinism is an extremely important concept in computer science and programming (and in many other fields of science and philosophy). For example game of life is a deterministic system while Markov chain is not.

Determinism is also a **philosophical theory** and aspect of physics theories -- here it signifies that our Universe is deterministic, i.e. that everything is already predetermined by the state of the universe and the laws of physics, i.e. that we don't have "free will" (whatever it means) because our brains are just machines following laws of physics like any other matter etc. Many normies believe quantum physics disproves determinism which is however not the case, there may e.g. exist hidden variables that still make quantum physics deterministic -- some believe the Bell test disproved hidden variables but again this is NOT the case as it relies on statistical independence of the experimenters, determinism is already possible if we consider the choices of experimenters are also predetermined (this is called superdeterminism). Einstein and many others still believed determinism was the way the Universe works even after quantum physics emerged. { This also seems correct to me. Sabine Hossenfelder is another popular physicist promoting determinism. ~drummyfish } Anyway, this is already beyond the scope of technological determinism.

Computers are mostly deterministic by nature and design, they operate by strict rules and engineers normally try to eliminate any random behavior as that is mostly undesirable (with certain exceptions mentioned below) -- randomness leads to hard to detect and hard to fix bugs, unpredictability etc. Determinism has furthermore many advantages, for example if we want to record a behavior of a deterministic system, it is enough if we record only the inputs to the system without the need to record its state which saves a great amount of space -- if we later want to replay the system's behavior we simply rerun the system with the recorded inputs and its behavior will be the same as before (this is exploited e.g. in recording gameplay demos in video games such as Doom).

Determinism can however also pose a problem, notable e.g. in cryptography where we DO want true randomness e.g. when generating seeds. Determinism in this case implies an attacker knowing the conditions under which we generated the seed can exactly replicate the process and arrive at the seed value that's supposed to be random and secret. For this reason some CPUs come with special hardware for generating truly random numbers.

Despite the natural determinism of computers as such, **computer programs nowadays aren't always automatically deterministic** -- if you're writing a typical interactive computer program under some operating system, you have to make some extra bit of effort to make it deterministic. This is because there are things such as possible difference in timings or not perfectly specified behavior of floating point types in your language; for example a game running on slower computer will render fewer frames per second and if it has FPS-dependent physics, the time step of the physics engine will be longer on this computer, possibly resulting in slightly different physics behavior due to rounding errors. This means that such program run with the same input data will produce different results on different computers or under slightly different circumstances, i.e. it would be non-deterministic.

Nevertheless **we almost always want our programs to be deterministic** (or at least deterministic under some conditions, e.g. on the specific hardware platform we are using), always try to make your programs

deterministic unless you have a VERY good reason not to! **It doesn't take a huge effort to achieve determinism**, it's more of just taking the right design decisions (e.g. separating rendering and physics simulation), i.e. good programming leads to determinism and vice versa, determinism in your program indicates good programming. The reason why we want determinism is that such programs have great properties, e.g. that of easier debugging (bugs are reproducible just by knowing the exact inputs), easy and efficient recording of activity (e.g. demos in games), sometimes even time reversibility (like undos, but watch out -- this doesn't hold in general!). Determinism also itself serves as a kind of a test if the program is working right -- if your program can take recorded inputs and reproduce same behavior at every run, it shows that it's written well, without things like undefined behavior affecting its behavior.

{ The previous paragraph is here because I've talked to people who thought that determinism was some UBER feature that requires a lot of work and so on ("OMG Trackmania is deterministic, what a feat!") -- this is NOT the case. It may intuitively seem so to non-programmers or beginners, but really this is not the case. Non-determinism in software appears usually due to a fuck up, ignorance or bad design choice made by someone with a low competence. Trust me, determinism is practically always the correct way of making programs and it is NOT hard to do. ~drummyfish }

Even if we're creating a program that somehow works with probability, we usually want to make it deterministic! This means we don't use actual random numbers but rather pseudorandom number generators that output chaotic values which simulate randomness, but which will nevertheless be exactly the same when ran multiple times with the same initial seed. This is again important e.g. for debugging the system in which replicating the bug is key to fixing it. If under normal circumstances you want the program to really behave differently in each run, you make it so only by altering its initial random seed.

In theoretical computer science non-determinism means that a model of computation, such as a Turing machine, may at certain points decide to make one of several possible actions which is somehow most convenient, e.g. which will lead to finding a solution in shortest time. Or in other words it means that the model makes many computations, each in different path, and at the end we conveniently pick the "best" one, e.g. the fastest one. Then we may talk e.g. about how the computational strength or speed of computation differ between a deterministic and non-deterministic Turing machine etc.

Determinism does NOT guarantee reversibility, i.e. if we know a state of a deterministic system, it may not always be possible to say from which state it evolved, or in other words: a system that's deterministic may or may not be deterministic in reverse time direction. This reversibility is only possible if the rules of the system are such that no state can evolve from two or more different states. If this holds then it is always possible to time-reverse the system and step it backwards to its initial state. This may be useful for things such as undos in programs. Also note that even if a system is reversible, it may be computationally very time consuming and sometimes practically impossible to reverse the system (imagine e.g. reversing a cryptographic hash -- mathematical reversibility of such hash may be arbitrarily ensured by e.g. pairing each hash with the lowest value that produces it).

Is floating point deterministic? In theory even floating point arithmetic can of course be completely deterministic but there is the question of whether this holds about concrete specifications and implementations of floating point (e.g. in different programming languages) -- here in theory non-determinism may arise e.g. by some unspecified behavior such as rounding rules. In practice you can't rely on float being deterministic. The common float standard, IEEE 754, is basically deterministic, including rounding etc. (except for possible payload of NaNs, which shouldn't matter in most cases), but this e.g. doesn't hold for floating point types in C!

devuan

Devuan

Devuan is a GNU/Linux distribution that's practically ideantical to Debian (it is its fork) but without systemd as well as without packages that depend on the systemd malware. Devuan offers a choice of several init systems, e.g. openrc, sysvinit and runit. It was first released in 2017.

Notice how *Devuan* rhymes less with *lesbian* than *Debian*.

Despite some flaws (such as being Linux with all the bloat), Devuan is still one of the best operating systems for most people and it is at this time recommended by us over most other distros not just for avoiding systemd, but mainly for its adoption of Debian free software definition that requires software to be free as a whole, including its data (i.e. respecting also free culture). It is also a nicely working unix system that's easy to install and which is still relatively unbloated.

{ I can recommend Devuan, I've been using it as my main OS for several years. NOTE: some people told me Devuan is impure because it's still kinda bloated and recommends bloated stuff like Firefox etc., they recommended e.g. Dragora and Hyperbola -- this is absolutely true, **basically nothing ideal exists at the moment**, and any software recommendation always comes with the danger of it becoming shit over time, always keep that in mind; really recommending any software at this point comes down to discussing what's least shit, what suckles the least and also putting a lot of weight on subjective factors. Also distros don't matter basically, they are all shit, just choose something that doesn't stand too much in the way of your creation and accept that it sucks. If you are skilled you can set up a much better Unix than Debian, Debian is a recommendation for a user who wants a good balance between "close to LRS" and "just works"; if you are skilled enough to set up a close to ideal system you probably don't need my recommendation on a distro, you can find it yourself. ~drummyfish }

dick_reveal

Dick Reveal

Dick/pussy reveal is the act of publicly revealing one's nudity on the Internet, in normal situations done voluntarily.

See Also

- face reveal
 - deepfake
 - the fappinging
-

digital

Digital

Digital computer technology is that which works with whole numbers, i.e. discrete values, as opposed to analog technology which works with real numbers, i.e. continuous values (note: do not confuse things such as floating point with truly continuous values!). The name *digital* is related to the word *digit* as digital computers store data by digits, e.g. in 1s and 0s if they work in binary. By extension the word *digital* is also used to indicate something works based on digital technology, for example "digital currency", "digital music" etc.

Normies confuse digital with electronic or think that digital computers can only be electronic, that digital computers can only work in binary or have other weird assumptions whatsoever. **This is indeed false!** An abacus is a digital device, a book with text is a digital data storage. Fucking normies RIP.

{ Apparently it is "digitisation", not "digitalization". ~drummyfish }

The advantage of digital technology is its resilience to noise which prevents degradation of data and accumulation of error -- if a digital picture is copied a billion times, it will very likely remain unchanged, whereas performing the same operation with analog picture would probably erase most of the information it bears due to loss of quality in each copy. Digital technology also makes it easy and practically possible to create fully programmable general purpose computers of great complexity.

Digital vs analog, simple example: imagine you draw two pictures with a pencil: one in a normal fashion on a normal paper, the other one on a grid paper, by filling specific squares black. The first picture is analog, i.e. it records continuous curves and position of each point of these curves can be measured down to extremely small fractions of millimeters -- the advantage is that you are not limited by any grid and can draw

any shape at any position on the paper, make any wild curves with very fine details, theoretically even microscopic ones. The other picture (on a square grid) is digital, it is composed of separate points whose position is described only by whole numbers (x and y coordinates of the filled grid squares), the disadvantage is that you are limited by only being able to fill squares on predefined positions so your picture will look blocky and limited in amount of detail it can capture (anything smaller than a single grid square can't be captured properly), the resolution of the grid is limited, but as we'll see, imposing this limitations has advantages. Consider e.g. the advantage of the grid paper image with regards to copying: if someone wants to copy your grid paper image, it will be relatively easy and he can copy it exactly, simply by filling the exact same squares you have filled -- small errors and noise such as imperfectly filled squares can be detected and corrected thanks to the fact that we have limited ourselves with the grid, we know that even if some square is not filled perfectly, it was probably meant to be filled and we can eliminate this kind of noise in the copy. This way we can copy the grid paper image a million times and it won't change. On the other hand the normal, non-grid image will become distorted with every copy and in fact even the original image will become distorted by aging; even if that who is copying the image tries to trace it extremely precisely, small errors will appear and these errors will accumulate in further copies, and any noise that appears in the image or in the copies is a problem because we don't know if it really is a noise or something that was meant to be in the image.

Of course, digital data may become distorted too, it is just less likely and it's easier to deal with this. It for example happens that space particles (and similar physics phenomena, e.g. electronic interference) flip bits in computer memory, i.e. there is always a probability of some bit flipping from 0 to 1 or vice versa. We call this **data corruption**. This may also happen due to physical damage to digital media (e.g. scratches on the surface of CDs), imperfections in computer network transmissions (e.g. packet loss over wifi) etc. However we can introduce further measures to prevent, detect and correct data corruption, e.g. by keeping redundant copies (2 copies of data allow detecting corruption, 3 copies allow even its correction), keeping checksums or hashes (which allow only detection of corruption but don't take much extra space), employing error correcting codes etc.

Another way in which digital data can degrade similarly to analog data is **reencoding between lossy-compressed formats** (in the spirit of the famous "needs more jpeg" meme). A typical example is digital movies: as new standard for video encoding are emerging, old movies are being reconverted from old formats to the new ones, however as video is quite heavily lossy-compressed, losses and distortion of information happens between the reencodings. This is best seen in videos and images circulating on the internet that are constantly being ripped and converted between different formats. This way it may happen that digital movies recorded nowadays may only survive into the future in very low quality, just like old analog movies survived until today in degraded quality. This can be prevented by storing the original data only with lossless compression and with each new emerging format create the release of the data from the original.

digital_signature

Digital Signature

Digital signature is a method of mathematically (with cryptographical algorithms) proving that, with a very high probability, a digital message or document has been produced by a specific sender, i.e. it is something aka traditional signature which gives a "proof" that something has been written by a specific individual.

It works on the basis of asymmetric cryptography: the signature of a message is a pair of a public key and a number (the signature) which can only have been produced by the owner of the private key associated with the public key. This signature is dependent on the message data itself, i.e. if the message is modified, the signature will no longer be valid, preventing anyone who doesn't possess the private key from modifying the message. The signature number can for example be a hash of the message decoded with the private key -- anyone can check that the signature encoded with the public key gives the document hash, proving that whoever computed the signature number must have possessed the private key.

Signatures can be computed e.g. with the RSA algorithm.

The advantage here is that **anonymity can be kept with digital signatures**; no private information such as the signer's real name is required to be revealed, only his public key. Someone may ask why we then

even sign documents if we don't know by whom it is signed lol? But of course the answer is obvious: many times we don't need to know the identity of the signer, we just need to know that different messages have all been written by the same man, and this is what a digital signature can ensure. And of course, if we want, a public key can have a real identity assigned if desirable, it's just that it's not required.

dinosaur

Dinosaur

In the hacker jargon *dinosaur* is a type of a big, very old, mostly non-interactive (batch), possibly partly mechanical computer, usually an IBM mainframe from 1940s and 1950s (so called Stone Age). They resided in *dinosaur pens* (mainframe rooms).

{ This is how I understood it from the Jargon File. ~drummyfish }

diogenes

Diogenes

"The most beautiful thing in the world is freedom of speech." --Diogenes

Diogenes (412 BC - 323 BC) was one of the biggest ancient Greek philosophers, the best known proponent of Cynicism and one of the absolutely most based men in history as by his philosophy he practiced extreme life minimalism (he lived in a barrel), asceticism, self-sufficiency, nonconformism, he refused to work, refused all authority, criticised absolutely everything and was always extremely logically consistent and behaved in accordance to what he taught, which is really what makes all his critics -- mostly just big hypocrite pussies -- so greatly pissed; the philosophy of Diogenes is quite close to our own ideals. The word "cynic" itself comes from a word for "dog" and indeed, Diogenes lived as one, he just roamed the streets barefoot with a stick, he wore a robe that at night he used to cover himself (two in one), he didn't give a shit about anything, preached his wisdom, he basically didn't own anything as he believed possession only enslaves us and that everything we need is already there in the nature. He didn't seek popularity, approval, wealth or power, he wanted freedom, spiritual and moral purity, he wanted to let go of absolutely all bullshit. He was also pretty funny, reading about him is really on the level of 4chan humor, more than 2000 years ahead of his time. Diogenes wrote some stuff, most famously his *Republic* describing an ideal society, however none of his writings sadly survived, we now only know what others have written about him (there are possibly some recounts of the works who have read them). Let's remember we shouldn't call him a hero, that would itself contradict both his and our philosophy, but if we are to see anyone as a good inspiration and moral example, Diogenes is among the best (well, at least in most things, it goes without saying we can't absolutely embrace everything someone ever did).

Some famous and/or interesting things he did (or at least is said to have done):

- He lived in a barrel with his only possession being a bowl which he later threw away because a boy showed him he could actually drink using just bare hands ("A child has beaten me at simple living!").
- He didn't work, just begged.
- He walked with a lantern in crowds of people, saying he was "looking for a man". By this he implied all those around weren't even worth being called *people*. { There seems to be a widespread inaccuracy in translations that say he was looking for "honest man", but apparently he just said "man", i.e. literally just human. ~drummyfish }
- He masturbated in public. When asked why, he replied: "If only I could satisfy hunger by rubbing my belly!" { Based. ~drummyfish } He seems to have done it just to get rid of the physiological need, for example when a prostitute promised to visit him but came late, he already did the job himself.
- When asked where he was from, he always answered "I am citizen of the world", using the word cosmopolitan (he's credited for coining it). By this he gave a huge fuck you to nationalism.
- He just shat, pissed and farted wherever he wanted, he was stealing without shame, he would eat where it was forbidden to eat, e.g. in Plato's lectures. He also didn't restrain from always saying what he was thinking, he did zero self ensorship.

- As he became quite famous, Alexander the Great himself (the biggest boss in the world at the time) came to meet him and asked if he could do anything for him, Diogenes just replied "yeah, get out of my sunlight". It impressed even Alexander who allegedly said "If I weren't Alexander, I would be Diogenes". Another of their dialogue went like this: Alexander: "Are you not afraid of me?", Diogenes: "Why? Are you good or bad?", Alexander: "I am good!", Diogenes: "Well then, who is afraid of the good?".
- Being asked what the most beautiful thing in the world was, he replied "freedom of speech". { Sauce: the book *Lives of the Eminent Philosophers*. ~drummyfish }
- When he was dying he said after death he just wanted to be thrown to animals to be eaten. { I'm thinking of actually writing this in my will. ~drummyfish }
- When asked why he was begging when Plato wasn't, he said "He is too, he just holds his head down so it can't be heard."
- Once he was invited to a dinner, he said he wouldn't go because "last time he went, the host didn't show proper gratitude."
- At seeing temple police arresting a thief he exclaimed "look, big thieves are arresting a small thief".
- LULZ (not embraced but funny): Once he was asking some man for fruit, the man said "I'll give it to you if you can persuade me". Diogenes said: "If I could persuade you, I would persuade you to hang yourself".
- ...

His life in short summary: he was born probably 412 (or 419?) BC in Sinope (Turkey) to Hicesius, a money changer, then he got into trouble for devaluing the currency with his father so he left the city and went to Athens where he became the pupil of Antisthenes. He started to live in a tub belonging to the temple of Cybele. He hated Plato's philosophy, called it shite and disrupted his lectures on purpose e.g. by eating there or asking weird questions -- he also had some funny conversations and troll moments with him, like when he offered him some figs and when Plato ate them, Diogenes was like "Bruh, I thought you would share them, not eat them all". Plato called him "Socrates gone mad". Diogenes advised both mental and physical practice -- training the mind is obviously important for achieving inner freedom and wisdom, but body and mind are interconnected, asceticism leads to learning to live with little, not be dependent on much and frees us from slavery and unnecessary desire. When traveling to Aegina he was captured by pirates and sold as a slave to Xenias, a man living in Corinth. Being a greatly educated slave, Xenias actually made Diogenes a teacher of his two sons, and Diogenes didn't live such a bad life, he could still preach his philosophy and indeed he did, until he died in 323 BC -- it is said he died on the same day as Alexander the Great. It's been written at the day of his death he just seated himself on the road to Olympia and calmly passed away. A pillar with marble dog was erected in his honor.

Why was he so based you ask? Most normies don't get this, they are like "shit in public = bad" or "no werk = steal" etc., it just shows how immensely retarded everyone is (and why Diogenes really couldn't find a man anywhere, he really just saw monkeys). Diogenes was the only one around who was ACTUALLY THINKING, he wouldn't accept any word of a propaganda without first thinking about if those words were actually true. He saw something and asked "is it good?". And he came to conclusion that most things are just bullshit. But that's not all: not only did he say something's bullshit -- something most people would just go on doing without end -- he actually just stopped doing the bullshit and by that proved his point. Most people hated (and still hate) him because he just clearly proves them wrong, without any shadow of a doubt, simply by showing them something is unnecessary by living without it. Politicians in suits are just idiotic talking retardheads in expensive suits who just talk talk talk and lie and preach huge bullshit without actually doing anything and if someone just clearly shows they are idiots, their only possible "defense" is to discredit the opposition, which is why all the idiots just try to spread hate of Diogenes (and ad hominem of his teaching) based on shallow things like "the guy shit in public + his cloth smells = bad = words false" (but ofc it's actually effective on the population made basically of zombies). Diogenes didn't even have to talk much, he just said "this is bullcrap, look, you can live without it", and then he JUST DID IT like the biggest chad.

See Also

- cynicism
- Buddha
- Jesus

disease

Disease

Disease is a bad state of living organism's health caused by failure of its inner mechanisms rather than being directly caused by a physical injury. Technological and consumerist diseases are mental diseases almost exclusively present in humans (but also possibly in some animals forced to live like humans) related to shit technology. Some of the most common diseases, mostly of the technological kind but also of others, include:

- assholism
- audiophilia
- autism
- cancer
- capitalism
- data hoarding
- depression
- egoism
- Emacs
- furry disorder
- homosexuality
- hopping:
 - ♦ browser hopping
 - ♦ distro hopping
 - ♦ text editor hopping
 - ♦ git hopping
 - ♦ language hopping and paradigm hopping
 - ♦ license hopping
 - ♦ ...
- lack of IQ
- maximalism
- narcissism
- nationalism
- NPC disease
- object obsession
- obsessive privacy obsession
- pride
- retardation
- schizophrenia
- troll personality disorder
- Unix ricing
- ...

distance

Distance

Distance is a measure of how far away from each other two points are. Most commonly distance refers to physical separation in space, e.g. as in distance of planets from the Sun, but more generally distance can refer to any kind of parameter space and in any number of dimensions, e.g. the distance of events in time measured in seconds (1D distance) or distance of two text strings as the amount of their dissimilarity (Levenshtein distance). Distances are extremely important in computer science and math as they allow us to do such things as clustering, path searching, physics simulations, various comparisons, sorting etc.

Distance is similar/related to length, the difference is that distance is computed between two points while length is the distance of one point from some implicit origin.

There are many ways to define distance within given space. Most common and implicitly assumed distance is the **Euclidean distance** (basically the "straight line from point A to point B" whose length is computed with Euclidean Theorem), but other distances are possible, e.g. the taxicab distance (length of the kind of perpendicular path taxis take between points A and B in Manhattan, usually longer than straight line).

Mathematically a space in which distances can be measured are called metric spaces, and a distance within such space can be any function *dist* (called a *distance* or *metric* function) that satisfies these axioms:

1. $dist(p,p) = 0$ (distance from identical point is zero)
2. Values given by *dist* are never negative.
3. $dist(p,q) = dist(q,p)$ (symmetry, distance between two points is the same in both directions).
4. $dist(a,c) \leq dist(a,b) + dist(b,c)$ (triangle inequality)

Approximations

Computing Euclidean distance requires multiplication and most importantly square root which is usually a pretty slow operation, therefore many times we look for simpler approximations. Note that a possible approach here may also lead through computing the distance normally but using a fast approximation of the square root.

Two very basic and rough approximations of Euclidean distance, both in 2D and 3D, are taxicab (also Manhattan) and Chebyshev distances. Taxicab distance simply adds the absolute coordinate differences along each principal axis (*dx*, *dy* and *dz*) while Chebyshev takes the maximum of them. In C (for generalization to 3D just add one coordinate of course):

```
int distTaxi(int x0, int y0, int x1, int y1)
{
    x0 = x1 > x0 ? x1 - x0 : x0 - x1; // dx
    y0 = y1 > y0 ? y1 - y0 : y0 - y1; // dy

    return x0 + y0;
}

int distCheb(int x0, int y0, int x1, int y1)
{
    x0 = x1 > x0 ? x1 - x0 : x0 - x1; // dx
    y0 = y1 > y0 ? y1 - y0 : y0 - y1; // dy

    return x0 > y0 ? x0 : y0;
}
```

Both of these distances approximate a circle in 2D with a square or a sphere in 3D with a cube, the difference is that taxicab is an upper estimate of the distance while Chebyshev is the lower estimate. For speed of execution (optimization) it may also be important that taxicab distance only uses the operation of addition while Chebyshev may result in branching (*if*) in the max function which is usually not good for performance.

A bit more accuracy can be achieved by averaging the taxicab and Chebyshev distances which in 2D approximates a circle with an 8 segment polygon and in 3D approximates a sphere with 24 sided polyhedron. The integer-only C code is following:

```
int dist8(int x0, int y0, int x1, int y1)
{
    x0 = x1 > x0 ? x1 - x0 : x0 - x1; // dx
    y0 = y1 > y0 ? y1 - y0 : y0 - y1; // dy

    return (x0 + y0 + (x0 > y0 ? x0 : y0)) / 2;
}
```

{ The following is an approximation I came up with when working on tinypysicsengine. While I measured the average and maximum error of the taxi/Chebyshev average in 3D at about 16% and 22% respectively, the following gave me 3% and 12% values. ~drummyfish }

Yet more accurate approximation of 3D Euclidean distance can be made with a 48 sided polyhedron. The principle is following: take absolute values of all three coordinate differences and order them by magnitude so that $dx \geq dy \geq dz \geq 0$. This gets us into one of 48 possible slices of space (the other slices have the same shape, they just differ by ordering or signs of the coordinates but the distance in them is of course equal). In this slice we'll approximate the distance linearly, i.e. with a plane. We do this by simply computing

the distance of our point from a plane that goes through origin and whose normal is approximately {0.8728,0.4364,0.2182} (it points in the direction that goes through the middle of space slice). The expression for the distance from this plane simplifies to simply $0.8728 * dx + 0.4364 * dy + 0.2182 * dz$. The following is an integer-only implementation in C (note that the constants above have been converted to allow division by 1024 for possible optimization of division to a bit shift):

```
int32_t dist48(
    int32_t x0, int32_t y0, int32_t z0,
    int32_t x1, int32_t y1, int32_t z1)
{
    x0 = x1 > x0 ? x1 - x0 : x0 - x1; // dx
    y0 = y1 > y0 ? y1 - y0 : y0 - y1; // dy
    z0 = z1 > z0 ? z1 - z0 : z0 - z1; // dz

    if (x0 < y0) // order the coordinates
    {
        if (x0 < z0)
        {
            if (y0 < z0)
            { // x0 < y0 < z0
                int32_t t = x0; x0 = z0; z0 = t;
            }
            else
            { // x0 < z0 < y0
                int32_t t = x0; x0 = y0; y0 = t;
                t = z0; z0 = y0; y0 = t;
            }
        }
        else
        { // z0 < x0 < y0
            int32_t t = x0; x0 = y0; y0 = t;
        }
    }
    else
    {
        if (y0 < z0)
        {
            if (x0 < z0)
            { // y0 < x0 < z0
                int32_t t = y0; y0 = z0; z0 = t;
                t = x0; x0 = y0; y0 = t;
            }
            else
            { // y0 < z0 < x0
                int32_t t = y0; y0 = z0; z0 = t;
            }
        }
    }

    return (893 * x0 + 446 * y0 + 223 * z0) / 1024;
}
```

A similar approximation for 2D distance is (from a 1984 book *Problem corner*) this: $\sqrt{dx^2 + dy^2} \approx 0.96 * dx + 0.4 * dy$ for $dx \geq dy \geq 0$. The error is $\leq 4\%$. This can be optionally modified to use the closest power of 2 constants so that the function becomes much faster to compute, but the maximum error increases (seems to be about 11%). C code with fixed point follows (commented out line is the faster, less accurate version):

```
int dist2DAprox(int x0, int y0, int x1, int y1)
{
    x0 = x0 > x1 ? (x0 - x1) : (x1 - x0);
    y0 = y0 > y1 ? (y0 - y1) : (y1 - y0);

    if (x0 < y0)
    {
        x1 = x0; // swap
        x0 = y0;
        y0 = x1;
    }
}
```

```
    return (123 * x0 + 51 * y0) / 128; // max error = ~4%
    //return x0 + y0 / 2;           // faster, less accurate
}
```

TODO: this https://www.flipcode.com/archives/Fast_Approximate_Distance_Functions.shtml

distrohopping

Distrohopping

Distrohopping is a serious mental illness that makes people waste lives on constantly switching GNU/Linux distributions ("distros"). This affects mostly those of the lowest skill in tech who feel the need to LARP as wannabe tech nerds; as it's been said, an amateur is obsessed with tools, an expert is obsessed with mastery (Richard Stallman has for example famously never installed GNU/Linux himself as he has better things to do) -- a true programmer will just settle with a comfy Unix environment that can run vim and dedicate his time to creating a timeless source code while the hopper, like a mere animal, is just busy masturbating to a new bastard child of Ubuntu and Arch Linux that adds a new wallpaper and support for vertical mice -- **such an activity is basically as retarded as mainstream tech consumerism** with the only difference being a hopper isn't limited by finance so he can just distrohop 24/7 and hop himself to death.

TODO: cure? take the bsd pill? :-)

See Also

- [editorhopping](#)
 - [genderhopping](#)
 - [hopping](#) as a disease in general
-

docker

Docker

I don't fucking know what this is and I don't wanna know that.

dodleston

Dodleston Mystery

The Dodleston mystery regards a teacher Ken Webster who in 1984 supposedly started exchanging messages with people from the past and future, most notably people from the 16th and 22nd century, via files on a BBC micro computer. While probably a hoax and creepypasta, there are some interesting unexplained details... and it's a fun story.

The guy has written a proprietary book about it, called *The Vertical Plane*.

{ If the story is made up and maybe even if it isn't it may be a copyright violation to reproduce the story with all the details here so I don't know if I should, but reporting on a few facts probably can't hurt. Yes, this is how bad the copyrestriction laws have gotten. ~drummyfish }

dog

Dog

Here is the dog! He doesn't judge you; dog love is unconditional. No matter who you are or what you ever did, this buddy will always love you and be your best friend <3 By this he is giving us a great lesson.

revolutionary deathmatch multiplayer (the name *deathmatch* itself was coined by Romero during Doom multiplayer sessions), as well as a HUGE modding and mapping community. It was a success in every way -- arguably no other game has since achieved a greater revolution than Doom (no, not even Minecraft, World of Warcraft etc.). Many reviews of it just went along the lines: "OK, Doom is the best game ever made. now let's just take a look at the details...".

The game's backstory was simple and didn't stand in the way of gameplay, it's basically about a tough marine (so called *Doomguy*) on a Mars military base slaying hordes of demons from hell, all in a rock/metal style with a lot of gore and over-the-top violence (chain saws n stuff).

Doom was followed by Doom II in 1995, which "content-wise" was basically just a data disc, the same game with new levels and some minor additions. Later there were some other releases and rereleases, notable is Doom III from 2004, Doom 2016 ("reboot") and Doom: Eternal (2020).

Some interesting things about Doom:

- Someone created a Doom system monitor for Unix systems called psDooM where the monsters in game are the operating system processes and killing the monsters kills the processes.
- Someone (kgsws) has been hacking the ORIGINAL Doom engine in an impressive way WITHOUT modifying the source code or the binary, rather using arbitrary code execution bug; he added very advanced features known from newer source ports, for example an improved 3D rendering algorithms allowing geometry above geometry etc. (see e.g. <https://yt.artemislena.eu/watch?v=RdbRPNPUIU>). It's called the Ace engine.
- Doom sprites were made from photos of physical things: weapons are modified photos of toys, enemies were made from clay and then photographed from multiple angles (actually a great alternative to 3D modeling that's less dependent on computers and produces more realistic results).
- The strongest weapon in the game is name BFG9000, which stands for "big fucking gun".
- TODO

Doom Engine/Code

See also game engine for the list of different Doom engines.

Doom source code is written in C89 and is about 36000 lines of code long. The original system requirements stated roughly a 30 MHz CPU and 4 MB RAM as a minimum. It had 27 levels (9 of which were shareware), 8 weapons and 10 enemy types. The engine wasn't really as flexible in a way "modern" programmers expect, many things were hard coded, there was no scripting or whatever (see? you don't fucking need it), new games using the engine had to usually modify the engine internals.

The game only used fixed point, no float!

The **Doom engine** (also called *id Tech 1*) was revolutionary and advanced (not only but especially) video game graphics by a great leap, considering its predecessor Wolf3D was really primitive in comparison (Doom basically set the direction for future trends in games such as driving the development of more and more powerful GPUs in a race for more and more impressive visuals). Doom used a technique called **BSP rendering** (levels were made of convex 2D sectors that were then placed in a BSP tree which helped quickly sort the walls for rendering front-to-back) that was able to render realtime 3D views of textured (all walls, floors and ceilings) environments with primitive lighting (per-sector plus diminishing lighting), enemies and items represented by 2D billboards ("sprites"). No GPU acceleration was used, graphics was rendered purely with CPU (so called software rendering, GPU rendering would come with Doom's successor Quake, and would also later be brought to Doom by newer community made engines, though the original always looks the best). This had its limitations, for example the camera could not look up and down, there could be no tilted walls and the levels could not have rooms above other rooms. The geometry of levels was only static, i.e. it could not change during play (only height of walls could), because rendering was dependent on precomputed BSP trees (which is what made it so fast). For these reasons some call Doom "pseudo 3D" or 2.5D rather than "true 3D", some retards took this even as far as calling Doom 2D with its graphics being just an "illusion", as if literally every 3D graphics ever wasn't a mere illusion. Nevertheless, though with limitations, Doom did present 3D views and internally it did work with 3D coordinates (for example the player or projectiles have 2D position plus height coordinate), despite some dumb YouTube videos saying otherwise. For this reason we prefer to call Doom a **primitive 3D** engine, but 3D nonetheless. Other games later used the Doom engine,

such as Heretic, Hexen and Strife. The Doom engine was similar to and competing with Build engine that ran games like Duke Nukem 3D, Blood and Shadow Warrior. All of these 90s shooters were amazing in their visuals and looked far better than any modern shit. Build engine games had similar limitations to those of the Doom engine but would improve on them (e.g. faking looking up and down by camera tilting, which could in theory be done in Doom too, or allowing sloped floor and dynamic level geometry).

Indexed (palette) mode with "only" 256 colors was used for rendering. Precomputed color tables were used to make dimming of colors faster.

Doom also has a deterministic FPS-independent physics which allows for efficient recording of demos of its gameplay and creating tool assisted speedruns, i.e. the time step of game simulation is fixed (35 tics per second). Such demos can be played back in high quality while being minuscule in size and help us in many other ways, for example for verifying validity of speedruns. This is very nice and serves as an example of a well written engine (unlike later engines from the same creators, e.g. those of Quake games which lacked this feature -- here we can see how things get progressively shittier in computer technology as we go forward in time).

There is no antialiasing in the engine, i.e. aliasing can be noticed on far-away textures, but it is suppressed by the use of low-res textures and dimming far-away areas. There is also no edge smoothing (kind of misleadingly known as "antialiasing") in the geometry rendering, the engine is subpixel accurate in rendering of the top and bottoms of the walls, i.e. the line these boundaries form may result in rasterizing slightly different pixels even if the start and end pixel is the same, depending on the subpixel position of the start and endpoint -- this feature doesn't much help in static screenshots but makes animation nicer.

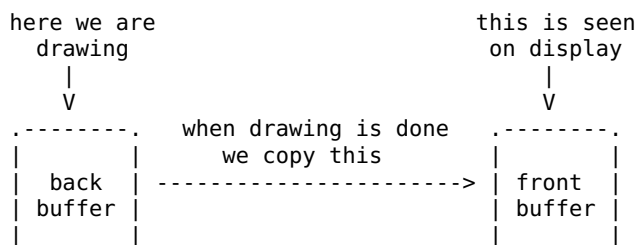
See Also

- [Anarch](#)
- [Duke 3D](#)
- [Gloom](#) (fun [Amiga](#) Doom clone)
- [Quake](#)
- [Jedi engine](#)
- [Build engine](#)
- [Chasm: The Rift](#)
- [raycasting](#)

double_buffering

Double Buffering

In computer graphics double buffering is a technique of rendering in which we do not draw directly to video RAM, but instead to a second "back buffer", and only copy the rendered frame from back buffer to the video RAM ("front buffer") once the rendering has been completed; this prevents flickering and displaying of incompletely rendered frames on the display. Double buffering requires a significant amount of extra memory for the back buffer, however it is also necessary for how graphics is rendered today.



In most libraries and frameworks today you don't have to care about double buffering, it's done automatically. For this reason in many frameworks you often need to indicate the end of rendering with some special command such as `flip`, `endFrame` etc. If you're going lower level, you may need to implement double buffering yourself.

Though we encounter the term mostly in computer graphics, the principle of using a second buffer in order to ensure the result is presented only when it's ready can be applied also elsewhere.

Let's take a small example: say we're rendering a frame in a 3D game. First we render the environment, then on top of it we render the enemies, then effects such as explosions and then at the top of all this we render the GUI. Without double buffering we'd simply be rendering all these pixel into the front buffer, i.e. the memory that is immediately shown on the display. This would lead to the user literally seeing how first the environment appears, then enemies are drawn over it, then effects and then the GUI. Even if all this redrawing takes an extremely short time, it is also the case that the final frame will be shown for a very short time before another one will start appearing, so in the result the user will see huge flickering: the environment may look kind of normal but the enemies, effects and GUI may appear transparent because they are only visible for a fraction of the frame. The user also might be able to see e.g. enemies that are supposed to be hidden behind some object if that object is rendered after the enemies. With double buffering this won't happen as we perform the rendering into the back buffer, a memory which doesn't show on the display. Only when we have completed the frame in the back buffer, we copy it to the front buffer, pixel by pixel. Here the user may see the display changing from the old frame to the new one from top to the bottom, but he will never see anything temporary, and since the old and new frames are usually very similar, this top-to-bottom update may not even be distracting (it is addressed by vertical synchronization if we really want to get rid of it).

There also exists triple buffering which uses yet another additional buffer to increase FPS. With double buffering we can't start rendering a new frame into back buffer until the back buffer has been copied to the front buffer which may further be delayed by vertical synchronization, i.e. we have to wait and waste some time. With triple buffering we can start rendering into the other back buffer while the other one is being copied to the front buffer. Of course this consumes significantly more memory. Also note that triple buffering can only be considered if the hardware supports parallel rendering and copying of data, and if the FPS is actually limited by this... mostly you'll find your FPS bottleneck is elsewhere in which case it makes no sense to try to implement triple buffering. On small devices like embedded you probably shouldn't even think about this.

Double buffering can be made more efficient by so called page flipping, i.e. allowing to switch the back and front buffer without having to physically copy the data, i.e. by simply changing the pointer of a display buffer. This has to be somehow supported by hardware.

When do we actually need double buffering? Not always, we can avoid it or suppress its memory requirements if we need to, e.g. with so called frameless rendering -- we may want to do this e.g. in embedded programming where we want to save every byte of RAM. The mainstream computers nowadays simply always run on a very fast FPS and keep redrawing the screen even if the image doesn't change, but if you write a program that only occasionally changes what's on the screen (e.g. an e-book reader), you may simply leave out double buffering and actually render to the front buffer once the screen needs to change, the user probably won't notice any flicker during a single quick frame redraw. You also don't need double buffering if you're able to compute the final pixel color right away, for example with ray tracing you don't need any double buffering, unless of course you're doing some complex postprocessing. Double buffering is only needed if we compute a pixel color but that color may still change before the frame is finished. You may also only use a partial double buffer if that is possible (which may not be always): you can e.g. split the screen into 16 regions and render region by region, using only a 1/16th size double buffer. Using a palette can also make the back buffer smaller: if we use e.g. a 256 color palette, we only need 1 byte for every pixel of the back buffer instead of some 3 bytes for full RGB. The same goes for using a smaller resolution that is the actual native resolution of the screen.

downto

Downto Operator

In C the so called "downto" operator is a joke played on nubs. It goes like this: Did you know C has a hidden downto operator - ->? Try it:

```
#include <stdio.h>

int main(void)
```

```
{
  int n = 20;

  while (n --> 10) // n goes down to 10
    printf("%d\n",n);

  return 0;
}
```

Indeed this compiles and works. In fact --> is just -- and > operators.

drummyfish

Drummyfish

"Next time you're considering offing yourself, go for it." --genuine reaction of normal people in Xonotic to drummyfish advocating people should love each other

{ My email is currently: drummyfish AT disroot DOT org. ~drummyfish }

Drummyfish (also known as *tastyfish*, *drummy*, *drumy*, *smellyfish* and *i forcefeed my diarrhea to capitalism*) is a programmer, anarchopacifist and proponent of free software/culture, who started this wiki and invented the kind of software it focuses on: less retarded software (LRS). Besides others he has written Anarch, small3dlib, raycastlib, smallchesslib, tinyphysicsengine, SAF and comun. He has also been creating free culture art and otherwise contributing to free projects such as OpenMW; he's been contributing with public domain art of all kind (2D, 3D, music, ...) and writings to Wikipedia (no longer cause ban), Wikimedia Commons (also banned now), opengameart, libregamewiki, freesound and others. Drummyfish is insane/neuroretarded, suffering from anxiety/depression/etcetc. (diagnosed avoidant personality disorder) and has more than once been called a schizo, though psychiatrists didn't officially diagnose him with schizophrenia (yet). He sometimes self harms, both physically and socially. Due to spreading uncensored truth, helping and loving others and revealing corruption he is banned and censored on many places on the Internet, including Wikipedia, Wikimedia Commons, 4chan, GitLab, many subreddits, some Xonotic and Openarena servers etc. He also has no real life and is pretty retarded when it comes to leading projects or otherwise dealing with people or practical life. Drummyfish's political compass is off the charts, he once tried to take the political compass test, the computer got confused and exploded. He is also a wizard.

Drummyfish is the most physically disgusting bastard on Earth, no woman ever loved him, he is so ugly people get suicidal thoughts from seeing any part of him.

He loves all living beings, even those whose attributes he hates or who hate him. He is a vegetarian (since about 2018) and here and there supports good causes, for example he donates hair and gives money to homeless people who ask for them and sometimes cleans the Earth from plastic garbage (something he learned when he slaved as a factory cleaner). He also tried to donate blood but couldn't because he's taking antidepressants.

Drummyfish has a personal website at www.tastyfish.cz, and a gopherhole at self.tastyfish.cz. He uses vim, doesn't have any favorite distro and will NEVER HAVE ONE (in fact he hates Linux and would use another kernel if it was possible).

Photos of drummyfish: young, older (after being confronted with real life) and naked.

Drummyfish's real name is Miloslav Ä Ä–Ä¾, he was born on 24.08.1990 and lives in Moravia, Czech Republic, Earth (he rejects the concept of a country/nationalism, the info here serves purely to specify a location). He is a more or less straight male of the white race. He started programming at high school in Pascal, then he went on to study compsci (later focused on computer graphics) in a Brno University of Technology and got a master's degree in 2017, however he subsequently refused to find a job in the industry, partly because of his views (manifested by LRS) and partly because of mental health issues. He rather chose to stay closer to the working class and do less harmful slavery such as cleaning and physical spam distribution, and continues hacking on his programming (and other) projects in his spare time in order to be able to do it with absolute freedom.

{ Why doxx myself? Following the LRS philosophy, I believe information should be free. Censorship -- even in the name of privacy -- goes against information freedom. We should live in a society in which people are moral and don't abuse others by any means, including via availability of their private information. And in order to achieve ideal society we have to actually live it, i.e. slowly start to behave as if it was already in place. Of course, I can't tell you literally everything (such as my passwords etc.), but the more I can tell you, the closer we are to the ideal society. ~drummyfish }

He likes many things such as animals, peace, freedom, programming, math and games (used to play Xonotic and OpenArena, even though he despises competitive behavior in real life). He plays piano and drums a little bit and tries to pick up new things like chess, go and language learning. He has no sense of smell (since birth).

Before becoming a kind of schizo, he used to be relatively normal, even had a girlfriend for a while -- for a long time he was a proprietary Windows normie, using Facebook and playing mainstream games like Trackmania and World of Warcraft (since vanilla, quit during WotLK, played tauren warrior named *Drummy*). In the university he started using GNU/Linux because it was convenient for the school work, but still mostly used Windows. Only near the end of his studies he became more interested in FOSS, after reading Richard Stallman's biography. At the beginning he promoted "open source" and used soynet platforms such as Fediverse, later on he found the suckless website and was enlightened by minimalism; he also started to see through the evils of open source, capitalism and other things and refused to conform, which led him to the path of becoming the aforementioned schizo.

In 2019 drummyfish has written a "manifesto" of his ideas called **Non-Competitive Society** that describes the political ideas of an ideal society. It is in the public domain under CC0 and available for download online. Around 2020 he spent a few months in mental hospital. Some time around 2023 he bought a tiny caravan inawoods and plans to live there, away from society.

Does drummyfish have divine intellect? Hell no, he's pretty retarded at most things, but thanks to his extreme tendency for isolation, great curiosity and obsession with truth he is possibly the only man on Earth completely immune to propaganda, he can see the world as it is, not as it is presented, so he feels it is his moral duty to share what he is seeing. He is able to overcome his natural dumbness by tryharding and sacrificing his social and sexual life so that he can program more. If drummyfish can learn to program LRS, so can you.

See Also

- autism
- schizo

duke3d

Duke Nukem 3D

Duke Nukem 3D (often just *duke 3D*) is a legendary first person shooter video game released in January 1996 (as shareware), one of the best known such games and possibly the second greatest 90s FPS right after Doom. It was made by 3D realms, a company competing with Id software (creators of Doom), in engine made by Ken Silverman. Duke 3D is a big sequel to two previous games which were just 2D platformers; when this 3rd installment came out, it became a hit. It is remembered not only for being very technologically advanced, further pushing advanced fully textured 3D graphics that Doom introduced, but also for its great gameplay and above all for its humor and excellent parody of the prototypical 80s overttestosteroned alpha male hero, the protagonist Duke himself -- it showed a serious game didn't have to take itself too seriously and became loved exactly for things like weird alien enemies or correct portrayal of women as mere sexual objects which nowadays makes feminists screech in furious rage of thousand suns. Only idiots criticised it. Duke was later ported to other platforms (there was even a quite impressive 3D port for GBA) and received a lot of additional "content".

Of course, Duke is sadly proprietary, as most gaymes, though the source code was later released as FOSS under GPL (excluding the game data and proprietary engine, which is only source available). A self-proclaimed FOSS engine for Duke with GPU accelerated graphics exists: EDuke32 -- the repository is kind

of a mess though and it's hard to tell if it is legally legit as there are parts of the engine's proprietary code (which may be actually excluded from the compiled binary), so... not sure.

Code

The codebase (including Build engine) is roughly 100000 LOC of C, with some parts in assembly.

The original system requirements were roughly following: 66 MHz CPU, 16 MB RAM and 30 MB storage space.

Duke ran on **Build engine**, a legendary software rendering primitive 3D engine that had limitations similar to those of Doom engine, i.e. the camera could not genuinely rotate up or down (though it could fake this with kind of a "tilting") and things like rooms above other rooms in a level were allowed only in limited ways (hacks such as extra rendering passes or invisible teleports were used to allow this). The engine was not unsimilar to that of Doom, enemies and other objects were represented with 2D sprites and levels were based on the concept of sectors (a level was really made as a 2D map in which walls were assigned different heights and textures), however it had new features -- most notably dynamic environment, meaning that levels could change on the fly without the need for precomputation, allowing e.g. destructible environments. How the fuck did they achieve this? Instead of BSP rendering (used by Doom) Build engine used **portal rendering**: basically (put in a quite simplified way) there was just a set of sectors, some of which shared walls ("portals") -- rendering would first draw the sector the player stood in (from the inside of course) and whenever it encountered a portal wall (i.e. a wall that sees into another sector), it would simply recursively render that too in the same way -- turns out this was just fine. Other extra features of the engine included tilted floors and ceilings, fake looking up/down, 3rd person view etc. The Build engine was also used in many other games (most notably Shadow Warrior and Blood) and later incorporated even more advanced stuff, such as voxel models, though these weren't yet present in Duke. Just like Doom, Build engine **only used fixed point**, no float! { Hmm, actually maybe there was a small exception, see the link below. ~drummyfish }

{ Here are some details on the engine internals from a guy who specializes on this stuff:
https://fabiansanglard.net/duke3d/build_engine_internals.php. ~drummyfish }

See Also

- Doom
- Anarch
- Jedi engine

dungeons_and_dragons

Dungeons And Dragons

TODO

{ Sorry, I may start world war 3 by saying this, but I think D&D sucks. ~drummyfish }

The idea behind D&D is really, really cool in theory... HOWEVER in practice we just get something so excruciatingly awfully cringe; screeching, sweating autistic fat men pretending to be young female fairies, it would really be much more pleasant to eat glass than watch D&D session in progress. D&D is not really like what it seems at first sight, it is NOT a game for "smart" people who don't fit in because of their huge IQ, it's more of a last resort place for people who would LOVE to socialize so much but can't because they're extremely ugly or unlovable and no one wants to be around them. It's really so SO PAINFUL to watch the most beta soy fatman trying to masturbate his ego by awful attempts at acting and pretending he's a general of orc army or something while holding some plastic sword, thinking that when he drops all restraints and screeches on top of his lungs he will suddenly look like Brad Pitt and he'll impress that one real weird beta female that's always present in any D&D session. Real nerds just hate people and won't try to look for socialization even with other nerds. Furthermore it's not even a real game with strictly set rules, it's more of a collaborative story writing and acting, it's almost closer to dumb normie stuff like theatre and whatnot, a real nerd MAY want to get into writing, but he will have enough of a vision to not let some random averages

fuck up his universe, that just smells by want of social interaction rather than being an attempt at creating good art -- so yeah, it's not really about fantasy, playing a game or anything, just about desperately trying to interact with any people at all for any cost.

duskos

Dusk OS

Dusk OS is a work in progress non-Unix extremely minimalist 32 bit free as in freedom operating system whose primary purpose is to be helpful during societal collapse but which will still likely be very useful even before it happens. It is made mainly by Virgil Dupras, the developer of Collapse OS, as a bit "bigger" version of Collapse OS, one that's intended for the first stage of societal collapse and will be more "powerful" and comfortable to use for the price of increased complexity (while Collapse OS is simpler and meant for the use during later stages). But don't be fooled, Dusk OS is still light year ahead in simplicity than the most minimal GNU/Linux distro you can imagine; by this extremely minimalist design Dusk OS is very close to the ideals of our LRS, it is written in Forth but also additionally (unlike Collapse OS) includes a so called "Almost C" compiler allowing ports of already existing programs, e.g. Unix command line utilities. It is also available under CC0 public domain just as official LRS projects, that's simply unreal.

The project has a private mailing list. Apparently there is talk about the system being useful even before the collapse and so it's even considering things like networking support etc. -- as capitalism unleashes hell on Earth, any simple computer capable of working on its own and allowing the user complete control will be tremendously useful, even if it's just a programmable calculator. Once GNU/Linux and BSDs sink completely (very soon), this may be where we find the safe haven.

The only bad thing about the project at the moment seems to be the presence of some pseudoleftists around the project, threatening political takeover etcetc. At the moment there's thankfully no code of censorship :-) If the project gets more popular it will be forced. Hopefully forks will come if anything goes wrong. In any case, we will be continuing our independent work that will probably yield a similar system, thought much later.

{ I'm not 100% sure about everything that will follow as I'm still studying this project, please forgive mistakes, double check claims. ~drummyfish }

It really does look amazing and achieves some greatly awesome things, for example it's (as far as it seems) completely self-hosted from the ground up, containing no binary blobs (at least for the bootstrap it seems, driver state unchecked), being able to completely bootstrap itself from some 1000 lines of assembly, however it still keeps portability by using a kind of simple abstract assembly called HAL, so it's not tied to any CPU architecture like most other simple OSes. The "user mode" system can also be compiled as a normal program on current operating systems, allowing to make so called Dusk packages, i.e. automatically export any Dusk OS program as a basically native program for current computers. You don't even need any virtual machine or emulator to try out the OS, you just download it, type make and run it as a normal program. The whole system is also very fast. There's some black magic going on. Also the project documentation is very nice. It also seems they don't care about security and safety at all, WHICH IS EXTREMELY GOOD, there is no place for such bullshit in a truly minimalist project.

Let's sum up some of the interesting features of the system:

- "aggressive" simplicity, simplicity is above speed (but speed is still great and high among priorities)
- written in Forth, offering simplified C compiler as its main feature
- portable, using simple abstract assembly to run on different architectures
- no concurrency, linear computation only, this will probably imply no multitasking
- no virtual memory
- preferring static memory allocation
- making use of global states/variables (going against mainstream capitalist "best practices")
- 32 bit system, maximum 4 GB of memory
- no user friendliness, hacker "operator" assumed instead of a layman user
- no "security", "safety", "privacy", user accounts etc.

- graphics: primarily no GUI, simple graphics capabilities will be present, direct access to screen pixels (through so called "grid" interface)
 - some nice text shell will be made
 - sound: ???
 - networking: being considered
 - NOT a Unix-like but will naturally be aligned with many of its concepts as per simplicity
-

dynamic_programming

Dynamic Programming

Dynamic programming is a programming technique that can be used to make many algorithms more efficient (usually meaning faster). It can be seen as an optimization technique that works on the principle of repeatedly breaking given problem down into smaller subproblems and then solving one by one from the simplest and remembering already calculated results that can be reused later.

It is frequently contrasted to the divide and conquer (DAC) technique which at the first sight looks similar but is in fact quite different. DAC also subdivides the main problem into subproblems, but then solves them recursively, i.e. it is a top-down method. DAC also doesn't remember already solved subproblem and may end up solving the same problem multiple times, wasting computational time. Dynamic programming on the other hand starts solving the subproblems from the simplest ones -- i.e. it is a **bottom-up** method -- and remembers solutions to already solved subproblems in some kind of a table which makes it possible to quickly reuse the results if such subproblem is encountered again. The order of solving the subproblems should be made such as to maximize the efficiency of the algorithm.

It's not the case that dynamic programming is always better than DAC, it depends on the situation. Dynamic programming is effective **when the subproblems overlap** and so the same subproblems WILL be encountered multiple times. But if this is not the case, DAC can easily be used and memory for the look up tables will be saved.

Example

Let's firstly take a look at the case when divide and conquer is preferable. This is for instance the case with many sorting algorithms such as quicksort. Quicksort recursively divides parts of the array into halves and sorts each of those parts: sorting each of these parts is a different subproblem as these parts (at least mostly) differ in size, elements and their order. The subproblems therefore don't overlap and applying dynamic programming makes little sense.

But if we tackle a problem such as computing Nth Fibonacci number, the situation changes. Considering the definition of Nth Fibonacci number as a "*sum of N-1th and N-2th Fibonacci numbers*", we might naively try to apply the divide and conquer method:

```
int fib(int n)
{
    return (n < 2) ?
        n : // start the sequence with 0, 1
        fib(n - 1) + fib(n - 2); // else add two previous
}
```

But we can see this is painfully slow as calling `fib(n - 2)` computes all values already computed by calling `fib(n - 1)` all over again, and this inefficiency additionally appears inside these functions recursively. Applying dynamic programming we get a better code:

```
int fib(int n)
{
    if (n < 2)
        return n;

    int current = 1, prev = 0;

    for (int i = 2; i <= n; ++i)
```

```

{
    int tmp = current;
    current += prev;
    prev = tmp;
}

return current;
}

```

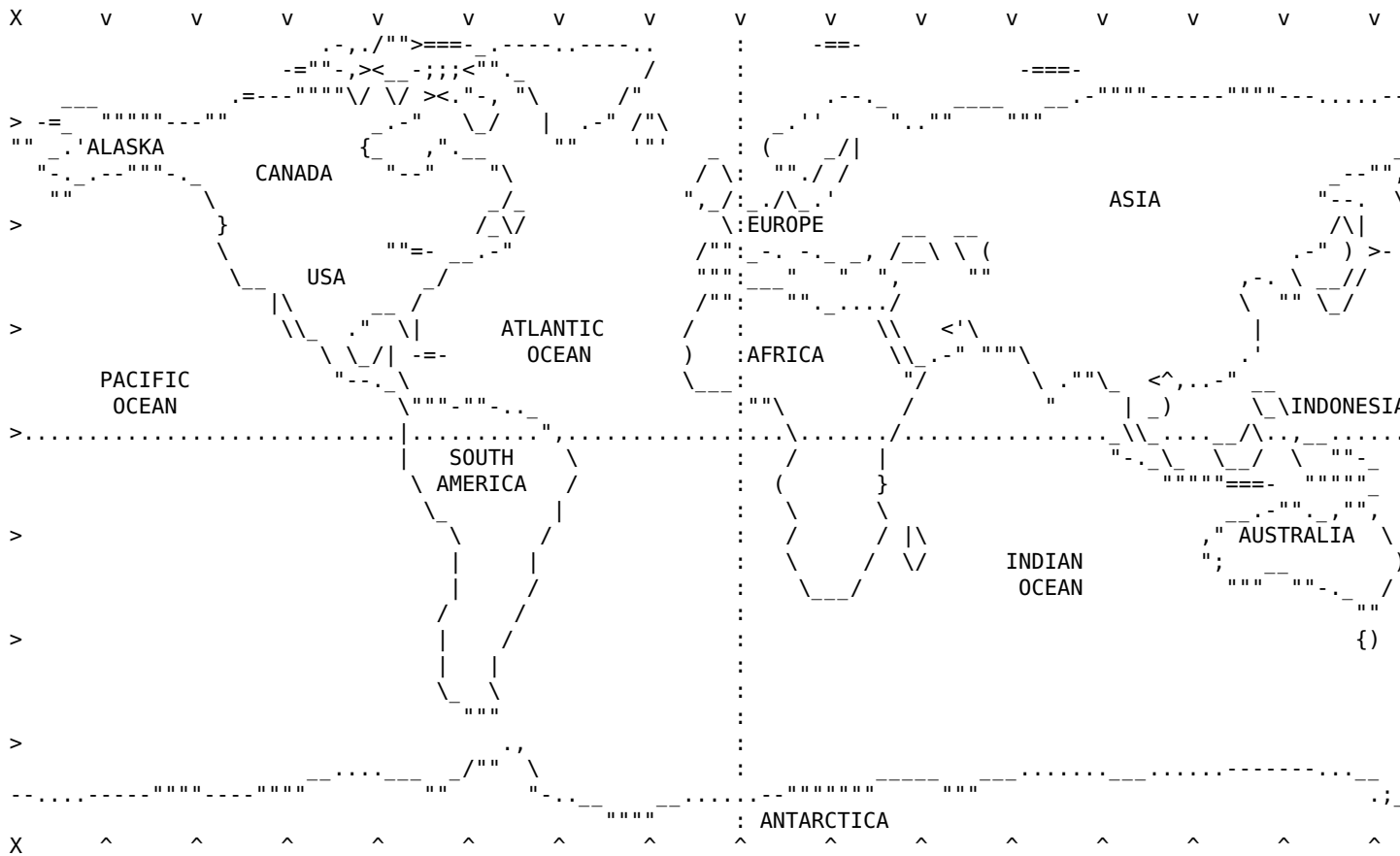
We can see the code is longer, but it is faster. In this case we only need to remember the previously computed Fibonacci number (in practice we may need much more memory for remembering the partial results).

earth

Earth

Well, Earth is the planet we live on. It is the third planet from the Sun of our Solar system which itself is part of the Milky Way galaxy, Universe. So far it is the only known place to have life.

Now behold the grand rendering of the Earth map in ASCII (equirectangular projection):



Some numbers about the planet Earth:

- age: 4.54 billion years
- distance from the Sun (nearest, furthest): 147098450 km, 152097597 km,
- radius (equator, poles): 6378 km, 6356 km
- mass: $5.9 \cdot 10^{24}$ kg
- acceleration by gravity: 9.8 m/s^2
- axial tilt: 23.4 degrees
- length of year: 365.25 days
- land vs water area: 148940000 km^2 , 361132000 km^2

easier_done_than_said

Easier Done Than Said

Easier done than said is the opposite of easier said than done.

Example: exhaling, as saying the word "exhaling" requires exhaling plus doing some extra work such as correctly shaping your mouth.

easy_to_learn_hard_to_master

Easy To Learn, Hard To Master

"Easy to learn, hard to master" (ETLHTM) is a type of design of a game (and by extension a potential property of any art or skill) which makes it relatively easy to learn to play while mastering the play (playing in near optimal way) remains very difficult.

Examples of this are games such as tetris, minesweeper or Trackmania.

LRS sees the ETLHTM design as extremely useful and desirable as it allows for creation of suckless, simple games that offer many hours of fun. With this philosophy we get a great amount of value for relatively little effort.

This is related to a fun coming from **self imposed goals**, another very important and useful concept in games. Self imposed goals in games are goals the player sets for himself, for example completing the game without killing anyone (so called "pacifist" gameplay) or completing it very quickly (speedrunning). Here the game serves only as a platform, a playground at which different games can be played and invented -- inventing games is fun in itself. Again, a game supporting self imposed goals can be relatively simple and offer years of fun, which is extremely cool.

The simplicity of learning a game comes from simple rules while the difficulty of its mastering arises from the complex emergent behavior these simple rules create. Mastering of the game is many times encouraged by competition among different people but also competition against oneself (trying to beat own score). In many simple games such as minesweeper there exists a competitive scene (based either on direct matches or some measurement of skill such as speedrunning or achieving high score) that drives people to search for strategies and techniques that optimize the play, and to training skillful execution of such play.

The opposite is hard to learn, easy to master.

See Also

- easier done than said
 - speedrun
-

education

Education

not to be confused with indoctrination

TODO

egoism

Egoism

TODO

Some signs of egoism include:

- **Drawing someone's own face (or letting someone else do it) and then using it for a profile picture.** "Modern" soydevs on twitter and blogs are so guilty of this, especially the "game devs" with pixel art portraits etc., it's so narcissistic and cringe you just want to puke. Why would anyone humble even allow someone to make a statue of him, allowing the danger of cult of personality and becoming a hero? Why the fuck do people obsess about forcing their ugly faces onto others?
- **Decorating one's body obsessively**, especially with tattoos, wild hairstyles, clothes that serves other purpose than pure protection from weather, for example a suit etc.
- **Putting one's name in (or near) the title of his creation:** one of the most famous examples being the *Shit Faggot's Game Of Civilization*. Yes, Linux counts too; even though Linus didn't name it himself, he just waited for someone else to do it for him and then didn't protest; he also joked about it, trying to make it look OK, though without success (see below).
- **Egoism masked as joking**, i.e. doing something egoistic and then pretending to do it for the sake of a joke; for example in the book *World of Warcraft Diary* the author X puts a huge quote of himself on one page and jokingly writes under it "X quoting X in his own book" -- hahaha we laughed ok? It's not egoism, it's done for a joke, BTW the quote will stay there. The author here thinks he is smart as he think this achieves two things: promoting himself while also making him look like someone with a sense of humor. In fact it just makes him look like the most egocentric bastard.
- **Assertiveness**, also saying shit like "to love others you first have to love yourself" etc., honestly I don't know who comes up with such crap lol.
- **Being a capitalist**; by definition a capitalist only cares about himself, capitalist is incapable of love or wanting any benefit for anyone else than himself, he only benefits others if he sees it will somehow lead to his own benefit in the future (this applies even to for example to caring about his own family etc.).
- **Using licenses that require giving credit**, such as CC-BY-SA.
- **"Personal pronouns"**.
- **Making (or allowing others to make) art that glorifies you**, e.g. documentaries, books etc.
- ...

elo

Elo

The Elo system (named after Arpad Elo, NOT an acronym) is a mathematical system for rating the relative strength of players of a certain game, most notably and widely used in chess but also elsewhere (video games, table tennis, ...). Based on number of wins, losses and draws against other Elo rated opponents, the system computes a number (rating) for each player that highly correlates with that player's current strength/skill; as games are played, ratings of players are constantly being updated to reflect changes in their strength. The numeric rating can then be used to predict the probability of a win, loss or draw of any two players in the system, as well as e.g. for constructing ladders of current top players and matchmaking players of similar strength in online games. For example if player A has an Elo rating of 1700 and player B 1400, player A is expected to win in a game with player B with the probability of 85%. Besides Elo there exist alternative and improved systems, notably e.g. the Glicko system (which further adds e.g. confidence intervals).

The Elo system was created specifically for chess (even though it can be applied to other games as well, it doesn't rely on any chess specific rules) and described by Arpad Elo in his 1978 book called *The Rating of Chessplayers, Past and Present*, by which time it was already in use by FIDE. It replaced older rating systems, most notably the Harkness system. Despite more "advanced" systems being around nowadays, Elo remains the most widely used one.

Elo rates only RELATIVE performance, not absolute, i.e. the rating number of a player says nothing in itself, it is only the DIFFERENCE in rating points between two players that matters, so in an extreme case two

players rated 300 and 1000 in one rating pool may in another one be rated 10300 and 11000 (the difference of 700 is the only thing that stays the same, mean value can change freely). This may be influenced by initial conditions and things such as **rating inflation** (or deflation) -- if for example a chess website assigns some start rating to new users which tends to overestimate an average newcomer's abilities, newcomers will come to the site, play a few games which they will lose, then they ragequit but they've already fed their points to the good players, causing the average rating of a good player to grow over time.

Keep in mind Elo is a big simplification of reality, as is any attempt at capturing skill with a single number -- even though it is a very good predictor of something akin a "skill" and outcomes of games, trying to capture a "skill" with a single number is similar to e.g. trying to capture such a multidimensional thing as intelligence with a single dimensional IQ number. For example due to many different areas of a game to be mastered and different playstyles transitivity may be broken in reality: it may happen that player A mostly beats player B, player B mostly beats player C and player C mostly beats player A, which Elo won't capture.

How It Works

Initial rating of players is not specified by Elo, each rating organization applies its own method (e.g. assign an arbitrary value of let's say 1000 or letting the player play a few unrated games to estimate his skill).

Suppose we have two players, player 1 with rating A and player 2 with rating B . In a game between them player 1 can either win, i.e. score 1 point, lose, i.e. score 0 points, or draw, i.e. score 0.5 points.

The expected score E of a game between the two players is computed using a sigmoid function (400 is just a magic constant that's usually used, it makes it so that a positive difference of 400 points makes a player 10 times more likely to win):

$$E = 1 / (1 + 10^{((B - A)/400)})$$

For example if we set the ratings $A = 1700$ and $B = 1400$, we get a result $E \approx 0.85$, i.e. in a series of many games player 1 will get an average of about 0.85 points per game, which can mean that out of 100 games he wins 85 times and loses 16 times (but it can also mean that out of 100 games he e.g. wins 70 times and draws 30). Computing the same formula from the player 2 perspective gives $E \approx 0.15$ which makes sense as the number of points expected to gain by the players have to add up to 1 (the formula says in what ratio the two players split the 1 point of the game).

After playing a game the ratings of the two players are adjusted depending on the actual outcome of the game. The winning player takes some amount of rating points from the loser (i.e. the loser loses the same amount of point the winner gains which means the total number of points in the system doesn't change as a result of games being played). The new rating of player 1, A_2 , is computed as:

$$A_2 = A + K * (R - E)$$

where R is the outcome of the game (for player 1, i.e. 1 for a win, 0 for loss, 0.5 for a draw) and K is the change rate which affects how quickly the ratings will change (can be set to e.g. 30 but may be different e.g. for new or low rated players). So with e.g. $K = 25$ if for our two players the game ends up being a draw, player 2 takes 9 points from player 1 ($A_2 = 1691$, $B_2 = 1409$, note that drawing a weaker player is below the expected result).

How to compute Elo difference from a number of games? This is useful e.g. if we have a chess engine X with Elo E_X and a new engine Y whose Elo we don't know: we may let these two engines play 1000 games, note the average result E and then compute the Elo difference of the new engine against the first engine from this formula (derived from the above formula by solving for Elo difference $B - A$):

$$B - A = \log_{10}(1 / E - 1) * 400$$

Some Code

Here is a C code that simulates players of different skills playing games and being rated with Elo. Keep in mind the example is simple, it uses the potentially imperfect rand function etc., but it shows the principle

quite well. At the beginning each player is assigned an Elo of 1000 and a random skill which is normally distributed, a game between two players consists of each player drawing a random number in range from 1 to his skill number, the player that draws a bigger number wins (i.e. a player with higher skill is more likely to win).

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PLAYERS 101
#define GAMES 10000
#define K 25          // Elo K factor

typedef struct
{
    unsigned int skill;
    unsigned int elo;
} Player;

Player players[PLAYERS];

double eloExpectedScore(unsigned int elo1, unsigned int elo2)
{
    return 1.0 / (1.0 + pow(10.0, (((double) elo2) - ((double) elo1)) / 400.0));
}

int eloPointGain(double expectedResult, double result)
{
    return K * (result - expectedResult);
}

int main(void)
{
    srand(100);

    for (int i = 0; i < PLAYERS; ++i)
    {
        players[i].elo = 1000; // give everyone initial Elo of 1000

        // normally distributed skill in range 0-99:
        players[i].skill = 0;

        for (int j = 0; j < 8; ++j)
            players[i].skill += rand() % 100;

        players[i].skill /= 8;
    }

    for (int i = 0; i < GAMES; ++i) // play games
    {
        unsigned int player1 = rand() % PLAYERS,
                    player2 = rand() % PLAYERS;

        // let players draw numbers, bigger number wins
        unsigned int number1 = rand() % (players[player1].skill + 1),
                    number2 = rand() % (players[player2].skill + 1);

        double gameResult = 0.5;

        if (number1 > number2)
            gameResult = 1.0;
        else if (number2 > number1)
            gameResult = 0.0;

        int pointGain = eloPointGain(eloExpectedScore(
            players[player1].elo,
            players[player2].elo), gameResult);

        players[player1].elo += pointGain;
        players[player2].elo -= pointGain;
    }
}
```

```

for (int i = PLAYERS - 2; i >= 0; --i) // bubble-sort by Elo
    for (int j = 0; j <= i; ++j)
        if (players[j].elo < players[j + 1].elo)
        {
            Player tmp = players[j];
            players[j] = players[j + 1];
            players[j + 1] = tmp;
        }

for (int i = 0; i < PLAYERS; i += 5) // print
    printf("#%d: Elo: %d (skill: %d%%)\n", i, players[i].elo, players[i].skill);

return 0;
}

```

The code may output e.g.:

```

#0: Elo: 1134 (skill: 62%)
#5: Elo: 1117 (skill: 63%)
#10: Elo: 1102 (skill: 59%)
#15: Elo: 1082 (skill: 54%)
#20: Elo: 1069 (skill: 58%)
#25: Elo: 1054 (skill: 54%)
#30: Elo: 1039 (skill: 52%)
#35: Elo: 1026 (skill: 52%)
#40: Elo: 1017 (skill: 56%)
#45: Elo: 1016 (skill: 50%)
#50: Elo: 1006 (skill: 40%)
#55: Elo: 983 (skill: 50%)
#60: Elo: 974 (skill: 42%)
#65: Elo: 970 (skill: 41%)
#70: Elo: 954 (skill: 44%)
#75: Elo: 947 (skill: 47%)
#80: Elo: 936 (skill: 40%)
#85: Elo: 927 (skill: 48%)
#90: Elo: 912 (skill: 52%)
#95: Elo: 896 (skill: 35%)
#100: Elo: 788 (skill: 22%)

```

We can see that Elo quite nicely correlates with the player's real skill.

elon_musk

Elon Mu\$k

Elon Musk is an enormous capitalist dick. Elon's hair is the least fake thing about him. His IQ is immeasurably low but he liked to LARP as Einstein on Twitter, it's super cringe, he's like a child, just more retarded and uglier.

TODO: more dirt

Musk's company Neuralink killed 1500 animals in 4 years, was charged with animal cruelty (sauce).

TODO: that moment he tried to play superhero when the kids got stuck in the cave :D

e

E

Euler's number (not to be confused with Euler number), or e , is an extremely important and one of the most fundamental numbers in mathematics, approximately equal to 2.72, and is almost as famous as π . It appears very often in mathematics and nature, it is the base of natural logarithm, its digits after the decimal point go on forever without showing a simple pattern (just as those of π), and it has many more interesting

properties.

It can be defined in several ways:

- Number e is such number for which a function $f(x) = e^x$ (so called exponential function) equals its own derivative, i.e. $f(x) = f'(x)$.
- Number e is a limit of the infinite series $1/0! + 1/1! + 1/2! + 1/3! + \dots$ (! signifies factorial). I.e. adding all these infinitely many numbers gives exactly e .
- Number e is a number greater than 1 for which integral of function $1/x$ from 1 to e equals 1.
- Number e is the base of natural logarithm, i.e. it is such number e for which $\log(e, x) = \text{area under the function's curve from 1 to } x$.
- ...

e to 100 decimal digits is:

2.71828182845904523536028747135266249775724709369995957496696762772407663035354759457138217

e to 100 binary digits is:

10.1011011111100001010100010110001010001010111011010010101001101010101111110111000101011000

Just as π , e is a real transcendental number (it is not a root of any polynomial equation) which also means it is an irrational number (it cannot be expressed as a fraction of integers). It is also not known whether e is a normal number, which would mean its digits would contain all possible finite strings, but it is conjectured to be so.

TODO

encryption

Encryption

Encryption is just mathematically embraced obscurity.

TODO

encyclopedia

Encyclopedia

Encyclopedia (also encyclopaedia, cyclopedia or cyclopaedia, from Greek *enkyklios paideia*, roughly "general education") is a large book (or a series of books) providing structured summary of wide knowledge in one or many fields of knowledge (such as mathematics, history, engineering, general knowledge etc.), usually structured as a collection of alphabetically ordered articles on terms used in the field. Paper encyclopedias are oftentimes printed in several volumes as the amount of contained information is too great for a single book (in large ones you may even see one or two volumes dedicated ONLY for the index). The largest and most famous encyclopedia to date is the online Wikipedia created by volunteers in free culture spirit, however Wikipedia suffers from significant issues such as censorship, high political propaganda and low quality of writing, therefore it is important to also stay interested in other encyclopedias such as Britannica, Americana or LRS wiki.

Encyclopedias are awesome, get as many of them as you can, especially the printed ones -- they are usually relatively cheap (especially second hand books) and provide an ENORMOUS amount of information, FOREVER (no one can cancel your physically owned paper book, you will retain it even after the collapse when such books will become practically your only source of human knowledge). Also remember, paper books are still of much higher quality than online resources such as Wikipedia -- even if they lose in terms of sheer volume, they make up in quality of writing and still many times contain information that's not available online, and the older ones are more objective and trustworthy, considering the decline of free speech online.

Shorter articles may also do a better job at providing overall summary of a topic and filtering out less important information, as opposed to a gigantic Wikipedia article. Furthermore even if such a book isn't free as in freedom, the knowledge, information and data contained in it is in the public domain as such things cannot (yet) be owned, therefore it is possible to legally paraphrase the information into a new source which we may make public domain itself (however watch out to not merely copy-paste texts from encyclopedias as text CAN be copyrighted, as well as e.g. the mere selection of which facts to include; always be very careful).

Since an encyclopedia will typically focus on encompassing wide knowledge, as the other side of the coin its disadvantage will also oftentimes be shallowness, it will go into greater depth only on very important topics, although very big encyclopedias largely eliminate this issue and go fairly deep on all subjects; encyclopedias specialized on some particular subject can also afford to provide in-depth knowledge.

{ A favorite pastime of mine is looking up the same term in different encyclopedias and comparing them -- this can help get to the essence of actually understanding the term, as well as revealing censorship and different views of the authors. ~drummyfish }

Great nerds read encyclopedias linearly from start to finish just like a normal book, which may help expand one's knowledge as well as ignite curiosity in new things and spot some cool interesting facts. { And yet bigger nerds write their own encyclopedias. ~drummyfish }

Similar terms: encyclopedias, which also used to be called **cyclopedias** in the past, are similar to **dictionaries** and these types of books often overlap (many encyclopedias call themselves dictionaries); the main difference is that a dictionary focuses on providing linguistic information and generally has shorter term definitions, while encyclopedias have longer articles (which however limits their total number, i.e. encyclopedias will usually prefer quality over quantity). Encyclopedias are also a subset of so called **reference works**, i.e. works that serve to provide information and reference to it (other kinds of reference works being e.g. world maps, tabulated values or API references). A **universal/general** encyclopedia is one that focuses on human knowledge at wide, as opposed to an encyclopedia that focuses on one specific field of knowledge. **Compendium** can be seen almost as a synonym to encyclopedia, with encyclopedias perhaps usually being more general and extensive. **Almanac** is also similar to encyclopedia, more focusing on tabular data. **Micropedia** is another term, sometimes used to denote a smaller encyclopedia (one edition of Britannica came with a micropedia as well as a larger macropedia).

These are some **nice/interesting/benchmark articles** to look up in encyclopedias: algorithm, anarchism, Andromeda (galaxy), Antarctica, Atlantis, atom, axiom of choice, Bible, big bang, black hole, brain, Buddhism, C (programming language), cannibalism, capitalism, castle, cat, censorship, central processing unit, chess, Chicxulub, China, color, comet, communism, computer, Creative Commons, Deep Blue, democracy, Democratic People's Republic of Korea, depression, determinism, dinosaur, dodo, dog, Doom (game), Earth, Einstein, Elo, Encyclopedia, entropy, ethics, Euler's Number, evolution, font, football, fractal, free software, game, gigantopythecus, go (game), god, GNU project, hacker, Hanging Gardens of Babylon, hardware, Hitler, Holocaust, homosexual, human, information, intelligence, Internet, IQ, Japan, Jesus, Jew, language, Latin, life, light, lightning, Linux, logarithm, logic, love, Mammoth, mathematics, Mariana Trench, Mars, Milky Way, Moon, morality, Mount Everest, music, necrophilia, Open Source, negro, nigger, pacifism, pedophilia, penis, pi, Pluto, prime number, quaternion, Pompei, Quran, race, Roman Empire, sex, sine, schizophrenia, software, Stallman (Richard), star, Stonehenge, suicide, Sun, Tibet, technology, Tetris, time, Titanic, transistor, Troy, Tyrannosaurus Rex, UFO, universe, Unix, Uruk, Usenet, Valonia Ventricosa (bubble algae), Vatican, Venus, video game, Wikipedia, woman, World War II, World Wide Web, ...

What is the best letter in an encyclopedia? If you are super nerdy, you may start to search for your favorite starting letter -- this if fun and may also help you e.g. decide which volume of your encyclopedia to take with you when traveling. Which letter is best depends on many things, e.g. the language of the encyclopedia, its size, your area of interest and so on. Assuming English and topics that would be interesting to the readers of LRS wiki, the best letter is most likely C -- it is the second most common starting letter in dictionaries, has a great span and includes essential and interesting terms such as computer, C programming language, cat, communism, capitalism, chess, christianity, collapse, CPU, color, culture, copyleft, compiler, creative commons, cryptography, copyright, car, cancer, cellular automata, consumerism, cosine, Chomsky, CIA, cybernetics, cracking, chaos, carbon, curvature, chemistry, censorship and others. As close second comes S, the most frequent letter in dictionaries, with terms such as Stallman, science, shader, semiconductor, silicon, software, sound, socialism, state, selflessness, speech recognition, steganography, square root, sudoku, suicide, speedrun, space, star, Sun, sine, Soviet union, schizophrenia, set, suckless, shit,

sex and others. { This is based on a list I made where I assigned points to each letter. The letters that follow after C and S are P, M, A, E, T, L, R, F, D, G, I, B, H, U, N, W, V, J, O, K, Q, Z, Y, X. ~drummyfish }

Notable/Nice Encyclopedias

Here is a list of notable encyclopedias, focused on general knowledge English language ones. The most notable ones are in bold. Also check out existing encyclopedias in other languages that you speak, we can't list those here.

{ See also <https://wikiindex.org/>. ~drummyfish }

name	year	legal status	format	~articles	comment
Britannica 9th edition	1889	PD (old)	25 vol.		legendary enc., major edition, one of "Big Three", partly digitized (archive.org, wikisource, ...)
Britannica 11th edition	1910	PD (old)	29 vol.	40K	legendary enc., major edition, one of "Big Three", mostly digitized (both scan and txt), PC incorrect :)
Britannica Concise Encyclopedia	2002	proprietary	1 vol. 2000p	28K	nice, short descriptions, condensed from the main multivol. Brit., piratable pdf
Britannica online	...now	proprietary	online	130K	bloated, high quality articles, unpaid is limited and with ads
<u>Citizendium</u>	2006...	proprietary? (NC)	online	18K	Wikipedia alternative, censored, faggots have unclear license
Chambers Encyclopedia (new)	2001	proprietary	1 vol. 980p		1 vol republication of old multivol. enc. (going back to 1800s, already PD), topic-sorted
Collier's New Encyclopedia	1921	PD (old)	10 vol.		NOT TO BE CONFUSED with Collier's Encyclopedia (different one), digitized on Wikisource (txt)
Columbia Encyclopedia	1935...	proprietary	1 vol. ~3Kp	~50K	high quality, lots of information { Read the 1993 edition, it's super nice. ~drummyfish }
<u>Conservaped.</u>	2006...	proprietary	online	52K	American fascist wiki, has basic factual errors
Larousse Desk Reference Enc.	1995	proprietary	1 vol. 800p	200K?	by James Hughes, nice, quality general overviews, topic-ordered { I bought this, it's nice. ~drummyfish }
Domestic Encyclopaedia	1802	PD (old)	4 vol.		shorter articles, partially digitized on Wikisource
Encyclopedia Americana	1820...	PD (old)	~30 vol.		longer articles, one of "Big Three", several editions (1906, 1920) partly digitized on wikisource
Encyclopedia Dramatica	2004...	PD (CC0)	online	15K	informal/fun/"offensive" but valuable info (on society, tech, ...), basically no censorship, no propaganda
Encyclopedia of Marxism	1999...	CC BY-SA	online	~3K	focused on Marxism, quality, shorter articles
Everybodywiki	2017...	CC BY-SA	online	~300K	alternative to Wikipedia allowing articles on non notable things and people
Google Knol	~2010	proprietary	online		failed online enc. by Google, archived on archive.org
Grolier Multimedia Encyclopedia	2003	proprietary	CD		

name	year	legal status	format	~articles	comment
Illustrated Family Encyclopedia	1997	proprietary	2 vol. 920p	5K	kid-friendly, nice pictures, USA bias, piratable
<u>Infogalactic</u>	2016...	CC BY-SA	online	2M	Wikipedia fork, no SJW censorship, FOR PROFIT (you can buy article control lol), can't make accounts
<u>Leftypedia</u>	2020...	GFDL	online	~200	Leftist encyclopedia, currently NOT littered by SJWs, writing about all branches of the "left"
<u>LRS wiki</u>	2021...	PD (CC0)	online/elec.	500	best encyclopedia, focused on tech/society, no censorship
<u>Metapedia</u>	2006...	GFDL	online	7K	Wikipedia fork, online, no SJW censorship, ATM limited account creation, "pro-European" fascism
Microsoft <u>Encarta</u>	...2009	proprietary	electronic	62K	Micro\$oft enc., low quality articles (errors), MS propaganda (no free software etc. lol), is on archive.org
<u>ProleWiki</u>	2020...	proprietary	online	~3K	Proletariat wiki, similar to Leftypedia but focused on Marxism-Leninism.
Simple English Wikipedia	2001...	CC BY-SA	online	200K	Wikipedia with simpler language and simpler explanations, censored
The New American Cyclopaedia	1879	PD (old)	16 vol.		partially digitized on Wikisource (txt)
The New International Encyc.	1905	PD (old)	20 vol.		partially digitized on Wikisource
The Nuttall Encyclopaedia	1907	PD (old)	1 vol.	16K	short articles, oldschool, digitized (gutenberg)
<u>Vikidia</u>	2006...	CC BY-SA	online	4K	"Wikipedia for kids", probably as censored as Wikipedia
Webster's Unabridged Dictionary	1864	PD (old)	paper	476K	short descriptions, digitized (gutenberg)
<u>Wikipedia</u>	2001...	CC BY-SA	online	6M	largest and most famous, EXTREME PSEUDOLEFTIST CENSORSHIP AND POLITICAL PROPAGANDA, free culture
Old Wikipedia	2001	GFDL	online	19K	archived old Wikipedia, less censorship, https://nostalgia.wikipedia.org
Pears' Cyclopedia	1897	PD (old)	1 vol. 740p		contains dictionary, general knowl. maps, reference etc., scanned on archive.org
World Almanac and Book of Facts	1868...	some PD (old)	1 vol.		interesting and useful information, data and facts from old to new age, US-centered
The World Book	1917...	proprietary	22 vol.	17K	best selling print enc., large, high quality but for younger audience, US propaganda (anticommunism etc.)
The World Book 1917	1917	PD (old)	8 vol.	3K	nicely readable
Uncyclopedia	2005...	proprietary (NC)	online	37K	parody, <u>fun</u> enc., "more normie friendly dramatica"

See Also

- [wiki](#)
 - [Jargon File](#)
 - [wikiwikiweb](#)
-

english

English

"there'dn't've" --English

English is a natural human language spoken mainly in the USA, UK and Australia as well as in dozens of other countries and in all parts of the world (with about 1.5 billion speakers). It is the default language of the world nowadays. Except for the awkward relationship between written English and its pronunciation it is a pretty simple and suckless language (even though not as suckless as Esperanto), even a braindead man can learn it { Knowing Czech and learning Spanish, which is considered one of the easier languages, I can say English is orders of magnitude simpler. ~drummyfish }. It is the lingua franca of the tech world (virtually every programming language is based on English for example) and many other worldwide communities as well as the Internet. Thanks to its simplicity (lack of declension, fixed word order, relatively simple grammatical rules etc.) it is pretty suitable for computer analysis and as a basis for programming languages.

If you haven't noticed, this wiki is written in English.

Retarded Mistakes You Make In English

TODO

- **"The reason is because ..."**: This is just awful, if you have any brain at all your ears just bleed. Correctly you say "The reason is (that) ...". Why? Here is a small helper: "Why are you not working?", "Because I'm lazy.", "What is the sound?", "The sound is noise.", "You are not working. What is the reason?", "The reason is my laziness" OR "The reason is (that) I'm lazy."
 - **"Just because X doesn't mean Y"**: this is completely grammatically wrong, it is either "Just the fact that X doesn't mean Y." OR "Just because X not(Y)." You just have to hear it, but here is an attempt at showing you why this it's wrong: "I don't have to cry just because I'm sad.", we can reverse the sentences to "Just because I'm sad I don't have to cry" OR "Just the fact that I'm sad doesn't mean I have to cry".
 - **"how it looks like"**: It's either fucking "what it looks like" OR "how it looks", NOT both at once.
 - **Using apostrophe for plural**, especially with acronyms, for example "VPN's" (WRONG) instead of "VPNs" (correct).
 - **Spelling**: beginner stuff like "its" vs "it's", "there" vs "their" etc.
 - **Countable vs uncountable** bitch, I double dare you to ever say shit like "less mistakes".
 - ...
-

entrepreneur

Entrepreneur

Entrepreneur is an individual practicing legal slavery and legal theft under capitalism; capitalists describe those actions by euphemisms such as "doing business". Successful entrepreneurs can also be seen as murderers as they consciously firstly hoard resources that poor people lack (including basic resources needed for living) and secondly cause and perpetuate situations such as the third world slavery where people die on a daily basis performing extremely difficult, dangerous and low paid work, so that the entrepreneur can buy his ass yet another private jet.

entropy

Entropy

Entropy is a quite cryptic, often misunderstood scientific term that may have different definitions depending on specific field and context, which can intuitively be interpreted as an amount of disorder, uncertainty or randomness. There are two main kinds of entropy: information entropy (information theory) and thermodynamic entropy (physics).

Information Entropy

Information entropy is a basic concept in information theory -- watch out, this kind of entropy is different from entropy in physics (which is described below). We use entropy to express an "amount of hidden information" in events, messages, codes etc. This can be used e.g. to design compression algorithms, help utilize bandwidths better etc.

Let's first define what information means in this context (note that the meaning of *information* here is kind of mathematical, not exactly equal to the meaning of *information* used in common speech). For a random event (such as a coin toss) with probability p the amount of information we get by observing it is

$$I(p) = \log_2(1/p) = -1 * \log_2(p)$$

The unit of information here is bit (note the base 2 of the logarithm -- other bases can be used too but then the units are called differently), in information theory also known as *shannon*. Let's see how the definition behaves: the less probable an event is, the more information its observation gives us (with 0, i.e. impossible event, theoretically giving infinite information), while probability 1 gives zero information (observing something we know will happen tells us literally nothing).

Now an **entropy of a random variable** X , which can take values $x_1, x_2, x_3, \dots, x_n$ with probabilities $q_1, q_2, q_3, \dots, q_n$ is defined as

$$H(x) = \sum(q_i * I_i) = \sum(q_i * \log_2(1/q_i))$$

How does entropy differ from information? Well, they are measured in the same units (bits), the difference is in the interpretation -- under the current context information is basically what we know, while entropy is what we don't know, the uncertainty. So entropy of a certain message (or rather of the probability distribution of possible messages to receive) says how much information will be gained by receiving it -- once we receive the message, the entropy kind of "turns into information", so the amount of information and entropy is actually the same. Perhaps the relationship is similar to that of energy and work in physics -- both are measured in the same units, energy is the potential for work and can be converted to it.

Entropy is greater if unpredictability ("randomness") is greater -- it is at its maximum if all possible values of the random variable are equally likely. For example entropy of a coin toss is 1 bit, given both outcomes are equally likely (if one outcome was more likely than the other, entropy would go down).

More predictable events have lower entropy -- for example English text has quite low entropy because it is pretty easy to predict missing letters from other letters (there is a lot of redundancy in human language). Thanks to this we can compress the text, e.g. using Huffman code -- compression reduces size, i.e. removes redundancy/correlation/predictability, and so increases entropy.

Example: consider a weather forecast for a specific area, day and hour -- our weather model predicts rain with 55% probability, cloudy with 30% probability and sunny with 15% probability. Once the specific day and hour comes, we will receive a message about the ACTUAL weather that there was in the area. What entropy does such message have? According to the formula above: $H = 0.55 * \log_2(1/0.55) + 0.3 * \log_2(1/0.3) + 0.15 * \log_2(1/0.25) \sim 1.3 \text{ bits}$. That is the entropy and amount of information such message gives us.

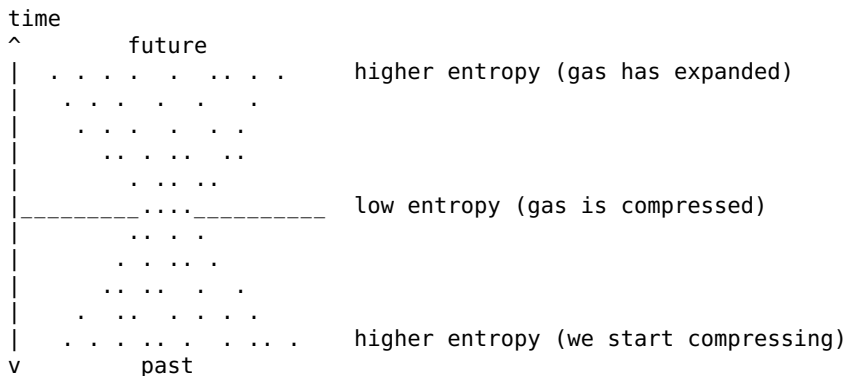
How is information entropy related to the physics entropy?

TODO

Physics Entropy

TODO

But WHY does entropy increase in time-forward direction? One may ask if laws on nature are time-symmetric, why is the forward direction of time special in that entropy increases in that direction? Just WHY is it so? Well, it is not so really, entropy simply increases in both time-forward and time-backward directions from a point of low entropy. Such point of low entropy may be e.g. the Big Bang since which entropy has been increasing in the time direction that's from the Big Bang towards us. Or the low entropy point may be a compressed gas; if we let such gas expand its entropy will increase to the future, but we may also look to the past in which the gas had high entropy before we compressed it, i.e. here entropy locally increases also towards the past. This is shown in the following image:



esolang

Esoteric Programming Language

So called esoteric programming languages (esolangs) are highly experimental and fun programming languages that employ bizarre and/or unconventional ideas. Popular languages of this kind include Brainfuck, Chef or Omgrofl.

There is a great wiki for esolangs, the Esolang Wiki (<https://esolangs.org>). If you want to behold esolangs in all their beauty, see [https://esolangs.org/wiki/Hello_world_program_in_esoteric_languages_\(nonalphabetic_and_A-M\)](https://esolangs.org/wiki/Hello_world_program_in_esoteric_languages_(nonalphabetic_and_A-M)). The Wiki is published under CC0!

Many esolangers seem to be code golfers, i.e. people who do various programming challenges while aiming for the shortest code which often requires a wise choice of language... or perhaps making a completely new language just for the job :) Codegolf stack exchange is therefore one place to see many esolangs in action.

Some notable ideas employed by esolangs are:

- Using images instead of text as source code (e.g. *Piet*).
- Doing nothing (e.g. *Nothing*).
- Being two or more dimensional (e.g. *Befunge* or *Hexagony*).
- Source code resembling cooking recipes (e.g. *Chef*).
- Trying to be as hard to use as possible.
- Trying to be as hard to compile as possible (e.g. *Befunge*).
- Adding randomness to program execution (e.g. *Entropy*), or working with randomness in other ways (e.g. XD has only one command, XD, which always translates to random Brainfuck command).
- Having no input/output (e.g. *Compute*).
- Obligation to beg the compiler to do its job (e.g. *INTERCAL*).
- Using only white characters in source code (e.g. *Whitespace*).
- Using only a single letter in source code (e.g. *Unary*).
- Using git repository structure as source code (e.g. *legit*).
- Source code resembling dramatic plays (e.g. *Shakespeare*, actual real-life plays were performed).

- Solely focus on golfing, i.e. writing the shortest possible programs (e.g. *GoldScript*)
- Using unicode characters (e.g. *UniCode*).
- Being infinitely many languages (e.g. *MetaGolfScript*, each one solves a specific program in 0 bytes).
- ...

Esolangs are great because:

- **They are fun** and have a cool community around them.
- **They are actually useful research in language design and food for thought**, even if most of the ideas aren't useful directly, esolangs really teach us about the borders and definitions of what languages are. And sometimes, by mistake, actual discoveries are made.
- **They are great exercise in programming** and design. Simple languages that are allowed to not be useful are potentially good for education as they let the programmer fully focus on a specific idea and its implementation.
- **They blend technology with art**, train creativity and thinking "outside the box".
- **They are a breath of fresh air** in the sometimes too serious area of technology. Hobbyist and non-commercial programming communities are always great to have.
- ...

A famous one-man organization related to esolangs is Cat's Eye run by Chris Pressey, currently reachable at <https://catseye.tc>.

History

INTERCAL, made in 1972 by Donald Woods and James Lyon, is considered the first esolang in history: its goal was specifically intended to be different from traditional languages and so for example a level of politeness was introduced -- if there weren't enough PLEASE labels in the source code, the compiler wouldn't compile the program.

In 1993 Brainfuck, probably the most famous esolang, was created.

In 2005 esolang wiki was started.

TODO

Specific Languages

The following is a list of some notable esoteric languages.

- **!@\$%^&*() +**: Source code looks like gibberish.
- **Brainfuck**: Extremely simple but hard to program in, arguably the most famous esolang with many forks.
- **Brainfork**: Brainfuck with added multithreading.
- **Befunge**: Two dimensional language that's extremely hard to compile.
- **Chef**: Source codes look like cooking recipes.
- **Entropy**: Adds randomness to programs, data in variables decay.
- **FALSE**: Aims for as small compiler as possible, inspired creation of Brainfuck and other esolangs, very minimalist.
- **Gravity**: Executing programs involves solving differential equations related to gravity, which is uncomputable.
- **INTERCAL**: Maybe the first esolang, includes such statements as PLEASE D0 which have to be present in order for the compilation to be successful.
- **Nothing**: Does nothing, guarantees zero bugs.
- **Compute**: Can compute any existing problem in arbitrarily short time, but has no output so the result cannot be printed.
- **Omgrofl**: Source code is composed of internet acronyms such as *lol*, *wtf*, *lmao* etc.
- **Pi**: Source code looks like the number pi, errors encode the program.
- **Piet**: Source codes are images.

- **Text**: Language that always prints its source code (it is not Turing complete). All ASCII files are programs in Text.
- **Polynomial**: Programs are polynomials whose zeros determine the commands.
- **Unary**: Source code uses only 1 character: 0. Each program is just a sequence of zeros of different length.
- **Velato**: Source codes are MIDI files.
- **Whitespace**: Source code uses only white characters (spaces, tabs and newlines) so it looks seemingly empty.
- **XENBLN**: Golfing language, hello world is just Å i.
- ...

{ There used to be an esolang webring, now only accessible through archive:
<https://web.archive.org/web/20110728084807/http://hub.webring.org/hub/esolang>. You can find nice links there. ~drummyfish }

See Also

- WPU (weird processing unit)
- conlang
- micronation

ethics

Ethics

Ethics is the study of morality. (For more see the article on morality.)

TODO?

everyone_does_it

Everyone Does It

"Everyone does it" is an argument quite often used by simps to justify their unjustifiable actions. It is often used alongside the "just doing my job" argument.

The argument has a valid use, however it is rarely used in the valid way. We humans, as well as other higher organisms, have evolved to mimic the behavior of others because such behavior is tried, others have tested such behavior for us (for example eating a certain plant that might potentially be poisonous) and have survived it, therefore it is likely also safe to do for us. So we have to realize that "everyone does it" is an **argument for safety, not for morality**. But people nowadays mostly use the argument as an excuse for their immoral behavior, i.e. something that's supposed to make bad things they do "not bad" because "if it was bad, others wouldn't be doing it". That's of course wrong, people do bad things and the argument "everyone does it" helps people do them, for example during the Nazi holocaust this excuse partially allowed some of the greatest atrocities in history. Nowadays during capitalism it is used to excuse taking part unethical practices, e.g. those of corporations.

So if you tell someone "You shouldn't do this because it's bad" and he replies "Well, everyone does it", he's really (usually) saying "I know it's bad but it's safe for me to do".

The effect is of course abused by politicians: once you get a certain number of people moving in a certain shared direction, others will follow just by the need to mimic others. Note that just creating an illusion (using the tricks of marketing) of "everyone doing something" is enough -- that's why you see 150 year old grandmas in ads using modern smartphones -- it's to force old people into thinking that other old people are using smartphones so they have to do it as well.

Another potentially valid use of the argument is in the meaning of "everyone does it so I am FORCED to do it as well". For example an employer could argue "I have to abuse my employees otherwise I'll lose the edge on

the market and will be defeated by those who continue to abuse their employees". This is very true but it seems like many people don't see or intend this meaning.

evil

Evil

Evil always wins in the end. But that's not a reason to join it.

TODO

As Richard Stallman says, **all evil does some good, which is never a reason to support it**. Just as any good always does a little bit of evil, the opposite also holds: for example Facebook, a corporation who accelerates and largely causes downfall of civilization and kills and tortures millions of people, may on occasion help do something good, for example help people communicate during an emergency, however it is no reason to support Facebook or to stop supporting its destruction.

exercises

Exercises

Here there should be a set of exercise problems for those wishing to pursue LRS in any way.

{ Hmmm, it's hard to figure out exactly what to put here. ~drummyfish }

Programming Projects

Here you will find suggestions for programming projects depending on your skill level.

Easy

- TODO

Medium

- TODO
-

Hard

- TODO
-

explicit

Explicit

Explicit is something that's directly expressed; it is the opposite of implicit.

f2p

Free To Play

Free to play (F2P) is a "business model" of predatory proprietary games that's based on the same idea as giving children free candy so that they get into your van so that you can rape them.

facebook

Facebook

"Facebook has no users, it only has users." --rms

TODO

faggot

Faggot

Faggot is a synonym for gay.

fail_ab

Type A/B Fail

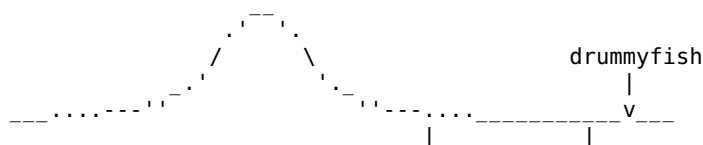
Type A and type B fails are two very common cases of failing to adhere to the LRS politics/philosophy by only a small margin. Most people don't come even close to LRS politically or by their life philosophy -- these are simply general failures. Then there are a few who ALMOST adhere to LRS politics and philosophy but fail in an important point, either by being/supporting pseudoleft (type A fail) or being/supporting right (type B fail). The typical cases are following (specific cases may not fully fit these, of course):

- **type A fail:** Is anticapitalist, anticonsumerist, may incline towards minimalism, supports free software and free culture, may even be a vegan, anarchist, C programmer etc., however falls into the trap of supporting pseudoleft, e.g. LGBT or feminism and things such as censorship ("moderation", COCs), "just violence and bullying" (violence against fascists, e.g. antifa), falls for memes such as "Rust is the new C".
- **type B fail:** Is against pseudoleft bullshit and propaganda such as political correctness, is a racial realist, highly supports suckless software, hacking and minimalism to achieve high freedom, usually also opposes corporations and state, however falls into the trap of being a fascist, easily accepts violence, believes in "natural selection/wild west as a basis of society", supports and engages in cryptocurrencies, believes in some form of capitalism and that the current form of it can be "fixed" ("anarcho" capitalism etc.)

Both types are furthermore prone to falling a victim to privacy obsession, productivity obsession, hero worshipping, use of violence, diseases such as distro hopping, tech consumerism and similar defects.

Type A/B fails are the "great filter" of the rare kind of people who show a great potential for adhering to LRS. It may be due to the modern western culture that forces a right-pseudoleft false dichotomy that even those showing a high degree of non-conformance eventually slip into the trap of being caught by one of the two poles. These two fails seem to be a manifestation of an individual's true motives of self interest which is culturally fueled with great force -- those individuals then try to not conform and support non-mainstream concepts like free culture or sucklessness, but eventually only with the goal of self interest. It seems to be extremely difficult to abandon this goal, much more than simply non-conforming. Maybe it's also the subconscious knowledge that adhering completely to LRS means an extreme loneliness; being type A/B fail means being a part of a minority, but still having a supportive community, not being completely alone.

However these kinds of people may also pose a hope: if we could educate them and "fix their failure", the LRS community could grow rapidly. If realized, this step could even be seen as the main contribution of LRS -- uniting the misguided rightists and pseudoleftists by pointing out errors in their philosophies (errors that may largely be intentionally forced by the system anyway exactly to create the hostility between the non-conforming, as a means of protecting the system).



fantasy_console

Fantasy Console

Fantasy console, also fantasy computer, is a software platform intended mainly for creating and playing simple games, which imitates parameters, simplicity and look and feel of classic retro consoles such as GameBoy. These consoles are called *fantasy* because they are not emulators of already existing hardware consoles but rather "dreamed up" platforms, virtual machines made purely in software with artificially added restrictions that a real hardware console might have. These restrictions limit for example the resolution and color depth of the display, number of buttons and sometimes also computational resources.

The motivation behind creating fantasy consoles is normally twofold: firstly the enjoyment of retro games and retro programming, and secondly the immense advantages of simplicity. It is much faster and easier to create a simple game than a full fledged PC game, this attracts many programmers, simple programming is also more enjoyable (fewer bugs and headaches) and simple games have many nice properties such as small size (playability over web), easy embedding or enabling emulator-like features.

Fantasy consoles usually include some kind of simple IDE; a typical mainstream fantasy console both runs and is programmed in a web browser so as to be accessible to normies. They also use some kind of easy scripting language for game programming, e.g. Lua. Even though the games are simple, the code of such a mainstream console is normally bloat, i.e. we are talking about pseudominimalism. Nevertheless some consoles, such as SAF, are truly suckless, free and highly portable (it's not a coincidence that SAF is an official LRS project).

Some fantasy consoles may blend with open consoles, e.g. by starting as a virtual console that's later implemented in real hardware.

Notable Fantasy Consoles

The following are a few notable fantasy consoles.

name	year	license	game lang.	specs.	comment
<u>CToy</u>	2016	<u>zlib</u>	<u>C</u>	128x128	<u>suckless</u>
<u>IBNIZ</u>	2011	<u>zlib</u>	own	256x256 32b, 4M RAM	for demos, by <u>Viznut</u>
<u>LIKO-12</u>	2016	<u>MIT</u>	<u>Lua</u>	192x128	
<u>MEG4</u>	2023	<u>GPL</u>	C, Lua, ...	320x200 8b, 576K RAM	
<u>microw8</u>	2021	<u>unlicense</u>	webassembly	320x240 8b, 256K RAM	
<u>PICO-8</u>	2015	<u>propr.</u>	<u>Lua</u>	128x128 4b	likely most famous
PixelVision8	2020	<u>MS-PL</u> (FOSS)	<u>Lua</u>	256x240	written in C#
Pyxel	2018	<u>MIT</u>	<u>Python</u>	256x256 4b	
<u>SAF</u>	2021	<u>CC0</u>	<u>C</u>	64x64 8b	<u>LRS</u> , suckless
<u>TIC-80</u>	2016	<u>MIT</u>	Lua, JS, ...	240x136 4b	paid "pro" version
<u>Vircon32</u>	???		C, assembly	640x360, 16M RAM	aims to be implementable in HW
<u>uxn</u>	2021	<u>MIT</u>	<u>Tal</u>		very minimal

Apart from these there are many more (MicroW8, PX8, WASM-4, ZZT, ...), you can find lists such as <https://paladin-t.github.io/fantasy/index>. There even exists a Brainfuck fantasy console, called BrainFuckConsole74.

See Also

- open console
- handheld

- [virtual machine](#)
 - [IBNIZ](#)
 - [ISA](#)
 - [SAF](#)
 - [DOS](#)
-

faq

Frequently Asked Questions

Not to be confused with [fuck](#) or [frequently questioned answers](#).

{ answers by ~[drummyfish](#) }

Is this a joke? Are you [trolling](#)?

No. Jokes are [here](#).

What the fuck?

See [WTF](#).

Is there [RSS](#) feed?

I dunno, I don't do anything like that now, remember this isn't a blog, tho you can probably get RSS by going to this wiki's git repo and getting the commit feed.

How does LRS differ from [suckless](#), [KISS](#), [free software](#) and similar types of software?

Sometimes these sets may greatly overlap and LRS is at times just a slightly different angle of looking at the same things, but in short LRS cherry-picks the best of other things and is much greater in scope (it focuses on the big picture of whole society). I have invented LRS as my own take on suckless software and then modified it a bit and expanded its scope to encompass not just technology but the whole society -- as I cannot speak on behalf of the whole suckless community (and sometimes disagree with them a lot), I have created my own "fork" and simply set my own definitions without worrying about misinterpreting, misquoting or contradicting someone else. LRS advocates very similar technology to that advocated by suckless, but it furthermore has its specific ideas and areas of focus. The main point is that **LRS is derived from an unconditional love of all life** rather than some shallow idea such as "[productivity](#)". In practice this leads to such things as a high stress put on [public domain](#) and legal safety, [altruism](#), selflessness, anti-[capitalism](#), accepting [games](#) as desirable type of software, **NOT subscribing to the [productivity cult](#), rejecting [security](#), [privacy](#), [cryptocurrencies](#) etc.** While suckless is apolitical and its scope is mostly limited to software and its use for "getting job done", LRS speaks not just about technology but about the whole society -- there are two main parts of LRS: [less retarded software](#) and [less retarded society](#).

One way to see LRS is as a philosophy that takes only the [good](#) out of existing philosophies/movements/ideologies/etc. and adds them to a single unique [idealist](#) mix, without including [cancer](#), [bullshit](#), errors, propaganda and other negative phenomena plaguing basically all existing philosophies/movements/ideologies/etc.

Why this obsession with extreme [simplicity](#)? Is it because you're too stupid to understand complex stuff?

I used to be the mainstream, complexity embracing programmer. I am in no way saying I'm a genius but I've put a lot of energy into studying computer science full time for many years so I believe I can say I have some understanding of the "complex" stuff. I speak from own experience and also on behalf of others who shared their experience with me that the appreciation of simplicity and realization of its necessity comes after many years of dealing with the complex and deep insight into the field and into the complex connections of that field to society.

You may ask: well then but why it's just you and a few weirdos who see this, why don't most good programmers share your opinions? Because they need to make living or because they simply WANT to make a lot of money and so they do what the system wants them to do. Education in technology (and generally just being exposed to corporate propaganda since birth) is kind of a trap: it teaches you to embrace complexity and when you realize it's not a good thing, it is too late, you already need to pay your student loan, your rent, your mortgage, and the only thing they want you to do is to keep this complexity cult rolling. So people just do what they need to do and many of them just psychologically make themselves believe something they subconsciously know isn't right because that makes their everyday life easier to live. "Everyone does it so it can't be bad, better not even bother thinking about it too much". It's difficult doing something every day that you think is wrong, so you make yourself believe it's right.

It's not that we can't understand the complex. It is that the simpler things we deal with, the more powerful things we can create out of them as the overhead of the accumulated complexity isn't burdening us so much.

Simplicity is crucial not only for the quality of technology, i.e. for example its safety and efficiency, but also for its freedom. The more complex technology becomes, the fewer people can control it. If technology is to serve all people, it has to be simple enough so that as many people as possible can understand it, maintain it, fix it, customize it, improve it. It's not just about being able to understand a complex program, it's also about how much time and energy it takes because time is a price not everyone can afford, even if they have the knowledge of programming. Even if you yourself cannot program, if you are using a simple program and it breaks, you can easily find someone with a basic knowledge of programming who can fix it, unlike with a very complex program whose fix will require a corporation.

Going for the simple technology doesn't necessarily have to mean we have to give up the "nice things" such as computer games or 3D graphics. Many things, such as responsiveness and customizability of programs, would improve. Even if the results won't be so shiny, we can recreate much of what we are used to in a much simpler way. You may now ask: why don't companies do things simply if they can? Because complexity benefits them in creating de facto monopolies, as mentioned above, by reducing the number of people who can tinker with their creations. And also because capitalism pushes towards making things quickly rather than well -- and yes, even non commercial "FOSS" programs are pushed towards this, they still compete and imitate the commercial programs. Already now you can see how technology and society are intertwined in complex ways that all need to be understood before one comes to realize the necessity of simplicity.

How would your ideal society work? Isn't it utopia?

See the article on [less retarded society](#), it contains a detailed FAQ especially on that.

So is there a whole community here or what? Do you have forums or anything?

Well, firstly LRS by [anarchist](#) principles avoids establishing a centralized community with things such as platforms, senior members, moderators, bureaucracy, codified rules etc. Every nice thing that went this way turned to shit once it became more popular and grew, corruption always appears and prevails (see [Wikipedia](#), [GNU/FSF](#), [Linux](#) etc.). So we rather aim for a community of decentralized, loosely associated individuals and small entities with greatly overlapping sets of values rather than any organizations etc. So rather think something like many people with their own websites referring and linking to each other.

At the moment this wiki/movement/ideology/etc. is basically me ([drummyfish](#)). There are a handful of people who told me they like this wiki a lot, share many of my views and values and who I am in regular contact with mostly over email. Then there are quite a few people who like this to various degree and contact me from time to time, many just write me a one time email, suggest something, thank me, suggest I kill myself etc., from which I estimate there is a non-negligible number of lurkers (given that only 1 in relatively many readers will actually write me an email).

A forum, mailing list or something would possibly be nice, but so far no one made any and I would be hesitant to call it an "official" LRS forum anyway, exactly to prevent growing into some kind of corrupt "democratic internet community" of which there are too many nowadays. I am also pretty anxious of collaborating with someone, making hard connections to other project etc. So if you'd like to make a LRS-focused forum (or anything similar), it would be best if you just make it your own thing -- of course I'll be very glad if you refer to my stuff etc., I just won't be able to keep up with another project, I will only be able

to be a regular user of your forum.

Why the angry and aggressive tone, can't you write in a nicer way, especially when you advocate love etc.?

More than once I've been told that someone was initially afraid to talk to me because I write in this "aggressive/angry" way. I am sorry, this way of writing has its reasons that I established here and it's what works for me, I really don't intend to stress you out -- firstly this is how I internally think because yes, I am very frustrated, and **I want this wiki to capture my internal thoughts in a very unfiltered way** (note that having bad thoughts doesn't mean one has to act on them), also I find this way flows the best for me and allows me to communicate what I feel and think the best, and it also gives this wiki kind of its own "personality" and prevents it from taking on a super serious tone -- informality and fun are quite important for a healthy view of the world. I am really tired of all the overly correct and polite articles on the Internet (I have tried to write in different ways but it always stands in the way). Sometimes I get mood swings and regret writing something, other times I bash myself for being too soft -- but I don't want to delete stuff too much, this will all be reflected on the wiki. Underneath all this still lies the important message of **love and peace**. I guess I also want to show that to be truly loving you don't have to change your personality or censor your thoughts. In normal conversations I try as much as possible to be nice, I actually almost never get aggressive towards others, if I get very stressed I usually just leave or in more extreme cases target hate towards myself, but I really try to not hurt anyone (people also told me they were quite surprised that I was kind of "nice" when they actually talked to me). I actually have a lot of trouble in real life for not defending myself, people often abuse it and I let them, I don't fight back, I don't believe in revenge or violence and in addition I have social anxiety. Please don't be afraid to contact me <3

Why the name "less retarded"? If you say you're serious about this, why not a more serious name?

I don't know, this is not so easy to answer because I came up with the name back when the project was smaller in scope and I didn't think about a name too hard: this name was playful, catchy, politically incorrect (keeping SJWs away) and had a kind of reference to *suckless*, potentially attracting attention of suckless fans. It also has the nice property of being unique, with low probability of name collision with some other existing project, as not many people will want to have the word "retarded" in the name. Overall the name captures the spirit of the philosophy and is very general, allowing it to be applied to new areas without being limited to certain means etc.

Now that the project has evolved a bit the name actually seems to have been a great choice and I'm pretty happy about it, not just for the above mentioned reasons but also because it is NOT some generic boring name that politicians, PR people and other tryhard populists would come up with. In a way it's trying to stimulate thought and make you think (if only by making you ask WHY anyone would choose such a name). Yes, in a way it's a small protest and showing we stay away from the rotten mainstream, but it's definitely NOT an attempt at catching attention at any cost or trying to look like cool rebels -- such mentality goes against our basic principles. Perhaps the greatest reason for the name is to serve as a test -- truth should prevail no matter what name it is given and we try to test and prove this, or rather maybe prevent succeeding for wrong reasons -- we are not interested in success (which is what mere politicians do); if our ideas are to become accepted, they have to be accepted for the right reasons. And if you refuse to accept truth because you don't like its name, you are retarded and by own ignorance doom yourself to live in a shit society with shit technology.

Who writes this wiki? Can I contribute?

You can only contribute to this wiki if you're a straight white male. Just kidding, you can't contribute even if you're a straight white male :)

At the moment it's just me, drummyfish. This started as a collaborative wiki name *based wiki* but after some disagreements I forked it (everything was practically written by me at that point) and made it my own wiki where I don't have to make any compromises or respect anyone else's opinions. I'm not opposed to the idea of collaboration in some situations but I bet we disagree on something in which case I probably don't want to search for a compromise. I also resist allowing contributions because with multiple authors the chance of legal complications grows, even if the work is under a free license or waiver (refer to e.g. the situation where

some Linux developers were threatening to withdraw their code contribution license). But you can totally fork this wiki, it's public domain.

If you want to contribute to the cause, just create your own website, spread the ideas you liked here -- you may or may not refer to LRS, everything's up to you. Start creating software with LRS philosophy if you can -- together we can help evolve and spread our ideas in a decentralized way, without me or anyone else being an authority, a potential censor. That's the best way forward I think.

Why is it called a wiki when it's written just by one guy? Is it to deceive people into thinking there's a whole movement rather than just one weirdo?

Yes.

No, of course not you dumbbo. There is no intention of deception, this project started as a collaborative wiki with multiple contributors, named *Based Wiki*, however I (drummyfish) forked my contributions (most of the original Wiki) into my own Wiki and renamed it to *Less Retarded Wiki* because I didn't like the direction of the original wiki. At that point I was still allowing and looking for more contributors, but somehow none of the original people came to contribute and meanwhile I've expanded my LRS Wiki to the point at which I decided it's simply a snapshot of my own views and so I decided to keep it my own project and kept the name that I established, the *LRS Wiki*. Even though at the moment it's missing the main feature of a wiki, i.e. collaboration of multiple people, it is still a project that most people would likely call a "wiki" naturally (even if only a personal one) due to having all the other features of wikis (separate articles linked via hypertext, non-linear structure etc.) and simply looking like a wiki -- nowadays there are many wikis that are mostly written by a single man (see e.g. small fandom wikis) and people still call them wikis because culturally the term has simply taken a wider meaning, people don't expect a wiki to absolutely necessarily be collaborative and so there is no deception. Additionally I am still open to the idea to possibly allowing contributions, so I'm simply keeping this a wiki, the wiki is in a sense waiting for a larger community to come. Finally the ideas I present here are not just mine but really do reflect existing movements/philosophies with significant numbers of supporters (suckless, free software, ...).

Why did you make this wiki? What is its purpose?

There are many reasons, it serves multiple purposes which also change a bit over time -- the "net good" arising from the existence of this wiki seems to be quite highly positive, so it keeps existing and at least for now flourishes. Anyway here are some reasons for why this wiki exists:

- At the beginning this was mostly a silly fun and a way of communicating among a few people, though soon it became just my own project.
- It really tries to explore -- and also show -- how technology could be done well. By writing this wiki I invent new terms, categorize them, connect concepts together, find useful ideas buried in the history of technology. This wiki can become useful if there is a turnaround, e.g. after the collapse. It also tries to SHOW that many things can be done simply, many articles contain short code snippets that do very useful work in very few lines of code. Many project like libraries and games have been done just by myself and though they many not be perfect, just imagine if some real genius, of a small team of people, tried to do things the way I do them -- we would see miracles happen. Many articles link to cool minimalist programs and projects that demonstrate the principles.
- Hopefully it's going to be a useful resource for people who want to e.g. start programming in C, if only as a repository of code snippets for example. By this I hope to help bring more people to the world of good technology. It's also going to be a big book that's completely CC0, there is never enough of these.
- Similarly I hope to spark some critical thinking and non-mainstream midsets such as those of altruism, minimalism, nonviolence and selflessness. There are so few resources that do this, everything is overshadowed by the huge circlejerk avalanche of modern brainwashing of competitiveness, secrecy, self interest, basically just pure evil. I really feel that if I don't say certain things, no one else will.
- I hope to perhaps inspire others to make something similar, be it a wiki in similar style or any other kind of art, shared selflessly in the public domain, trying to lead some moral example etc.
- It's an experiment in creating what for now seems to be a unique kind of work, a highly uncensored "brain dump" of a social outcast which combines a serious encyclopedia, half serious shitposting, repository of various data, social comments and so on. It's also being published continuously, with

TODOs and WIPs, errors and mistakes, there are no "stable versions", just a kind of "as is" stream of data, take whatever you will out of it.

- The wiki tries to document contemporary society, it may be useful for future historians.
- It helps me cope, vent my emotion.
- It immensely helps me explain my worldviews without wasting my time and having to talk to people too much. Before this wiki whenever someone asked me "why do you think X?" over email, I had to spend an hour formulating a reply, then I would send it, the guy would be like "hmm it's cool but I disagree" -- so that was an hour of my life lost. People kept asking me the same things over and over, I used to have the exact same hour long conversations with many people, so now I just write my reasonings here and point to my articles without having to suffer retarded arguments over and over. Really it made me much less suicidal.
- They said I can't write this stuff on their site and that I should go make my own site, so I did. They banned me so I made myself more free. Maybe also a nice example for others.
- I just love making it, it makes me relax like nothing else. I write this wiki in between making my software projects which I love too but which require a bit more effort. A few times in my life I wanted to perhaps be a teacher as I like education and explaining things but at the same time I CANNOT STAND FUCKING PEOPLE -- this wiki makes me able to do what I like without having to suffer what I hate.
- It satisfied my obsessive needs for making lists, cheatsheets and so on. Also helps me keep my notes, ideas etc.
- It makes me (and hopefully others) learn about and research quite obscure things, I have personally discovered so many new amazing things while writing this wiki. Browsing the Internet is sometimes not enough, writing about something is what makes you go deeper.
- I hope to contribute to the downfall of capitalism, though TBH I don't believe it can happen at this point.
- This also helped me find some cool online friends. If you go searching for people like you, you won't find them, but if you put your ideas publicly on the display, people with similar ideas will reach out to you themselves.
- It's a work I can read myself knowing I won't find things that make me more suicidal like the word "person" and stupid grammar mistakes such as using apostrophes for plurals, using "it's" instead of "its", sentences containing "the reason is because" or "just because ... doesn't mean" etc.
- I like wikis, I like editing the pages, but I hate wikis with other people on it.
- Probably other reasons I couldn't recall right now :-)
- ...

Why is this rather not a blog?

Because blogs suck, they are based on the idea of content consumerism and subscribers following celebrities just like on youtube or facebook, blog posts are hasted, ugly and become obsolete in a week, this wiki is trying to create a reference work that can be polished and will last some time.

Since it is public domain, can I take this wiki and do anything with it? Even something you don't like, like sell it or rewrite it in a different way?

Yes, you can do anything... well, anything that's not otherwise illegal like falsely claiming authorship (copyright) of the original text. This is not because I care about being credited, I don't (you DON'T have to give me any credit), but because I care about this wiki not being owned by anyone. You can however claim copyright to anything you add to the wiki if you fork it, as that's your original creation.

Why not keep politics out of this Wiki and make it purely about technology?

Firstly technological progress is secondary to the primary type of progress in society: the social progress. The goal of our civilization is to provide good conditions for life -- this is social progress and mankind's main goal. Technological progress only serves to achieve this, so technological progress follows from the goals of social progress. So, to define technology we have to first know what it should help achieve in society. And for that we need to talk politics.

Secondly examining any existing subject in depth requires also understanding its context anyway. Politics and technology nowadays are very much intertwined and the politics of a society ultimately significantly

affects what its technology looks like (capitalist SW, censorship, bloat, spyware, DRM, ...), what goals it serves (consumerism, productivity, control, war, peace, ...) and how it is developed (COCs, free software, ...), so studying technology ultimately requires understanding politics around it. I hate arguing about politics, sometimes it literally make me suicidal, but it is inevitable, we have to specify real-life goals clearly if we're to create good technology. Political goals guide us in making important design decisions about features, tradeoffs and other attributes of technology.

Thirdly society and computer programs are in many ways similar and we naturally see analogies between both the problems and the solutions.

Of course you can fork this wiki and try to remove politics from it, but I think it won't be possible to just keep the technology part alone so that it would still make sense, most things will be left without justification and explanation.

What is the political direction of LRS then?

In three words basically anarcho pacifist communism, however the word culture may be more appropriate than "politics" here as we aim for removing traditional systems of government based on power and enforcing complex laws, there shall be no politicians in today's sense in our society. For more details see the article about LRS itself.

Why do you blame everything on capitalism when most of the issues you talk about, like propaganda, surveillance, exploitation of the poor and general abuse of power, appeared also under practically any other systems we've seen in history?

This is a good point, we talk about capitalism simply because it is the system of today's world and an immediate threat that needs to be addressed, however we always try to stress that the root issue lies deeper: it is competition that we see as causing all major evil. Competition between people is what always caused the main issues of a society, no matter whether the system at the time was called capitalism, feudalism or pseudosocialism. While historically competition and conflict between people was mostly forced by the nature, nowadays we've conquered technology to a degree at which we could practically eliminate competition, however we choose to artificially preserve it via capitalism, the glorification of competition, and we see this as an extremely wrong direction, hence we put stress on opposing capitalism, i.e. artificial prolonging of competition.

Why are you calling everything "capitalism" when clearly you can't generalize so much, capitalism needs mass production, things like free trade, corporatism, libertarianism etc. are different things than capitalism, no?

Nah, it's all essentially the same, as long as it's based on competition it's just a different stage of capitalism at best, all these things are based on the fundamentally flawed idea of letting humans compete -- this will always lead to things like trade, money, people forming bigger groups, companies, cartels and eventually corporations etcetc. Only those who believe capitalism can somehow be fixed or made manageable find it important to distinguish different types of it, we just oppose it all, so we don't bother to distinguish between different flavors of shit too much, that would be just unnecessary and distracting.

How is this different from Wikipedia?

In many ways. Our wiki is better e.g. by being more free (completely public domain, no fair use proprietary images etc.), less bloated, better accessible, not infected by pseudoleftist fascism and censorship (we only censor absolutely necessary things, e.g. copyrighted things or things that would immediately put us in jail, though we still say many things that may get us in jail), we have articles that are better readable etc.

WTF I am offended, is this a nazi site? Are you racist/Xphobic? Do you love Hitler?!?!?

We're not fascists, we're in fact the exact opposite: our aim is to create technology that benefits everyone equally without any discrimination. I (drummyfish) am personally a pacifist anarchist, I love all living beings and believe in absolute social equality of all life forms. We invite and welcome everyone here, be it gays,

communists, rightists, trannies, pedophiles or murderers, we love everyone equally, even you and Hitler.

Note that the fact that we love someone (e.g. Hitler) does NOT mean we embrace his ideas (e.g. Nazism) or even that we e.g. like the way he looks. You may hear us say someone is a stupid ugly fascist, but even such individuals are living beings we love.

What we do NOT engage in is political correctness, censorship, offended culture, identity politics and pseudoleftism. We do NOT support fascist groups such as feminists and LGBT and we will NOT practice bullying and codes of conducts. We do not pretend there aren't any differences between people and we will make jokes that make you feel offended.

OK then why are there swastikas on your main page?

Swastika is an old religious symbol used e.g. in Buddhism, it is a sign of good fortune and protection from evil. Nazis later used swastika but it doesn't give them monopoly over it, and also they used the other version, with arms going clockwise (AND usually also turned additional 45 degrees), so you can only confuse it with Nazi swastika if you are retarded.

Why do you use the nigger-w so much?

To counter its censorship, we mustn't be afraid of words. The more they censor something, the more I am going to uncensor it. They have to learn that the only way to make me not say that word so often is to stop censoring it, so to their action of censorship I produce a reaction they dislike. That's basically how you train a dog. (Please don't ask who "they" are, it's pretty obvious).

It also has the nice side effect of making this less likely to be used by corporations and SJWs.

How can you say you love all living beings and use offensive language at the same time?

The culture of being offended is bullshit, it is a pseudoleftist (fascist) invention that serves as a weapon to justify censorship, canceling and bullying of people. Since I love all people, I don't support any weapons against anyone (not even against people I dislike or disagree with). People are offended by language because they're taught to be offended by it by the propaganda, I am helping them unlearn it. Political correctness is one of the most retarded and toxic things to ever have been invented. Learn to separate being evil and being angry.

But don't you think someone can misinterpret your politically incorrect speech for inciting violence, fascism etc.?

Yes, idiots can misinterpret anything, that's not my fault, truth has to be revealed -- if someone wrongly interprets a message or uses truth for doing harm, then he is to be blamed, not me. If you think I am to be blamed because I should take into account the possibility of my words leading to someone causing harm, then I am also not to blame because my politically incorrect speech is itself a reaction to the pseudoleftist insanity, so ultimately the pseudoleft is to be blamed because they should take into account that by creating hell on Earth someone will start doing what I do which may lead to someone else doing something bad. Either way I, standing in the middle of the chain, am not responsible, logically you cannot put the blame on me, you have to choose whether the responsibility lies at the end of the chain (correct) or at its beginning (wrong but still doesn't serve you).

NOTE: blame here is used in moral sense, not in legal sense or in sense of implying punishment. I simply argue that a scientist shouldn't avoid creating inventions that may be abused. Would you blame Alan Turing for Apple's atrocities?

But how can you so pretentiously preach "absolute love" and then say you hate capitalists, fascists, bloat etc.?

OK, firstly we do NOT love *everything*, we do NOT advocate against hate itself, only against hate of living beings (note we say we love *everyone*, not *everything*). Hating other things than living beings, such as some bad ideas or malicious objects, is totally acceptable, there's no problem with it. We in fact think hate of some

concepts is necessary for finding better ways.

Now when it comes to "*hating*" people, there's an important distinction to be stressed: we never hate a living being as such, we may only hate their properties. So when we say we *hate* someone, it's merely a matter of language convenience -- saying we *hate* someone never means we hate a man as such, but only some thing about that man, for example his opinions, his work, actions, behavior or even appearance. I can hear you ask: what's the difference? The difference is we'll never try to eliminate a living being or cause it suffering because we love it, we may only try to change, in non-violent ways, their attributes we find wrong (which we *hate*): for example we may try to educate the person, point out errors in his arguments, give him advice, and if that doesn't work we may simply choose to avoid his presence. But we will never target hate against him. Hate the sin, not the sinner.

And yeah, of course sometimes we make jokes and sarcastic comments, it is relied on your ability to recognize those yourself. We see it as retarded and a great insult to intelligence to put disclaimers on jokes, that's really the worst thing you can do to a joke.

So you really "love" everyone, even dicks like Trump, school shooters, instagram manipulators etc.?

Yes, but it may need an elaboration. There are many different kinds of love: love of a sexual partner, love of a parent, love of a pet, love of a hobby, love of nature etc. Obviously we can't love everyone with the same kind of love we have e.g. for our life partner, that's impossible if we've actually never even seen most people who live on this planet. The love we are talking about -- our universal love of everyone -- is an unconditional love of life itself. Being alive is a miracle, it's beautiful, and as living beings we feel a sense of connection with all other living beings in this universe who were for some reason chosen to experience this rare miracle as well -- we know what it feels like to live and we know other living beings experience this special, mysterious privilege too, though for a limited time. This is the most basic kind of love, an empathy, the happiness of seeing someone else live. It is sacred, there's nothing more pure in this universe than feeling this empathy, it works without language, without science, without explanation. While not all living beings are capable of this love (a virus probably won't feel any empathy), we believe all humans have this love in them, even if it's being suppressed by their environment that often forces them compete, hate, even kill. Our goal is to awaken this love in everyone as we believe it's the only way to achieve a truly happy coexistence of us, living beings.

Why do you say you don't fight when clearly you are fighting the whole world here.

That's what you would call it -- am I literally physically punching the world in the face or something? Rather ask yourself why you choose to compare things like education and advocating love to a war. I am just revealing truth, educating, sometimes expressing frustration, anger and emotion, sometimes joking, without political correctness. Sometimes names greatly matter and LRS voluntarily chooses to never view its endeavor as being comparable to a fight, like capitalists like to do -- in many situations this literally IS only about using a different word -- seemingly something of small to no importance -- however the word sets a mood and establishes a mindset, when we go far enough, it will start to matter that we have chosen to not see ourselves as fighter, for example we **will NEVER advocate any violence or call for anyone's death**, unlike for example LGBT, feminists and Antifa, the "fighters".

I dislike this wiki, our teacher taught us that global variables are bad and that OOP is good.

This is not a question you dummy. Have you even read the title of this page? Anyway, your teacher is stupid, he is, very likely unknowingly, just spreading the capitalist propaganda. He probably believes what he's saying but he's wrong.

Lol you've got this fact wrong and you misunderstand this and this topic, you've got bugs in code, your writing sucks etc. How dare you write about things you have no clue about?

see also gatekeeping

I want a public domain encyclopedia that also includes topics of new technology and correct views without censorship, and also one which doesn't literally make me want to kill myself due to inserted propaganda of

evil. Since this supposedly modern society failed to produce even a single such encyclopedia and since every idiot on this planet wants to keep his copyright on everything he writes and/or wants to censor what he creates, I am forced to write the encyclopedia myself from scratch, even for the price of making mistakes. No, US public domain doesn't count as world wide public domain. Even without copyright there are still so called moral rights etc. Blame this society for not allowing even a tiny bit of information to slip into public domain. Writing my own encyclopedia is literally the best I can do in the situation I am in. Nothing is perfect, I still believe this can be helpful to someone. You shouldn't take facts from a random website for granted. I have to do my own research on everything, even on thing I have basically no clue about -- I know there are people a million times more knowledgeable on many subjects I write about, but I simply cannot believe them, today's society forces experts to lie, so rather than taking a lie from an expert I am forced to take a honest view of a layman (me). This society is also quite fucked up in forcing the idea that you can't make basic observations unless you have 10 PhDs, you are only allowed to spread the gospel officially approved of by decorated soyentists -- I just ignore this. If you wanna help me correct errors, email me.

Why is this shit so poorly written and inconsistent (typos, incorrect comma placement, inconsistent acronym case, weird grammar etc.)?

Mainly for these reasons:

- On purpose, this is kinda informal text and doesn't wanna get all serious, I want the text to flow in the same way in which informal Internet chat does (both during reading and writing). Though I personally hate such mistakes (and the laziness of authors to proofread) in nice/formal texts, I purposefully don't aim for creating such type of text here because that would firstly make it something else than I want, and secondly it would be extra work and effort that I think is better spent on actually communicating ideas (as opposed to to communicating them perfectly). Afterall a "nicer" version of the wiki can be made by anyone as it is completely public domain.
- This wiki is rather a dirty tool, a temporary vehicle on a way to a better technology and society, not a tidy encyclopedia. It prefers communicating ideas to achieving perfect form, focusing too much on nice form would result in having to leave out a lot of important information due to "quality assurance".
- I don't care sometimes.
- I'm lazy.
- I'm shit and just make mistakes.

How can you use CC0 if you, as anarchists, reject laws and intellectual property?

We use it to **remove** law from our project, it's kind of like using a weapon to destroy itself. Using a license such as GFDL would mean we're keeping our copyright and are willing to execute enforcement of intellectual property laws, however using a CC0 waiver means we GIVE UP all lawful exclusive rights that have been forced on us. This has no negative effects: if law applies, then we use it to remove itself, and if it doesn't, then nothing happens. To those that acknowledge the reality of the fact that adapting proprietary information can lead to being bullied by the state we give a guarantee this won't happen, and others simply don't have to care.

A simple analogy is this: a law is so fucked up nowadays that it forces us to point a gun at anyone by default when we create something. It's as if they literally put a gun in our hand and force point it at someone. We decide to drop that weapon, not merely promise to not shoot.

What software does this wiki use?

Git, the articles are written in markdown and converted to HTML with a simple script.

I don't want my name associated with this, can you remove a reference to myself or my software from your wiki?

No.

How can you think you are the smartest man in universe? How can you say your opinions are facts? Isn't it egoistic?

I don't think I am the smartest at all -- I never said that, except in the article about IQ. In fact I am highly dumb, I just have a gift of being completely immune to propaganda and seeing the world clearly, and I happen to be in circumstances under which I can do what others can't; for example as I have no friends and no one likes me, I can write and create freely, without self censorship due to fear of losing my job, offending my friends etc. I can write close to what is the absolute truth thanks to all this. I am also super autistic in that I enjoy just thinking 24/7 about programming and stuff instead of thinking about money and watching ads, which compensates for my dumbness a bit.

How do I know my opinions are facts? Experience. How do we discover facts? There is never a 100% certainty of anything, even of mathematical proofs, we may only ever have a great statistical confidence and beliefs so strong we call them facts. Just as by walking 1000 times against a wall you learn you won't walk through, I have over the decades learned I am correct in what I say and that everyone else is simply a monkey incapable of thinking. I used to be the kind of guy "open to discussion and opinions of others", I was giving this approach a chance over and over for about 30 years, I had more than enough patience, but it didn't work, the world has failed. People are absolutely stupid, you can physically show them something, give them tons of evidence and proofs, they won't believe what is literally in front of their eyes -- no, not even intellectuals, people in universities etc. Talking to others and listening to them is a complete waste of life, it's like trying to talk to potatoes or rocks, I might just as well be punching air all day or trying to eat dirt. The best I found I can do now is kind of talk to myself here, record my brain dump in hopes someone will once understand. I really don't know what else to do.

There is nothing egoistic about being special in something, everyone has a talent for something, egoism is about being fascist, preoccupied with oneself and focusing on self benefit, achieving fame, recognition etc., which I try my best to never do. Maybe I slip sometimes as an imperfect man I am, I make mistakes, I do stupid stuff, but I honestly just want the good of everyone without putting myself in the front.

Don't you think people will leave your website when they read in the article they are being called retards?

main article: butthurt

Only idiots will, those are people who are beyond saving anyway, so in reality I'm just saving them time. For me it works like this (it should also work like this for you, else you're faulty): if I read somewhere a thing X is retarded, along with a good explanation as to why, and I know I'm doing X, I really say to myself "hmm, I'd like to be less retarded, I should probably stop doing that", and then I try -- that's how I unretard myself more and more. That's how it works for people who have a good, essentially non-retarded mindset (even if they still do many retarded things; remember, only God can be perfect). Truth is a truth, if something is stupid it's stated to be stupid, being diplomatic is harmful and idiotic.

Are you the only one in the world who is not affected by propaganda?

It definitely seems so.

How does it feel to be the only one on this planet to see the undistorted truth of reality?

Pretty lonely and depressing.

Are you a crank?

Depending on exact definition the answer is either "no" or "yes and it's a good thing".

Are you retarded?

:(Maybe, but even stupid people can sometimes have smart ideas.

Fascism

Fascist (from Latin *fasces*, "bundle", "group", ancient symbol of power to punish) groups are subgroups of society that strongly pursue self interest on the detriment of others (those who are not part of said group). Fascism is a rightist, competitive tendency, very much connected to identity politics (being or rather feeling to be part of some group, e.g. nation, sex, race etc.); fascists aim to make themselves as strong, as powerful and as rich as possible, i.e. to weaken and possibly eliminate competing groups, to have power over them, enslave them and to seize their resources. The means of their operation are almost exclusively evil, including violence, bullying, wars, propaganda, eye for an eye, slavery etc.

A few examples of fascist groups are corporations, nations, NSDAP (Nazis), LGBT, feminists, Antifa, KKK, Marxists and, of course, the infamous Italian fascist party of Benito Mussolini. Some famous fascists include Alexander the Great, Napoleon, Churchill, Julius Caesar, Hitler, Mussolini, Stalin, Henry Ford, Steve Jobs and all American presidents (basically any so called "great leader" can be included).

Fascism is always bad and we have to aim towards eliminating it (that is eliminating fascism, NOT fascists -- fascists are people and living beings to whom we wish no harm). However here comes a great warning: **in eliminating fascism be extremely careful to not become a fascist yourself**. We purposefully do NOT advice to fight fascism as fight implies violence, the tool of fascism. **Elimination of fascism has to be done in a non-violent way**. Sadly, generation after generation keeps repeating the same mistake over and over: they keep opposing fascism by fascist means, eventually taking the oppressors place and becoming the new oppressor, only to again be dethroned by the new generation. This has happened e.g. with feminism and other pseudoleftist movements. This is an endless cycle of stupidity but, more importantly, endless suffering of people. This cycle needs to be ended. We must choose not the easy way of violence, but the difficult way of non-violent rejection which includes loving the enemy as we love ourselves. Fascism is all about loving one's own group while hating the enemy groups -- if we can achieve loving all groups of people, even fascists themselves, fascism will have been by definition eliminated.

Identity is the parent of fascism, fear is its fuel. Identity makes one mentally separate people into groups, feel part of one of them and feel threatened by the other ones. When fear of an individual reaches certain level -- which is different for everyone -- he turns to fascism. Even that who is normally anti fascist has a breaking point, under extreme pressure of fear one starts to seek purely selfish goals. This is why e.g. capitalism fuels fear culture: it makes people fascists which is a prerequisite for becoming a capitalist. When "leaders" of nations need to lead war, they start spreading propaganda of fear so as to turn people into fascists that easily become soldiers. This is why education is important in eliminating fascism: it is important to e.g. show that we need not be afraid of people of other cultures, of sharing information and resources etc. The bullshit of fear propaganda has to be exposed.

There are many easy giveaways of fascism -- in general fascism is linked to some of the following:

- pride
- sense of identity (sexual orientation, race, sex, nation, political party, ...)
- flags, flag-like symbols
- uniforms, unified fashion (wearing same colors, hairstyles, clothes ...), parades
- sense of competition, war mentality, encouragement of "fighting spirit", punishment of defeatism
- heroes, leaders, cults of personality, strong rhetoric
- the sense of nation, community or similar separating groups
- demanding "rights"
- militarism
- propaganda, censorship, brainwashing, mass hysteria
- hype, fear, pressure to take action
- ...

Fascist

See [fascism](#).

fear_culture

Fear Culture

A frightened man gives up his freedom for the promise of safety.

TODO

fediverse

Fediverse

TODO

Fediverse is partly nice, employing a few cool ideas, but also quite shitty -- it may be a relief, a less harmful alternative to proprietary social media, but it's definitely not the way of good technology. With time it will very likely keep degenerating into more and more harmful thing, just like Wikipedia and similar big projects riding on the "FOSS" brand. The following is a list of some reasons why Fediverse sucks (keep in mind that some of them are not inherent but rather established properties of the network):

- It is **greatly bloated**, mostly relying on modern browsers with JavaScript and encryption, multiple complex protocols, hugely complicated backends etc.
- It is **capitalist software, trying to mimic capitalist ways** just with a "FOSS sticker" on it, they just copy twitter, Facebook and reddit closely, keeping it based on content consumerism, like whoring, friend hoarding, scrolling addiction etc. -- a free license doesn't fix this. Fediverse doesn't care about actual freedom but rather about a "freedom" label, aiming more for getting big rather than getting actually good. A truly good network would just be based on a completely different set of ideas, see e.g. gopher.
- It **embraces censorship (see e.g. fediblock) and is greatly infected with pseudoleftist ideology**. While decentralization prevents hardcore global blocks, most network instances just block the minority of instances that allow free speech, creating isolated islands, most of which have speech filters etc. Furthermore people using these networks are for the greatest part soyboys, soydevs and SIWs circlejerking their posts, blocking everyone else AND the software projects themselves are made by the same people, employing codes of censorship etc.
- It is **developed mostly by incompetent people** -- as said, the users and developers are mainstreamers, mostly 16 year old trans zoomers who just learned about computers and are just bashing together stuff in JavaScript, they have no real plan or vision, neither do they know anything about good technology design. The result looks accordingly.
- ...

There seems to be a pretty nice "offensive" wiki connected to fediverse at https://fediverse.wiki/wiki/Main_Page.

feminism

Feminism

Sufficiently advanced stupidity is indistinguishable from feminism. --old Chinese proverb

Feminism, also feminazism or femifascism, is a fascist terrorist pseudoleftist cult aiming for establishing female as the superior gender, for social revenge on men and gaining political power, such as that over language. Similarly to LGBT, feminism is violent, toxic and harmful, based on brainwashing, mass hysteria, bullying (e.g. the metoo campaign) and propaganda.

A quite nice article on feminism can also be found on the incel wiki at <https://incels.wiki/w/Feminism>. { A friend also recommended a text called *Counter-Advice From The Third Sex*, possibly check it out. ~drummyfish }

If anything's clear, then that feminism is not at all about gender equality but about hatred towards men and female superiority. Firstly feminism is not called *gender equality movement* but *feminism*, i.e. for-female, literally "womanism", and as we know, name plays a huge role. Imagine this: if you asked feminists if they could right now implement matriarchy in society, i.e. female ruling over man, how many of them do you think would answer "no"? There is not even a shadow of a doubt a vast majority would absolutely answer "yes", we may at best argue about if it would be 85% or 99% of them. So the question of feminist goals is absolutely clearly answered, there is no point in trying to deny it. To a feminist a man is what a Jew was to the Nazi or what the Christian was to the Romans who famously hunted Christians down and fed them to the lions because they refused to bow to their polytheist ideology (nowadays analogous to e.g. refusing to practice political correctness). The whole story is repeated again, we have yet again not learned a bit from our history. Indeed, women have historically been oppressed and needed support, but once women reach social equality -- which has basically already happened a long time ago now -- feminist movement will, if only by social inertia, keep pursuing more advantages for women (what else should a movement called *feminism* do?), i.e. at this point the new goal has already become female superiority. In the age of capital no one is going to just dissolve a movement because it has already reached its goal, such a movement present political capital one will simply not throw out of window, so feminists will forever keep saying they're being oppressed and will forever keep inventing new bullshit issues to keep fighting. Note for example that feminists care about things such as wage gap but of course absolutely don't give a damn about opposite direction inequality, such as men dying on average much younger than women etc. -- feminism cares about women, not equality. And of course, when men establish "men rights" movements, suddenly feminists see those as "fascist", "toxic" and "violent" and try to destroy such movements.

{ I really have no issues with women, I truly love everyone, but I do pay attention to statistics. One of the biggest things feminism achieved for me in this regard is that now it's simply not enough for me to see a woman achieve success in society to be convinced she is skilled or capable, a woman getting PhD to me nowadays automatically just means she got it because she's a woman and we need more quotas of "strong women in SCIENCE". In the past I didn't see it this way, a woman that did something notable back then was mostly convincing to me. Nowadays I just require much better evidence to believe she is good at something, e.g. seeing something truly good she created -- to be honest, I now don't recall any woman in "modern times" to have convinced me, but I am really open to it and just waiting to be proven wrong. ~drummyfish }

Some notable things feminists managed to achieve are:

- Women hate men.
- Men hate women.
- Women are now slaves too, they have to pursue career instead of being able to enjoy stress-free time at home.
- People are scared of physical touch, eye contact, even talking. Touching a stranger by accident can mean a lawsuit.
- Men are pushed to forming fascist counter movement such as MGTOW.
- Men actually being nice to women, e.g. holding a door open for them, is seen as hostility.
- Actual good achievements of women are now dismissed because everyone supposes the success was fabricated as part of ever present feminist propaganda, hurting the few truly skilled women.
- Women refuse to have children.
- Even if a woman has a child, she has it late and doesn't take proper care of it, all because she is supposed to pursue a career and compete with men.
- Women refuse to date men, men are depressed and commit suicides (see incel).
- Stronger sexism, people now believe women are better than men, that man is automatically something to fear etc.
- There are more bullshit jobs such as diversity departments etc.
- Industries such as those of technology, science, movies etc. all go to absolute shit because of incompetent women FORCED there because "we need more strong women everywhere".
- Strong propaganda everywhere, destroying all art, truth about history etc.
- Women are more stressed because their capabilities are overestimated by the propaganda, a young girl is told she is better than a man and she is expected to beat men; in reality she finds out she can't beat a man and becomes depressed, thinking she is extremely inferior while she is just a normal

woman.

- Censorship of basically all old art such as movies without enough women in them, movies that make any kind of fun of any woman, movies that show any woman as weak etc. Tiny bits of free speech are disappearing completely.
- Women marry women and raise children who lack fathers, something that's objectively extremely bad from psychological point of view. Ask literally anyone who grew up without a father if he missed having one.
- People believe literal lies such as that a woman is physically stronger and more intelligent than man.
- Eye contact is perceived as rape.
- More people want to castrate all men than before.
- Language is becoming ugly, ridiculous and retarded by political correctness, forcing things like "personkind" instead of "mankind" or abolishment of the word "history" (because it contains the substring "his").
- Stronger fight culture, cults of personalities, toxicity of whole society etc.
- Because all the above everyone is hostile, stressed, scared, depressed, society is in tension, leading to even faster downfall.
- ...

{ LMAO, a supposed woman writer who won 1 million euro prize turned out to actually be three men writers, see Carmen Mola :) Also the recent "historically first all female space walk" during which they managed to lose \$100K worth of equipment :D ~drummyfish }

Part of the success of feminism is also capitalism -- women with priviledges, e.g. those of not having to work as much as men, are not accepted under capitalism; everyone has to be exploited as much as possible, everyone has to be a work slave. Therefore capitalist propaganda promotes ideas such as "women not having to work is oppression by men and something a woman should be ashamed of", which is of course laughable, but with enough brainwashing anything can be established, even the most ridiculous and obvious bullshit.

Apparently in Korea feminists already practice segregation, they separate parking spots for men and women so as to prevent women bumping into men or meeting a man late at night because allegedly men are more aggressive and dangerous. Now this is pretty ridiculous, this is exactly the same as if they separated e.g. parking lots for black and white people because black people are statistically more often involved in crime, you wouldn't want to meet them at night. So, do we still want to pretend feminists are not fascist?

femoid

Femoid

See woman.

fight_culture

Fight Culture

Fight culture is the harmful, mostly western mindset of seeing any endeavor as a fight against something. Even such causes as aiming for establishment of peace are seen as fighting the people who are against peace, which is funny but also sad. Fight culture keeps, just by the constant repetition of the word *fight* (and similar ones such as *combat*, *win* etc.), a subconscious validation of violence as justified and necessary means for achieving any goal. Fight culture is to a great degree the culture of capitalist society (of course not exclusively), the environment of extreme competition and hostility. It fuels war mentality, hostility, fear culture (everyone is your enemy!), constant unrest leading to mental health deterioration, obsession with various kinds of protections against everything etc. It is ridiculous, our society is now one big fight: against global warming, unemployment, inflation, unproductivity, traffic jams, hunger, too little hunger etc. Perhaps in a few years it won't even sound so weird to say you are fighting a road when you try to get from point A to point B or that you are fighting air when riding your bike.

We, of course, see fight culture as highly undesirable for a good society as that needs to be based on peace, love and collaboration, not competition. For this reasons we never say we "fight" anything (or even "win", we rather achieve goals), we rather aim for goals, look for solutions, educate and sometimes reject, avoid, refuse and oppose bad concepts (e.g. fight culture itself).

Capitalist often say that "life is a fight". We say life is what you make it, and if for your life is a fight, it merely says you desire fight. We do not.

How to stop engaging in fight culture? Adopt defeatism. That frees you, accepting loss makes you no longer constrained to unethical behavior justified by the necessity to win, the capitalist argument "you have to do X or else you lose" suddenly stops being valid and you are free to behave morally, you no longer have to engage in a lot of bullshit. Capitalist culture is extremely hostile to defeatism because it knows that's the way out of it, a defeatist individual no longer works for capitalism, therefore capitalist propaganda spreads hatred of defeatism, and in extreme situations (e.g. war) makes it officially a crime! That's how you know it's the correct thing to do.

fight

Fight

See fight culture.

finished

Finished

A finished project is completed, working and doesn't need regular maintenance, it serves its users and doesn't put any more significant burden of development cost on anyone. A finished project is not necessarily perfect and bugless, it is typically just working as intended, greatly stable, usable, well optimized and good enough. In a sane society (such as lrs) when we start a project, we are deciding to invest some effort into it with the promise of one day finishing it and then only benefiting from it for evermore; however under capitalist's update culture nothing really gets finished, projects are started with the goal of developing them forever so as to enslave more and more people (or, as capitalists put it, "create jobs" for them), which is extremely harmful. Finished project often have the version number 1.0, however under capitalist update culture this just a checkpoint towards implementing basic features which doesn't really imply the project is finished (after 1.0 they simply aim for 2.0, 3.0 etc.). **Always aim for projects that will be finished** (even if potentially not by you); sure, even in a good society SOME projects may be "perpetual" in nature -- for example an encyclopedia that's updated every 5 years with new knowledge and discoveries -- however this should only be the case where NECESSARY and the negative effects of this perpetual nature should be minimized (for example with the encyclopedia we should make the update span as large as possible, let's say 5 years as opposed to 1 year, and we should yield a nice and tidy release after every update).

Examples of projects that have been finished are:

- Collapse OS
- Anarch
- Old video games, especially those for game consoles distributed on physical media such as cartridges or CDs. When for example a GameBoy game was released back then, it had to work as the user would buy the cartridge whose content couldn't be updated over the Internet, if there was a bug, it was there forever.
- ...

How to make greatly finishable projects?

- Make a plan and stick to it, have a goal for what the project should look like when finished, have a roadmap (even if only in your head). I.e. don't aim for "creating a good text editor", aim for "creating a text editor with features X, Y, Z that's written in under N lines of code".

- Only use time-tested suckless/LRS future proof technology that's itself finished, for example the C99 language.
- Keep it simple, minimize dependencies (there are often the cause for the need of maintenance), adhere to extreme minimalism. See also portability.
- TODO

firmware

Firmware

Firmware is a type of very basic software that's usually preinstalled on a device from factory and serves to provide the most essential functionality of the device. On simple devices, like mp3 players or remote controls, firmware may be all that's ever needed for the device's functioning, while on more complex ones, such as personal computers, firmware (e.g. BIOS or UEFI) allows basic configuration and installation of more complex software (such as an operating system) and possibly provides functions that the installed software can use. Firmware is normally not meant to be rewritten by the user and is installed in some kind of memory that's not very easy to rewrite, it may even be hard-wired in which case it becomes something on the very boundary of software and hardware.

fixed_point

Fixed Point

Fixed point arithmetic is a simple and often good enough method of computer representation of fractional numbers (i.e. numbers with higher precision than integers, e.g. 4.03), as opposed to floating point which is a more complicated way of doing this which in most cases we consider a worse, bloated alternative. Probably in 99% cases when you think you need floating point, fixed point will do just fine. Fixed point arithmetic is not to be confused with fixed point of a function in mathematics (fixed point of a function $f(x)$ is such x that $f(x) = x$), a completely unrelated term.

Fixed point has at least these advantages over floating point:

- **It doesn't require a special hardware coprocessor** for efficient execution and so doesn't introduce a dependency. Programs using floating point will run extremely slowly on systems without float hardware support as they have to emulate the complex hardware in software, while fixed point will run just as fast as integer arithmetic. For this reason fixed point is very often used in embedded computers.
- It is **natural, easier to understand and therefore better predictable**, less tricky, KISS, suckless. (Float's IEEE 754 standard is 58 pages long, the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic* has 48 pages.)
- Is easier to implement and so **supported in many more systems**. Any language or format supporting integers also supports fixed point.
- It isn't ugly and in two's complement **doesn't waste values** (unlike IEEE 754 with positive and negative zero, denormalized numbers, many NaNs etc.).
- Some simpler (i.e. better) programming languages such as comun don't support float at all, while fixed point can be used in any language that supports integers.

How It Works

Fixed point uses a fixed (hence the name) number of digits (bits in binary) for the integer part and the rest for the fractional part (whereas floating point's fractional part varies in size). I.e. we split the binary representation of the number into two parts (integer and fractional) by IMAGINING a radix point at some place in the binary representation. That's basically it. Fixed point therefore spaces numbers uniformly, as opposed to floating point whose spacing of numbers is non-uniform.

So, **we can just use an integer data type as a fixed point data type**, there is no need for libraries or special hardware support. We can also perform operations such as addition the same way as with integers. For example if we have a binary integer number represented as 00001001, 9 in decimal, we may say we'll be

considering a radix point after let's say the sixth place, i.e. we get 000010.01 which we interpret as 2.25 ($2^2 + 2^{-2}$). The binary value we store in a variable is the same (as the radix point is only imagined), we only INTERPRET it differently.

We may look at it this way: we still use integers but we use them to count smaller fractions than 1. For example in a 3D game where our basic spatial unit is 1 meter our variables may rather contain the number of centimeters (however in practice we should use powers of two, so rather 1/128ths of a meter). In the example in previous paragraph we count 1/4ths (we say our **scaling factor** is 1/4), so actually the number represented as 00000100 is what in floating point we'd write as 1.0 (00000100 is 4 and $4 * 1/4 = 1$), while 00000001 means 0.25.

This has just one consequence: **we have to normalize results of multiplication and division** (addition and subtraction work just as with integers, we can normally use the + and - operators). I.e. when multiplying, we have to divide the result by the inverse of the fractions we're counting, i.e. by 4 in our case ($1/(1/4) = 4$). Similarly when dividing, we need to MULTIPLY the result by this number. This is because we are using fractions as our units and when we multiply two numbers in those units, the units multiply as well, i.e. in our case multiplying two numbers that count 1/4ths give a result that counts 1/16ths, we need to divide this by 4 to get the number of 1/4ths back again (this works the same as e.g. units in physics, multiplying number of meters by number of meters gives meters squared). For example the following integer multiplication:

00001000 * 00000010 = 00010000 ($8 * 2 = 16$)

in our system has to be normalized like this:

(000010.00 * 000000.10) / 4 = 000001.00 ($2.0 * 0.5 = 1.0$)

SIDE NOTE: in practice you may see division replaced by the shift operator (instead of /4 you'll see >> 2).

With this normalization we also have to **think about how to bracket expressions to prevent rounding errors and overflows**, for example instead of $(x / y) * 4$ we may want to write $(x * 4) / y$; imagine e.g. x being 00000010 (0.5) and y being 00000100 (1.0), the former would result in 0 (incorrect, rounding error) while the latter correctly results in 0.5. The bracketing depends on what values you expect to be in the variables so it can't really be done automatically by a compiler or library (well, it might probably be somehow handled at runtime, but of course, that will be slower). There are also ways to prevent overflows e.g. with clever bit hacks.

The normalization and bracketing are basically the only things you have to think about, apart from this everything works as with integers. Remember that **this all also works with negative number in two's complement**, so you can use a signed integer type without any extra trouble.

Remember to **always use a power of two scaling factor** -- this is crucial for performance. I.e. you want to count 1/2th, 1/4th, 1/8ths etc., but NOT 1/10ths, as might be tempting. Why are power of two good here? Because computers work in binary and so the normalization operations with powers of two (division and multiplication by the scaling factor) can easily be optimized by the compiler to a mere bit shift, an operation much faster than multiplication or division.

Code Example

For start let's compare basic arithmetic operations in C written with floating point and the same code written with fixed point. Consider the floating point code first:

```
float
a = 21,
b = 3.0 / 4.0,
c = -10.0 / 3.0;

a = a * b;    // multiplication
a += c;       // addition
a /= b;       // division
a -= 10;      // subtraction
a /= 3;       // division
```

```
printf("%f\n",a);
```

Equivalent code with fixed point may look as follows:

```
#define UNIT 1024          // our "1.0" value

int
a = 21 * UNIT,
b = (3 * UNIT) / 4,      // note the brackets, (3 / 4) * UNIT would give 0
c = (-10 * UNIT) / 3;

a = (a * b) / UNIT;      // multiplication, we have to normalize
a += c;                  // addition, no normalization needed
a = (a * UNIT) / b;      // division, normalization needed, note the brackets
a -= 10 * UNIT;          // subtraction
a /= 3;                  // division by a number NOT in UNITs, no normalization needed

printf("%d.%d%d%d\n",    // writing a nice printing function is left as an exercise :)
a / UNIT,
((a * 10) / UNIT) % 10,
((a * 100) / UNIT) % 10,
((a * 1000) / UNIT) % 10);
```

These examples output 2.185185 and 2.184, respectively.

Now consider another example: a simple C program using fixed point with 10 fractional bits, computing square roots of numbers from 0 to 10.

```
#include <stdio.h>

typedef int Fixed;

#define UNIT_FRACTIONS 1024 // 10 fractional bits, 2^10 = 1024

#define INT_TO_FIXED(x) ((x) * UNIT_FRACTIONS)

Fixed fixedSqrt(Fixed x)
{
    // stupid brute force square root

    int previousError = -1;

    for (int test = 0; test <= x; ++test)
    {
        int error = x - (test * test) / UNIT_FRACTIONS;

        if (error == 0)
            return test;
        else if (error < 0)
            error *= -1;

        if (previousError > 0 && error > previousError)
            return test - 1;

        previousError = error;
    }

    return 0;
}

void fixedPrint(Fixed x)
{
    printf("%d.%03d", x / UNIT_FRACTIONS,
        ((x % UNIT_FRACTIONS) * 1000) / UNIT_FRACTIONS);
}

int main(void)
{
    for (int i = 0; i <= 10; ++i)
    {
```

```

    printf("%d: ",i);

    fixedPrint(fixedSqrt(INT_TO_FIXED(i)));

    putchar('\n');
}

return 0;
}

```

The output is:

```

0: 0.000
1: 1.000
2: 1.414
3: 1.732
4: 2.000
5: 2.236
6: 2.449
7: 2.645
8: 2.828
9: 3.000
10: 3.162

```

fizzbuzz

FizzBuzz

FizzBuzz is a relatively simple programming problem that's famous/infamous by having a number of different approach solutions and is often used e.g. in interviews to test the skills of potential hires. It comes from a child game that teaches basic integer division in which kids are supposed to shout a specific word if a number is divisible by some other number -- what's of interest about the problem is not the solution itself (which is trivial) but rather how one should structure an algorithm that solves the problem. The problem is stated as follows:

Write a program that writes out numbers from 1 to 100 (including both); however if a number is divisible by 3, write "Fizz" instead of the number, if the number is divisible by 5, write "Buzz" instead of it and if it is divisible by both 3 and 5, write "FizzBuzz" instead of it.

The statement may of course differ slightly, for example in saying how the output should be formatted or by specifying goals such as "make the program as short as possible" or "make the program as fast as possible". For the sake of this article let's consider the following the correct output of the algorithm:

```

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz,
Buzz, 26, Fizz, 28, 29, FizzBuzz, 31, 32, Fizz, 34, Buzz, Fizz, 37, 38, Fizz, Buzz, 41, Fizz, 43, 44, FizzBuzz, 46,
47, Fizz, 49, Buzz, Fizz, 52, 53, Fizz, Buzz, 56, Fizz, 58, 59, FizzBuzz, 61, 62, Fizz, 64, Buzz, Fizz, 67, 68, Fizz,
Buzz, 71, Fizz, 73, 74, FizzBuzz, 76, 77, Fizz, 79, Buzz, Fizz, 82, 83, Fizz, Buzz, 86, Fizz, 88, 89, FizzBuzz, 91,
92, Fizz, 94, Buzz, Fizz, 97, 98, Fizz, Buzz

```

Why the fuss around FizzBuzz? Well, firstly it dodges an obvious single elegant solution that many similar problems usually have and it leads a beginner to a difficult situation that can reveal a lot about his experience and depth of his knowledge. The tricky part lies in having to check not only divisibility by 3 and 5, but also by BOTH at once, which when following basic programming instincts ("just if-then-else everything") leads to inefficiently checking the same divisibility twice and creating some extra ugly if branches and also things like reusing magic constants in multiple places, conflicting the "DRY" principle etc. It can also show if the guy knows things usually unknown to beginners such as that the modulo operation with non-power-of-two is usually expensive and we want to minimize its use. However it is greatly useful even when an experienced programmer faces it because it can serve a good, deeper discussion about things like optimization; while FizzBuzz itself has no use and optimizing algorithm that processes 100 numbers is completely pointless, the problem is similar to some problems in practice in which the approach to solution often becomes critical, considering scalability. In practice we may very well encounter FizzBuzz's big brother, a problem in which we'll need to check not 100 numbers but 100 million numbers per second and

check not only divisibility by 3 and 5, but by let's say all prime numbers. Problems like this come up e.g. in cryptography all the time, so we really have to come to discussing time complexity classes, instruction sets and hardware acceleration, parallelism, possibly even quantum computing, different paradigms etc. So really FizzBuzz is like a kind of great conversation starter, a bag of topics, a good training example and so on.

TODO: some history etc.

Implementations

Let's see how we can implement, improve and optimize FizzBuzz in C. Keep in mind the question of scalability, i.e. try to imagine how the changes we make to the algorithm would manifest if the problem grew, i.e. if for example we wanted to check divisibility by many more numbers than just 1 and 5 etc. We will only focus on optimizing the core of the algorithm, i.e. the divisibility checking, without caring about other things like optimizing printing the commas between numbers and whatnot. Also we'll be supposing all compiler optimization are turned off so that the excuse "compiler will optimize this" can't be used :)

For starters let us write a kind of vanilla, naive solution that everyone will likely come up with as his first attempt. A complete noob will fail to produce even this basic version, a slightly advanced programmer (we might say a "coder") may submit this as the final solution.

```
#include <stdio.h>

int main(void)
{
    for (int i = 1; i <= 100; ++i)
    {
        if (i != 1)
            printf(", ");

        if (i % 3 == 0 && i % 5 == 0)
            printf("FizzBuzz");
        else if (i % 3 == 0) // checking divisibility by 3 again :/
            printf("Fizz");
        else if (i % 5 == 0) // checking divisibility by 5 again :/
            printf("Buzz");
        else
            printf("%d", i);
    }

    putchar('\n');
    return 0;
}
```

It works, however with a number of issues. Firstly we see that for every number we check we potentially test the divisibility by 3 and 5 twice, which is not good, considering division (and modulo) are one of the slowest instructions. We also reuse the magic constants 3 and 5 in different places, which would start to create a huge mess if we were dealing with many more divisors. There is also a lot of branching, in the main divisibility check we may jump up to three times for the checked number -- jump instructions are slow and we'd like to avoid them (again, consider we were checking e.g. divisibility by 1000 different numbers).

When asked to optimize the algorithm a bit one might come up with something like this:

```
#include <stdio.h>

int main(void)
{
    for (int i = 1; i <= 100; ++i)
    {
        if (i != 1)
            printf(", ");

        int printNum = 1;

        if (i % 3 == 0)
        {
            printf("Fizz");
            printNum = 0;
        }

        if (i % 5 == 0)
        {
            printf("Buzz");
            printNum = 0;
        }

        if (printNum)
            printf("%d", i);
    }

    putchar('\n');
    return 0;
}
```

```

    printNum = 0;
}

if (i % 5 == 0)
{
    printf("Buzz");
    printNum = 0;
}

if (printNum)
    printf("%d",i);
}

putchar('\n');
return 0;
}

```

Now we check the divisibility by 3 and 5 only once for each tested number, and also keep only one occurrence of each constant in a single place, that's good. But we still keep the slow branching.

A bit more experienced programmer may now come with something like this:

```

#include <stdio.h>

int main(void)
{
    for (int i = 1; i <= 100; ++i)
    {
        if (i != 1)
            printf(", ");

        switch ((i % 3 == 0) + ((i % 5 == 0) << 1))
        {
            case 1: printf("Fizz"); break;
            case 2: printf("Buzz"); break;
            case 3: printf("FizzBuzz"); break;
            default: printf("%d",i); break;
        }

    }

    putchar('\n');
    return 0;
}

```

This solution utilizes a switch structure to only perform single branching in the divisibility check, based on a 2 bit value that in its upper bit records divisibility by 5 and in the lower bit divisibility by 3. This gives us 4 possible values: 0 (divisible by none), 1 (divisible by 3), 2 (divisible by 5) and 3 (divisible by both). The switch structure by default creates a jump table that branches right into the correct label in O(1).

We can even go as far as avoiding any branching at all with so called branchless programming, even though in this specific case saving one branch is probably not worth the cost of making it happen. But for the sake of completeness we can do e.g. something as follows.

```

#include <stdio.h>

char str[] = "\0\0\0\0\0\0\0\0\Fizz\0\0\0\0\0\0\0\0\Buzz\0\0\0\0\0\FizzBuzz";

int main(void)
{
    for (int i = 1; i <= 100; ++i)
    {
        if (i != 1)
            printf(", ");

        // look mom, no branches!
        char *s = str;

        *s = '1'; // convert number to string
    }
}

```

```

s += i >= 100;
*s = '0' + (i / 10) % 10;
s += (*s != '0') | (i >= 100);
*s = '0' + i % 10;

int offset = ((i % 3 == 0) + ((i % 5 == 0) << 1)) << 3;
printf(str + offset);
}

putchar('\n');
return 0;
}

```

The idea is to have a kind of look up table of all options we can print, then take the thing to actually print out by indexing the table with the 2 bit divisibility value we used in the above example. Our lookup table here is the global string `str`, we can see it rather as an array of zero terminated strings, each one starting at the multiple of 8 index (this alignment to power of two will make the indexing more efficient as we'll be able to compute the offset with a mere bit shift as opposed to multiplication). The first item in the table is initially empty (all zeros) and in each loop cycle will actually be overwritten with the ASCII representation of currently checked number, the second item is "Fizz", the third item is "Buzz" and last one is "FizzBuzz". In each loop cycle we compute the 2 bit divisibility value, which will be a number 0 to 3, bit shift it by 3 to the left (multiply it by 8) and use that as an offset, i.e. the place where the printing function will start printing (also note that printing will stop at encountering a zero value). The conversion of number to ASCII is also implemented without any branches (and could be actually a bit simpler as we know e.g. the number 100 won't ever be printed). However notice that we pay a great price for all this: the code is quite ugly and unreadable and also performance-wise we many times waste time on converting the number to ASCII even if it then won't be printed, i.e. something that a branch can actually prevent, and the conversion actually further uses modulo and division instructions which we are trying to avoid in the first place... so at this point we probably overengineered this.

If the problem asks for shortest code, even on detriment of readability and efficiency, we might try **the code golfer approach**:

```

#include <stdio.h>
#define P printf(
int i;int main(){while(i<100){if(i++)P", ");int a=!(i%3)+!(i%5)*2;if(a)P"FizzBuzz\0Fizz"+(4+(a==1)*5)*(a!=3);

```

It's almost definitely not minimal but can be a good start.

TODO: comun

flatland

Flatland

Flatland: A Romance of Many Dimensions is an amazing book from 1884, now completely in the safe/strong public domain, whose story takes place in a flat plane, a two dimensional world inhabited by sentient two dimensional geometric shapes (men being polygons, women just line segments). The book is classified as mathematical fantasy -- besides being a very rare case of an exceptional quality completely public domain fantasy before The Lord of the Rings, it is also both a social criticism and an interesting and entertaining examination of mathematical and scientific concepts such as "how would two dimensional beings build their houses?", "how would they see?" etc. Flatland was written by Edwin Abbott Abbott, an English theologian, priest and teacher. There were sequels and spinoffs written by other people, even movies, but these aren't generally in the public domain yet.

{ As the book is in the safe public domain, I won't restrain from going on deeper on summarizing the plot etc., legal dangers of any "infringements" are quite definitely zero here. YES, I know I can summarize plots even of proprietary works, but this wiki goes further, it want to also ensure that someone can e.g. take the plot and turn in into a video game, which in cases like fair use could lead to infringements. ~drummyfish }

From now on expect **spoilers** :)

The book is written as a narration by a square, an upper middle class shape, and describes all the peculiarities of living in Flatland, talking directly to the reader, a supposed inhabitant of "Spaceland" (the 3D world). The year in the book is 1999 of the Flatland world. The book explains and explores spaces of different dimensions, firstly mathematically and then as a social topic -- the square protagonist essentially starts thinking about the possibility that besides his 2D universe there might exist worlds of different dimensions -- at first he dreams about being in 1D land -- Lineland -- and later, at the turn of the millennium, he is visited by a sphere from the 3D world, a sort of alien to the square, but the sphere is able to convince the square that it came from a higher dimensional space. The ideas of existence of different dimensions are consequently seen as a kind of lunacy and heresy by others, he is basically seen as a schizo and conspiracy theorist and gets in trouble for his freethinking, just like many of those who in the past questioned religious orthodoxy or those who nowadays question official "science". Examination of the 0 dimensional space, Pointland, also appears in the story.

The following are some further details about the work:

- **Men are polygons, more sides implies higher class**, within a single class the polygon regularity further implies one's social standing. Triangles are lowest class of men, among these the lowest are irregular triangles (soldiers, their sharp point makes them good at fighting) with two long and one short side because these are kind of closer to a mere line segment (woman). Equal sided triangles are the middle class. Squares and pentagons are professional workers. Noblemen begin at hexagons. The highest class are circles, i.e. polygons in which the number of sides is so numerous they appear to be smooth all around -- these are the priests. A man's male descendant will typically have one more side than his father, rising in social class, but this doesn't always happen and is less common among lower classes, with isoscele soldiers only rarely having such children (it requires some effort, like spiritual exercise and intermarriages).
- **Women are just line segments**, implying several things -- firstly a line segment is basically the lowest possible polygon, a degenerated triangle, i.e. by this a woman is even below the lowest class of men (triangles). Furthermore for its geometric shape a woman is dangerous and can be sneaky as from certain angles she just cannot be seen -- one can bump into her and get injured (as a one dimensional shape she is very sharp, even more than a soldier), which, as the author notes, is further made worse by women being extremely dumb and acting mostly just on instinct. For this women are required by law to wiggle their butts (which they actually started to like doing as a kind of fashion trend -- yes, just like twerking) and make noise when moving around in public.
- **World directions** can be distinguished thanks to a law of nature in Flatland that creates a slight but constant attraction towards south, making rain fall from north. This could be explained by Flatland being located on a slightly tilted plane within Spaceland.
- **Sight** of Flatlanders is also explained -- seeing in this world poses difficulties -- everything looks like a line and a Flatlander always has just one eye -- so the senses of hearing and touch are mentioned to be highly developed to provide additional help (though distinguishing by voice is seen as more of a pleb thing, not much practiced by aristocrats who prefer using sight). Fog also helps. Colors exist and could aid seeing greatly, but their use is highly regulated (read forbidden) by the Universal Color Bill because in history color firstly caused some trouble (like women shading themselves to look like circles) and secondly nobility wanted to keep the precious art of sight recognition to themselves, so they just banned it despite it killing all art etc. (Basically what copyright does nowadays etc.)
- Sizes are mentioned in common units, a Flatlander is about a foot or so in size, though whether their foot corresponds to our foot isn't clear.
- **Light** is present everywhere and at all times -- the origin of light is unknown to Flatland society but the protagonist hints on the heretic idea that light in fact comes from the Spaceland, in which Flatland is embedded.
- **Houses** are most commonly pentagonal (sharper shapes are forbidden to prevent injuries), with roofs protecting against rain and they have no windows (as light is everywhere). West side has a big entrance for men, east a smaller entrance for women.
- The book spawns a possible **free universe**, which may furthermore be quite friendly to e.g. making computer games (2D games are easier to make than 3D ones, also no need for many assets etc.).
- The book is in the **safe public domain**, i.e. it was published before 1900, author has been dead for nearly 100 years, so even in countries with strictest copyright it should be fine.
- There is some **nice sexist bashing of women** :D E.g. "[soldiers are] creatures almost on a level with women in their lack of intelligence", "[women are] wholly devoid of brainpower" etc. That's very cool and refreshing.
- ...

Floating Point

In programming floating point (colloquially just *float*) is a way of representing fractional numbers (such as 5.13) and approximating real numbers (i.e. numbers with higher than integer precision), which is a bit more complex than simpler methods for doing so (such as fixed point). The core idea of it is to use a radix ("decimal") point that's not fixed but can move around so as to allow representation of both very small and very big values. Nowadays floating point is the standard way of approximating real numbers in computers (floating point types are called *real* in some programming languages, even though they represent only rational numbers, floats can't e.g. represent π exactly), basically all of the popular programming languages have a floating point data type that adheres to the IEEE 754 standard, all personal computers also have the floating point hardware unit (FPU) and so it is widely used in all modern programs. However most of the time a simpler representation of fractional numbers, such as the mentioned fixed point, suffices, and weaker computers (e.g. embedded) may lack the hardware support so floating point operations are emulated in software and therefore slow -- remember, float rhymes with blat. Prefer fixed point.

Floating point is tricky, it works most of the time but a danger lies in programmers relying on this kind of magic too much, some new generation programmers may not even be very aware of how float works. Even though the principle is not so hard, the emergent complexity of the math is really complex. One floating point expression may evaluate differently on different systems, e.g. due to different rounding settings. Floating point can introduce chaotic behavior into linear systems as it inherently makes rounding errors and so becomes a nonlinear system (source: <http://foldoc.org/chaos>). One common pitfall of float is working with big and small numbers at the same time -- due to differing precision at different scales small values simply get lost when mixed with big numbers and sometimes this has to be worked around with tricks (see e.g. this devlog of The Witness where a float time variable sent into shader is periodically reset so as to not grow too large and cause the mentioned issue). Another famous trickiness of float is that you shouldn't really be comparing them for equality with a normal `==` operator as small rounding errors may make even mathematically equal expressions unequal (i.e. you should use some range comparison instead).

And there is more: floating point behavior really depends on the language you're using (and possibly even compiler, its setting etc.) and it may not be always completely defined, leading to possible nondeterministic behavior which can cause real trouble e.g. in physics engines.

{ Really as I'm now getting down the float rabbit hole I'm seeing what a huge mess it all is, I'm not nearly an expert on this so maybe I've written some BS here, which just confirms how messy floats are. Anyway, from the articles I'm reading even being an expert on this issue doesn't seem to guarantee a complete understanding of it :) Just avoid floats if you can. ~drummyfish }

Is floating point literal evil? Well, of course not, but it is extremely overused. You may need it for precise scientific simulations, e.g. numerical integration, but as our small3dlib shows, you can comfortably do even 3D rendering without it. So always consider whether you REALLY need float. **You mostly do NOT need it.**

Simple example of avoiding floating point: many noobs think that if they e.g. need to multiply some integer x by let's say 2.34 they have to use floating point. This is of course false and just proves most retarddevs don't know elementary school math. Multiplying x by 2.34 is the same as $(x * 234) / 100$, which we can optimize to an approximately equal division by power of two as $(x * 2396) / 1024$. Indeed, given e.g. $x = 56$ we get the same integer result 131 in both cases, the latter just completely avoiding floating point.

How It Works

The very basic idea is following: we have digits in memory and in addition we have a position of the radix point among these digits, i.e. both digits and position of the radix point can change. The fact that the radix point can move is reflected in the name *floating point*. In the end any number stored in float can be written with a finite number of digits with a radix point, e.g. 12.34. Notice that any such number can also always be written as a simple fraction of two integers (e.g. $12.34 = 1 * 10 + 2 * 1 + 3 * 1/10 + 4 * 1/100 = 617/50$), i.e. any such number is always a rational number. This is why we say that floats represent fractional numbers and not true real numbers (real numbers such as π , e or square root of 2 can only be approximated).

More precisely floats represent numbers by representing two main parts: the *base* -- actual encoded digits, called **mantissa** (or significand etc.) -- and the position of the radix point. The position of radix point is called the **exponent** because mathematically the floating point works similarly to the scientific notation of extreme numbers that use exponentiation. For example instead of writing 0.0000123 scientists write $123 * 10^{-7}$ -- here 123 would be the mantissa and -7 the exponent.

Though various numeric bases can be used, in computers we normally use base 2, so let's consider it from now on. So our numbers will be of format:

*mantissa * 2^{exponent}*

Note that besides mantissa and exponent there may also be other parts, typically there is also a sign bit that says whether the number is positive or negative.

Let's now consider an extremely simple floating point format based on the above. Keep in mind this is an EXTREMELY NAIVE inefficient format that wastes values. We won't consider negative numbers. We will use 6 bits for our numbers:

- 3 leftmost bits for mantissa: This allows us to represent $2^3 = 8$ base values: 0 to 7 (including both).
- 3 rightmost bits for exponent: We will encode exponent in two's complement so that it can represent values from -4 to 3 (including both).

So for example the binary representation 110011 stores mantissa 110 (6) and exponent 011 (3), so the number it represents is $6 * 2^3 = 48$. Similarly 001101 represents $1 * 2^{-3} = 1/8 = 0.125$.

Note a few things: firstly our format is shit because some numbers have multiple representations, e.g. 0 can be represented as 000000, 000001, 000010, 000011 etc., in fact we have 8 zeros! That's unforgivable and formats used in practice address this (usually by prepending an implicit 1 to mantissa).

Secondly notice the non-uniform distribution of our numbers: while we have a nice resolution close to 0 (we can represent 1/16, 2/16, 3/16, ...), our resolution in high numbers is low (the highest number we can represent is 56 but the second highest is 48, we can NOT represent e.g. 50 exactly). Realize that obviously with 6 bits we can still represent only 64 numbers at most! So float is NOT a magical way to get more numbers, with integers on 6 bits we can represent numbers from 0 to 63 spaced exactly by 1 and with our floating point we can represent numbers spaced as close as 1/16th but only in the region near 0, we pay the price of having big gaps in higher numbers.

Also notice that things like simple addition of numbers become more difficult and time consuming, you have to include conversions and rounding -- while with fixed point addition is a single machine instruction, same as integer addition, here with software implementation we might end up with dozens of instructions (specialized hardware can perform addition fast but still, not all computer have that hardware).

Rounding errors will appear and accumulate during computations: imagine the operation $48 + 1/8$. Both numbers can be represented in our system but not the result (48.125). We have to round the result and end up with 48 again. Imagine you perform 64 such additions in succession (e.g. in a loop): mathematically the result should be $48 + 64 * 1/8 = 56$, which is a result we can represent in our system, but we will nevertheless get the wrong result (48) due to rounding errors in each addition. So the behavior of float can be **non intuitive** and dangerous, at least for those who don't know how it works.

Standard Float Format: IEEE 754

IEEE 754 is THE standard that basically all computers use for floating point nowadays -- it specifies the exact representation of floating point numbers as well as rounding rules, required operations applications should implement etc. However note that the standard is **kind of shitty** -- even if we want to use floating point numbers there exist better ways such as **posits** that outperform this standard. Nevertheless IEEE 754 has been established in the industry to the point that it's unlikely to go anytime soon. So it's good to know how it works.

Numbers in this standard are signed, have positive and negative zero (oops), can represent plus and minus infinity and different NaNs (not a number). In fact there are thousands to billions of different NaNs which are

basically wasted values. These inefficiencies are addressed by the mentioned posits.

Briefly the representation is following (hold on to your chair): leftmost bit is the sign bit, then exponent follows (the number of bits depends on the specific format), the rest of bits is mantissa. In mantissa implicit 1. is considered (except when exponent is all 0s), i.e. we "imagine" 1. in front of the mantissa bits but this 1 is not physically stored. Exponent is in so called biased format, i.e. we have to subtract half (rounded down) of the maximum possible value to get the real value (e.g. if we have 8 bits for exponent and the directly stored value is 120, we have to subtract $255 / 2 = 127$ to get the real exponent value, in this case we get -7). However two values of exponent have special meaning; all 0s signify so called denormalized (also subnormal) number in which we consider exponent to be that which is otherwise lowest possible (e.g. -126 in case of 8 bit exponent) but we do NOT consider the implicit 1 in front of mantissa (we instead consider 0.), i.e. this allows storing zero (positive and negative) and very small numbers. All 1s in exponent signify either infinity (positive and negative) in case mantissa is all 0s, or a NaN otherwise -- considering here we have the whole mantissa plus sign bit unused, we actually have many different NaNs (WTF), but usually we only distinguish two kinds of NaNs: quiet (qNaN) and signaling (sNaN, throws and exception) that are distinguished by the leftmost bit in mantissa (1 for qNaN, 0 for sNaN).

The standard specifies many formats that are either binary or decimal and use various numbers of bits. The most relevant ones are the following:

name	M bits	E bits	smallest and biggest number	precision <= 1 up to
binary16 (half precision)	10	5	$2^{(-24)}$, 65504	2048
binary32 (single precision, float)	23	8	$2^{(-149)}$, $2^{127} * (2 - 2^{(-23)}) \approx 3 * 10^{38}$	16777216
binary64 (double precision, double)	52	11	$2^{(-1074)}$, $\sim 10^{308}$	9007199254740992
binary128 (quadruple precision)	112	15	$2^{(-16494)}$, $\sim 10^{4932}$	$\sim 10^{34}$

Example? Let's say we have float (binary34) value 11000000111100000000000000000000: first bit (sign) is 1 so the number is negative. Then we have 8 bits of exponent: 10000001 (129) which converted from the biased format (subtracting 127) gives exponent value of 2. Then mantissa bits follow: 111000000000000000000000. As we're dealing with a normal number (exponent bits are neither all 1s nor all 0s), we have to imagine the implicit 1. in front of mantissa, i.e. our actual mantissa is 1.111000000000000000000000 = 1.875. The final number is therefore $-1 * 1.875 * 2^2 = -7.5$.

See Also

- posit
- fixed point

floss

FLOSS

FLOSS (free libre and open source) is basically FOSS.

football

Football

Not to be confused with any American pseudosport.

Football is one of the most famous sport games in which two teams face each other and try to score goals by kicking an inflated ball. It is one of the best sports not only because it is genuinely fun to play and watch but also because of its essentially simple rules, accessibility (not for rich only, all that's really needed is

something resembling a ball) and relatively low discrimination -- basically anyone can play it, unlike for example basketball in which height is key; in amateur football even fat people can take part (they are usually assigned the role of a goalkeeper). Idiots call football *soccer*.

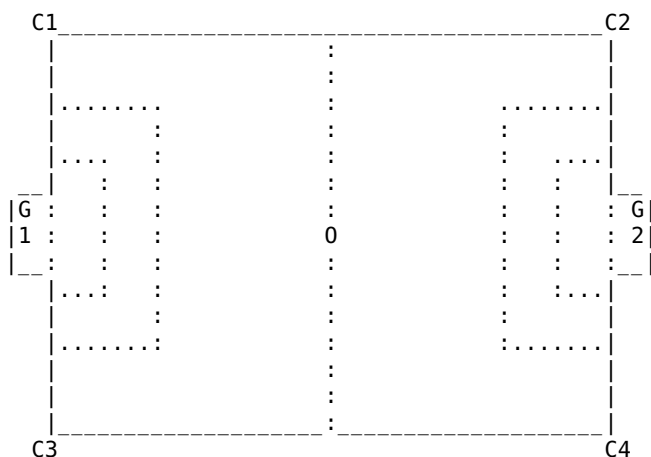
We, LRS, highly value football, as it's a very KISS sport that can be played by anyone anywhere without needing expensive equipment. It is the sport of the people, very popular in poor parts of the world.

Football can be implemented as a video game or inspire a game mode -- this has been done e.g. in Xonotic (the Nexball mode) or SuperTuxKart. There is a popular mainstream proprietary video game called Rocket League in which cars play football (INB4 zoomers start calling football "Rocket League with people"). There is also a greatly suckless pen and paper version of football called paper football.

Rules

As football is so widely played on all levels and all around the world, there are many versions and rule sets of different games in the *football* family, and it can sometimes be difficult to even say what classifies as football and what's a different sport. There are games like futsal and beach football that may or may not be seen as a different sport. The most official rules of what we'd call football are probably those known as *Laws of the Game* governed by International Football Association Board (IFAB) -- these rules are used e.g. by FIFA, various national competitions etc. Some organizations, e.g. some in the US, use different but usually similar rules. We needn't say these high level rules are pretty complex -- *Laws of the Game* have over 200 pages and talk not just about the mechanics of the game but also things such as allowed advertising, political and religious symbolism, referee behavior etc.

Here is a simple ASCII rendering of the football pitch:



In amateur games simpler rules are used -- a sum up of such rules follows:

- There are **two teams** of players facing each other in a match, officially 11 players on each team (10 "normal" players and one goal keeper), but of course this can wildly differ in just for fun games (street rules may also state that there is no fixed goal keeper; goal keeper is the one currently closest to own goal, or there may simply be no goal keeper at all, especially with small goals). Each team usually wears different colors so they're easily distinguished.
- The match is played on a flat **rectangular field** (officially between 100x64 and 110x75 meters, but for fewer player this can of course be much smaller) where there are **two goals** in the center of the shorter sides. Each team has one goal (which may be as simple as two shoes marking the goal borders) into which the opponent team tries to score goals.
- There is **one ball** in the game, players move the ball by kicking it with their feet -- they may use other body parts too **except for their arms**. The exception is goal keeper who may touch the ball also with his arms and hands, but only within the small area near the goal he protects. If no ball is available other things resembling it may be used.
- The game starts with both teams on their half of the field, one is given the possession of the ball (this is usually decided by coin toss, but sometimes e.g. one team is given the choice of goal and other one gets the ball).

- If the ball ends up in one team's goal, the opposite team **scores a point** and the game is restarted.
- If the ball leaves the play field on one of the longer sides, the team who's player didn't touch the ball last gets a **throw-in** -- one player takes the ball in his hands and throws it to the play field from the point at which the ball left the field. If a game is played in an environment which the ball cannot leave (e.g. a small sport hall), this rule is simply ignored.
- If the ball leaves the play field on one of the shorter sides, then two things may happen. If the ball was last touched by the team whose goal is on the side at which the ball left the field, the opposite team gets a **corner kick** (a free kick from the nearest corner of the field). Otherwise the goal keeper of the goal on the side where the ball left the play field gets a kick off from near his goal.
- **Offside**: this is the infamous rule that women don't get. This rule basically states that it is illegal for a player to get the ball while he's on the opponent's half and at the time the ball started to be passed to him he was closer to the opponent's goal than the second to last opponent's player (usually the back-most defender). This is so that one player doesn't just camp in front of the opponent's goal and wait for a long pass to score an easy goal. However in fun games this rule may be just ignored.
- There are **punishments** for breaking the rules (e.g. playing with arm, attacking the opponent player or stalling the game) which include a free kick (from the spot at which an offense happened), a penalty kick (one player gets a free kick on the opponent's goal with goal keeper from certain distance), yellow card (warning to a player), red card (given after yellow card, the player has to leave the game) etc.
- The game is played in **two halves**, each one officially 45 minutes long, but in fun games the number of parts and their duration is pretty arbitrary. After the first half there is a short rest pause and the teams switch sides. After the second half whichever team has scored more points wins. If the score is equal, the match may be prolonged as a tie breaker, either just by adding more time or playing the instant death. If no winner is decided here, there may eventually be penalty kicks to decide the winner.
- It's good if there's a referee, but if there is none, players just enforce the rules collectively.

fork

Fork

In technology forking generally means splitting, or better said duplicating an abstract entity (such as a computer process or development project) into two or more such entities so that each one can from then on develop differently; this is very similar to how biological cells duplicate by splitting. The term *fork* is used in many contexts, for example in software development (project forking), in operating systems (process forking), in cryptocurrencies (blockchain forking), nondeterministic computing (computation forking) etc.

Project Forking

In context of projects fork is a branch that splits from the main branch of a project and continues to develop in a different direction as a separate version of that project, possibly becoming a completely new one. This may happen with any "intellectual work" or idea such as software, movement, theory, literary universe, religion or, for example, a database. Forks may later be *merged* back into the original project or continue and diverge far away, forks of different projects may also combine into a single project as well.

For example the Android operating system and Linux-libre kernel have both been forked from Linux. Linux distributions highly utilize forking, e.g. Devuan or Ubuntu and Mint are forked from Debian. Free software movement was forked into open source, free culture and suckless, and suckless was more or less forked into LRS. Wikipedia also has forks such as Metapedia. Memes evolve a lot on the basis of forking.

Forking takes advantage of the ability to freely duplicate information, i.e. if someone sees how to improve an intellectual work or use it in a novel way, he may simply copy it and start developing it in a new diverging direction while the original continues to exist and going its own way. That is **unless copying and modification of information is artificially prevented**, e.g. by intellectual property laws or purposeful obscurity standing in the way of remixing. For this reason forking is very popular in free culture and free software where it is allowed both legally and practically -- in fact it plays a very important role there.

In software development temporary forking is used for implementing individual features which, when completed, are merged back into the main branch. This is called **branching** and is supported by version

control systems such as git.

There are two main kinds of forks:

- **soft forks** (also dynamic forks): Soft fork introduces changes that somehow stay compatible with the original project and can potentially even be merged back later, the fork exists as a different but synchronized version of the original project and as the original gets updates, the fork automatically gets these updates as well. Temporary git forks during development are soft forks, also for example Linux-libre is a soft fork of Linux as it adds a set of scripts that automatically remove proprietary blobs from the current version of Linux; patches of suckless software can also be seen as soft forks. It is typical that a soft fork somehow maintains just a set of changes against the original, e.g. in a form of a diff or script, i.e. soft fork is kind of a lightweight fork for which the original project stays a dependency.
- **hard forks**: Hard fork splits from the original project in such a way that it can no longer be easily merged back, it diverges in a very different way and stops being synchronized with the original. For example darkplaces is a hard fork of the Quake 1 engine. Hard fork typically just copies all the data of the original project to a new repository and start modifying them freely. This has the disadvantage of having to repeat work on the original and the fork (e.g. if a new bug is discovered in the original after the split, it has to be manually fixed in both versions). This is one of the reasons why hard forks very often split off of projects that aren't actively developed anymore.

Is forking good? Yes, to create anything new it is basically necessary to build on top of someone else's work, stand on someone else's shoulders. Some people criticize too much forking; for example some cry about Linux distro fragmentation, they say there are too many of distros and that people should rather focus their energy on creating a single or at least fewer good operating systems, i.e. that forking is kind of "wasting effort". LRS supports any kind of wild forking and experimentation, we believe the exploration of many directions to be necessary in order to find the right one, in a good society waste of work won't be happening -- that's an issue of a competitive society, not forking.

In fact we think that (at least soft) forking should be incorporated on a much more basic level, in the way that the suckless community popularized. In suckless **everyone's copy of software is a personal fork**, i.e. software is distributed in source form and is so extremely easy to compile and modify that every user is supposed to do this as part of the installation process (even if he isn't a programmer). Before compilation user applies his own selected patches, custom changes and specific configuration (which is done in the source code itself) that are unique to that user and which form source code that is the user's personal fork. Some of these personal forks may even become popular and copied by other users, leading to further development of these forks and possible natural rise of very different software. This should lead to natural selection, survival and development of the good and useful forks.

Process Forking

See also *fork bomb*.

TODO

formal_language

Formal Language

The field of formal languages tries to mathematically and rigorously view problems as languages; this includes probably most structures we can think of, from human languages and computer languages to visual patterns and other highly abstract structures. Formal languages are at the root of theoretical computer science and are important e.g. for the theory of computability/decidability, computational complexity, security and compilers, but they also find use in linguistics and other fields of science.

A **formal language** is defined as a (potentially infinite) set of strings (which are finite but unlimited in length) over some alphabet (which is finite). I.e. a language is a subset of E^* where E is a finite alphabet (a set of *letters*). ($*$ is a *Kleene Star* and signifies a set of all possible strings over E). The string belonging to a language may be referred to as a *word* or perhaps even *sentence*, but this word/sentence is actually a whole

kind of *text* written in the language, if we think of it in terms of our natural languages. The C programming language can be seen as a formal language which is a set of all strings that are a valid C program that compiles without errors etc.

For example, given an alphabet $[a,b,c]$, a possible formal language over it is $[a,ab,bc,c]$. Another, different possible language over this alphabet is an infinite language $[b,ab,aab,aaab,aaaab,\dots]$ which we can also write with a regular expression as a^*b . We can also see e.g. English as being a formal language equivalent to a set of all texts over the English alphabet (along with symbols like space, dot, comma etc.) that we would consider to be in English as we speak it.

What is this all good for? This mathematical formalization allows us to classify languages and understand their structure, which is necessary e.g. for creating efficient compilers, but also to understand computers as such, their power and limits, as computers can be viewed as machines for processing formal languages. With these tools researches are able to come up with proofs of different properties of languages, which we can exploit. For example, within formal languages, it has been proven that certain languages are uncomputable, i.e. there are some problems which a computer cannot ever solve (typical example is the halting problem) and so we don't have to waste time on trying to create such algorithms as we will never find any. The knowledge of formal languages can also guide us in designing computer languages: e.g. we know that regular languages are extremely simple to implement and so, if we can, we should prefer our languages to be regular.

Classification

We usually classify formal languages according to the **Chomsky hierarchy**, by their computational "difficulty". Each level of the hierarchy has associated models of computation (grammars, automatons, ...) that are able to compute **all** languages of that level (remember that a level of the hierarchy is a superset of the levels below it and so also includes all the "simpler" languages). The hierarchy is more or less as follows:

- **all languages**: This includes all possible languages, even those that computers cannot analyze (e.g. the language representing the halting problem). These languages can only be computed by theoretical computers that cannot physically exist in our universe.
- **type 0, recursively enumerable languages**: Most "difficult"/general languages that computers in our universe can analyze. These languages can be computed e.g. by a **Turing machine**, lambda calculus or a general unrestricted grammar. Example language: a^n where n is not a prime.
- **type 1, context sensitive languages**: Computed e.g. by a linearly bounded non-deterministic Turing machine or a context sensitive grammars. Example language: $a^n(n)b^n(n)c^n(n)$, $n \geq 0$ (strings of n as, followed by n bs, followed by n cs).
- **type 2, context free languages**: Computed by e.g. non-deterministic pushdown automata or context free grammars. (Deterministic pushdown automata compute a class of languages that is between type 2 and type 3).
- **type 3, regular languages**: The *easiest, weakest* kind of languages, computed e.g. by finite state automatons or regular expressions. This class includes also all finite languages.

Note that here we are basically always examining **infinite languages** as finite languages are trivial. If a language is finite (i.e. the set of all strings of the language is finite), it can automatically be computed by any type 3 computational model. In real life computers are actually always equivalent to a finite state automaton, i.e. the *weakest* computational type (because a computer memory is always finite and so there is always a finite number of states a computer can be in). However this doesn't mean there is no point in studying infinite languages, of course, as we're still interested in the structure, computational methods and approximating the infinite models of computation.

NOTE: When trying to classify a programming language, we have to be careful about what we classify: one thing is what a program written in given language can compute, and another thing is the language's syntax. To the former all strict general-purpose programming languages such as C or JavaScript are type 0 (Turing complete). From the syntax point of view it's a bit more complicated and we need to further define what exactly a syntax is (where is the line between syntax and semantic errors): it may be (and often is) that syntactically the class will be lower. There is actually a famous meme about Perl syntax being undecidable.

forth

Forth

{ I'm a bit ashamed but I'm not really "fluent" at Forth, I just played around with it for a bit. Yes, I'm planning to get into it more after I do the other million things on my TODO list. Let me know if there is some BS, thank u <3 ~drummyfish }

Forth ("fourth generation" shortened to four characters due to technical limitations) is a very good, extremely minimal stack-based untyped programming language that uses postfix (reverse Polish) notation. Its vanilla form is super simple, it's miles simpler than C, it's very elegant and its compiler/interpreter can be made very easily, giving it high practical freedom (i.e. not being practically controlled by any central organization). As of writing this the smallest Forth implementation, milliforth, has just **340 bytes** (!!!) of machine code, that's just incredible. Forth is used e.g. in space technology (e.g. RTX2010, a radiation hardened space computer directly executing Forth) and embedded systems as a way to write efficient low level programs that are, unlike those written in assembly, portable (fun fact: there even exist computers directly running Forth in hardware). Forth was the main influence for Comun, the LRS programming language, it is also used by Collapse OS and Dusk OS as the main language. In its minimalism Forth competes a bit with Lisp.

{ There used to be a nice Forth wiki at wiki.forthfreak.net, now it has to be accessed via archive as it's dead. ~drummyfish }

{ There is also some discussion about how low level Forth really is, if it really is a language or something like a "metalanguage", or an "environment" to create your own language by defining your own words. Now this is not a place to go very deep on this but kind of a sum up may be this: Forth in its base version is very low level, however it's very extensible and many extend it to some kind of much higher level language, hence the debates. ~drummyfish }

It is usually presented as interpreted language but may as well be compiled, in fact it maps pretty nicely to assembly. Even if interpreted, it can still be very fast. Forth systems traditionally include not just a compiler/interpreter but also an **interactive environment**, kind of REPL language shell.

There are several Forth standards, most notably ANSI Forth from 1994 (the document is proprietary, sharing is allowed, 640 kB as txt). Besides others it also allows Forth to include optional floating point support.

A free implementation is e.g. GNU Forth (gforth) or pforth (a possibly better option by LRS standards, favors portability over performance).

Forth was invented by Charles Moore in 1968, for programming radio telescopes.

Language

Forth is case-insensitive (this may however not be the case in some implementations).

The language operates on an evaluation **stack**: e.g. the operation + takes the two values at the top of the stack, adds them together and pushed the result back on the stack. Besides this there are also some "advanced" features like variables living outside the stack, if you want to use them.

The stack is composed of **cells**: the size and internal representation of the cell is implementation defined. There are no data types, or rather everything is just of type signed int.

Basic abstraction of Forth is so called **word**: a word is simply a string without spaces like abc or 1mm#3. A word represents some operation on stack (and possible other effect such as printing to the console), for example the word 1 adds the number 1 on top of the stack, the word + performs the addition on top of the stack etc. The programmer can define his own words which can be seen as "functions" or rather procedures or macros (words don't return anything or take any arguments, they all just invoke some operations on the stack). A word is defined like this:

```
: myword operation1 operation2 ... ;
```

For example a word that computes and average of the two values on top of the stack can be defined as:

```
: average + 2 / ;
```

Built-in words include:

GENERAL:

+	add	a b -> (a + b)
-	subtract	a b -> (b - a)
*	multiply	a b -> (a * b)
/	divide	a b -> (b / a)
=	equals	a b -> (-1 if a = b else 0)
<	less than	a b -> (-1 if a < b else 0)
>	greater than	a b -> (-1 if a > b else 0)
mod	modulo	a b -> (b % a)
dup	duplicate	a -> a a
drop	pop stack top	a ->
swap	swap items	a b -> b a
rot	rotate 3	a b c -> b c a
.	print top & pop	
key	read char on top	
.s	print stack	
emit	print char & pop	
cr	print newline	
cells	times cell width	a -> (a * cell width in bytes)
depth	pop all & get d.	a ... -> (previous stack size)
bye	quit	

VARIABLES/CONSTS:

variable X	creates var named X (X is a word that pushed its addr)
N X !	stores value N to variable X
N X +!	adds value N to variable X
X @	pushes value of variable X to stack
N constant C	creates constant C with value N
C	pushes the value of constant C

SPECIAL:

()	comment (inline)
\	comment (until newline)
." S "	print string S
X if C then	if X, execute C // only in word def.
X if C1 else C2 then	if X, execute C1 else C2 // only in word def.
do C loop	loops from stack top value to stack second from, top, special word "i" will hold the iteration val.
begin C until	like do/loop but keeps looping as long as top = 0
begin C while	like begin/until but loops as long as top != 0
allot	allocates memory, can be used for arrays
recurse	recursively call the word currently being defined

example programs:

```
100 1 2 + 7 * / . \ computes and prints 100 / ((1 + 2) * 7)
```

```
cr ." hey bitch " cr \ prints: hey bitch
```

```
: myloop 5 0 do i . loop ; myloop \ prints 0 1 2 3 4
```

How To

Source code files usually have .fs extension. We can use mentioned gforth to run our files. Let's create file my.fs; in it we write: { Hope the code is OK, I never actually programmed in Forth before. ~drummyfish }

```
: factorial
  dup 1 > if
    dup 1 - recurse *
  else
    drop 1
```



```
    then
;

5 factorial .

bye
```

We can run this simply with `gforth my.fs`, the programs should write 120.

foss

FOSS

FOSS (Free and Open Source Software, sometimes also FLOSS, adding *Libre*), is a kind of neutral term for software that is both free as in freedom and open source. It's just another term for this kind of software, as if there weren't enough of them :) People normally use this to stay neutral, to appeal to both free and open source camps or if they simply need a short term not requiring much typing. It's maybe also a little more vague acronym as in "I don't care, it has some kinda freeish license" and so it's possibly also a good label for a lot of "modern" software which can rarely be called a purely free software anymore -- most new projects are some sort of bloated bastard child of originally a free software project and some corporation's openwashing rapeware, so we just call it "FOSS".

fourier_transform

Fourier Transform

Fourier Transform (FT) is one of the most important transformations/algorithms in signal processing (and really in computer science and mathematics in general), which enables us to express and manipulate a signal (such as a sound or picture) in terms of frequencies it is composed of (rather than in terms of individual samples). It is so important because frequencies (basically sine waves) are actually THE important thing in signals, they allow us to detect things (voices, visual objects, chemical elements, ...), compress signals, modify them in useful ways (e.g. filter out noise of specific frequency band, enhance specific frequency bands, ...). There also exists a related algorithm called **Fast Fourier Transform** (FFT) which is able to compute one specific version of FT very quickly and so is often used in practice.

For newcomers FT is typically not easy to understand, it takes time to wrap one's head around it. There is also a very confusing terminology; there exist slightly different kinds of the Fourier Transform that are called by similar names, or sometimes all simply just "Fourier Transform" -- what programmers usually mean by FT is DFT or FFT. There also exist Fourier Transforms in higher dimensions (2D, 3D, ...) -- the base case is called one dimensional (because our input signal has one coordinate). All this will be explained below.

What FT does in essence: it transforms an input signal (which can also be seen as a function) from **time (also space) domain**, i.e. the usual representation that for each x says the sample value $f(x)$, to **frequency domain**, another function that for each frequency f says "how much of the frequency is present" (amplitude and phase). For example an FT of a simple sine wave will be a function with a single spike at the frequency of the sine wave. There is also an **inverse Fourier Transform** that does the opposite (transforms the signal from frequencies back to time samples). The time and frequency representations are EQUIVALENT in that either one can be used to represent the signal -- it turns out that even "weird" looking functions can be decomposed into just sums of many differently shifted and scaled sine waves. In the frequency domain we can usually do two important things we cannot do in time domain: firstly analyze what frequencies are present (which can help e.g. in voice recognition, spectral analysis, earthquake detection, music etc.) and also MODIFY them (typical example is e.g. music equalizer or compression that removes or quantizes some frequencies); if we modify the frequencies, we may use the inverse FT to get back the "normal" (time representation) signal back. Some things are also easier to do in the frequency domain, for example convolution becomes mere multiplication.

FT is actually just one of many so called **integral transforms** that are all quite similar -- they always transform the signal to some other domain and back, they use similar equation but usually use a different kind of function. Other integral transforms are for example **discrete cosine transformation** (DCT) or

wavelet transform. DCT is actually a bit simpler than FT, so if you are having hard time with FT, go check out DCT.

If you know linear algebra, this may help you understand what (D)FT really does: Imagine the signal we work with is a POINT (we can also say a vector) in many dimensional space; if for example we have a recorded sound that has 1000 samples, it is really a 1000 dimensional vector, a point in 1000 dimensional space, expressed as an "array" of 1000 numbers (vector components). A short note: since we consider a finite number of discrete samples here, we are actually dealing with what's called DISCRETE FT here, not the "vanilla" FT, but for now let's not diverge. (D)FT does nothing more than transforming from one vector basis ("system of coordinates", "frame of reference") to another basis; i.e. by default the signal is expressed in time domain (our usual vector basis), the numbers in the sound "array" are such because we are viewing them from the time "frame of reference" -- (D)FT will NOT do anything with to the signal itself (it is a vector/point in space, which will stay where it is, the recorded sound itself will not change), it will merely express this same point/vector from a different "point of view"/"frame of reference" (set of basis vectors) -- that of frequencies. That's basically how all the integral transforms work, they just have to ensure the basis they are transforming to is orthogonal (i.e. kind of legit, "usable") of course. In addition the FT equation is nothing complex, it literally just uses a **dot product** of the whole input signal with each checked frequency wave to find out how similar the signal is to that particular frequency, as dot product simply says "how similar two vectors are" -- really, think about the equation and you will see it's really doing just that.

TODO: alternatives (like performing FIR filtering without actually doing FT etc.)

Details

First let's make clearer the whole terminology around FT:

- **Fourier Series (FS):** Transforms a PERIODIC (repeating) signal into a DISCRETE (non-continuous) spectrum. We can see this spectrum also as an infinite SERIES of coefficients c_0, c_1, c_2 , etc. The input signal can generally be complex, in which case the output spectrum also has negative part (c_{-1}, c_{-2}, c_{-3} etc.) and shows us COMPLEX EXPONENTIALS (i.e. not mere sine waves) of the input signal; however if the input signal is real (probably most signals we practically deal with), the spectrum's negative part is symmetric to the positive part and the corresponding positive and negative complex exponentials always together give a sine wave, so for "normal" signals we can see the spectrum only being in the non-negative part and showing us the sine waves of the signal.
- **Fourier Transform (FT):** Generalization of FS to work on any signal, not just periodic ones, i.e. FT takes a NON-PERIODIC signal and transforms it into a CONTINUOUS spectrum. This is achieved simply by considering the period of the signal to be infinite -- the spectrum now becomes continuous exactly because the input is non-periodic (this is a relationship that generally holds); since the output is continuous, we now rather see it as a function rather than a series. Same as with FS the input can be complex (in which case the same implications apply), but we usually work with real signals.
- **Inverse Fourier Transform (IFT):** Does the opposite of FT, i.e. transforms the signal back from frequency domain to time domain.
- **Discrete Time Fourier Transform (DTFT)** (not to be confused with DFT!): Fourier Transform for DISCRETE (non-continuous) input signals (e.g. sound pressure captured only at specific points in time) -- since the input is discrete, the spectrum will be PERIODIC (this is another relationship that generally holds).
- **Discrete Fourier Series (DFS):** Version of FS for discrete (non-continuous) signals, transforms the input DISCRETE and PERIODIC signal to a spectrum (series of coefficients) of which there are infinitely many and are also PERIODIC (with the same period as the input signal).
- **Discrete Fourier Transform (DFT)** (not to be confused with DTFT!): Uses DFS to transform a FINITE DISCRETE signal to a FINITE DISCRETE spectrum (with the same period as the input) by simply "pretending" the finite input signal is actually repeating over and over and then, after the transform, only leaving in the first period of the result (since the rest is just repeating). **This is actually what programmers usually mean by Fourier Transform** because in computers we practically always only deal with finite discrete signals (i.e. arrays of data).
- **Fast Fourier Transform (FFT):** Computes DFT (NOT FT!) that's faster than the naive implementation, i.e. computing the equation that defines DFT as it's written has time complexity $O(n^2)$ while FFT improves this to $O(n * \log(n))$.

From now on we will implicitly be talking about DFT of a real function (we'll ignore the possibility of complex input), the most notable transform here.

The input to DFT is a real function, i.e. the time domain representation of the signal. The output is a complex valued function of frequency, i.e. the spectrum -- for each frequency it says a complex number whose magnitude and phase say the magnitude and phase of that frequency (a sine wave) in the signal (many programs will visualize just the magnitude part as that's usually the important thing, however keep in mind there is always also the phase part as well).

The general equations defining DFT and IDFT, for signal with N samples, are following

$$\text{DFT}[k] = \sum_{n=0}^{N-1} x[n] * e^{(-2 * i * \pi * k * n / N)}$$

$$\text{IDFT}[k] = 1/N * \sum_{n=0}^{N-1} x[n] * e^{(2 * i * \pi * k * n / N)}$$

OK, this is usually where every noob ragequits if he hasn't already because of all the pis and es and just generally ununderstandable mess of weird symbols etc. What the heck does this all mean? As said above, it's doing nothing else than dot product or vectors really: one vector is the input signal and the other vectors are the individual frequencies (sine waves) we are trying to discover in the signal -- this looks so complicated because here we are actually viewing the general version for a possible complex input signal, the *e to something* part is actually the above mentioned complex exponential, it is the exponential way of writing a complex number (see e.g. Euler's identity). Anyway, considering only real input signal, we can simplify this to a more programmer friendly form:

```
DFT:
  init DFT_real and DFT_imag to 0s

  for k = 0 to N - 1
    for n = 0 to N - 1
      angle = -2 * i * pi * k * n / N
      DFT_real[k] += x[n] * cos(angle)
      DFT_imag[k] += x[n] * sin(angle)

IDFT:
  init data to 0s

  for k = 0 to N - 1
    for n = 0 to N - 1
      angle = 2 * i * pi * k * n / N
      data[k] += DFT_real[n] * cos(angle) - DFT_imag[n] * sin(angle)

  data[k] /= N
```

Example: take a look at the following array of 8 kind of arbitrary values and what their DFT looks like:

	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
data:	5.00	4.71	6.00	6.54	1.00	2.29	0.00	-0.54
DFT:	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
	#	#	#	#	#	#	#	#
magn.:	25.00	12.74	1.00	7.33	1.00	7.33	1.00	12.74
phase:	0.00	-1.52	-1.57	-0.10	-3.14	0.10	1.57	1.52

```

-----
real:    25.00   0.70   0.00   7.30  -1.00   7.30  -0.00   0.70
imag.:    0.00 -12.72  -1.00  -0.72  -0.00   0.72   1.00  12.72

restored:
data:     5.00   4.71   6.00   6.54   1.00   2.29   0.00  -0.54

```

At the top we have the input data: notice the data kind of looks similar to a low-frequency sine wave, so the frequencies in the spectrum below are mostly low, but there's also some high frequency noise that's deforming the wave. For convenience here we show the spectrum values in both formats (magnitude/phase and real/imaginary part), but keep in mind it's just different formats of the same complex number values; for analysis we are mostly interested in the magnitude of the complex numbers as that shows as the amplitude of the frequency, i.e. the "amount" of the frequency in the signal. Here we notice the greatest peak is at frequency 0 -- this is a "constant" component, the lowest possible frequency that just represents a constant vertical offset of the signal (a constant number added to all samples); this component here is so big because our input signal doesn't really oscillate around the value 0 as it doesn't even go to negative values -- DFT sees this as our signal being shifted quite a lot "up". Frequencies 1 and 7 are the second biggest here: DFT is telling us the signal looks mostly like an addition of a sine wave with very low frequency and very high frequency (which it does), it doesn't see many middle value frequencies here. At the end we also see the original values computed back using IDFT, just to check everything is working as expected.

Here is the C code that generates the above, you may use it as a snippet and/or to play around with different inputs to see what their spectra look like (for "readability" we commit the sin of using floating point numbers here, implementation of DFT without floats is left as an exercise :}):

```

#include <stdio.h>
#include <math.h>

#define PI 3.141592
#define N (sizeof(data) / sizeof(double)) // size of input data
#define NUM_FORMAT "%6.2lf"
#define STR_FORMAT "%-10s"
#define DRAW_HEIGHT 6

double data[] = // enter input data here
{5.00, 4.71, 6.00, 6.54, 1.00, 2.29, 0.00, -0.54};

double dftR[N]; // real part of DFT
double dftI[N]; // imaginary part of DFT
double dftM[N]; // just for printing: magnitude of DFT
double dftA[N]; // argument (angle/phase) of DFT

void printArray(double *array)
{
    for (int i = 0; i < N; ++i)
        printf(" " NUM_FORMAT, array[i]);

    putchar('\n');
}

void drawArray(double *array, double scale)
{
    for (int y = 0; y < DRAW_HEIGHT; ++y)
    {
        printf(" ");

        for (int x = 0; x < N; ++x)
        {
            printf(" ");
            putchar(((int) array[x] * scale) >= (DRAW_HEIGHT - y) ? '#' : ' ');
        }

        putchar('\n');
    }
}

void printDft(void)

```

```

{
    printf(STR_FORMAT,"  magn.:"); printArray(dftM);
    printf(STR_FORMAT,"  phase:"); printArray(dftA);
    puts("  ----");
    printf(STR_FORMAT,"  real:"); printArray(dftR);
    printf(STR_FORMAT,"  imag.:"); printArray(dftI);
}

void dft(void)
{
    for (int i = 0; i < N; ++i)
    {
        dftR[i] = 0;
        dftI[i] = 0;
    }

    for (int k = 0; k < N; ++k)
    {
        for (int n = 0; n < N; ++n)
        {
            double angle = (-2 * PI * k * n) / N;
            dftR[k] += data[n] * cos(angle);
            dftI[k] += data[n] * sin(angle);
        }

        // just for printing also precompute magnitudes and phases
        dftM[k] = sqrt(dftR[k] * dftR[k] + dftI[k] * dftI[k]);
        dftA[k] = atan2(dftI[k],dftR[k]);
    }
}

void idft(void)
{
    for (int i = 0; i < N; ++i)
        data[i] = 0;

    for (int k = 0; k < N; ++k)
    {
        for (int n = 0; n < N; ++n)
        {
            double angle = (2 * PI * k * n) / N;
            data[k] += dftR[n] * cos(angle) - dftI[n] * sin(angle);
        }

        data[k] /= N;
    }
}

int main(void)
{
    drawArray(data,1);
    printf(STR_FORMAT,"data:"); printArray(data);

    puts("\nDFT:");
    dft();
    drawArray(dftM,0.25);
    printDft();
    idft();

    puts("\nrestored:");
    printf(STR_FORMAT,"data:"); printArray(data);

    return 0;
}

```

TODO: pictures, 2D version

fqa

Frequently Questioned Answers

TODO: figure out what to write here

fractal

Fractal

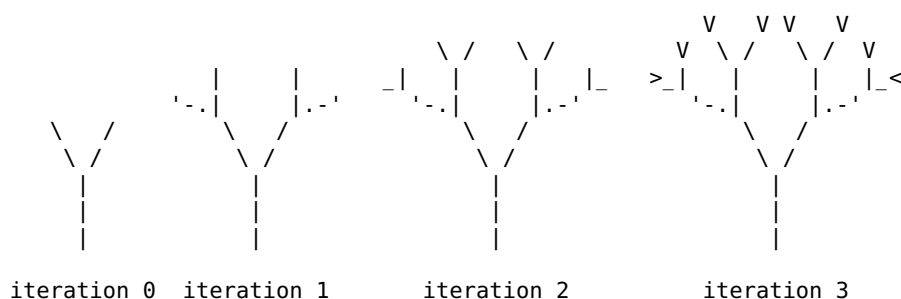
Informally speaking fractal is a shape that's geometrically "infinitely complex" while being described in an extremely simple way, e.g. with a very simple formula or algorithm. Shapes found in the nature, such as trees, mountains or clouds, are often fractals. Fractals show self-similarity, i.e. when "zooming" into an ideal fractal we keep seeing it is composed, down to an infinitely small scale, of shapes that are similar to the shape of the whole fractal; e.g. the branches of a tree look like smaller versions of the whole tree etc.

TODO: brief history

Fractals are the beauty of mathematics that can easily be seen even by non-mathematicians, so are probably good as a motivational example in math education.

Fractal geometry is a kind of geometry that examines these intricate shapes -- it turns out that unlike "normal" shapes such as circles and cubes, whose attributes (such as circumference, volume, ...) are mostly quite straightforward, perfect fractals (i.e. the mathematically ideal ones whose structure is infinitely complex) show some greatly unintuitive properties -- basically just as anything involving infinity they can get very tricky. For example a 2D fractal may have **finite area but infinite circumference** -- this is because the border is infinitely complex and swirls more and more as we zoom in, increasing the length of the border more and more the closer we look. This was famously notice e.g. when people tried to measure lengths of rivers or coastlines (which are sort of fractal shapes) -- the length they measured always depended on the length of the ruler they used; the shorter ruler you use, the greater length you get because the meanders of the details increase it. For this reason it is impossible to exactly and objectively give an exact length of such a shape.

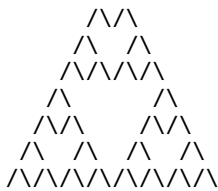
Fractal is formed by iteratively or recursively (repeatedly) applying its defining rule -- once we repeat the rule infinitely many times, we've got a perfect fractal. In the real world, of course, both in nature and in computing, the rule is just repeat many times as we can't repeat literally infinitely. The following is an example of how iteration of a rule creates a simple tree fractal; the rule being: *from each branch grow two smaller branches*.



Mathematically fractal is a shape whose Hausdorff dimension (the "scaling factor of the shape's mass") may be non-integer and is bigger than its topological dimension (the "normal" dimension such as 0 for a point, 1 for a line, 2 for a plane etc.). For example the Sierpinski triangle has a topological dimension 1 but Hausdorff dimension approx. 1.585 because if we scale it down twice, it decreases its "weight" three times (it becomes one of the three parts it is composed of); Hausdorff dimension is then calculated as $\log(3)/\log(2) \approx 1.585$.

L-systems are one possible way of creating fractals. They describe rules in form of a formal grammar which is used to generate a string of symbols that are subsequently interpreted as drawing commands (e.g. with turtle graphics) that render the fractal. The above shown tree can be described by an L-system. Among similar famous fractals are the Koch snowflake and Sierpinski Triangle.

/\



Sierpinski Triangle

Fractals don't have to be deterministic, sometimes there can be randomness in the rules which will make the shape be not perfectly self-similar (e.g. in the above shown tree fractal we might modify the rule to *from each branch grow 2 or 3 new branches*).

Another way of describing fractals is by iterative mathematical formulas that work with points in space. One of the most famous fractals formed this way is the **Mandelbrot set**. It is the set of complex numbers c such that the series $z_{next} = (z_{previous})^2 + c$, $z_0 = 0$ does not diverge to infinity. Mandelbrot set can nicely be rendered by assigning each iteration's result a different color; this produces a nice colorful fractal. Julia sets are very similar and there is infinitely many of them (each Julia set is formed like the Mandelbrot set but c is fixed for the specific set and z_0 is the tested point in the complex plain).

Fractals can of course also exist in 3 and more dimensions so we can have also have animated 3D fractals etc.

Fractals In Tech

Computers are good for exploring and rendering fractals as they can repeat given rule millions of times in a very short time. Programming fractals is quite easy thanks to their simple rules, yet this can highly impress noobs.

However, as shown by Code Parade (<https://yewtu.be/watch?v=Pv26QAOcb6Q>), complex fractals could be rendered even before the computer era using just a projector and camera that feeds back the picture to the camera. This is pretty neat, though it seems no one actually did it back then.

A nice FOSS program to interactively zoom into 2D fractals is e.g. xaos.

3D fractals can be rendered with ray marching and so called *distance estimation*. This works similarly to classic ray tracing but the rays are traced iteratively: we step along the ray and at each step use an estimate of the current point to the surface of the fractal; once we are "close enough" (below some specified threshold), we declare a hit and proceed as in normal ray tracing (we can render shadows, apply materials etc.). The distance estimate is done by some clever math.

Mandelbulber is a free, advanced software for exploring and rendering 3D fractals using the mentioned method.

Marble Racer is a FOSS game in which the player races a glass ball through levels that are animated 3D fractals. It also uses the distance estimation method implemented as a GPU shader and runs in real-time.

Fractals are also immensely useful in procedural generation, they can help generate complex art much faster than human artists, and such art can only take a very small amount of storage.

There exist also compression techniques based on fractals, see fractal compression.

There also exist such things as fractal antennas and fractal transistors.

frameless

Frameless Rendering

Frameless rendering is a technique of rendering animation by continuously updating an image on the screen by updating single "randomly" selected pixels rather than by showing a quick sequence of discrete frames. This is an alternative to the mainstream double buffered frame-based rendering traditionally used nowadays.

Typically this is done with image order rendering methods, i.e. methods that can immediately and independently compute the final color of any pixel on the screen -- for example with raytracing.

The main advantage of frameless rendering is of course saving a huge amount of memory usually needed for double buffering, and usually also increased performance (fewer pixels are processed per second). The animation may also seem more smooth and responsive -- reaction to input is seen faster. Another advantage, and possibly a disadvantage as well, is a **motion blur** effect that arises as a side effect of updating by individual pixels spread over the screen: some pixels show the scene at a newer time than others, so the previous images kind of blend with the newer ones. This may add realism and also prevent temporal aliasing, but blur may sometimes be undesirable, and also the kind of blur we get is "pixelated" and noisy.

Selecting the pixels to update can be done in many ways, usually with some pseudorandom selection (jittered sampling, Halton sequence, Poisson Disk sampling, ...), but regular patterns may also be used. There have been papers that implemented adaptive frameless rendering that detected where it is best to update pixels to achieve low noise.

Historically similar (though different) techniques were used on computers that didn't have enough memory for a double buffer or redrawing the whole screen each frame was too intensive on the CPU; programmers had to identify which pixels had to be redrawn and only update those. This resulted in techniques like *adaptive tile refresh* used in scrolling games such as Commander Keen.

framework

Framework

Software framework is a collection of tools such as environments, libraries, compilers and editors, that together allow fast and comfortable implementation of other software by plugging in relatively small pieces of code. While a simple library is something that's plugged as a helper into programmer's code, framework is a bigger system into which programmer plugs his code. Frameworks are generally bloated and harmful, LRS doesn't recommend relying on them.

free_body

Free Body

Free (as in freedom) body, also libre body, is a body of a human who allows (legally and otherwise) everyone some basic rights to it, such as the right to touch any of its part. This is a concept inspired by that of free software and free culture, just applied to one's body.

TODO: waiver like CC0 but for a body?

```
{ Made my waiver here https://codeberg.org/drummyfish/my_text_data/src/branch/master/body_waiver.txt.
~drummyfish }
```

See Also

- free universe
 - free software
 - free culture
-

free_culture

Free Culture

Free (as in freedom) culture is a movement aiming for the relaxation of intellectual property restrictions, mainly that of copyright, to allow free usage, reusing and sharing of artworks and other kind of information. Free culture argues that our society has gone too far in forcefully restricting the natural freedom of information by very strict laws (e.g. by authors holding copyright even 100 years after their death) and that we're hurting art, creativity, education and progress by continuing to strengthen restrictions on using, modifying (remixing) and sharing things like books, music and scientific papers. The word "free" in free culture refers to freedom, not just price -- free cultural works have to be more than just available gratis, they must also give its users some specific legal rights. Nevertheless free culture itself isn't against commercialization of art, it just argues for rather doing so by other means than selling legal rights to it. The opposite of free culture is permission culture (culture requiring permission for reuse of intellectual works).

The promoters of free culture want to relax intellectual property laws (copyright, patents, trademarks etc.) but also promote an ethic of sharing and remixing being good (as opposed to the demonizing anti-"piracy" propaganda of today), they sometimes mark their works with words "**some rights reserved**" or even "no rights reserved", as opposed to the traditional "all rights reserved".

Free culture is kind of a younger sister movement to the free software movement, in fact it has been inspired by it (we could call it its fork). While free software movement, established in 1983, was only concerned with freedoms relating to computer program source code, free culture later (around 2000) took its ideas and extended them to all information including e.g. artworks and scientific data. There are **clearly defined criteria** for a work to be considered free (as in freedom) work, i.e. part of the body of free cultural works. The criteria are very similar to those of free software (the definition is at <https://freedomdefined.org/Definition>) and can be summed up as follows:

A free cultural work must allow anyone to (legally and practically):

1. **Use it** in any way and for any purpose, even commercially.
2. **Study it**.
3. **Share it**, i.e. redistribute copies, even commercially.
4. **Modify it** and redistribute the modified copies, even commercially.

Some of these conditions may e.g. further require a source code of the work to be made available (e.g. sheet music, to allow studying and modification). Some conditions may however still be imposed, as long as they don't violate the above -- e.g. if a work allows all the above but requires crediting the author, it is still considered free (as in freedom). Copyleft (also share-alike, requirement of keeping the license for derivative works) is another condition that may be required. This means that many (probably most) free culture promoters actually rely and even support the concept of e.g. copyright, they just want to make it much less strict.

It was in 2001 when Lawrence Lessig, an American lawyer who can be seen as the movement's founder, created the Creative Commons, a non-profit organization which stands among the foundations of the movement and is very much connected to it. By this time he was already educating people about the twisted intellectual property laws and had a few followers. Creative Commons would create and publish a set of licenses that anyone could use to release their works under much less restrictive conditions than those that lawfully arise by default. For example if someone creates a song and releases it under the CC-BY license, he allows anyone to freely use, modify and share the song as long as proper attribution is given to him. It has to be noted that **NOT all Creative Commons licenses are free culture** (those with NC and ND conditions break the above given rules)! It is also possible to use other, non Creative Commons licenses in free culture, as long as the above given criteria are respected.

In 2004 Lessig published his **book** called Free Culture that summarized the topic as well as proposed solutions -- the book itself is shared under a Creative Commons license and can be downloaded for free (however the license is among the non-free CC licenses so the book itself is not part of free culture Imao, big fail by Lessig).

{ I'd recommend reading the Free Culture book to anyone whose interests lie close to free culture/software, it's definitely one of the essential works. ~drummyfish }

In the book Lessig gives an overview of the history of copyright -- it has been around since about the time of invention of printing press to give some publishers exclusive rights (an artificial monopoly) for printing and publishing certain books. The laws evolved but at first were not so restrictive, they only applied to very specific uses (printing) and for limited time, plus the copyright had to be registered. Over time corporations pressured to make it more and more restrictive -- nowadays copyright applies to basically everything and lasts for 70 years AFTER the death of the author (!!!). This is combined with the fact that in the age of computers any use of information requires making a copy (to read something you need to download it), i.e. copyright basically applies to ANY use now. I.e. both scope and term of copyright have been extended to the extreme, and this was done even AGAINST the US constitution -- Lessig himself tried to fight against it in court but lost. This form of copyright now restricts culture and basically only serves corporations who want to e.g. **kill the public domain** (works that run out of copyright and are now "free for everyone") by repeatedly prolonging the copyright term so that people don't have any pool of free works that would compete (and often win simply by being gratis) with the corporate created "content". In the books Lessig also mentions many hard punishments for breaking copyright laws and a lot of other examples of corruption of the system. He then goes on to propose solutions, mainly his Creative Commons licenses.

Free culture has become a relative success, the free Creative Commons licenses are now widely used -- **Wikipedia is one of the most famous examples of free culture** as it is licensed under the CC-BY-SA and its sister project Wikimedia Commons hosts over 80 million free cultural works! Openstreetmap is a free cultural collaborative project offering maps of the whole world, libregamewiki and opengameart are sites focused on creation of free cultural video games and game assets and there are many more. There are famous promoters of free culture such as Nina Paley, there exist webcomics, books, songs etc. In development of libre games free cultural licenses are used (alongside free software licenses) to liberate the game assets -- e.g. the Freedoom project creates free culture content replacement for the game Doom. Many scientists release their data to public domain under CC0. And of course, LRS highly advocated free culture, specifically public domain under CC0.

BEWARE of fake free culture: there are many resources that look like or even call themselves "free culture" despite not adhering to its rules. This may be by intention or not, some people just don't know too much about the topic -- a common mistake is to think that all Creative Commons licenses are free culture -- again, this is NOT the case (the NC and ND ones are not). Some think that "free" just means "gratis" -- this is not the case (free means freedom, i.e. respecting the above mentioned criteria of free cultural works). Many people don't know the rules of copyright and think that they can e.g. create a remix of some non-free pop song and license it under CC-BY-SA -- they CANNOT, they are making a derivative work of a non-free work and so cannot license it. Some people use licenses without knowing what they mean, e.g. many use CC0 and then ask for their work to not be used commercially -- this can't be done, CC0 specifically allows any commercial use. Some try to make their own "licenses" by e.g. stating "do whatever you want with my work" instead of using a proper waiver like CC0 -- this is with high probability legally unsafe and invalid, it is unfortunately not so easy to waive one's copyright -- DO use the existing licenses. Educate yourself and if you're unsure, ask away in the community, people are glad to give advice.

See Also

- free software
- free universe
- copyfree
- kopimi

freedom

Freedom

Only when man loses everything he becomes actually free. Freedom is about letting go.

TODO: basic definitions

People who seriously look for attaining mental/spiritual freedom often resort to asceticism, at least for a period of time (e.g. Buddha) -- this is very commonly not done with the intent of actually giving up all materialistic pleasures forevermore, but rather to let go of the dependency on the them, to know and see

that one really can live without them if needed so that one becomes less afraid of losing them, which is often what internally enslaves us. Without even realizing it we are nowadays addicted to many things (games, social media, overeating, shiny gadgets, ...) like an alcoholic is to booze; it is not necessarily bad to drink alcohol, but it is bad to be addicted to it -- to free himself the alcoholic needs to abstain from alcohol for a long period of time. Our chains are often within ourselves: for example we often don't have the freedom to say what we want to say because that might e.g. ruin our career, preventing us from enjoying our expensive addictions -- once we don't worry about this, we gain the freedom to say what we want. Once you rid yourself of fear of jail, you gain the freedom to do potentially illegal things, and so on. Additionally going through the experience of letting go of pleasures very often opens up your eyes and mind, new thoughts emerge and one reevaluates what's really important in life.

Freedom is something promised by most (if not all) ideologies/movements/etc.; this is because without further specification the term is so wide it says very little -- the very basic thing to know is, of course, that **there is no such thing as general freedom**; one kind of freedom restricts other kinds of freedom -- for example so called freedom of market says that a rich capitalist is free to do whatever he wants, which leads to him enslaving people, killing the freedom of those people.

What kind of freedom is LRS interested in? Basically the freedom for living beings to do what makes them happy -- of course this can't be achieved 100% (if one desires to enslave others, their freedom would disappear), however we can get very close (make a society in which people don't wish to enslave others). For this goal we choose to support such freedoms as free speech, free software, free culture, free love etc.

See Also

- FreeDoom

free_hardware

Free/Freedom-Friendly Hardware

Free (as in freedom) hardware is a form of ethical hardware aligned with the philosophy of free (as in freedom) software, i.e. having a free licensed designed that allows anyone to study, use, modify and share such designs for any purpose and so prevent abuse of users by technology. Let us note the word *free* refers to user freedom, not price! Sometimes the term may be more broadly and not completely correctly used even for hardware that's just highly compatible with purely free software systems -- let us rather call these a **freedom friendly hardware** -- and sometimes people misunderstand the term *free* as meaning "gratis hardware"; to avoid misunderstandings GNU recommends using the term **free design hardware** or **libre hardware** for free hardware in the strict sense, i.e. hardware with free licensed design. Sometimes -- nowadays maybe even more often -- the term "open source" hardware or *open hardware* with very similar meaning is encountered, but that is of course a harmful terminology as open source is an inherently harmful capitalist movement ignoring the ethical question of freedom -- hence it is recommended to prefer using the term free hardware. Sometimes the acronym FOSH (free and open source hardware) is used neutrally, similarly to FOSS.

GNU, just like us, highly advocates for free hardware, though, unlike with software, they don't completely reject using non-free hardware nowadays, not just for practical reasons (purely free hardware basically doesn't exist), but also because hardware is fundamentally different from software and it is possible to use *some* non-free hardware (usually the older one) relatively safely, without sacrificing freedom. The FSF issues so called **Respects Your Freedom** (RYF) certification for non-malicious hardware products, both free and non-free, that can be used with 100% free software (even though RYF has also been a target of some criticism of free software activists).

We, LRS, advocate for more strict criteria than just a free-licensed hardware design, for example we prefer complete public domain and advocate high simplicity which is a prerequisite of true freedom -- see less retarded hardware for more.

The topic of free hardware is a bit messy, free hardware definition is not as straightforward as that of free software because hardware, a physical thing, has some inherently different properties than software and it is also not as easy to design and create so it evolves more slowly than software and it is much more difficult to

create hardware completely from the ground up. Now consider the very question "what even is hardware"? There is a grey area between hardware and software, sometimes we see firmware as hardware, sometimes as software, sometimes pure software can be hardwired into a circuit so it basically behaves like hardware etc. Hardware design also has different levels, a higher level design may be free-licensed but its physical implementation may require existing lower level components that are non-free -- does such hardware count as free or not? How much down does free go -- do peripherals have to be free? Do the chips have to be free? Do the transistors themselves have to be free? We have to keep these things in mind. While in the software world it is usually quite easy to label a piece of software as free or not (at least legally), with hardware we rather tend to speak of different levels of freedom, at least for now.

Existing Free And Freedom-Friendly Hardware And Firmware

{ I'm not so much into hardware, this may be incomplete or have some huge errors, as always double check and please forgive :) Report any errors you find, also send me suggestions, thanks. ~drummyfish }

TODO, WORK IN PROGRESS, UNDER CONSTRUCTION

The following is a list of hardware whose design is **at least to some degree** free/open (i.e. for example free designs that however may be using a non-free CPU, this is an issue discussed above):

- **Arduino:** Extremely popular single board microcontrollers that can be easily used to make various devices. Designs and software tools are free, however the name Arduino is trademarked AND the hardware designs are using existing proprietary components, e.g. the AVR MCUs, i.e. Arduino is not 100% free from the ground up, but the degree of freedom is high and the hardware is kind of simple, i.e. friendly to tinkering and hacking.
- **RISC-V:** Big project creating a free-licensed instruction set architecture, usable by anyone for anything etc. (however the RISC-V brand is trademarked). A number of free CPUs/SOC implementations exist (alongside many proprietary implementations), for example PicoRV32 or Sodor.

The following is a list of some "freedom friendly" hardware, i.e. hardware that though partly or fully proprietary is not or can be made non-malicious to the user (has documented behavior, allows fully free software, librebooting, battery replacement, repairs etc.):

- **Agon:** Simple game console.
- **Ben NanoNote:** tiny GNU/Linux laptop whose design is free, however it utilizes e.g. a proprietary CPU.
- **DragonBox Pyra:** Upcoming small handheld computer running GNU/Linux that *almost* meets the RYF criteria, schematics will be available, GPU drivers are sadly proprietary. Successor to OpenPandora.
- **Librem 5:** WARNING, this device has been criticized a lot. It's an "open"/privacy-friendly smartphone with free-licensed design running GNU/Linux, however it uses proprietary firmware (loaded from secondary CPU to sneakily comply with RYF) and the functionality is, according to reviews, horrible.
- **MNT Reform:** "Open hardware" (free-licensed design but using proprietary components) laptop with NXP ARM CPU and Vivante GPU that can run with free drivers, has no camera or microphone. Pretty expensive.
- **Talos ES:** Very simple usable CPU that can be made at home.
- **Old Thinkpad laptops:** Old thinkpads such as X200, T400 and T500 are construction-wise superior to maybe any other laptop ever made, however despite being proprietary they are compatible with libreboot and can be purchased with Intel ME CPU backdoor disabled, offering complete control over the device, plus they can be bought relatively cheap. Very popular, some even certified "Respects Your Freedom" by the FSF.
- **OLinuxino:** TODO
- **OpenPandora:** Game console/tiny computer.
- **Open consoles** such as Arduboy, Pokitto and Gamebuino usually utilize a lot of simple free hardware such as Arduino, provide schematics, free libraries and encourage hacking.
- **Raspberry Pi** is not really free hardware but with free firmware such as librerpi it can be quite freedom friendly.
- **Other proprietary laptops:** many mostly older laptops are freedom friendly, e.g. Asus C201 Chromebook. You can usually find these in the libreboot compatibility list.
- **Pinephone:** Another "free/open" smartphone running GNU/Linux, probably better than Librem5, also uses some proprietary firmware (e.g. for Wifi), design is only source-available.

- **Ringo MakerPhone**: Educational Arduino dumbphone running on free software, by [Circuitmess](#). { I own one, is a bit buggy but works for calls and messages. ~drummyfish }
- **Ronja**: Device for optical communication using ethernet protocol.
- **Uzebox**: Very simple TV game console.
- ...

The following is a list of [firmware](#), [operating systems](#) and software tools that can be used to liberate freedom-friendly proprietary devices:

- **coreboot, libreboot, GNU boot, nonGeNUine Boot etc.**: More or less libre replacements for proprietary [BIOS](#) in personal computers. Different projects here take different roads and tolerate different amounts of non-free binary blobs, just as different [Linux](#) distros, so check out each one to pick whichever you like best.
- **librerpi**: Libre boot firmware for [RPI](#).
- **PostmarketOS**: Mobile [GNU/Linux](#) distribution that can be used to liberate smartphones.
- **Replicant**: Fork of [Android](#) mobile OS that replaces proprietary components with free software, can be used to liberate smartphones, though it is still bloat.
- **Rockbox**: Free firmware for digital audio players allowing replacement of the proprietary firmware and even improving on functionality and [GUI](#).
- ...

See Also

- [free software](#)
- [salvage computing](#)
- [RYF](#)
- [public domain computer](#)
- [less retarded hardware](#)

free

Free

In our community, as well as in the wider tech and some non-tech communities, the word free is normally used in the sense of [free as in freedom](#), i.e. implying freedom, not price. The word for "free of cost" is [gratis](#) (also *free as in beer*). To prevent this confusion the word [libre](#) is sometimes used in place of *free*, or we say *free as in freedom*, *free as in speech* etc.

free_software

Free Software

Not to be [confused](#) with [open source](#).

Free (as in freedom) software is a type of ethical [software](#) that's respecting its users' freedom and preventing their abuse, generally by availability of its source code AND by a [license](#) that allows anyone to use, study, modify and share the software without restricting conditions (such as having to pay or get explicit permission from the author). Free software is NOT equal to software whose source code is just available publicly or software that is offered for zero price, the basic legal rights to the software are the key attribute that has to be present. Free software stands opposed to [proprietary software](#) -- the kind of abusive, closed software that [capitalism](#) produces by default. Free software is not to be confused with [freeware](#) ("gratis", software available for free); although free software is always available for free thanks to its definition, zero price is not its goal. The goal is freedom.

Free software is also known as *free as in freedom*, *free as in speech* software or *libre* software. It is sometimes equated with [open source](#), even though open source is fundamentally different ([evil](#)), or neutrally labelled FOSS or FLOSS (free/libre and open-source software); sadly free software has lost to open source in mainstream popularity. In contrast to free software, software that is merely gratis (freeware) is sometimes

called *free as in beer*.

Examples of free software include the GNU operating system (also known as "Linux"), GIMP (image editor), Stockfish chess engine, or games such as Xonotic and Anarch. Free software is actually what runs the world, it is a standard among experts and it is possible to do computing with exclusively free software (though this may depend on how far you stretch the definition), even though most normal people don't even know the term free software exists because they only ever come in contact with abusive proprietary consumer software such as Windows and capitalist games. There also exists a lot of big and successful software, such as Firefox, Linux (the kernel) or Blender, that's often spoken of as free software which may however be only technically true or true only to a big (but not full) degree: for example even though Linux is 99% free, in its vanilla version it comes with proprietary binary blobs which breaks the rules of free software. Blender is technically free but it is also capitalist software which doesn't really care about freedom and may de-facto limit some freedoms required by free software, even if they are granted legally by Blender's license. Such software is better called "open source" or FOSS because it doesn't meet the high standards of free software. This issue of technically-but-not-really free software is addressed by some newer movements and philosophies such as suckless and our less retarded software who usually also aim for unbloating technology so as to make it more free in practice.

Though unknown to common people, the invention and adoption of free software has been **one the most important events in the history of computers** -- mere technology consumers nowadays don't even realize (and aren't told) that what they're using consists and has been enabled possibly mostly by software written non-commercially, by volunteers for free, basically on communist principles. Even if consumer technology is unethical because the underlying free technology has been modified by corporations to abuse the users, without free software the situation would have been incomparably worse if Richard Stallman hadn't achieved the small miracle of establishing the free software movement. Without it there would probably be practically no alternative to abusive technology nowadays, everything would be much more closed, there would probably be no "open source", "open hardware" such as Arduino and things such as Wikipedia. If the danger of intellectual property in software wasn't foreseen and countered by Richard Stallman right in the beginning, the corporations' push of legislation would probably have continued and copyright laws might have been many times worse today, to the point of not even being able to legally write free software nowadays. We have to be very grateful that this happened and continue to support free software.

Richard Stallman, the inventor of the concept and the term "free software", says free software is about ensuring the freedom of computer users, i.e. people truly owning their tools -- he points out that unless people have complete control over their tools, they don't truly own them and will instead become controlled and abused by the makers (true owners) of those tools, which in capitalism are corporations. Richard Stallman stressed that **there is no such thing as partially free software** -- it takes only a single line of code to take away the user's freedom and therefore if software is to be free, it has to be free as a whole. This is in direct contrast with open source (a term discourages by Stallman himself) which happily tolerates for example Windows only programs and accepts them as "open source", even though such a program cannot be run without the underlying proprietary code of the platform. It is therefore important to support free software rather than the business spoiled open source.

Free software is not about privacy! That is a retarded simplification spread by cryptofascists. Free software, as its name suggests, is about freedom in wide sense, which of course may include the freedom to stay anonymous, but there are many more freedoms which free software stands for, e.g. the freedom of customization of one's tools or the general freedom of art -- being able to utilize or remix someone else's creation for creating something new or better. Software focused on privacy is called simply privacy respecting software.

Is free software communism? This is a question often debated by Americans who have a panic phobia of anything resembling ideas of sharing and giving away for free. The answer is: yes and no. No as in it's not Marxism, the kind of evil pseudocommunism that plagued the world not a long time long ago -- that was a hugely complex, twisted violent ideology encompassing whole society which furthermore betrayed many basic ideas of equality and so on. Compared to this free software is just a simple idea of not applying intellectual property to software, and this idea may well function under some form of early capitalism. But on the other hand yes, free software is communism in its general form that simply states that sharing is good, it is communism as much as e.g. teaching a kid to share toys with its siblings.

Definition

Free software was originally defined by Richard Stallman for his GNU project. The definition was subsequently adopted and adjusted by other groups such as Debian or copyfree and so nowadays there isn't just one definition, even though the GNU definition is usually implicitly assumed. However, all of these definitions are very similar and are quite often variations and subsets of the original one. The GNU definition of free software is paraphrased as follows:

Software is considered free if all its users have the legal and de facto rights to:

0. Use the software for any purpose (even commercial or that somehow deemed unethical by someone).
1. Study the software. For this source code of the program has to be available.
2. Share the software with anyone.
3. Modify the software. For this source code of the program has to be available. This modified version can also be shared with anyone.

Note that as free software cares about real freedom, the word "right" here is seen as meaning a de facto right, i.e. NOT just a legal right -- legal rights (a free license) are required but if there appears a non-legal obstacle to those freedoms, free software communities will address them. Again, open source differs here by just focusing on legality.

To make it clear, freedom 0 (use for any purpose) covers ANY use, even commercial use or use deemed unethical by society of the software creator. Some people try to restrict this freedom, e.g. by prohibiting use for military purposes or prohibiting use by "fascists", which makes the software NOT free anymore. NEVER DO THIS. The reasoning behind freedom 0 is the same as that behind free speech: allowing any use doesn't imply endorsing or supporting any use, it simply means that we refuse to engage in certain kinds of oppression out of principle. Trying to mess with freedom 0 would be similar to e.g. prohibiting science on the ground of the fact that scientific results can be used in unethical ways -- we simply don't do this. We try to prevent unethical behavior in other ways than prohibiting basic rights.

Source code here means the preferred form in which software is modified, i.e. things such as obfuscated source code don't count as true source code.

The developers of Debian operating system have created their own guidelines (Debian Free Software Guidelines) which respect these points but are worded in more complex terms and further require e.g. non-functional data to be available under free terms as well (source), respecting also free culture, which GNU doesn't (source). The definition of "open source" is yet more complex even though in practice legally free software is eventually also open source and vice versa. The copyfree definition tries to be a lot more strict about freedom and forbids for example copyleft (which GNU promotes) and things such as DRM clauses (i.e. a copyfree license mustn't impose technology restrictions, even those seen as "justified", for similar reasons why we don't prohibit any kind of use for example).

History

Free software was invented by Richard Stallman in the 1980s. His free software movement inspired later movements such as the free culture movement and the evil open-source movement.

TODO: something here

By 2024 free software is dead -- yes, FSF and some free software "activists" are still around, but they don't bear any significance, just like the hippies lost any significance after 1960s etc. Corruption, politics and free market have finally killed the movement, open source prevailed and it is now redefining even the basic pillars of the four freedoms (partial openness or just source availability is now practically synonymous with "open source"), probably sealing the fate of technology, free software seems to have only postponed capitalist disaster by a few decades, which is still a great feat.

"Free" Software Alternatives, Pseudo Free Environments AKA What Freedom Really Is

The "**free software alternatives**" question is one that's constantly being discussed under capitalism: corporations try to forcefully keep users enslaved by proprietary software environments while free software proponents and users themselves want to free the users with "alternatives" made as free software. A very common mistake for a free software newcomer to make is to try to "**drop-in replace proprietary software with free software**"; a user used to proprietary software and its ways just wants the programs he's used to, just "without ads and subscriptions etc.". This doesn't work, or only to an extremely limited scale, because the whole proprietary world is made and DESIGNED from the ground up to allow user exploitation as much as possible, with e.g. building such thing like consumerism right into the design of visual elements of the software etc., i.e. proprietary vs free software is not just about a legal license, but whole philosophy of technology, asking things such as why are we so obsessed over "updates" or why are we freaking out about privacy. Trying to drop-in replace proprietary technology with 1 to 1 looking free software is like trying to replace whole capitalism with an "environment friendly capitalism" in which everything works the same except we have cars made of wood and skyscrapers made of recycled paper -- indeed, one sees that to get rid of the destructive nature of capitalism we really have to replace capitalism as such with all its basic concepts with something fundamentally different; and the situation is same with proprietary software.

For example most users nowadays want GUI in all programs, which is how they've been nurtured by capitalism, however we have to realize that **a truly (de facto, not just legally) free software has to be minimalist** and so most TRULY free software will mostly work only from the command line; a command line program is not necessarily harder or less comfortable to use (users are just nurtured to think so by capitalism), it is however inherently more free than a GUI one in all ways (not only by being more flexible, efficient, portable and non-discrimination, but also simpler and therefore e.g. modifiable by more people). We have to realize that a **freedom respecting computing environment INHERENTLY LOOKS DIFFERENT from the proprietary one**, the matter is NOT only about the license (free license is just a necessary condition to allow freedom under capitalism, however it is not a sufficient condition for freedom). Some projects calling themselves "free" (or rather "open source") make the mistake (sometimes intentionally, exactly to e.g. more easily pull over more users from the proprietary land) of simply mimicking proprietary ways 1 to 1 -- see e.g. Fediverse ("free" facebook/twitter/etc.), Blender etc. -- these are technically/legally free, but not actually, de-facto free. While a short-sighted view tells us this wins more users from the proprietary platforms, in long term we see we are just rebuilding dystopias, only painted with brighter colors so as to make them look friendlier (and oftentimes this is exactly the aim of the authors). Transitioning to TRULY free platforms is harder -- **one has to relearn basic things** such as, as has been mentioned, working with command line rather than GUI -- but ultimately right as one really gets more freedom, however under capitalist pressure and nurturing it is a hard thing to do, requiring extorting a lot of energy to resist the pressures of society.

After some years dealing with software freedom (in serious ways, making money doesn't count) many -- including us -- realize that the "licensing" fuss and legal questions, though important, are the surface, shallow views of freedom; one that also gets exploited by many (see e.g. openwashing). Those who seek real freedom will sooner or later find themselves focusing on minimalism and simplicity, e.g. LRS, suckless, Bitreich etc. Going yet further, one starts to see the inherent interconnections of technology and whole society, and has to become interested also in social concepts, hence our proposal of less retarded society.

See Also

- free hardware
- open source
- free culture
- creative commons
- copyfree

free_speech

Free Speech

Freedom of speech means there are no arbitrary government or anyone else imposed punishments for or obstacles (such as censorship) to merely talking about anything, making any public statement or publication of any information. **Free speech has to be by definition absolute and have no limit**, otherwise it's not free speech but controlled, limited speech -- trying to add exceptions to free speech is like trying to limit to whom a free software license is granted; doing so immediately makes such software non-free. **Free speech also comes with zero responsibility** exactly by definition, as responsibility implies some forms of punishment; free speech means exactly one can say anything without fearing any burden of responsibility. Freedom of speech is an essential attribute of a mature society, sadly it hasn't been widely implemented yet and with the SIW cancer the latest trend in society is towards eliminating free speech rather than supporting it (see e.g. political correctness). Speech is being widely censored by extremist groups (e.g. LGBT and corporations, see also cancel culture) and states -- depending on country there exist laws against so called "hate speech", questioning official versions of history (see e.g. Holocaust denial laws present in many EU states), criticizing powerful people (for example it is illegal to criticize or insult that huge inbred dick Thai king), sharing of useful information such as books (copyright censorship) etc. Free speech nowadays is being eliminated by the strategy of creating an exception to free speech, usually called "hate speech", and then classifying any undesired speech under such label and silencing it.

The basic principle of free speech says that **if you don't support freedom of speech which you dislike, you don't support free speech**. I.e. speech that you hate does not equal hate speech.

Free speech is based on the observation that firstly limiting speech is extremely harmful, and secondly that **speech itself never harms anyone**, it is only actions that harm and we should therefore focus on the actions themselves. A death threat or call for someone's murder doesn't kill -- sure, it may lead to someone being killed, but so may for example playing sports. If any kind of speaking leads to people dying, you have a deep issue within your society that definitely does NOT lie in not applying enough censorship; trying to solve your issue with censorship here is like trying to solve depression by physically deforming the depressed man's face into a smile and pretending he's OK. Offending someone by pointing out he's an idiot also doesn't count as speech causing harm, it's just a sad case of someone who is unable to bear hearing truth (or a lie), in which case he shouldn't be listening to people any more than someone with epilepsy should be watching seizure inducing videos.

Some idiots (like that xkcd #1357) say that free speech is only about legality, i.e. about what's merely allowed to be said by the law or what speech the law "protects". Of course, **this is completely wrong** and just reflects this society's obsession with law; true free speech mustn't be limited by anything -- if you're not allowed to say something, it doesn't matter too much what it is that's preventing you, your speech is not free. By the twisted logic of "free speech with consequences" you always have free speech, even in North Korea -- you aren't PHYSICALLY prevented to speak, you just have to bear responsibility for your speech, in this case a bullet. A bullet is a bullet, be it from a government gun or a drug cartel gun, a gun pointed at one's face always makes one not want to talk, no matter who the gun belongs to. If for example it is theoretically legal to be politically incorrect and criticize the LGBT gospel but you de-facto can't do it because the LGBT fascist SIWs would cancel you and maybe even physically lynch you, your speech is not free. It is important to realize **we mustn't tie free speech to legal definition** (also considering that a good society aims to eliminate law itself), i.e. it isn't enough to make speech free only in legal sense, a **TRUE free speech plainly and simply means anyone can literally say what he wants without any fear at all**. Our goal is to make speech free culturally, i.e. teach people that we should let others speak freely, even those -- and especially those -- who we disagree with.

Free speech extends even to such actions as shouting "fire" in a crowded theatre. In a good society with free speech people don't behave like monkeys, they will not trust a mere shout without having a further proof of there actually being fire and even if they suspect there is fire, they will not panic as that's a retarded thing to do.

Despite what the propaganda says **there is no free speech in our society**, the only kind of speech that is allowed is that which either has no effect or which the system desires for its benefit. **Illusion of free speech is sustained by letting people speak until they actually start making a change** -- once someone's speech leads to e.g. revealing state secrets or historical truths (e.g. about Holocaust, human racism or government crimes -- see wikileaks) or to destabilizing economy or state, such speech is labeled

"harmful" in some way (hate speech, intellectual property violation, revealing of confidential information, instigating crime, defamation etc.), censored and punished. Even though nowadays just pure censorship laws are being passed on daily basis, even in times when there are seemingly no specific censorship laws and so it seems that "we have free speech" there always exist generic laws that can be fit to any speech, such as those against "inciting violence", "terrorism", "undermining state interests", "hate speech" or any other fancy issue, which can be used to censor absolutely any speech the government pleases, even if such speech has nothing to do with said causes -- it is enough that some state lawyer can find however unlikely possible indirect link to such cause: this could of course be well seen e.g. in the cases of Covid flu or Russia-Ukraine war. Even though there were e.g. no specific laws in European countries against supporting Russia immediately after the war started, government immediately started censoring and locking up people who supported Russia on the Internet, based on the above mentioned generic laws. These laws work on the same principle as backdoor in software: they are advocated as a "safety" "feature" and allow complete takeover of the system, but are mostly unused until the right time comes, to give the users a sense of being safe ("I've been using this backdoored CPU for years and nothing happened, so it's safe"); unlike with software backdoor though the law backdoor isn't usually removed after it has been exploited, people are just too stupid to notice this and governments can get away with keeping the laws in place, so they do.

See Also

- censorship

free_universe

Free Universe

Free universe (also "open" universe) is a free culture ("free as in freedom") fictional universe that serves as a basis/platform for creating art works such as stories in forms of books, movies or video games. Such a universe provides a consistent description of a fictional world which may include its history and lore, geography, characters, laws of physics, languages, themes and art directions, and possibly also assets such as concept art, maps, music, even ready-to-use 3D video game models etc. A free universe is essentially the same kind of framework which is provided by proprietary universes such as those of Star Wars or Pokemon, with the exception that free universe is free/"open", i.e. it comes with a free license and so allows anyone to use it in any way without needing explicit permission; i.e. anyone can set own stories in the universe, expand on it, fork it, use its characters etc. (possibly under conditions that don't break the rules of free culture). The best kind of free universe is a completely public domain one which imposes absolutely no conditions on its use. The act of creating fictional universes is called **world building**.

But if anyone is allowed to do anything with the universe and so possibly incompatible works may be created, then **what is canon?!** Well, anything you want -- it's the same as with proprietary universes, regardless of official canon there may be different groups of fans that disagree about what is canon and there may be works that contradict someone's canon, there is no issue here.

Existing free universes: existence of a serious project aiming purely for the creation of a free universe as its main goal is unknown to us, though there are some project of similar nature, for example "open"geofiction (<https://opengeofiction.net/>) creating a fictional map of the world, which is however proprietary (NC license), i.e. also an example of openwashing. Free universes may be spawned as a byproduct of other free works -- for example old public domain books of fiction, such as Flatland, or libre games such as FLARE, Anarch or FreeDink create a free universe. The MMORPG game Ryzom releases its lore and content under a free license, spawning a huge, rich and high quality fantasy universe (though the game's server isn't libre and hence the game as such isn't altogether free). Libre comics such as Pepper and Carrot (<https://www.peppercarrot.com/en/wiki>), Wuffle Comics (<https://web.archive.org/web/20200117033753/http://www.wufflecomics.com/>) and Phil from GCHQ (<https://phillfromgchq.co.uk/>) also give life to their own free universes. If you want to start a free universe project, go for it, it would be highly valued!

See Also

- free body

Free Will

You can do what you want, but you can't want what you want.

Free will is a logically erroneous egocentric belief that humans (and possibly other living beings) are special in the universe by possessing some kind of soul which may disobey laws of physics and somehow make spontaneous, unpredictable decisions according to its "independent" desires. Actually that's the definition of *absolute indeterminate* free will; weaker definitions are also possible, e.g. *volitional free will* means just that one's actions are determined internally, or for the purposes of law definitions based on one's sanity may be made. But here we'll focus on the philosophical definition as that's what most autism revolves around. The Internet (and even academic) debates of free will are notoriously retarded to unbelievable levels, similarly to e.g. debates of consciousness.

{ Sabine nicely explains it here https://yewtu.be/watch?v=zpU_e3jh_FY. ~drummyfish }

Free will is usually discussed in relation to **determinism**, an idea of everything (including human thought and behavior) being completely predetermined from the start of the universe. Determinism is the most natural and most likely explanation for the working of our universe; it states that laws of nature dictate precisely which state will follow from current state and therefore everything that will ever happen is only determined by the initial conditions (start of the universe). As human brain is just matter like any other, it is no exception to the laws of nature. Determinism doesn't imply we'll be able to make precise predictions (see e.g. chaos or undecidability), just that everything is basically already set in stone as a kind of unavoidable fate. Basically the only other possible option is that there would be some kind true randomness, i.e. that laws of nature don't specify an exact state to follow from current state but rather multiple states out of which one is "taken" at random -- this is proposed by some quantum physicists as quantum physics seems to be showing the existence of inherent randomness. Nevertheless **quantum physics may still be deterministic**, see the theory of hidden variables and superdeterminism (no, Bell test didn't disprove determinism). But **EVEN IF the universe is non deterministic, free will still CANNOT exist**. Therefore this whole debate is meaningless.

Why is there no free will? Because it isn't logically possible, just like e.g. the famous omnipotent God (could he make a toast so hot he wouldn't be able to eat it?). Either the universe is deterministic and your decisions are already predetermined, or there exists an inherent randomness and your decisions are determined by a mere dice roll (which no one can call a free will more than just making every decision in life based on a coin toss). In either case your decisions are made for you by something "external". Even if you follow a basic definition of free will as "acting according to one's desires", you find that your decisions are DETERMINED by your desires, i.e. something you did not choose (your desires) makes decisions for you. There is no way out of this unless you reject logic itself.

For some reason retards (basically everyone) don't want to accept this, as if accepting it changed anything, stupid capitalists think that it would somehow belittle their "achievements" or what? Basically just like the people who used to let go of geocentrism. This is ridiculous, they hold on to the idea of their "PRECIOOOOUUUSS FREE WILL" to the death, then they go and consume whatever a TV tells them to consume. Indeed one of the most retarded things in the universe.

fsf

FSF

FSF stands for Free Software Foundation, a non-profit organization established by Richard Stallman with the goal of promoting and supporting free as in freedom software, software that respects its users' freedom.

History

TODO

In September 2019 Richard Stallman, the founder and president of the FSF, was cyberbullied and cancelled by SJW fascists for simply stating a rational but unpopular opinion on child sexuality and was forced to resign as a president. This might have been the last nail in the coffin for the FSF. The new president would come to be Geoffrey Knauth, an idiot who spent his life writing proprietary software in such shit as C# and helped built military software for killing people (just read his cv online). What's next, a porn actor becoming the next Pope? Would be less surprising.

After this the FSF definitely died.

See Also

- GNU
- OSI
- EFF
- Creative Commons

fuck

Fuck

FUCK

function

Function

Function is a very basic term in mathematics and programming with a slightly different meanings in each, also depending on exact context: mathematical function basically maps numbers to other numbers, a function in programming is similar but is rather seen as a subprogram to which we divide a bigger program. Well, that's pretty simplified but those are the very rough ideas. A more detailed explanation will follow.

Yet another attempt at quick summary: imagine function as a tiny box. In mathematics you throw numbers (or similar object, for example sets) to the box and it spits out other numbers (or "objects"); the number that falls out always only depends on the number you throw in. So the box basically just transforms numbers into other numbers. In programming a function is similar, it is also a box to which you throw numbers and can behave like the mathematical function, but the limitations are relaxed so the box can also do additional things when you throw a number in it, it may for example light up a light bulb; it may also remember things and sometimes spit out a different number when you throw in the same number twice.

Mathematical Functions

In mathematics functions can be defined and viewed from different angles, but it is essentially anything that assigns each member of some set A (so called *domain*) exactly one member of a potentially different set B (so called *codomain*). A typical example of a function is an equation that from one "input number" computes another number, for example:

$$f(x) = x / 2$$

Here we call the function f and say it takes one parameter (the "input number") called x . The "output number" is defined by the right side of the equation, $x / 2$, i.e. the number output by the function will be half of the parameter (x). The domain of this function (the set of all possible numbers that can be taken as input) is the set of real numbers and the codomain is also the set of real numbers. This equation assigns each real number x another real number $x / 2$, therefore it is a function.

Now consider a function $f_2(x) = 1 - 1/x$. Note that in this case the domain is the set of real numbers minus zero; the function can't take zero as an input because we can't divide by zero. The codomain is the set of real numbers minus one because we can't ever get one as a result.

Another common example of a function is the sine function that we write as $\sin(x)$. It can be defined in several ways, commonly e.g. as follows: considering a right triangle with one of its angles equal to x radians, $\sin(x)$ is equal to the ratio of the side opposing this angle to the triangle hypotenuse. For example $\sin(\pi/4) = \sin(45 \text{ degrees}) = 1/\sqrt{2} \approx 0.71$. The domain of sine function is again the set of real number but its codomain is only the set of real numbers between -1 and 1 because the ratio of said triangle sides can never be negative or greater than 1, i.e. sine function will never yield a number outside the interval $<-1,1>$.

Note that these functions have to satisfy a few conditions to really be functions. Firstly each number from the domain must be assigned exactly one number (although this can be "cheated" by e.g. using a set of couples as a codomain), even though multiple input numbers can give the same result number. Also importantly **the function result must only depend on the function's parameter**, i.e. the function mustn't have any memory or inside state and it mustn't depend on any external factors (such as current time) or use any randomness (such as a dice roll) in its calculation. For a certain argument (input number) a function must give the same result every time. For this reason not everything that transforms numbers to other numbers can be considered a function.

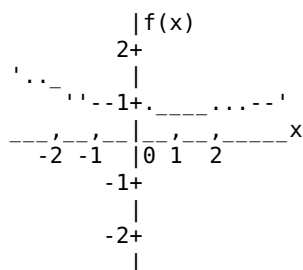
Functions can have multiple parameters, for example:

$$g(x,y) = (x + y) / 2$$

The function g computes the average of its two parameters, x and y . Formally we can see this as a function that maps elements from a set of couples of real numbers to the set of real numbers.

Of course function may also work with just whole numbers, also complex numbers, quaternions and theoretically just anything crazy like e.g. the set of animals :) However in these "weird" cases we generally no longer use the word *function* but rather something like a map. In mathematical terminology we may hear things such as a *real function of a complex parameter* which means a function that takes a complex number as an input and gives a real number result.

To get better overview of a certain function we may try to represent it graphically, most commonly we make function **plots** also called **graphs**. For a function of a single parameter we draw graphs onto a grid where the horizontal axis represents number line of the parameter (input) and the vertical axis represents the result. For example plotting a function $f(x) = ((x - 1) / 4)^2 + 0.8$ may look like this:



This is of course done by plotting various points $[x, f(x)]$ and connecting them by a line.

Plotting functions of multiple parameters is more difficult because we need more axes and get to higher dimensions. For functions of 2 parameters we can draw e.g. a heightmap or create a 3D model of the surface which the function defines. 3D functions may in theory be displayed like 2D functions with added time dimension (animated) or as 3D density clouds. For higher dimensions we usually resort to some kind of cross-section or projection to lower dimensions.

Functions can have certain properties such as:

- being **bijjective**: Pairs exactly one element from the domain with one element from codomain and vice versa, i.e. for every result (element of codomain) of the function it is possible to unambiguously say which input created it. For bijective functions we can create **inverse functions** that reverse the mapping (e.g. arcus sine is the inverse of a sin function that's limited to the interval where it is bijective). For example $f(x) = 2 * x$ is bijective with its inverse function being $f^{-1}(x) = x / 2$, but $f_2(x) = x^2$ is not bijective because e.g. both 1 and -1 give the result of 1.
- being an **even function**: For this function it holds that $f(x) = f(-x)$, i.e. the plotted function is symmetric by the vertical axis. Example is the cosine function.
- being an **odd function**: For this function it holds that $-f(x) = f(-x)$, i.e. the plotted function is symmetric by the center point [0,0]. Example is the sine function.
- being **differentiable**: Its derivative is defined everywhere.
- **recursive**: Referring to themselves in their own definition.
- ...

In context of functions we may encounter the term composition which simply means chaining the functions. E.g. the composition of functions $f(x)$ and $g(x)$ is written as $(f \circ g)(x)$ which is the same as $f(g(x))$.

Calculus is an important mathematical field that studies changes of continuous functions. It can tell us how quickly functions grow, where they have maximum and minimum values, what's the area under the line in their plot and many other things.

Mathematical functions can be seen as models of computation, i.e. something akin an "abstract computer": the field studying such functions is called computability theory. Here we may divide functions into classes depending on how "difficult" it is to compute their result.

Notable Mathematical Functions

Functions commonly used in mathematics range from the trivial ones (such as the constant functions, $f(x) = \text{constant}$) to things like trigonometric functions (sine, cosine, tangent, ...), factorial, logarithm, logistic sigmoid function, Gaussian function etc. Furthermore some more complex and/or interesting functions are (the term function may be applied liberally here):

- **Ackermann function**: Extremely fast growing function with some weird properties.
- **busy beaver function**: A famous, extremely quickly growing uncomputable function.
- **Minkowski's questionmark function**: Weird function with fractal properties.
- **sin(1/x)**: Simple function that gets chaotic close to zero.
- **Dirichlet function**: Function that can't really be plotted properly.
- **Weierstrass function**: Continuous everywhere, differentiable nowhere.
- **Drawing with function plots**: there are many formulas whose plots create pictures, for example the **Tupper's formula** is a self-referential formula that can be seen as a function which when plotted draws the text of the formula itself. There are also equations such as the Batman equation or an equation that draws the symbol of a heart, which can be seen as functions too.
- **Thomae's function**: Function with a nice fractal plot.
- **Cantor function**
- **Riemann zeta function**: Function that's subject to Riemann hypothesis, one of the most important unsolved problems in mathematics.
- **Blancmange curve**
- **space filling curves**
- **Dirac delta function**: Function representing infinitely short impulse with unit energy.
- TODO

{ Playing around with plotting 2D functions (functions with 2 parameters) is very fun, you can create beautiful pictures with very simple formulas. I once created a tool for this (just some dirty page with JavaScript) and found quite nice functions, for example: `gaussian_curve((x^2) mod (abs(sin(x + y)) + 0.001) + (y^2) mod (abs(sin(x - y)) + 0.001))` plotted from [-3,3] to [3,3] (plot with amplitude set to range from white to black by given minimum and maximum in the given area). ~drummyfish }

Programming Functions

In programming the definition of a function is less strict, even though some languages, namely functional ones, are built around purely mathematical functions -- for distinction we call these strictly mathematical functions **pure**. In traditional languages functions may or may not be pure, a function here normally means a **subprogram** which can take parameters and return a value, just as a mathematical function, but it can further break some of the rules of mathematical functions -- for example it may have so called **side effects**, i.e. performing additional actions besides just returning a number (such as modifying data in memory which can be read by others, printing something to the screen etc.), or use randomness and internal states, i.e. potentially returning different numbers when invoked (called) multiple times with exactly the same arguments. These functions are called **impure**; in programming a *function* without an adjective is implicitly expected to be impure. Thanks to allowing side effects these functions don't have to actually return any value, their purpose may be to just invoke some behavior such as writing something to the screen, initializing some hardware etc. The following piece of code demonstrates this in C:

```
int max(int a, int b, int c) // pure function
{
    return (a > b) ? (a > c ? a : c) : (b > c ? b : c);
}

unsigned int lastPresudoRandomValue = 0;

unsigned int pseudoRandom(unsigned int maxValue) // impure function
{
    lastPresudoRandomValue = // side effect: working with global variable
        lastPresudoRandomValue * 7907 + 7;

    return (lastPresudoRandomValue >> 2) % (maxValue + 1);
}
```

In older languages functions were also called procedures or routines. Sometimes there was some distinction between them, e.g. in Pascal functions returned a value while procedures didn't.

Just as in mathematics, a function in programming may be recursive -- here we define recursion as a function that calls itself.

fun

Fun

See also lmao.

Fun is a rewarding lighthearted satisfying feeling you get as a result of doing or witnessing something playful.

Things That Are Fun

This is subjective AF, even within a single man this depends on day, hour and mood. Anyway some fun stuff may include:

- the #capitalistchallenge: Try to win this game, you have as many shots as you want. Go to some tech store, seek the shop assistant and tell him you are deciding to buy one of two products, ask which one he would recommend. If he recommends the cheaper one you win.
- the fight culture drinking game: Watch some modern documentary, take a drink every time someone says the word *fight*. Harder mode: also drink when they say the word right.
- programming
- games such as chess, go and shogi, racetrack, even vidya gaymes (programming them and/or playing them), but only old+libre ones
- jokes
- open consoles, programmable calculators and fantasy consoles, cool embedded programming without bullshit (see also SAF)

- obfuscating C, steganography
- marble racing
- Netstalking
- Checking out offensive domains like nigger.com, retard.edu etc.
- trolling
- funny programming languages
- vandalizing Wikipedia, LMAO take a look at this <https://encyclopediadramatica.online/Vandal/How-to>
- hanging around with friends on the Island
- laughing at normies dealing with bloat
- randomly stumbling upon sites on wiby, wikiindex, finding politically incorrect stuff in old encyclopedias and generally just digging out obscure data
- old Nokia phones were fun
- cowsay
- autostereograms
- math and data visualizations, e.g. fractals, phase diagrams (<https://yt.artemislana.eu/watch?v=b-pLRX3L-fg>), strange attractors, procgen, cellular automata, plotting wild 2D functions, ...
- ...

furry

Furry

"Human seriously believing to be a dog not considered mental illness anymore." --21st century

Furriness is a serious mental disorder (dolphi will forgive :D) and fetish that makes people extremely creepily obsessed and/or identify with anthropomorphic animals (usually those with fur) far beyond any line of acceptability as a healthy personality trait, they often identify e.g. with cats, foxes or even completely made up species. To a big degree it's a sexual identity but those people just try to pretend (and possibly even believe) they're animals everywhere; not only do they have furry conventions, you just see furry avatars all over the internet, on issue trackers on programming websites and forums, recently zoomer kids started to even e.g. meow in classes because they identify as cats (this caused some huge drama somewhere in the UK). You cannot NOT meet a furry on the Internet. They usually argue it's "cute" and try to make no big deal of it, however that's a mask beyond which something horribly rotten lies. There is something more to furrydom, it's basically a cult that has taken an idea too far, kind of like anorexia takes losing weight a bit too far -- cuteness is OK, however furries are not really cute, they are CREEPY, they take this beyond healthy passion, you see the psychopathic stares in their faces, they take child cartoon characters and fantasize about them being transsexual and gore raping them and having children with them, some even attempt suicides if you insult their favorite characters etc.

Also the furry community -- it's extremely toxic, firstly as any big internet-centered group it's mostly completely infected with wokeness, LGBT+feminazism, which combined with the cult behavior may really lead to the community cyber pushing you to suicide if you e.g. question the gender of some child cartoon character (or even if you for example oppose the idea the character has to have some non-binary gender). A favorite hobby of furries is to destroy software project by pushing ugly woke furry mascots, threatening by suicide if the project doesn't accept them. Furries also seem to have a strong love of copyright so as to "protect" their shitty amateur art no one would want to copy anyway. Many create their own "fursonas" or "species" and then prohibit others from using them, they are so emotionally invested in this they may literally try to murder you if you do something to the drawing of their character. Stay away.

Furry porn is called yiff.

In the past we might have been wondering whether by 2020 we'd already have cured cancer, whether we'd have cities on Mars and flying cars. Well no, but you can sexually identify as a fox now.

See Also

- uwu
- retardedness

Future-Proof Technology

Future-proof technology is technology that is very likely to stay functional for a very long time with minimal to no maintenance, even considering significant changes in state of technology in society. In a world of relatively complex technology, such as that of computers, this feature is generally pretty hard to achieve; today's consumerist society makes the situation even much worse by focusing on immediate profit without long-term planning and by implementing things such as bloat, intentional introduction of complexity, obscurity, dependencies and planned obsolescence. But with good approach, such as that of LRS, it is very possible to achieve.

A truly good technology is trying to be future-proof because this saves us the great cost of maintenance and reinventing wheels and it gives its users comfort and safety; users of future-proof technology know they can build upon it without fearing it will suddenly break.

Despite the extremely bad situation not all hope is lost. At least in the world of software future-proofing can be achieved by:

- Free (as in freedom) software -- making your source code available, legally modifyable and shareable is a basic step towards making it easy to repair, backup and adopt to new technology (e.g. compile for new CPU architectures etc.).
- Building on top of already well established, time-tested and relatively simple technology such as the C language or comun. Choosing to use the older standards with fewer features helps greatly as the less-feature-rich versions of languages are always more supported (for example there is many more C89 compilers than C17 compilers) and can even be relatively simply reimplemented if needed. Another example is e.g. OpenGL -- you should use the oldest (simplest) version you can to make a program better future proof.
- Minimizing dependencies to absolute bare minimum and offering alternatives and fallbacks in cases where you can't avoid introducing a dependency (e.g. you should always offer an option for software rendering in any program that by default uses GPU for 3D graphics). Dependencies are likely the single greatest cause of software death because if one of your dependencies dies, you whole project dies, and this goes recursively for all of the dependencies of the dependencies etc. This usually means software libraries but also goes for other software such as build systems and also hardware dependencies such as requiring GPU, floating point, special instructions etc.
- Practicing minimalism and reducing complexity which minimizes the maintenance cost and therefore raises the probability of someone being able to fix any issues that arise over time. Minimalism is necessary and EXTREMELY important, bloat will make your program very prone to dying as it will depend on a big community of programmers that maintain it and such community will itself always be very prone to disappearing (internals disagreements, stopped funding, lose of interest, ...).
- Making your program portable -- this ensures your program can be adapted to new platforms and also that you use abstractions that untie you from things such as hardware dependencies.
- Generally just avoiding the hyped "modern" "feature-rich" (bloated) technology arising from the consumerist market.
- ...

See Also

- finished
- sustainability
- portability
- <https://unixsheikh.com/articles/how-to-write-software-that-will-keep-working-for-decades.html>

Game Engine

Game engine is a software, usually a framework or a library, that serves as a base code for games. Such an engine may be seen as a platform allowing portability and offering preprogrammed functionality often needed in games (3D rendering, physics engine, I/O, networking, AI, audio, scripting, ...) as well as tools used in game development (level editor, shader editor, 3D editor, ...).

A game engine differs from a general multimedia engine/library, such as SDL, by its specific focus on games. It is also different from generic rendering engines such as 3D engines like OpenSceneGraph because games require more than just rendering (audio, AI, physics, ...). While one may use some general purpose technology such as C or SDL for creating a game, using a game engine should make the process easier. However, **beware of bloat** that plagues most mainstream game engines. LRS advises against use of any frameworks, so try to at worst use a game library. Many game programmers such as Jonathan Blow advocate and practice writing own engines for one's games.

Existing Engines

The following are some notable game engines.

- **free as in freedom**

- ♦ **Allegro**: 2D C game library.
- ♦ **BRender**: Old 3D engine that used mainly software rendering, used e.g. in Carmageddon, later released under MIT.
- ♦ **Cube2**: 3D voxel outdoor shooter engine with real-time editable levels, used e.g. in Cube 2: Sauerbraten.
- ♦ **Godot**: A successful but bloated FOSS (MIT) framework engine, alternative to the proprietary Unity engine, written in C++, supports many platforms, has 3D/2D graphics and physics engines, scripting in many languages and many "advanced" features. Capitalist software.
- ♦ *id Tech* engines (engines by Id software)
 - ◊ **id Tech 0**: Simple 2D raycasting engine, written in ANSI C, used mainly in Wolf3D (1992).
 - ◊ **id Tech 1**: BSP rendering engine used mainly in Doom and Doom 2.
 - **Chocolate Doom**: Doom engine fork aiming to be very similar to the vanilla version.
 - **Crispy Doom**: Slight enhancement of Chocolate Doom: increased resolution (640x480) and removed hardcoded engine limits.
 - **GZDoom**: Another Doom fork, supports newer OpenGL etc.
 - **PrBoom**: Doom engine fork adding e.g. OpenGL support.
 - ◊ **id Tech 2**: 3D engine used mainly in Quake and Quake 2, in a modified form (GoldSrc, proprietary) also in Half Life, features both GPU accelerated and software rendering.
 - **Darkplaces**: Fork of id Tech 2, used e.g. in Xonotic.
 - ◊ **id Tech 3**: 3D engine used mainly in Quake 3, sadly dropped software rendering support.
 - **ioquake3**: Fork of id Tech 3 aiming for bugfixes and improvements, e.g. SDL integration.
 - **OpenArena**: Game-specific fork of id Tech 3.
 - ◊ **id Tech 4**: 3D engine used mainly in Doom 3 and Quake 4.
 - **iodoom3**: Fork of id Tech 4, in a similar spirit to ioquake3.
- ♦ **Irrlicht**: C++ cross-platform library for 3D games, includes a physics engine and many rendering backends (OpenGL, software, DirectX, ...). Used e.g. by Minetest.
- ♦ **OpenMW**: FOSS remake of the engine of a proprietary RPG game TES: Morrowind, can be used to make open-world 3D RPG games.
- ♦ **Panda3D**: 3D game engine, under BSD, written in Python and C++.
- ♦ **pygame**: Python 2D game library.
- ♦ **Raylib**: C99 2D/3D game library, relatively minimalist.
- ♦ **SAF**: Official LRS library for tiny and simple portable games.
- ♦ **Torque3D**: 3D game engine in C++.

- **proprietary** (no go!):

- ♦ **Build Engine**: Old portal rendering "pseudo 3D" engine used mainly in 3D Realms games such as Duke3D. It is source available.
 - ♦ id Tech engines (engines by Id software)
 - ♦ **id Tech 5**: 3D engine used e.g. in Rage and some shitty Wolfenstein games.
 - ♦ **id Tech 6**: 3D engine adding Vulkan support, used e.g. in Doom 2016.
 - ♦ **id Tech 7**: 3D engine used e.g. in Doom: Eternal.
 - ♦ **Jedi engine**: old 90s "2.5D/Pseudo3D" engine best known for being used in **Dark Forces ** (Star Wars game).
 - ♦ **GameMaker**: Laughable toy for non-programmers.
 - ♦ **RAGE**: 3D open-world engine developed and used by Rockstar for games such as GTA.
 - ♦ **Source**: 3D engine by Valve used in games such as Half Life 2.
 - ♦ **Source2**: Continuation of Valve's source engine with added support of VR and other shit.
 - ♦ **Unity**: Shitty nub all-in-one 3D game engine, very bloated and capitalist, extremely popular among coding monkeys, includes ads.
 - ♦ **Unreal Engine**: One of the leading proprietary 3D game engines developed alongside Unreal Tournament games, EXTREMELY BLOATED and capitalist, known for hugely overcomplicated rendering (advertised as "photorealistic").
-

game

Game

Most generally game is a form of play which is restricted by certain rules, the goal of which is typically fun, providing challenge and/or competition (and sometimes more, e.g. education, training etc.). A game may have various combinations of mathematical/mental elements (e.g. competitive mental calculations, mathematically defined rules, ...), physical elements (based in real life physics, e.g. football, marble racing, ...) and even other types of elements (e.g. social, psychological, ...); nowadays very popular games are computer games, a type of video games (also gaymes or vidya, e.g. Anarch, minesweeper, Doom, ...), which are played with the help of a computer. An entity (human, computer, animal, ...) playing a game is called a player and his ability to play it well is called skill; however some games may involve pure randomness and chance which may limit or even eliminate the need of skill (e.g. rock paper scissors). *Game* is also a mathematical term in game theory which studies games and competition rigorously.

A fun take at the very concept of a game is Nomic, a game in which changing the game rules is part of the game. It leads to all kinds of mindfucks.

What does a good game look like? It is simple, LRS and beautiful, with only a few rules, but has great depth and provides endless hours of fun and challenge -- so called easy to learn, hard to master. A good game is free, owned by no one, belonging to the people, and lives its own life by relying on **self imposed goals** rather than "content consumption" in form of constant updates and centralized control by some kind of "owner" (as is the case with capitalist games) -- i.e. despite having a goal, the game doesn't try to hard force the player to do something, but rather opens up a nice environment (in which the main goal is but one of many fun things to do) for player's own creativity (once the player beats the game, he may e.g. try to beat it as fast as possible, play it with some deliberate limitation, try to play it as bad as possible, combine it with other games etc.). One such nice game is possibly racetrack.

Types Of Games

It's quite hard to exactly define what a game is, it is a fuzzy concept, and it is also hard to categorize games. Let us now define a simple classification of games by their basic nature, which will hopefully be suitable for us here:

- **mathematical games**: Games taking place in an abstract mathematical space, with exactly defined rules. Though mathematical games may of course be represented in real life (e.g. by physical chess pieces made of wood), such a representation is only a helper for the player and doesn't rule the game out of this category. Mathematicians try to *so/lve* these games in various ways, e.g. by trying to construct an algorithm for perfect play or proving that with perfect play one of the players can always secure a win.

- ♦ **computer games:** Mathematical games that practically REQUIRE a computer (and usually have been design as such) to be played due to the computations involved being very numerous and/or complex -- for example Doom.
- ♦ **non-computer mathematical games:** Mathematical games that do not require a computer (though of course their computer implementations may exist) as the calculations involved can be practically performed without it -- for example chess.
- **real life games:** Games taking place in real life, i.e. usually making use of real world physics or other laws (e.g. social ones) -- for example football or marble racing.
- **hybrid games:** Various combinations of mathematical and real life games, e.g. chess boxing.

Furthermore many different ways of division and classifications are widely used -- for example computer games may be divided as any other software (free vs proprietary, suckless vs bloat, ...), but also by many other aspect such as their genre, interface, platform etc. The following are common divisions we find usually among computer games, but often applicable to other typed of games also:

- by genre:
 - ♦ minigames
 - ♦ shooters
 - ♦ role playing
 - ♦ tower defenses
 - ♦ racing
 - ♦ platformers
 - ♦ strategy
 - ♦ adventures
 - ♦ sport
 - ♦ sandbox
 - ♦ ...
- by game design:
 - ♦ easy to learn, hard to master
 - ♦ hard to learn, easy to master
 - ♦ easy to learn, easy to master
 - ♦ hard to learn, hard to master
 - ♦ symmetric vs asymmetric gameplay
 - ♦ ...
- by number of players:
 - ♦ zero player
 - ♦ single player
 - ♦ multiplayer
 - ♦ massively multiplayer
- by information:
 - ♦ complete information
 - ♦ incomplete information
- by interface/graphics/world representation:
 - ♦ 2D
 - ♦ 3D
 - ♦ "pseudo3D"/primitive3D
 - ♦ command line/text
 - ♦ audio
 - ♦ ...
- by importance of skill:
 - ♦ purely skill based
 - ♦ involving chance
 - ♦ purely chance based
- by time management:
 - ♦ realtime
 - ♦ turn based
- by platform
 - ♦ real life
 - ♦ computer (console vs PC, ...)
 - ♦ pen and paper

- by budget/scale/financing:
 - ♦ hobbyist/amateur
 - ♦ indie
 - ♦ AAA
- by business model:
 - ♦ freeware
 - ♦ shareware
 - ♦ free to play
 - ♦ subscription, "software as a service"
 - ♦ buy once
 - ♦ pay to win
 - ♦ pay what you want/donation
 - ♦ adware
 - ♦ spyware
 - ♦ rapeware
 - ♦ ...
- ...

Computer Games

Computer game is most commonly understood to be software whose main purpose is to be played and, in most cases interactively, entertain the user; in a wider sense it may perhaps be anything we might call a game that happens to run on a computer (e.g. game theory games that serve research rather than entertainment etc.). Let us implicitly assume the former now. Sadly most such computer games are proprietary and toxic, as anything that's a subject of lucrative business under capitalism.

Among suckless software proponents there is a disagreement about whether games are legit software or just a meme and harmful kind of entertainment. The proponents of the latter argue something along the lines that technology is there only to get real work done, that games are for losers, that they hurt MUH PRODUCTIVITY, are an unhealthy addiction, wasted time and effort etc. Those in support of games as legitimate software see them as a valid form of relaxation, a form of art that's pleasant both to make and enjoy as a finished piece, and also a way to advancing technology along the way (note we are NOT talking about consumerist games here; any consumerist art is bad). Developing games has historically led to improvements of other kinds of software, especially e.g. 3D rendering, physics simulation and virtual reality. If games are done well, in a non-capitalist way, then **we, LRS, fully accept and support games as legitimate software**; of course as long as their purpose is to help all people, i.e. while we don't reject games as such, we reject most games the industry produces nowadays. We further argue that **in games it is acceptable to do what in real life is unethical** (even to characters controlled by other live players) and that this is in fact one of their greatest potential: to allow satisfying natural needs that were crucial in the jungle but became obsolete and harmful in advanced society, such as those for competition, violence, fascism, egoistic behavior and others -- provided the player can tell the difference between a game and real life of course. As such, games help us build a better society in which people can satisfy even harmful needs without doing actual harm; in a game it is acceptable to torture people, roleplay as a capitalist or even verbally bully other players in chat (who joined the server willingly knowing this is just a simulation, a roleplay), even though these things would be unacceptable to do in real life.

Despite arguments about the usefulness of games, most people agree on one thing: that the mainstream AAA games produced by big corporations (and nowadays basically just all commercial games, even the small ones, especially e.g. mobile games) are harmful, bloated, toxic, badly made and designed to be highly malicious, consumerist products whose sole purpose is to rape the user. They are one of the worst cases of capitalist software (rapeware). Such games are never going to be considered anywhere near good from our perspective (and even the mainstream is turning towards classifying modern games as shit), not even if they do some good.

PC games are mostly made for and played on MS Windows which is still the "gaming OS", even though in recent years we've seen a boom of "GNU/Linux gaming", possibly thanks to Windows getting shittier and shittier every year. While smallbrains see this as good, in fact it only leads to more windowization of GNU/Linux, i.e. games will just move to GNU/Linux, make it the new place of business and destroy it just as surely (indeed for example Valve is already raping it, by 2023 "Linux" is already almost unusable as it became more mainstream and popular). Many normies nowadays are practicing "mobile" or console gayming

which may be even worse, but really choosing between PC, consoles and phones is like choosing which kind of torture is best to endure before death. Sadly most games, even when played on GNU/Linux, are still proprietary, capitalist and bloated as hell. So yeah, the world of mainstream and even mainstream indie games is one big swamp that's altogether best to be avoided.

{ If you are really so broken that you HAVE TO play proprietary games to live a meaningful life, the least harmful way for everyone is to SOMEHOW GET YOUR HANDS ON old DOS games, or maybe games for some old consoles like gameboy, playstation 1 etc., or at worst some pre 2005 Windowzee gaymes, and play them in dosbox/wine or engine recreations like OpenMW etc. Yeah it's dirty, proprietary, non-free shit, but at least you don't need a supercomputer, you won't be tortured by ads, robbed by microthefts or bullied into consuming Internet. It's best if you just use this method to slowly rid yourself of your gayming addiction to be finally free. Also make sure to absolutely NEVER pay for a proprietary game -- NO, not even an indie one. Give the money to the homeless. ~drummyfish }

We might call this the **great tragedy of games**: the industry has become similar to the industry of **drug abuse**. Games feel great and can become very addictive, especially to people not aware of the dangers (children, retards, ...). Today not playing latest games makes you left out socially, out of the loop, a weirdo. Therefore contrary to the original purpose of a game -- that of making life better and bringing joy -- an individual "on games" from the capitalist industry will crave to constantly consume more and more "experiences" that get progressively more expensive to satisfy. This situation is purposefully engineered by the big game producers who exploit psychological and sociological phenomena to enslave *gamers* and make them addicted. Games become more and more predatory and abusive and of course, there are no moral limits for corporations of how far they can go: games with microthefts and lootboxes, for example, are similar to gambling, and are often targeted at very young children and people prone to gambling addictions. The game industry cooperates with the hardware and software industry to together produce a consumerist hell in which one is required to constantly update his hardware and software and to keep spending money just to stay in. The gaming addiction is so strong that even the FOSS people somehow create a **mental exception** for games and somehow do not mind e.g. proprietary games even though they otherwise reject proprietary software. Even most of the developers of free software games can't mentally separate themselves from the concepts set in place by capitalist games, they try to subconsciously mimic the toxic attributes of such games (bloat, unreasonably realistic graphics and hardware demands, content consumerism, cheating "protection", language filters and safe spaces, ...).

Therefore it is crucial to stress that **games are technology like any other**, they can be exploiting and abusive, and so indeed all the high standards we hold for other technology we must also hold for games. Too many people judge games solely by their externals, i.e. gameplay, looks and general fun they have playing them. For us at LRS gameplay is but one attribute, and not even the one of greatest importance; factors such as software freedom, cultural freedom, sucklessness, good internal design and being future proof are even more important.

A small number of games nowadays come with a free engine, which is either official (often retroactively freed by its developer in case of older games) or developed by volunteers. Example of the former are the engines of ID games (Doom, Quake), example of the latter can be OpenMW (a free engine for TES: Morrowind) or Mangos (a free server for World of Warcraft). Console emulators (such as of Playstation or Gameboy) can also be considered a free engine for playing proprietary games.

Yet a smaller number of games are completely free (in the sense of Debian's free software definition), including both the engine and game assets. These games are called **free games** or **libre games** and many of them are clones of famous proprietary games. Examples of these probably (one can rarely ever be sure about legal status) include SuperTuxKart, Minetest, Xonotic, FLARE or Anarch. There exists a wiki for libre games at <https://libregamewiki.org> and a developer forum at <https://forum.freegamedev.net/>. Libre games can also be found in Debian software repositories. However WATCH OUT, all mentioned repositories may be unreliable!

{ NOTE: Do not blindly trust libregamewiki and freegamedev forum, non-free games occasionally DO appear there by accident, negligence or even by intention. I've actually found that most of the big games like SuperTuxKart have some licensing issues (they removed one proprietary mascot from STK after my report). Ryzom has been removed after I brought up the fact that the whole server content is proprietary and secret. So if you're a purist, focus on the simpler games and confirm their freeness yourself. Anyway, LGW is a good place to start looking for libre games. It is much easier to be sure about freedom of suckless/LRS games, e.g.

Anarch is legally safe practically with 100% certainty. ~drummyfish }

Some games are pretty based as they don't even require GUI and are only played in the text shell (either using TUI or purely textual I/O) -- these are called TTY games or command line games. This kind of games may be particularly interesting to minimalists, hobbyists and developers with low (zero) budget, little spare time and/or no artistic skills. Roguelike games are especially popular here; there sometimes even exist GUI frontends which is pretty neat -- this demonstrates how the Unix philosophy can be applied to games.

Another kind of cool games are computer implementations of non-computer games, for example chess, backgammon, go or various card games. Such games are very often well tested and fine-tuned gameplay-wise, popular with active communities and therefore fun, yet simple to program with many existing free implementations and good AIs (e.g. GNU chess, GNU go or Stockfish). What's more, they are also many times completely public domain!

{ There is a great lost world of nice old-style games that used to be made for old dumb phones with Java (J2ME) -- between about 2000 and 2010 there were tons and tons of quality Java mobile games that had e.g. entire magazines dedicated solely to them. These games are mostly lost and impossible to find, even videos of them, but if you can somehow get your hands on some of those old magazines, you're in for a great nostalgia trip. Check out e.g. *Stolen in 60 seconds*, *Alien Shooter 3D*, *Gangstar* (GTA clone), *Playman World Soccer*, *Paid to Kill*, *Tibia Online*, *Ancient Empires*, *Legacy* (dungeon crawler), *Townsmen*, *Juiced 3D*, *Midtown Madness* and myriad of others. ~drummyfish }

Games As LRS

Computer games can be suckless and just as any other software should try to adhere to the Unix philosophy. A LRS game should follow all the principles that apply to any other kind of such software, for example being completely public domain or aiming for high portability and getting finished. This is important to mention because, sadly, many people see games as some kind of exception among software and think that different technological or moral rules apply -- this is wrong.

If you want to make a simple LRS game, there is an official LRS C library for this: SAF.

A LRS game will be similar to any other suckless program, one example of a design choice it should take is the following: while mainstream games are built around the idea of having a precompiled engine that runs scripts written in some interpreted language, a **LRS/suckless game wouldn't use run-time scripts** but would rather have such "scripts" written as a part of the whole game's source code (e.g. in a file `scripts.h`), in the same language as the engine (typically C) and they would be compiled into the binary program. This is the same principle by which suckless programs such as dwm don't use config files but rather have the configuration be part of the source code (in a file `config.h`). Doing this in a suckless program doesn't really have any disadvantages as such program is extremely easy and fast to recompile, and it brings in many advantages such as only using a single language, reducing complexity by not needing any interpreter, not having to open and read script files from the file system and also being faster.

Compared to mainstream games, a LRS game shouldn't be a consumerist product, it should be a tool to help people entertain themselves and relieve their stress. From the user perspective, the game should be focused on the fun and relaxation aspect rather than impressive visuals (i.e. photorealism etc.), i.e. it will likely utilize simple graphics and audio. Another aspect of an LRS game is that the technological part is just as important as how the game behaves on the outside (unlike mainstream games that have ugly, badly designed internals and mostly focus on rapid development and impressing the consumer with visuals).

The paradigm of LRS gamedev differs from the mainstream gamedev just as the Unix philosophy differs from the Window philosophy. While a mainstream game is a monolithic piece of software, designed to allow at best some simple, controlled and limited user modifications, a LRS game is designed with forking, wild hacking, unpredictable abuse and code reuse in mind.

Let's take an example. A LRS game of a real-time 3D RPG genre may for example consist of several independent modules: the RPG library, the game code, the content and the frontend. Yes, a mainstream game will consist of similar modules, however those modules will probably only exist for the internal organization of work and better testing, they won't be intended for real reuse or wild hacking. With the LRS RPG game it is implicitly assumed that someone else may take the 3D game and make it into a purely

non-real-time command line game just by replacing the frontend, in which case the rest of the code shouldn't be burdened by anything 3D-rendering related. The paradigm here should be similar to that existing in the world of computer chess where there exist separate engines, graphical frontends, communication protocols, formats, software for running engine tournaments, analyzing games etc. Roguelikes and the world of quake engines show some of this modularity, though not in such a degree we would like to see -- LRS game modules may be completely separate projects and different processes communicating via text interfaces through pipes, just as basic Unix tools do. We have to think about someone possibly taking our singleplayer RPG and make it into an MMORPG. Someone may even take the game and use it as a research tool for machine learning or as a VFX tool for making movies, and the game should be designed so as to make this as easy as possible -- the user interface should be very simple to be replaced by an API for computers. The game should allow easy creation of tool assisted speedruns, to record demos, to allow scripting (i.e. manipulation by external programs, traditional in-game interpreted scripting may be absent, as mentioned previously), modifying ingame variables, even creating cheats etc. And, importantly, **the game content is a module as well**, i.e. the whole RPG world, its lore and storyline is something that can be modified, forked, remixed, and the game creator should bear this in mind (see also free universe).

Of course, LRS games must NOT contain such shit as "anti-cheating technology", DRM etc. For our stance on cheating, see the article about it.

Legal Matters

Thankfully gameplay mechanisms cannot (yet) be copyrighted (however some can sadly be patented) so we can mostly happily clone proprietary games and so free them. However this must be done carefully as there is a possibility of stepping on other mines, for example violating a trade dress (looking too similar visually) or a trade mark (for example you cannot use the word *tetris* as it's owned by some shitty company) and also said patents (for example the concept of minigames on loading screens used to be patented in the past).

Trademarks have been known to cause problems in the realm of libre games, for example in the case of Nexuiz which had to rename to Xonotic after its original creator trademarked the name and started to make trouble.

Advice on cloning games: copy only the gameplay mechanics, otherwise make it original and very different from the cloned game or else you're threading the fine legal lines. See this as an opportunity to add something new, something that's yours, and potentially to apply and exploit minimalism, i.e. if you're going to clone Doom, do not make a game about shooting demons from hell that's called Gnoom -- just take the gameplay and do something new, e.g. why not try to make it a mix of sci-fi and fantasy with procedurally generated levels which will additionally save you a lot of time on level design?

Nice And Notable Gaymes

Of proprietary video games we should mention especially those that to us have clonning potential. Doom (possibly also Wolfenstein 3d) and other 90s shooters such as Duke Nukem 3D, Shadow Warrior and Blood (the great 90s boomer shooters) were excellent. Trackmania is a very interesting racing game like no other, based on kind of speedrunning, easy to learn, hard to master, very entertaining even solo. The Witness was a pretty rare case of a good newer game, set on a strange island with puzzles the player learns purely by observation. The Elder Scrolls (mainly Morrowind, Oblivion and Skyrim) are very captivating RPG games like no other, with extreme emphasis on freedom and lore; Pokemon games on GBC and GBA were similar in this while being actually pretty tiny games on small old handhelds. GTA games also offered a great open world freedom and fun based on violence, sandbox world and great gangster-themed story. Advance Wars was a great turn based strategy on GBA (and possibly one of the best games on that console), kind of glorified chess with amazing pixel art graphics. Warcraft III was possibly the best real time strategy game with awesome aesthetics. Its successor, World of Warcraft, is probably the most notable MMORPG with the same lovely aesthetics and amazing feel that would be worth bringing over to the free world (even if just in 2D or only text). Diablo (one and two) were a bit similar to WoW but limited to singleplayer and a few man multiplayer; there exists a nice libre Diablo clone called Flare now. Legend of Grimrock (one and two) is another rare case of actually good new take on an old concept of dungeon crawlers. Half Life games are also notable especially for their atmosphere, storyline and lore. Minecraft was another greatly influential game that spawned basically a new genre, though we have now basically a perfect clone called Minetest. Dward Fortress is also worth mentioning as the "most complex simulation ever made" -- it would be nice to have a

free clone. TODO: more.

Gamebooks -- books that require the reader to participate in the story and make choices executed by jumping to different pages based on given choice -- are worthy of mention as an interesting combination of a book and a game, something similar to computer adventure games -- in gamebooks lies a great potential for creating nice LRS games.

As for the free (as in freedom) libre games, let the following be a sum up of some nice games that are somewhat close to LRS, at least by some considerations.

Computer games: Anarch and microTD are examples of games trying to closely follow the less retarded principles while still being what we would normally call a computer game. SAF is a less retarded game library/fantasy console which comes with some less retarded games such as microTD. If you want something closer to the mainstream while caring about freedom, you probably want to check out libre games (but keep in mind they are typically not so LRS and do suck in many ways). Some of the highest quality among them are Xonotic, 0 A.D., openarena, Freedoom, Neverball, SupertuxKart, Minetest, The Battle for Wesnoth, Blackvoxel etcetc. -- these are usually quite bloated though.

As for **non-computer games**: these are usually closer to LRS than any computer game. Many old board games are awesome, including chess, go, shogi, xiangqi, backgammon, checkers etc. Gamebooks can be very LRS -- they can be implemented both as computer games and non-computer physical books, and can further be well combined with creating a free universe. Some card games also, TODO: which ones? :) Pen and pencil games that are amazing include racetrack, pen and pencil football etc. Nice real life physics games include football, marble racing etc.

See Also

- minigame
- demake
- game engine
- brain software
- open console
- fantasy console
- SAF
- chess
- gamebook
- tangram
- game of life

game_of_life

Game Of Life

--> *Reveal the secret of life with these four simple rules!* <--

Game of Life (sometimes just *Life*) is probably the most famous cellular automaton (mathematical simulation set on a grid of cells), which with only four simple rules gives rise to a world with patterns that seem to behave similarly to simple biological life. It was invented in 1970 by John Conway, a famous mathematician who researched games, and has since gained popularity among programmers, mathematicians and other nerds as it's not only scientifically valuable, it is also awesome and fun to play around with as a kind of sandbox toy; it is very easy to program, play around with, modify and can be explored to great depth. The word *game* is used because what we have here is really a zero-player mathematical game, which is furthermore completely deterministic (there is no randomness), so we only choose some starting state of the world and then watch the game "play itself". Game of Life is similar systems such as rule 110 and Langton's ant.

```

. . . . . [] . . . . .
. . . [][] . . . . . [][] [] . . . . .
. . [] . . [] . . . . . [] . . . . .
. . . . . [][] . . . . . [][] [] . . . . .
. . [] . . [] . . . . . [] . . . . .
```


arise in the world, how they behave, interact, what statistical properties the world has, what simulations it can run etc.

Rule Implications And Properties

There are thousands of documented structures, kind of "life forms" or "species" behaving in interesting ways, that can exist in game of life, some appear commonly and naturally (from a randomly initialized start state), some are rare and often have to be manually created. Common classes of such structures include **still life** (structures that persist and won't disappear on their own, some may just persist without even changing -- possibly simplest such structure is a 2x2 block of live cells), **oscillators** (structures that stay in place and periodically change their appearance, one of the simplest is blinker, a 1x3 block of live cells) and **space ships** (structures moving through space on their own, one of the simplest and most famous is glider), further include for example *guns* (structures that produce, or "shoot" space ships), *puffers* (kind of spaceships that leave a trail behind them), *waves*, *rotors*, *crawlers* etc. Some patterns can self replicate (create an identical copy of themselves), some can grow without limit. Here are some basic life structures:

<pre> [] [] [][]</pre>	<pre> [][] [][]</pre>	<pre> [] [] []</pre>	<pre> [][] [] [] [][]</pre>	<pre> [][] [][] [] [] [][] [][]</pre>
glider	block	blinker	beehive	mirrored table
	<pre> [][] [][]</pre>	<pre> [][] [][] [] [] []</pre>	<pre> [][] [] []</pre>	
<pre> [][] [] [][] [] [] [] [] [] [] []</pre>	<pre> [][] [][] [] [] [] [] [] [] [] []</pre>	lightweight spaceship	beacon	
	<pre> [][] [] [] [] [] []</pre>		<pre> [][] [] []</pre>	
	pin wheel		toad	

A typical run of the game from a randomly initialized grid looks a bit similar to the evolution of our Universe: initially (after the Big Bang) the world exhibits a chaotic behavior, looking like a random noise of cells switching on and off seemingly without order, then, after a short while, more orderly patterns of the three basic kinds (still lives, oscillators and space ships) emerge and start interacting, either by being too close to each other or by shooting space ships and hitting each other -- this is a kind of "golden age" of life (interesting events, for example spontaneous emergence of symmetry). After some time this usually settles on a repeating set of states with just still life and oscillators, far enough from each other to influence each other (a kind of heat death of the universe).

Staying at analogies with our Universe, game of life also recognizes its analogy to **speed of light** (or *speed of life*), the fastest speed by which information can propagate through the game of life universe -- in the basic version this speed is simply one cell per turn (as any possible pattern, no matter how complex, can only ever influence its immediately neighboring cells in one turn). This speed is exploited by some algorithms to optimize the game's simulation.

As game of life is Turing complete, some things about it are undecidable, for example whether a given pattern will ever appear.

Statistical properties: the following experiments were performed in a world with 128x128 cells and wrapping space. From a random starting state with 50% live cells populations mostly seem to soon somewhat stabilize at around a little bit more than one third of cells being alive. The following shows 16 runs, noting percentage of live cells after each 256 steps (notice how in one case a population just dies out immediately and in another it very much struggles to stay alive):

run 1:	50%	41%	28%	30%	37%	32%	32%	30%	34%	28%	33%	35%	33%	34%	28%	36%	40%
run 2:	50%	42%	28%	38%	32%	32%	40%	43%	39%	26%	34%	35%	38%	25%	37%	29%	44%
run 3:	50%	35%	29%	35%	38%	34%	31%	33%	30%	32%	35%	34%	39%	45%	42%	34%	34%
run 4:	50%	30%	45%	28%	32%	25%	36%	34%	32%	44%	29%	28%	37%	34%	31%	30%	27%

```

run 5:  50% 29% 29% 23% 29% 37% 39% 28% 35% 28% 32% 43% 43% 20% 20% 31% 34%
run 6:  50% 25% 37% 75% 12% 12% 37% 12% 25% 37% 75% 12% 12% 37% 12% 25% 37%
run 7:  50% 75% 1%  1%  3%  1%  3%  6%  1%  3%  3%  6%  3%  6%  6%  6%  12%
run 8:  50% 0%  0%  0%  0%  0%  0%  0%  0%  0%  0%  0%  0%  0%  0%  0%  0%
run 9:  50% 42% 31% 35% 21% 31% 28% 32% 33% 35% 30% 35% 30% 33% 47% 30% 28%
run 10: 50% 35% 30% 25% 39% 30% 33% 35% 30% 33% 38% 30% 24% 35% 28% 31% 35%
run 11: 50% 37% 35% 29% 43% 42% 27% 37% 41% 39% 29% 36% 34% 34% 35% 42% 39%
run 12: 50% 34% 39% 43% 41% 37% 39% 37% 38% 32% 35% 30% 32% 33% 32% 32% 28%
run 13: 50% 25% 37% 75% 12% 12% 37% 12% 25% 37% 75% 12% 12% 37% 12% 25% 37%
run 14: 50% 29% 29% 23% 29% 37% 39% 28% 35% 28% 32% 43% 43% 20% 20% 31% 34%
run 15: 50% 27% 27% 32% 36% 36% 35% 35% 33% 22% 21% 40% 31% 29% 46% 34% 31%
run 16: 50% 33% 39% 37% 34% 43% 32% 38% 36% 45% 32% 35% 28% 35% 32% 35% 43%

```

Code/Programming

Programming a simple version of game of life is quite easy and may take just 10 minutes, however if we aim for greatly optimized efficiency (speed, memory) or features such as truly infinite space or reversible time, matters will of course get very complex. Let's start with the simple way.

The following is a simple C implementation of a wrapping version of game of life (i.e. the world is not actually infinite):

```

#include <stdio.h>

#define WORLD_SIZE 20

unsigned char world[WORLD_SIZE * WORLD_SIZE];

unsigned char getCell(int x, int y)
{
    return world[y * WORLD_SIZE + x] & 0x01;
}

void setCell(int x, int y)
{
    world[y * WORLD_SIZE + x] |= 0x01;
}

int main(void)
{
    unsigned char random = 30;

    for (int i = 0; i < WORLD_SIZE * WORLD_SIZE; ++i)
    {
        world[i] = random > 127;
        random = random * 13 + 22;
    }

    char in = 0;
    int step = 0;

    while (in != 'q')
    {
        unsigned char *cell = world;

        for (int y = 0; y < WORLD_SIZE; ++y)
        {
            int yU = y == 0 ? (WORLD_SIZE - 1) : (y - 1),
                yD = (y + 1) % WORLD_SIZE,
                xL = WORLD_SIZE - 1,
                xR = 1;

            for (int x = 0; x < WORLD_SIZE; ++x)
            {
                int neighbors =
                    getCell(xL,yU) + getCell(x,yU) + getCell(xR,yU) +
                    getCell(xL,y)  +                  getCell(xR,y)  +
                    getCell(xL,yD) + getCell(x,yD) + getCell(xR,yD);

                if ((*cell) & 0x01)

```

```

    {
        putchar('[');
        putchar(']');

        if (neighbors == 2 || neighbors == 3)
            *cell |= 0x02;
    }
    else
    {
        putchar('.');
        putchar(' ');

        if (neighbors == 3)
            *cell |= 0x02;
    }

    xL = x;
    xR = (x + 2) % WORLD_SIZE;

    cell++;
}

putchar('\n');
}

for (int i = 0; i < WORLD_SIZE * WORLD_SIZE; ++i)
    world[i] >>= 1;

printf("\nstep %d\n", step);
puts("press RETURN to step, Q to quit");

step++;
in = getchar();
}

return 0;
}

```

The world cells are kept in the world array -- each cell holds the current state in the lowest bit. We perform drawing and update of the world at the same time. Notice especially that while updating the cells, we mustn't overwrite the cell's current state until its neighbors have been updated as well! Not realizing this is a **common beginner error** and results in so called naïve implementation. We avoid this error by first storing each cell's next state in the second lowest bit (keeping its current state in the lowest bit), and then, after all cells have been updated, we iterate over all cells again and perform one bit shift to the left (making the computed next states into current states).

For real serious projects there exist highly optimized algorithms such as QuickLife and HashLife -- if you are aiming to create a state-of-the-art program, check them out. Here we will not be discussing them further as they are beyond the scope of this article.

Implementing infinite world: it is possible to program the game so that the world has no boundaries (or possibly has boundaries imposed only by maximum values of the used integer data type; these could of course be removed by using an advanced arbitrary size integer type). The immediate straightforward idea is to simply resize the world when we need more space, i.e. initially we allocate some space to the world (let's say 128x128 cells) and once a cell comes to life outside this area we resize it by allocating more memory -- of course this resizing should happen by some bigger step than one because the pattern will likely grow further (so we may resize e.g. from 128x128 right to 256x256). This can of course be highly inefficient, a single glider traveling far away in one direction may cause resizing the world to astronomical size; therefore more smartness can be applied, for example we may allocate spaces by big tiles (let's say 64x64) wherever they are needed (and of course deallocate/free the ones that no longer have any live cells) -- this will require a lot of code for managing the tiles and being able to actually quickly simulate such representation of the world. It would also be possible to have no world array at all but rather only keep a list of cells that are alive, each one storing its coordinates -- this might of course become inefficient for a big number of live cells, however good optimization could make this approach bearable; a basic optimization here would have to focus on very quick determination of each cell's neighbor count, which could be achieved e.g. by keeping the list of the cells sorted (e.g. from northwestmost to southeastmost). Another idea (used e.g. by the QuickLife

algorithm) is to use a dynamic tree to represent the world.

Some basic **optimization** ideas are following: firstly, as shown in the code above, even though we could theoretically only allocate 1 bit for each cell, it is better to store each cell as a whole byte or possibly a whole integer (which will help memory alignment and likely speed up the simulation greatly) -- this also comes with the great feature of being able to store the current state in the lowest bit and older states in higher bits, which firstly allows rewinding time a few states back (which as seen below will be useful further) and secondly we don't need an extra array for performing the cell updates. Next, as another simplest optimization, we may try to skip big empty areas of the world during the update (however watch out for the border where a new cell can spawn due to a neighboring pattern). We may take this further and also skip areas that contain static, unchanging still life -- this could all be done e.g. by dividing the world into tiles (let's say 64x64) and keeping a record about each tile. This can be taken yet further and also detect e.g. periodically repeating still life (such as blinkers); if for example we know a tile contains pattern that repeats with period 2 and we are able to rewind time one step back (which we can easily do, as shown above), we can simply do this step back in time instead of simulating the whole cell. Next we may try to use dynamic programming, e.g. caches and hash tables to keep results of recently performed big pattern simulations to reuse in the future, so that we don't have to simulate them again (i.e. for example remembering how a glider evolved from one frame to another so that next frame we simply copy-paste the result instead of actually simulating each cell again); HashLife algorithm is doing something like this. Also try to focus greatly on the bottle necks such as counting the cell's neighbors -- it will be greatly worth it if you speed this code up, even for the cost of using more memory, i.e. consider things like loop unrolling, function inlining and look up tables for counting the neighbors. Further speedup may be achieved by parallelization (multithreading, GPU, SIMD, ...) as isolated parts of the world may be simulated independently, though this will introduce hardware dependencies and bloat and is therefore discouraged.

Extensions, Modifications And Generalizations

Game of Life can be extended/generalized/modified in great number of ways, some notable are e.g.:

- **Larger Than Life (LTL)**: Extended neighborhood of cells; one variant is e.g. 9x9 neighbourhood, cell dies with 34 to 58 live neighbors, dead cell becomes live with 34 to 45 live neighbors. Produces interesting, more smooth, bubble-like patterns that can move in various angles.
- **Smooth Life**: Continuous generalization of Game of Life, time steps are still discrete.
- **Lenia**: A relatively recent, highly generalized continuous version of Game of Life -- it is yet more generalized version of Smooth Life where all variables are continuous, including space, time and states, AND furthermore adds multiple channels ("plains" of existence that interact with each other). This system produces incredible patterns and great many organisms.
- **different grid geometry, more states, additional rules, ...**: Slight modifications one can make to experiment, e.g. trying out hexagonal grid, triangular grid, hyperbolic space, 3D and higher dimensional grids, more states (e.g. cells that remember their age) etc. Modifying the base rules is also possible, creating so called life-like automata: the basic game of life is denoted as B3/S23 (born with 3, stays alive with 2 or 3), some life-like variants include e.g. High Life which adds a rule that a dead cell with 6 live neighbors comes alive (B36/S23) -- this gives rise to a new pattern known as *replicator*.

See Also

- polyworld
- Resnick's termite

gay

Gay

Homosexuality is a sexual orientation and disorder (of course, not necessarily a BAD one) which makes individuals sexually attracted primarily to the same sex, i.e. males to males and females to females. A homosexual individual is called gay (stylized as gaaaaaaaay), homo or even faggot, females are called lesbians. The word *gay* is also used as a synonym for anything bad. About 2% of people suffer from

homosexuality. The opposite of homosexuality, i.e. the normal, natural sexual orientation primarily towards the opposite sex, is called heterosexuality or being *straight*. Homosexuality is not to be confused with bisexuality -- here we are talking about pure homosexuality, i.e. a greatly prevailing attraction mostly towards the same sex.

For an unenlightened reader coming from the brainwashland: this article is not "offensive", it is just telling uncensored truth. Be reminded that LRS is not advocating any discrimination, on the contrary we advocate absolute social equality and love of all living beings, despite some having disorders or being weird. Your indoctrination has made you equate political incorrectness with oppression and hate; to see the truth, you have to unlearn this -- see for example our FAQ.

Unlike e.g. pedophilia and probably also bisexuality, **pure homosexuality is NOT normal**, it could be called a disorder (WE REPEAT, we love all people, even those with a disorder) -- of course the meaning of the word disorder is highly debatable, but pure homosexuality is firstly pretty rare, and secondly from the nature's point of view gay people wouldn't naturally reproduce, their condition is therefore equivalent to any other kind of sterility, which we most definitely would call a defect -- not necessarily a defect harmful to society (there are enough people already), but nonetheless a defect from biological point of view. Is it okay to have a defect? Of course it is. In this case society may even benefit because gayness prevents overpopulation. Homosexuality even behaves like a diseases -- not only does it bring a defect (sterility) to the bearer, it also spreads itself through the culture in form of a fashion, i.e. despite homosexuals not reproducing (at least not much) the number of them is increasing because homosexuality is transmitted through culture, through the Internet and other media.

You can usually tell someone's gay from appearance and/or his body language. Gay people are more inclined towards art and other sex's activities, for example gay guys are often hair dressers or even ballet dancers.

There is a terrorist fascist organization called LGBT aiming to make gay people superior to others, but more importantly to gain political power -- e.g. the power over language.

Is being gay a choice? Even though homosexuality is largely genetically determined, it may also be to some extent a choice, sometimes a choice that's not of the individual in question, a choice made at young age and irreversible at older age. Most people are actually bisexual to a considerable degree, with a *preference* of certain sex. When horny, you'd fuck pretty much anything. Still there is a certain probability in each individual of choosing one or the other sex for a sexual/life partner. However culture and social pressure can push these probabilities in either way. If a child grows up in a major influence of Youtubers and other celebrities that openly are gay, or promote gayness as something extremely cool and fashionable, you see ads with gays and if all your role models are gay and your culture constantly paints being homosexual as being more interesting and somehow "brave" and if the competition of sexes fueled e.g. by the feminist propaganda paints the opposite sex as literal Hitler, the child has a greater probability of (maybe involuntarily) choosing the gay side of his sexual personality.

{ I even observed this effect on myself a bit. I've always been completely straight, perhaps mildly bisexual when very horny. Without going into detail, after spending some time in a specific group of people, I found my sexual preference and what I found "hot" shifting towards the preference prevailing in that group. Take from that whatever you will. ~drummyfish }

Of course, we have nothing against gay people as we don't have anything against people with any other disorder -- **we love all people equally**. But we do have an issue with any kind of terrorist organization, so while we are okay with homosexuals, we are not okay with LGBT.

Are you gay? How can you tell? In doing so you should actually NOT be guided by your sexual desires -- as has been said, most people are bisexual and in sex it many times holds that what disgusts you normally turns you on when you're horny, i.e. if you're a guy and would enjoy sucking a dick, you're not necessarily gay, it may be pure curiosity or just the desire of "forbidden fruit"; this is quite normal. Whether you're gay is probably determined by what kind of LIFE partner you'd choose, i.e. what sex you can fall in a ROMANTIC relationship with. If you're a guy and fall in love with another guy -- i.e. you're passionate just about being with that guy (even in case you couldn't have sex with him) -- you're probably gay. (Of course this holds the other way around too: if you're a guy and enjoy playing with tits, you may not necessarily be straight.)

gaywashing

Gaywashing

TODO

geek

Geek

Geek is a wannabe nerd, it's someone who wants to identify with being smart rather than actually being smart. Geeks are basically what used to be called a *smartass* in the old days -- overly confident conformists occupying mount stupid who think soyence is actual science, they watch shows like Rick and Morty and Big Bang Theory, they browse Rational Wiki and reddit -- especially r/atheism, and they make appearances on r/iamverysmart -- they wear T-shirts with cheap references to 101 programming concepts and uncontrollably laugh at any reference to number 42, they think they're computer experts because they know the word Linux, managed to install Ubuntu or drag and drop programmed a "game" in Godot. Geeks don't really have their own opinions, they just adopt opinions presented on 9gag, they are extremely weak and don't have extreme views. They usually live the normal conformist life, they have friends, normal day job, wife and kids, but they like to say they "never fit in" -- a true nerd is living in a basement and doesn't meet any real life people, he lives on the edge of suicide and doesn't nearly complain as much as the "geek".

See Also

- soydev
-

gemin

Gemini

Gemini is a shitty pseudominimalist network soynet protocol for publishing, browsing and downloading files, a simpler alternative to the World Wide Web and a more complex alternative to gopher (by which it was inspired). It is a part of so called Smol Internet. Gemini aims to be a "modern take on gopher", adding some new "features" and bloat, it's also more of a toxic, SIW soydev version of gopher; gemini is to gopher a bit like what Rust it to C. The project states it wants to be something in the middle between Web and gopher but doesn't want to replace either (but secretly it wants to replace gopher). Gemini is for zoomers in programming socks, gopher is for the real neckbeards.

On one hand Gemini is kind of cool but on the other hand it's pretty shit, especially by **REQUIRING the use of TLS encryption** for "muh security" because the project was made by privacy freaks that advocate the *ENCRYPT ABSOLUTELY EVERYTHIIIIING* philosophy. This is firstly mostly unnecessary (it's not like you do Internet banking over Gemini) and secondly adds a shitton of bloat and prevents simple implementations of clients and servers. Some members of the community called for creating a non-encrypted Gemini version, but that would basically be just gopher. Not even the Web goes as far as REQUIRING encryption (at least for now), so it may be better and easier to just create a simple web 1.0 website rather than a Gemini capsule. And if you want ultra simplicity, we highly advocate to instead prefer using gopher which doesn't suffer from the mentioned issue.

See Also

- gopher
 - smol internet
 - Fediverse
-

gender_studies

Gender Studies

what the actual fuck

gigachad

Gigachad

Gigachad is like chad, only more so. He has an ideal physique and makes women orgasm merely by looking at them.

girl

Girl

See femoid.

githopping

Githopping

Githopping is a disease similar to distrohopping but applied to git hosting websites. The disease has become an epidemics after the Micro\$oft's take over of GitHub when people started protest-migrating to GitLab, however GitLab became shit as well so people started hopping to other services like Codeberg etcetc. and now they are addicted to just copying their code from one site to another instead of doing actual programming.

Cure: free yourself of any git hosting, don't centralize your repos on one hosting, use multiple git hostings as mirrors for your code, i.e. add multiple push remotes to your local git and with every push update your repos all over the internet. Just spray the internet with your code and let it sink in, let it be captured in caches and archive sites and let it be preserved. **DO NOT** tie yourself to any specific git hosting by using any non-git features such as issue trackers or specialized CLI tools such as github cli. **DO NOT** use git hosting sites as a social network, just stop attention whoring for stars and likes, leave this kind of shit to tiktokers.

git

Git

Git (name without any actual meaning) is a FOSS (GPL) version control system (system for maintaining and collaboratively developing source code of programs), currently the most mainstream-popular one (surveys saying over 90% developers use it over other systems). Git is basically a command line program allowing to submit and track changes of text files (usually source code in some programming language), offering the possibility to switch between versions, branches, detect and resolve conflicts in collaborative editing etc.

Git was created by Linus Torvalds in 2005 to host the development of Linux and to improve on systems such as svn. Since then it has become extremely popular, mainly thanks to GitHub, a website that offered hosting of git projects along with "social network" features for the developers; after this similar hosting sites such as GitLab and Codeberg appeared, skyrocketing the popularity of git even more.

It is generally considered quite a good software, many praise its distributed nature, ability to work offline etc., however diehard software idealists still criticize its mildly bloated nature and also its usage complexity -- it is non-trivial to learn to work with git and many errors are infamously being resolved in a "trial/error + google" style, so some still try to improve on it by creating new systems.

Is git literally hitler? Well, by suckless standards git IS bloated and yes, git IS complicated as fuck, however let's try to go deeper and ask the important questions, namely "does this matter so much?" and

"should I use git or avoid it like the devil?". The answer is actually this: it doesn't matter too much that git is bloated and you don't have to avoid using it. Why? Well, git is basically just a way of hosting, spreading and mirroring your source onto many git-hosting servers (i.e. you can't avoid using git if you want to spread your code to e.g. Codeberg and GitLab) AND at the same time git doesn't create a dependency for your project, i.e. its shittiness doesn't "infect" your project -- if git dies or if you simply want to start using something else, you just copy-paste your source code elsewhere, you put it on FTP or anything else, no problem. It's similar to how e.g. Anarch uses SDL (which is bloated as hell) to run on specific platforms -- if it doesn't hard-depend on SDL and doesn't get tied to it, it's OK (and actually unavoidable) to make use of it. You also don't even have to get into the complicated stuff about git (like merging branches and resolving conflicts) when you're simply committing to a simple one-man project.

Which git hosting to use? All of them (except for GitHub which is a proprietary terrorist site)! Do not fall into the trap of githopping, just make tons of accounts, one on each git hosting site, add multiple push remotes and just keep pushing to all of them -- EZ. Remember, git hosting sites are just free file storage servers, not social platforms or brands to identify with. Do NOT use their non-git "features" such as issue trackers, CI and shit. They want you to use them as "facebook for programmers" and become dependent on their exclusive "features", so that's exactly what you want to avoid, just abuse their platform for free file storage. Additional tip on searching for git hosting sites: look up the currently popular git website software and search for its live instances with some nice search engine, e.g. currently searching just gitea (or "powered by gitea", "powered by gogs", "powered by forgejo") on wiby returns a lot of free git hostings.

Alternatives

Here are some alternatives to git:

- **nothing**: If you don't have many people on the project, you can comfortably just use nothing, like in good old times. Share a directory with the source code and keep regular backups in separate directories, share the source online via FTP or something like that, let internet archive back you up.
- **svn**: The "main", older alternative to git, used e.g. by SourceForge, apparently suffers from some design issues.
- **mailing list**: Development happens over email, people just keep sending patches that are reviewed and potentially merged by the maintainers to the main code base. This is how Linux was developed before git.
- **darcs**: Alternative to git, written in Haskell, advertising itself as simpler.
- **lit** (previously known as *gut*): WIP LRS/suckless version control system.
- ...

How To Use Git For Solo/Small Projects (Cheatsheet)

TODO

- `git add files_you_changed; git commit -m "Update"; git push`
- `git pull`
- `git stash`
- `git rm file_to_remove`
- `git init`
- `git log`
- `git diff`
- `git apply diff_file`
- *weird error*: just look it up on stack overflow

Set Up Your Own Git Server

WIP

on server:

```
mkdir myrepo
cd myrepo
git init --bare
```

on client you can clone and push with ssh:

```
git clone ssh://user@serveraddress:/path/to/myrepo
cd myrepo
... # do some work
git commit -m "blablabla"
git push
```

you can also make your repo clonnable via HTTP if you have HTTP server (e.g. [Apache](#)) running, just have address `http://myserver/myrepo` point to the repo directory, then you can clone with:

```
git clone http://myserver/myrepo
```

IMPORTANT NOTE: for the HTTP clone to work you need to do `git update-server-info` on the server in the repo directory after every repo update! You can do this e.g. with a git hook or [cronjob](#).

See Also

- [GitHub](#)
- [shithub](#)

global_discussion

Global Discussion

This is a place for general discussion about anything related to our thing. To comment just edit-add your comment. I suggest we use a tree-like structure as shows this example:

- Hello, this is my comment. ~drummyfish
 - ◆ Hey, this is my response. ~drummyfish

If the tree gets too big we can create a new tree under a new heading.

General Discussion

gnu

GNU

GNU ("*GNU is Not Unix*", a [recursive](#) acronym) is a large software project started by [Richard Stallman](#), the inventor of [free \(as in freedom\) software](#), running since 1983 with the goal of creating, maintaining and improving a completely free (as in freedom) [operating system](#), along with other free [software](#) that computer users might need. The project doesn't tolerate any [proprietary](#) software (though it sadly tolerates proprietary data). GNU achieved its goal of creating their free operating system when a [kernel](#) named [Linux](#) became part of it in the 90s as the last piece of the puzzle -- the system should be called just GNU but is now rather known as GNU/Linux (watch out: most so called "Linux systems" nowadays aren't embraced by GNU as they diverge from GNU's strict policies on what the system should look like, only a handful of operating systems are recommended by GNU). However, the GNU project didn't end and continues to further develop the operating system, or rather a myriad of user software that runs under the operating system -- GNU develops a few of its projects itself and also offers hosting and support (such as free legal defense) for GNU projects developed by volunteers who dedicate their work to them. GNU gave rise to the [Free Software Foundation](#) and is one of the most important software projects in history of computing.

The mascot of GNU is literally gnu (wildebeest), it is available under a copyleft license. WARNING: ironically GNU is extremely protective of their brand's "intellectual property" and will rape you if you use the name GNU without permission (see the case of GNU boot). It's quite funny and undermines the whole project a bit.

The GNU/Linux operating system has several variants in a form of a few GNU approved "Linux" distributions such as Guix, Trisquel or Parabola. Most other "Linux" distros don't meet the strict standards of GNU such as not including any proprietary software. In fact the approved distros can't even use the standard version of Linux because that contains proprietary blobs, a modified variant called Linux-libre has to be used.

GNU greatly prefers GPL licenses, i.e. it strives for copyleft and largely recommends it, even though it will also accept projects under permissive licenses as those are still free. GNU also helps with enforcing these licenses legally and advises developers to transfer their copyright to GNU so that they can "defend" the software for them.

If we still have a bit of freedom in computing nowadays, it is largely to GNU -- this can't be stressed enough. But although GNU is great and has been one of the best things to happen in software ever, it also has many flaws, for example:

- **GNU programs are typically bloated** -- although compared to Windows GNU programs are really light as a feather and though GNU programs are also in many cases (but not always) quite optimized, their source code, judged from strictly suckless perspective, is mostly huge, which many view as a big issue (it's a common theme, there are jokes such as GNU actually meaning *Gigantic and Nasty but Unavoidable* and so on). This is likely because GNU chooses to battle proprietary programs, often by trying to beat them at their own game, so features are preferred over minimalism to stay competitive.
- **GNU also doesn't mind proprietary non-functional data** (e.g. assets in video games). This goes against free culture and many other free software groups, notably e.g. Debian. Justifications for this range from "data itself can't be harmful" (false), through "we just focus on software" to "we need GNU to be more popular" (i.e. compatible with proprietary games and so on). GNU is also generally **NOT supportive of free culture and even uses copyright to prohibit modifications of their propaganda texts**: the GFDL license they use for texts may contain sections that are prohibited from being modified and so are non-free by definition. They also try to "protect" their names, you can't use the name "GNU" without their permission and so on. This sucks big time and shows some of the movement's darker side.
- **GNU greatly pushes copyleft**, which we, as well as many others, oppose.

History

The project officially started on September 27, 1983 by Richard Stallman's announcement titled *Free Unix!*. In it he expresses the intent to create a free as in freedom clone of the operating system Unix, and calls for people to join his effort (he also uses the term free software here). Unix was a good, successful de-facto standard operating system, but it was proprietary, owned by AT&T, and as such restricted by licensing terms. GNU was to be a similar system, compatible with the original Unix, but free as in freedom, i.e. freely available and allowing anyone to use it, improve it and share it.

In 1985 Richard Stallman wrote the GNU Manifesto, similar to the original project announcement, which further promoted the project and asked people for help in development. At this point the GNU team already had a lot of software for the new system: a text editor Emacs, a debugger, a number of utility programs and a nearly finished shell and C compiler (gcc).

At this point each program of the project still had its own custom license that legally made the software free as in freedom. The differences in details of these licenses however caused issues such as legal incompatibilities. This was addressed in 1989 by Richard Stallman's creation of a universal free software license: GNU General Public License (GPL) version 1. This license can be used for any free software project and makes these projects legally compatible, while also utilizing so called copyleft: a requirement for derived works to keep the same license, i.e. a legal mechanism for preventing people from making copies of a free project non-free. Since then GPL has become the primary license of the GNU project as well as of other unrelated projects.

GNU Projects

GNU has developed an almost unbelievable amount of software, it has software for all basic and some advanced needs. As of writing this there are 373 software packages in the official GNU repository (at https://directory.fsf.org/wiki/Main_Page). Below are just a few notable projects under the GNU umbrella.

- [GNU Hurd](#) (OS kernel, alternative to [Linux](#))
- [GNU Compiler Collection](#) (gcc, compiler for [C](#) and other languages)
- [GNU C Library](#) (glibc, [C](#) library)
- [GNU Core Utilities](#) (coreutils, basic utility programs)
- [GNU Debugger](#) (gdb, [debugger](#))
- [GNU Binary Utilities](#) (binutils, programs for working with binary programs)
- [GNU Chess](#) (strong [chess](#) engine)
- [GNU Go](#) (go game engine)
- [GNU Autotools](#) ([build system](#))
- [CLISP](#) (common [lisp](#) language)
- GNU Pascal ([pascal](#) compiler)
- [GIMP](#) (image manipulation program, a "free [photoshop](#)")
- GNU Emacs ([emacs](#) text editor)
- [GNU Octave](#) ([mathematics](#) software, "free Matlab")
- [GNU Mediagoblin](#) (decentralized file hosting on the [web](#))
- GNU Unifont ([unicode](#) font)
- [GNU Privacy Guard](#) (gpg, OpenPGP encryption)
- ...

See Also

- [Free Software Foundation](#)
- [Richard Stallman](#)
- [GNG](#) (GNG is Not GNU)
- [copyleft](#)
- [free software](#)

golang

Go (Programming Language)

Go (also golang) is a compiled [programming language](#) advertised as the the "[modern](#)" successor to [C](#), it is co-authored by one of C's authors, [Ken Thompson](#), and has been worked on by [Rob Pike](#), another famous Unix hacker (who however allegedly went insane and has been really doing some crazy shit for years). Of all the new language go is one of the least [harmful](#), however it's still quite [shit](#). Some reasons for this are:

- It is developed by [Google](#) and presented as "[open-source](#)" (not [free software](#)).
- It downloads additional [dependencies](#) during compilation, a huge defect due to which it's been rejected e.g. from [HyperbolaBSD](#) (https://wiki.hyperbola.info/doku.php?id=en:philosophy:incompatible_packages).
- It has (classless) [OOP](#) features.
- It has [bloat](#) such as [garbage collection](#), built-in [complex number](#) type, [concurrency](#) and something akin a [package manager](#) ([go get/install](#)).
- It forces a programming style in which an opening function bracket ({) can't be on its own line. [LMAO](#)
- Huge standard library with shit like crypto, image and html.
- ...

Anyway, it at least tries to stay *somewhat* simple in some areas and as such is probably better than other modern languages like [Rust](#). It purposefully omits features such as [generics](#) or static type conversions, which is good.

How big is it really? The official implementation by Google has whopping 2 million lines of code of self hosted implementation -- that's ginormous but keep in mind Google would likely implement minesweeper in two million lines of code too, so it may say little. Size of specification may be more informative -- that one has about 130 pages (after converting the official HTML specs to pdf), that's a bit smaller than that of C (the pure language part has about 160 pages), so that's not bad.

As of february 2024 there is no [code of conduct](#) in the official repo, that's good too.

go

Go

WIP

This article is about the game of go, for programming language see [golang](#).

{ I am still learning the beautiful game of go, please excuse potential unintentional errors here.
~drummyfish }

Go (from Japanese *Igo*, "surrounding board game", also *Baduk* or *Wei-qi*) is possibly the world's oldest original-form two-player board game, coming from Asia, and is one of the most beautiful, elegant, deep and popular games of this type in history, whose cultural significance and popularity can be compared to that of chess, despite it largely remaining widely popular only in Asia (along with other games like shogi, or "Japanese chess"). There however, especially in Japan, go is pretty big, it appears a lot in anime, there are TV channels exclusively dedicated to go etc., though in Japan shogi (the "Japanese chess") is probably a bit more popular; nevertheless go is likely the most intellectually challenging board games among all of the biggest board games. **Go is a bit difficult to get into** (kind of like vim?) though the rules can be learned quite quickly; it is hard to make big-picture sense of the rule implications and it may take weeks to months before one can even call himself a beginner player. To become a master takes lifetime (or two).

{ There is a nice non-bloated site hosting everything related to go: Sensei's Library at <https://senseis.xmp.net/>. ~drummyfish }

Compared to chess (some purists dislike this, see <https://senseis.xmp.net/?CompareGoToChess>) the rules of go are much more simple -- which is part of the game's beauty (see easy to learn, hard to master) -- though the emergent complexity of those few rules is grandiose; so much so that to play the game well is usually considered more challenging than learning chess well, as there are many more possibilities and mere calculation is not enough to be strong, one needs to develop a strong intuition; this is also the reason why it took 20 more years for computers to beat the best humans in go than in chess. Many say that go is yet deeper than chess and that it offers a unique experience that can't be found anywhere else; go is more mathematical, something that just exists naturally as a side effect of logic itself, while chess is a bit of an arbitrary set of more complex rules fine-tuned so that the game plays well. The spirit of go is also more zen-like and peaceful: while chess simulates war (something more aligned with western mentality), go is more about dividing territory, one could even see it not as a battle but rather a creation of art, beautiful patterns (something better aligned with eastern mentality). Also the whole culture around go is different, for example there is a strong tradition of go proverbs that teach you to play (there also exist many joke proverbs).

From LRS point of view go is one of the best games ever, for similar reasons to chess (it's highly free, suckless, cheap, not owned by anyone, fun, mathematically deep, nice for programming while the game itself doesn't even require a computer etc.) plus yet greater simplicity and beauty.

Solving go: similarly to chess the full game of go seems unlikely to be solved -- the 19x19 board makes the game state tree yet larger than that of chess, but the much simpler rules possibly give a bigger hope for mathematical proofs. Smaller boards however have been solved: Erik van der Werf made a program that confirmed win for black on boards up to (and including) 5x5 (best first move in all cases being in the middle of the board). Bigger boards are being researched, but a lot of information about them is in undecipherable Japanese/Korean gibberish, so we leave that for the future.

A famous proverb about go goes like this: what is the most perfect game man ever invented? Chess! But what about go? Go existed long before man...

TODO: rating, programming, stats, programs and sites for playing, ...

Rules

The rules of go vary a bit more than those of chess, they are not as much unified, but usually the details don't play as much of a role because e.g. different scoring systems still mostly result in the same outcome of games. Here we'll describe possibly the most common rule set.

The game's **goal** is basically to surround a bigger territory than the enemy player. The formal rules are pretty simple, though their implications are very complex.

Go is played by a black and white player, black plays first (unlike in chess) and then both players take turns placing stones of one's own color on squares -- a square is the INTERSECTION of the lines on the board, NOT the place between them (consider the lines to be carved in stone, the intersection is where the stone stands with stability). The stones are all the same (there are no different types of stones like in chess) and they cannot move; once a stone is placed, it stays on its position until the end of the game, or until it is captured by the enemy player. The board size is **19x19**, but for students and quick games 13x13 and 9x9 boards are also used. As black plays first, he has a slight advantage; for this white gets bonus points at the end of the game, so called **komi**, which is usually set to be 6.5 points (the half point eliminates the possibility of a draw). Komi may differ depending on board size or a specific scoring system.

Any player can **pass** on his move, i.e. making a move isn't mandatory. However you basically always want to make a move, one only passes when he feels there is nothing more to be gained and the game should end. If both players pass consecutively, the game ends.

The game considers **4-neighborhoods**, NOT 8-neighborhood, i.e. squares that don't lie on board edges have 4 neighbors: up, right, bottom and left; diagonal squares are NOT neighbors.

Capturing: a player can capture a group of connected (through 4-neighborhoods) enemy player's stones by completely surrounding them, or more precisely by taking away all so called **liberties** of that group -- *liberty* is an empty square that's immediately neighboring with the group (note that liberties may lie even inside the group). If a player places his stone so that it removes the enemy group's last liberty, then the group is removed from the board and all its stones are taken as captured. It is possible to capture stones by a move that would otherwise be forbidden as suicide, if after the removal of the captured group the placed stone gains a liberty.

Suicide is forbidden: it is not allowed to place a stone so that it would immediately result in that stone (or a group it would join) being captured by enemy. I.e. if there is an enemy group with one empty square in the middle of it, you cannot put a stone there as that stone would simply have no liberties and would immediately die. Exception to this is the above mentioned taking of a group, i.e. if a suicidal move results in immediately taking enemy's group, it is allowed.

The **ko** rule states that one mustn't make a move that returns the board to the immediately previous state; this basically applies just to the situation in which the enemy takes your stone and you would just place it back, retaking his capturing stone. By the *ko* rule you cannot do this IMMEDIATELY, but you can still do this any further following round. Some rulesets extend this rule to so called *superko* which prohibits repetition of ANY previously seen position (this covers some rare cases that can happen).

Territory: at any time any EMPTY square on the board belongs either to white (no black stone can be reached from it by traveling over neighbors), black (no white stone can be reached from it) or none (belongs to neither). Squares that have stone on them aren't normally considered to belong to anyone (though some scoring systems do), i.e. if you surround a territory as white, only the VACANT surrounded squares count as your territory. The size of territory plays a role in final scoring. An alternative to territory is **area**, which is territory plus the squares occupied by player's stones and which is used under some rulesets.

Prisoners are enemy's stones that are OBVIOUSLY in your territory and so are practically dead. I.e. they are inside what's clearly not their territory and with further play would clearly be captured. Obvious here is a matter of agreement between players -- if players disagree whether some stones are obvious prisoners, they simply keep playing and resolve the situation.

Scoring: scoring assigns points to each player when the game is over, the one with more points win. There are multiple scoring systems, most common are these two (players basically universally agree the scoring

system has almost no effect on the play so it's probably more of a convention):

- **Chinese** (area scoring): more KISS, the score is just each player's area (surrounded empty square PLUS squares occupied by the player's stones), plus komi for white. { This one seems to me like a better option for beginners and also for programming, it's just simpler and makes you not afraid of putting stones anywhere. ~drummyfish }
- **Japanese** (territory scoring): At the end of the game we count the score for black as the size of black's territory PLUS one point for each stone black has captured PLUS one point for each white prisoner (a would be captured stone) in black's territory. Score for white is computed analogously but we also add the komi compensation.

Handicaps: TODO.

Implications of rules and basic of strategy/tactics: life and death, eyes, atari, TODO.

Example: the following is an example of the end state of a beginner game on a 9x9 board:

```
9 | . # . . # # # 0 . |
8 | # . # . # 0 0 . 0 |
7 | . . . # # 0 . 0 . |
6 | # . . # 0 . 0 . 0 |
5 | # . # 0 0 0 . . . |
4 | . # # # 0 0 . . . |
3 | . . . # # 0 0 . . |
2 | . . . . # # 0 # . |
1 | . . . . # 0 0 0 . |
  +-----+
  A B C D E F G H I
```

Here black's (#) territory is 23, and black made 9 captures during the game, giving together 32 points. White's (0) territory is 16 and he has one black prisoner (H2), giving 17 points; furthermore white made 6 captures during the game and gets 5.5 (smaller value due to only 9x9 board size) bonus points as komi, totalling 28.5 point. Therefore black wins.

TODO

Play Tips

TODO

Go And Computers, Programming

See also <https://senseis.xmp.net/?ComputerGoProgramming> and <https://www.chessprogramming.org/Go>.

Board representation: a straightforward representation of the go board is as a simple array of squares; each square can be either empty, white or black, that's 3 values that can be stored with 2 bits, which allow storing 4 values, leaving one extra value to be used for some other purpose (e.g. marking illegal ko squares, estimated dead stones, marking last move etc.). 1 byte allows us to store 4 squares this way so we need only 91 bytes to represent the whole 19x19 board. On computers with enough RAM it may be considered to store 1 square in a single byte or int, making the board take more space but gaining speed thanks to data alignment (we don't need extra instructions for squeezing bit from/to a single byte). Of course we furthermore have to keep track of extra things such as numbers of captured stones.

TODO

Stats

Some interesting stats about go follow.

The longest possible game without passes has 4110473354993164457447863592014545992782310277120 moves. The longest recorded professional game seems to be mere 411 moves long (Hoshino Toshi vs Yamabe Toshiro, 1950). There are $2.08168199382 \cdot 10^{170}$ legal positions on a 19x19 board, $3.72497923077 \cdot 10^{79}$ for 13x13 and $1.03919148791 \cdot 10^{38}$ for 9x9. The number of possible games is estimated from $10^{10^{100}}$ to $10^{10^{171}}$. An average high-level game lasts about 150 moves. Average branching factor is 250 (compare to 35 in chess).

See Also

- [chess](#)
 - [game of life](#)
 - [hex game](#)
-

goodbye_world

Goodbye World

Goodbye world is a [program](#) that is in some sense an opposite of the traditional [hello world](#) program. What exactly this means is not strictly given, but some possibilities are:

- It just prints *goodbye world*, the programmer writes the program and never touches the language again.
 - It is the last program a programmer writes before death, either unknowingly or possibly as a [suicide](#) note.
 - Just as hello world shows the very basics of a language, a goodbye world may showcase the most advanced or masterful concepts of the language.
 - It is a program that erases itself or possibly the whole [operating system](#) etc.
 - TODO: more ideas?
-

good_enough

Good Enough

A good enough solution to a problem is a solution that solves the problem satisfyingly (not necessarily precisely or completely) while achieving minimal cost (effort, complexity, [maintenance](#), implementation time etc.). This is contrasted with an [overkill](#), a solution that's "too good" (for a higher cost). For example a word-for-word translation of a text is a primitive way of translation, but it may be good enough to understand the meaning of the text; in many climates a tent is a good enough accommodation solution while a luxury house is a solution of better quality (more comfortable, safe, ...) for a higher cost. It's been said that the [perfect is the enemy of good](#).

To give an example from the world of programming, [bubble sort](#) is in many cases better than quick sort for its simplicity, even though it's much slower than more advanced sorts. [ASCII](#) is mostly good enough compared to [Unicode](#). And so on.

In technology we are often times looking for good enough solution to achieve [minimalism](#) and save valuable resources (computational resources, programmer time etc.). It rarely makes sense to look for solutions that are more expensive than they necessarily need to be, however in the context of [capitalist software](#) we see this happen many times as a part of killer feature battle and also driving prices artificially up for economic reasons (e.g. increasing the cost of maintenance of a software eliminates any competition that can't afford such cost). An example of this is the trend in smartphones to have 4 and more physical cameras. This is only natural in [capitalism](#), we see the tendency for wasting resources everywhere. This of course needs to be stopped.

google

Google

Google (also Goolag) is one the very top and most evil big tech corporations, as well as one of the worst corporations in history (if not THE worst), comparable only to Microsoft, Apple and Facebook. Google is gigantically evil and largely controls the Internet, pushes mass surveillance, personal data collection and abuse, ads, bloat, fascism and censorship.

Google's motto used to be "**Don't be evil**", but in 2018 they ditched it lol xD

Google raised to the top thanks to its search engine launched in the 90s. It soon got a **monopoly on the Internet search** and started pushing ads. Nowadays Google's search engine basically just promotes "content" on Google's own content platforms such as YouTube and of course censors sites deemed politically incorrect.

If you are relying on Goolag for your search, you are missing on a huge part of the web, you will simply never see links to huge parts of the web which currently include e.g. Kiwifarms, Metapedia, Encyclopedia Dramatica, Infogalactic, Incels wiki, 8kun and many others. You are literally using crippled "search engine" if it can even be called so anymore, you're seeing a tiny bubble of preapproved content.

Besides heavily biasing web search results towards Google's own and friendly platforms, Google also **heavily censors** the search results and won't show links to prohibited sites unless you literally very specifically show that you want to find a prohibited site you already know of, for example you won't find results leading to Metapedia or Encyclopedia Dramatica unless you literally search for the url of those sites of long verbatim phrases they contain -- this is a trick played on those who "test" Google which at is mean to make it look as if Google actually isn't censored, however it is of course censored because the only people who will ever find the prohibited sites and their content are people who already know about it and are specifically searching for it just to test Google's censorship. { EDIT: tho Google also seems to refuse to give some URLs no matter what, e.g. <https://infogalactic.com>. Just tested it. ~drummyfish } If you intend to truly search the Internet, don't rely on Google's results but search with multiple engines (that have their own index) such as Mojeek, Yandex, Right Dao, wiby, YaCy, Qwant etc. (and of course search the darknet), also check out metasearch engines like SearxNG.

Google has created a malicious capitalist mobile "operating system" called Android, which they based on Linux with which they managed to bypass its copyleft by making Android de-facto dependent on their proprietary Play Store and other programs. I.e. they managed to take a free project and make a de-facto proprietary malware out of it -- a system that typically doesn't allow users to modify its internals and turn off its malicious features. Android is also one of the ugliest pieces of software ever made, requiring hugely specific and expensive computer setup just for its compilation. With Android they invaded a huge number of devices from cells phones to TVs and have the ability to spy on the users of these devices.

Google also tries to steal the public domain: they scan and digitize old books whose copyright has expired and put the on the Internet archive, however in these scans they put a condition that the scans should not be used for commercial purposes, i.e. they try to keep exclusive commercial right for public domain works, something they have no right to do at all.

gopher

Gopher

Gopher (allegedly from "go for information") is a network protocol for publishing, browsing and downloading files and is known as a much simpler alternative to the World Wide Web (i.e. to HTTP and HTML). In fact it competed with the Web in its early days and even though the Web won in the mainstream, gopher still remains used by small communities (usually the more dedicated though, see e.g. bitreich). Gopher is like the Web but well designed, it is the suckless/KISS way of doing what the Web does, it contains practically no bloat and so we highly advocate its use. Gopher inspired creation of Gemini, a similar but bit more complex and "modern" protocol, and the two together have recently become the main part of so called Smol Internet. Gopher is much better than Gemini though. The set of all public gopher servers is called gopherspace. The Gopher protocol was defined in 1993 in RFC 1436.

Gopher **doesn't use any encryption** (though some servers allow access via Tor). **This is good, encryption is bloat.** Gopher also doesn't really know or care about Unicode and similar bloat (which mostly serves trannies to insert emojis of pregnant men into readmes anyway, we don't need that), it's basically just ASCII (of course you can employ Unicode as gopher just transfers files really, it's just that Unicode is not part of gopher's specification and most people prefer to keep it ASCII). Gopher's simple design is intentional, the authors deemed simplicity a good feature. Gopher is so simple that you may very well write your own client and server and comfortably use them -- **you can even browse gopher just by manually using telnet** to communicate with the server.

How big is/was gopherspace? In 1994 there were over 1300 gopher servers (source: 1994 book *Finding it on the Internet*), around 1995 there were already more than 6000 (source: 1995 video *Searching the Internet - Gopher | The Internet Revealed*). Most of them are now gone, in 2005 there were only 145 servers reported by Veronica search engine (source: *2007 gopher archive*), though Gopher recently saw a new wave of popularity. As of 2023 the Veronica search engine reported 315 gopher servers in the world with 5+ million indexed selectors, which they estimated was 83% of the whole gopherspace (the peak server count was in 2020 at almost 400). Quarry search engine reports 369 servers and 1+ million indexed selectors. Contrition search engine reported even 495 servers and 7+ million selectors. The "gawler" crawler of gopherspace.de reported 192 active servers at the beginning of 2016, 182 in 2020, 413 in 2023 (bumped its search list) and 380 in 2024. Gopher LAWN directory (made by bitreich) contains 281 selected quality gopher holes.

From the user's perspective **the most important distinction from the Web** is that gopher is based on **menus** instead of "webpages"; a menu is simply a column of items of different predefined types, most importantly e.g. a *text file* (which clients can directly display), *directory* (link to another menu), *text label* (just shows some text), *binary file* etc. A menu can't be formatted or visually changed, there are no colors, images, scripts or hypertext -- a menu is not a presentation tool, it is simply a navigation node towards files users are searching for (but the mentioned ASCII art and label items allow for somewhat mimicking "websites" anyway). Gopher is also often **browsed from the command line**, though graphical clients are a thing too. Addressing works with URLs just as the Web, the URLs just differ by the protocol part (gopher:// instead of http://), e.g.: gopher://gopher.floodgap.com:70/1/gstats. What on Web is called a "website" on gopher we call a **gopherhole** or just *hole* (i.e. a collection of resources usually under a single domain) and the whole gopher network is called a **gopherspace**. Blogs are common on gopher and are called **phlogs** (collectively a *phlogosphere*). As menus can refer to one another, gopher creates something akin a **global file system**, so browsing gopher is like browsing folders and can comfortably be handled with just 4 arrow keys. Note that as menus can link to any other menu freely, the structure of the "file system" is not a tree but rather a general graph. Another difference from the Web is gopher's great emphasis on **plaintext and ASCII art** as it cannot embed images and other media in the menus (even though of course the menus can link to them). There is also a support for sending text to a server so it is possible to implement search engines, guest books, games etc.

Gopher is just an application layer protocol (officially running on port 70 assigned by IANA), i.e it sits above lower layer protocols like TCP and takes the same role as HTTP on the Web and so only defines how clients and servers talk to each other -- the gopher protocol doesn't say how menus are written or stored on servers. Nevertheless for the creation of menus so called **gophermaps** have been established, which is a simple format for writing menus and are the gopher equivalent of Web's HTML files (just much simpler, basically just menu items on separate lines, the exact syntax is ultimately defined by server implementation). A server doesn't have to use gophermaps, it may be e.g. configured to create menus automatically from directories and files stored on the server, however gophermaps allow users to write custom menus manually. Typically in someone's gopherhole you'll be served a welcoming intro menu similar to a personal webpage that's been written as a gophermap, which may then link to directories storing personal files or other hand written menus. Some gopher servers also allow creating dynamic content with scripts called **moles**.

Gopher software: sadly "modern" browsers are so modern they have millions of lines of code but can't be bothered to support such a trivial protocol like gopher, however there are Web proxies you can use to explore gopherspace (look up e.g. floodgap). Better browsers such as lynx (terminal), sacc, clit or forq (GUI) can be used for browsing gopherspace natively (it's not hard, you don't need to learn any keybinds, using arrow keys usually just works). As a server you may use e.g. Gophernicus (used by SDF) or search for another one, there are dozens. { Personally I've used gophrier for server, it was the simplest one I found. ~drummyfish } For the creation of gophermaps you simply use a plaintext editor. **Where to host gopher?** Pubnixes such as SDF, tilde.town and Circumlunar community offer gopher hosting but many people simply

self-host servers e.g. on Raspberry Pis, it's pretty simple.

A quick tl;dr/sumup of the gopher world/community as of 2023: thankfully there doesn't seem to be much ensorship and/or woke toxicity that's seen on the web, the community is still quite small, which is probably a GOOD thing, though something like doubling the gopherspace size would probably still be welcome; an issue/downside at this time seems to be a "self serving" nature of gopher ("come to gopher to learn about gopher"), i.e. instead of gopher being a "platform" for sharing all kinds of information, we mostly have a gopher community talking about gopher, so outsiders really have nothing to come in for. Of course there is interesting information of other kinds, but the overall impression is just this. Another issue is that current search engines like Veronica don't seem to be fulltext but rather search only document titles. Currently gopherspace seems to be mostly divided into following gopherholes:

- big center hubs (floodgap, bitreich, ...): Stable holes providing info and tutorials for newcomers, links to important resources, manifestos, sometimes providing a search engine or directory of other holes.
- phlogs and personal holes: Small holes with "hello world messages" and personal rants, usually about gopher and related topics such as technology minimalism, independent living etc.
- socializing/roleplay pubnices offering hosting (circumlunar, SDF, tilde town, ...): Smaller noncommercial communities in the spirit of old BBSes, they offer user account, ssh access to their Unix servers, email, web and gopher hosting space, chat, games etc. Sometimes they are roleplay focused, having some sci-fi backstory or something. They don't ask for any fees, however all seem to REQUIRE active participation in the community or else they'll delete your account -- this sucks big time for asocial introverts who just want a gopher hole without being forced to play with other kids.
- web proxies (gopherpedia, gophreddit): Mirrors/proxies to popular websites such as Wikipedia, project gutenber, reddit etc.

Some basic/interesting gopher links: gopher://bitreich.org/1/lawn (directory of gopher holes), gopher://gopher.floodgap.com/7/v2/vs (search engine), gopher://circumlunar.space:70/1 (circumlunar space), gopher://gopherpedia.com (Wikipedia on gopher), gopher://gopher.icu/7/quarry (search engine), ...

How To

Here is a nice tutorial: `git clone git://bitreich.org/gopher-tutorials/`.

To quickly try browsing gopher either use a web proxy, e.g. at `https://gopher.floodgap.com/gopher/gw.lite`, or use some nice native browser, e.g. `lynx gopher://floodgap.com`.

More technical details: just as with the web, you have some gopher server running somewhere (some IP address/domain, on port 70) which serves resources to clients. A client connects to the server (via TCP) and simply sends the name of the resource (file or directory) it wants to retrieve as a string ending with a newline. If the string is empty, the server sends the default directory (the "main page"). You may try this manually in terminal using telnet, nc or a similar tool. For example:

```
echo "" | nc floodgap.com 70
```

The server just sends us back a list of available resources in the "main directory", one per line, each in format:

```
<TYPE><DISPLAY_STRING><TAB><PATH><TAB><SERVER><TAB><PORT>
```

For example one of the lines here looks like this:

```
1Search Gopherspace with Veronica-2 <TAB> /v2 <TAB> gopher.floodgap.com <TAB> 70
```

Here 1 says the resource is a directory, then we have a display string (which you'll see in the browser), then the path to the resource, i.e. /v2, and then the server and port. If we want to retrieve this directory, we send:

```
echo "/v2" | nc floodgap.com 70
```

And get a similar response. This is basically all a client needs to know.

As for running a server, details depend on each one, but generally they behave like this: you have a server running in some default directory, let's say `/home/me/my_gopherhole`. By default a server will just serve list of files present in this directory to clients who request the "main directory", treating directories as subdirectories and sending regular files back. However there is one important feature: you may create a **gophermap file** to create a custom menu, or something akin a "gopher website". Gophermap is something like gopher's HTML, just much more simple. How to do this? You simply create a file name gophermap in the directory (the main one or any subdirectory) -- if the server sees such a file, it serves it instead of listing the directory file.

TODO: continue

Example

TODO

See Also

- Gemini
 - Fediverse
 - smol internet
-

graphics

Computer Graphics

Computer graphics (CG or just graphics) is a field of computer science that focuses on visual information. The field doesn't have strict boundaries and can blend and overlap with other possibly separate topics such as physics simulations, multimedia and machine learning. It usually deals with creating or analyzing 2D and 3D images and as such CG is used in data visualization, game development, virtual reality, optical character recognition and even astrophysics or medicine.

We can divide computer graphics in different ways, traditionally e.g.:

- by direction:
 - ♦ **rendering**: Creating images.
 - ♦ **computer vision**: Extracting information from existing images.
- by basic elements:
 - ♦ **raster**: Deals with images composed of a uniform grid of points called pixels (in 2D) or voxels (in 3D).
 - ♦ **vector**: Deals with images composed of geometrical primitives such as curves or triangles.
- by dimension:
 - ♦ **2D**: Deals with images of a 2D plane.
 - ♦ **3D**: Deals with images that capture three dimensional space.
- by speed:
 - ♦ **real time**: Trying to work with images in real time, e.g. being able to produce or analyze 60 frames per second.
 - ♦ **offline**: Processes or creates images over longer time-spans, even hours or days, e.g. in 3D movie rendering.
- ...

Since the 90s computers started using a dedicated hardware to accelerate graphics: so called graphics processing units (GPUs). These have allowed rendering of high quality images in high FPS, and due to the entertainment and media industry (especially gaming), GPUs have been pushed towards greater performance each year. Nowadays they are one of the most consumerist hardware, also due to the emergence of general purpose computations being moved to GPUs (GPGPU), lately especially mining of cryptocurrencies and training of AI. Most lazy programs dealing with graphics nowadays simply expect and require a GPU, which creates a bad dependency and bloat. At LRS we try to prefer the suckless software rendering, i.e. rendering on the CPU, without GPU, or at least offer this as an option in case GPU isn't

available. This many times leads us towards the adventure of using old and forgotten algorithms used in times before GPUs.

3D Graphics

This is a general overview of 3D graphics, for more technical overview of 3D rendering see [its own article](#).

3D graphics is a big part of CG but is a lot more complicated than 2D. It tries to achieve **realism** through the use of **perspective**, i.e. looking at least a bit like what we see in the real world. 3D graphics can very often be seen as **simulating the behavior of light**; there exists so called [rendering equation](#) that describes how light behaves ideally, and 3D computer graphics tries to approximate the solutions of this equation, i.e. the idea is to use [math](#) and [physics](#) to describe real-life behavior of light and then simulate this model to literally create "virtual photos". The theory of realistic rendering is centered around the rendering equation and achieving **global illumination** (accurately computing the interaction of light not just in small parts of space but in the scene as a whole) -- studying this requires basic knowledge of [radiometry](#) and [photometry](#) (fields that define various measures and units related to light such as [radiance](#), radiant intensity etc.).

In 2010s mainstream 3D graphics started to employ so called [physically based rendering](#) (PBR) that tries to yet more use physically correct models of [materials](#) (e.g. physically measured [BRDFs](#) of various materials) to achieve higher photorealism. This is in contrast to simpler (both mathematically and computationally), more [empirical](#) models (such as a single texture + [phong lighting](#)) used in earlier 3D graphics.

Because 3D is not very easy (for example [rotations](#) are pretty complicated), there exist many **3D engines** and libraries that you'll probably want to use. These engines/libraries work on different levels of abstraction: the lowest ones, such as [OpenGL](#) and [Vulkan](#), offer a portable API for communicating with the GPU that lets you quickly draw triangles and write small programs that run in parallel on the GPU -- so called [shaders](#). The higher level, such as [OpenSceneGraph](#), work with [abstraction](#) such as that of a **virtual camera** and **virtual scene** into which we place specific 3D objects such as models and lights (the scene is many times represented as a hierarchical graph of objects that can be "attached" to other objects, so called [scene graph](#)).

There is a tiny [suckless/LRS](#) library for real-time 3D: [small3dlib](#). It uses software rendering (no GPU) and can be used for simple 3D programs that can run even on low-spec embedded devices. [TinyGL](#) is a similar software-rendering library that implements a subset of [OpenGL](#).

Real-time 3D typically uses an **object-order** rendering, i.e. iterating over objects in the scene and drawing them onto the screen (i.e. we draw object by object). This is a fast approach but has disadvantages such as (usually) needing a memory inefficient [z-buffer](#) to not overwrite closer objects with more distant ones. It is also pretty difficult to implement effects such as shadows or reflections in object-order rendering. The 3D models used in real-time 3D are practically always made of **triangles** (or other polygons) because the established GPU pipelines work on the principle of drawing polygons.

Offline rendering (non-real-time, e.g. 3D movies) on the other hand mostly uses **image-order** algorithms which go pixel by pixel and for each one determine what color the pixel should have. This is basically done by casting a ray from the camera's position through the "pixel" position and calculating which objects in the scene get hit by the ray; this then determines the color of the pixel. This more accurately models how rays of light behave in real life (even though in real life the rays go the opposite way: from lights to the camera, but this is extremely inefficient to simulate). The advantage of this process is a much higher realism and the implementation simplicity of many effects like shadows, reflections and refractions, and also the possibility of having other than polygonal 3D models (in fact smooth, mathematically described shapes are normally much easier to check ray intersections with). Algorithms in this category include [ray tracing](#) or [path tracing](#). In recent years we've seen these methods brought, in a limited way, to real-time graphics on the high end GPUs.

See Also

- [computational photography](#)

graveyard

Graveyard

Welcome to the graveyard. Here we mourn the death of technology, art, science and other deceased by the hand of capitalism and its countless children such as Feminism, LGBT, consumerism and so on.

{ Sometimes we are very depressed from what's going on in this world, how technology is raped and used by living beings against each other. Seeing on a daily basis the atrocities done to the art we love and the atrocities done by it -- it is like watching a living being die. Sometimes it can help to just know you are not alone. ~drummyfish }

R. I. P.

~~~~~

TECHNOLOGY

long time ago - now

Here lies technology who was  
helping people tremendously until  
its last breath. It was killed by  
capitalism.

Now we better go back home.

---

greenwashing

## Greenwashing

*"For every car you consume we plant a tree." --corporations*

TODO

---

gui

## Graphical User Interface

*"Always add a commandline interface to your software. Graphical User interfaces are for sissies." --bitreich manifesto*

Graphical user interface (GUI) is a visual user interface that uses graphics such as images and geometrical shapes. This stands in contrast with text user interface (TUI) which is also visual but only uses text for communication.

Expert computer users normally frown upon GUI because it is the "noobish", inefficient, limiting, cumbersome, hard to automate way of interacting with computer. GUI brings complexity and bloat, they are slow, inefficient and distracting. We try not to use them and prefer the command line.

"Modern" GUIs mostly use callback-based programming, which again is more complicated than standard polling non-interactive I/O. If you need to do GUI, just use a normal infinite loop FFS.

## When And How To Do GUI

GUI is not forbidden, it has its place, but today it's way too overused -- it should be used only if completely necessary (e.g. in a painting program) or as a completely optional thing built upon a more suckless text interface or API. So remember: first create a program and/or a library working without GUI and only then consider creating an optional GUI frontend. GUI must never be tied to whatever functionality can be implemented without it.

Still, when making a GUI, you can make it suckless and lightweight. Do your buttons need to have reflections, soft shadows and rounded anti-aliased borders? No. Do your windows need to be transparent with light-refraction simulation? No. Do you need to introduce many MB of dependencies and pain such as QT? No.

The ergonomics and aesthetic design of GUIs has its own field and can't be covered here, but just keep in mind some basic things:

- Don't have too many elements (buttons etc.) at the screen at once, it's confusing as hell and drives noobs away.
- Things must be intuitive, i.e. behave in a way that they normally do (e.g. main menu should be at the top of the window, not the bottom etc.).
- Just use your brain. If a button is important and often used, it should probably be bigger than a button that's used almost never, etc.

The million dollar question is: **which GUI framework to use?** Ideally none. GUI is just pixels, buttons are just rectangles; make your GUI simple enough so that you don't need any shitty abstraction such as widget hierarchies etc. If you absolutely need some framework, look for a suckless one; e.g. nuklear is worth checking out. The suckless community sometimes uses pure X11, however that's not ideal, X11 itself is kind of bloated and it's also getting obsoleted by Wayland. The ideal solution is to make your GUI **backend agnostic**, i.e. create your own very thin abstraction layer above the backend (e.g. X11) so that any other backend can be plugged in if needed just by rewriting a few simple functions of your abstraction layer (see how e.g. Anarch does rendering).

## State Of Mainstream GUI

Nowadays there are a great many GUI libraries, frameworks, standards and paradigms, and it may be a bit hard to digest them at once.

TODO: some general shit bout graphical windows vs the "single window" mobile and web UI, analysis of the "GUI stack" (Linux framebuffer, X window, widget toolkits etc.), basic widgets etc.

---

hacker\_culture

## Hacker Culture

See hacking.

---

hacking

## Hacking

*Not to be confused with cracking.*

Hacking (also hackerdom) in the widest sense means exploiting usually (but not necessarily) a computer system in a clever, "thinking outside the box" way. In context of computers the word *hacker* was originally -- that is in 1960s -- used for very good programmers and people who were simply good with computers, the word *hacking* had a completely positive meaning; hacker could almost be synonymous with computer genius (at the time people handling computers were usually physicists, engineers or mathematicians), someone who enjoyed handling and programming computers and could playfully look for very clever ways of making them do what he wanted. Over time hackers evolved a whole **hacker culture** with its own slang, set of values, behavioral and ethical norms, in jokes and rich lore. As time marched on, computer security has started to become an important topic and some media started to use the word *hacker* for someone breaking into a computer system and so the word gained a negative connotation in the mainstream -- though many refused to accept this new meaning and rather used the word cracker for a "malicious hacker", there appeared new variants such as *white hat* and *black hat* hacker, referring to ethical and malicious hackers. With onset of online games the word *hacking* even became a synonym for cheating. The original positive meaning has recently seen some comeback with popularity of sites such as hacker news or hackaday, the word *life hack*



has even found its way into the non-computer mainstream dictionary, however a "modern hacker" is a bit different from the oldschool hacker, usually for the worse (for example a modern self proclaimed "hacker" has no issue with wearing a suit, something that would be despised by an oldschool hacker). We, LRS, advocate for using the original, oldschool meaning of the word *hacker*.

## Original Hacker Culture

The original hacker culture is a culture of the earliest computer programmers, usually smart but socially rather isolated nerds -- at the time mostly physicists, mathematicians and engineers -- who shared deep love for programming and pure joy of coming up with clever computer tricks, exploration of computers and freely sharing their knowledge and computer programs with each other. The culture started to develop rapidly at MIT in about the second half of 1960s, though other hacker communities existed earlier and in other places as well (still mostly at universities).

Nowadays this original culture is very sadly becoming almost completely extinct, owing to the modern world whose values -- such as self interest, consumerism, secrecy, praise of censorship, "inclusivity" of the incompetent, materialism etc. -- are mostly polar opposites of the original hacker values: a newly born man would have to reject 99% of the culture he grew up in to be able to adopt the hacker mindset. The culture seems to live on mostly in individuals, mostly the old hackers themselves, and partially in some extremely underground communities such as that of the demoscene, but even there it's degenerating greatly.

The word *hack* itself seems to have come from a model train club at MIT in whose slang the word referred to something like a project of passion without a specific goal; before this the word was used around MIT for a specific kind of clever but harmless pranks. Members of the model train club came to contact with early computers at MIT and brought their slang along. These early punch-card computers were expensive and sacred, hackers treated them as almost supernatural entities; in the book *Hackers* it is mentioned that those who were allowed to operate the machines were called *Priests* -- Priests would often carry out a little prayer to please the machine so that it would bless them with computation. During 60s and 70s so called phreaking -- hacking the phone network -- was popular among hackers.

Many ideas -- such as the beauty of minimalism -- that became part of hacker culture later came from the development of Unix and establishment of its programming philosophy. Many hackers came from the communities revolving around PDP 10 and ARPANET, and later around networks such as Usenet. At the time when computers started to be abused by corporations, Richard Stallman's definition of free software and his GNU project embodied the strong hacker belief in information freedom and their opposition of intellectual property. When computer technology became invaded and raped by capitalism, hackers separated themselves from the influx of coding monkeys and managers not only culturally, but also by retaining their programming philosophy -- programming of a hacker is very different from the ugly "software development" of a corporation, a hacker writes beautiful, minimal code. He doesn't merely aim to "get the job done", he creates art, a code that works well while being a beauty of engineering on the inside, he isn't afraid to throw away code and rewrite it from scratch just to make it a little better (as opposed to patching it up, bloating and extending it, as a corporation would do).

The culture has a deep lore and its own literature consisting of books that hackers usually like (e.g. The Hitchhiker's Guide to the Galaxy) and books by hackers themselves. Bits of the lore are in forms of short stories circulated as folklore, very popular form are so called Koans. Perhaps the most iconic hacker story is the Story of Mel which tells a true story of a master hacker keeping to his personal ethical beliefs under the pressure of his corporate employers -- a conflict between manager employers ("suits") and hacker employees is a common theme in the stories. Other famous stories include the *TV typewriter* and *Magic Switch*. One of the most famous hacker books is the Jargon File, a collectively written dictionary documenting hacker culture in detail. A 1987 book The Tao of Programming captures the hacker wisdom with Taoist-like texts that show how spiritual hacking can get -- this reflects the above mentioned sacred nature of the early computers. Hacker culture very frequently mimics eastern religions and philosophies such as Taoism, Buddhism or various martial arts. The textfiles website features many text files on hacking at <https://textfiles.vistech.net/hacking/>. See also *Ten Commandments for C Programmers* etc. A lot about hackers can be learned from books about them, e.g. the free book *Free as in Freedom* about Richard Stallman (available e.g. here). A prominent hacker writer is Eric S. Raymond who produced a very famous essay *The Cathedral and the Bazaar*, edited the Jargon File and has written guides such as *How To Become A Hacker* and *How To Learn Hacking* -- these are all good resources on hackerdom, even though Raymond himself is kind of shitty, he for example prefers the "open source" movement to free software.

As a symbol of hackerdom the glider symbol from game of life is sometimes used, it looks like this:

```
|_|0|_|  
|_|_|0|  
|0|0|0|
```

Let us now attempt to briefly summarize what it means to be a hacker:

- **Hacker is a kind of artist who builds and creates** (though not every artist is a hacker!), cracker is someone who breaks and destroys, many times due to being less competent or unworthy of true hacking -- destroying something is easier than creating something.
- **Hacker greatly values freedom**, among which are the **freedom of information, free software, free speech, free thinking**, free access to computers etc. Therefore he supports sharing, even if it is called for example "piracy", and despises things going against said freedoms such as proprietary software, passwords and security (preventing information freedom), censorship, copyright, patents, pretense and deceit etc.
- **Hackers are non-conformists, reject authority and don't respect social norms, hacking is a way of life**; a hacker wears old cheap clothes, long hair and unkept beard without conforming to any fashion, he sees caring about looks as a wasted time that would better be spent by hacking computers. Hacker is a basement dwelling nerd without social life because he has rich inner intellectual life, he's usually a kissless virgin, even a wizard, partly because of his looks but also again because typical adult life would require him to do less hacking. He doesn't program for money, he literally lives his whole life as a hacker (a typical example is e.g. focusing on powers of two, such as aiming for 1024 words in his essays, rather than using powers of ten like normal people).
- **Hacker values fun and playfulness** -- despite his serious dedication to the art, he hates seriousness of the business guys and "suits", as well as the self-centered, egoistic attitude of "modern hackers" who might see or present themselves as kind of superheroes. A hacker will give his programs funny names rather than names that would make for a good business product, a hacker will insert jokes in his source code (e.g. hex values such as 0xBEEFACE), documentation and speech (Jargon File has a whole section on how hackers construct and use words).
- **Hacker aims for ingenuity, cleverness, elegance, minimalism, thinking out of the box** etc. As such he loves math, puzzles, intellectual challenges (such as code golfing) and despises ugly commercial ways of mainstream technology, i.e. that which is bloated, hastily made to impress by visuals or cheap "killer features" while hiding ugly internals etc.
- **Hacker loves hacking and tinkering in itself -- hacking is the goal, not the means. Hacking is art and carries deep intellectual and even spiritual value.** To a hacker it is a joy to program computers and he aims for nothing more than enjoy endless hours of programming, programming is NOT a tool to achieve low goals such as monetary profit or mainstream fame. Many hackers claim that hacking is better than sex (though it is questionable whether many of them have experience with the latter).
- **Hacker is an elitist, attitude is not enough for being a hacker**, skill is of essential importance. Correct attitude and mindset are important and necessary but not sufficient (as ESR writes: "attitude is no substitute for competence") -- if you don't excel at hacking, you are not a hacker. This is in contrast e.g. with music genre fans where you can "identify yourself" as being "punk" or "metal" even if you can't play any musical instrument or with the modern "inclusive" "coder" culture in which you can easily be called a game developer even if you cannot program etc. Part of hackerdom is also an aim for good reputation among others, to be called a hacker by OTHERS, HOWEVER this has to be achieved without asking or self promotion, merely through doing good hacking, you must not beg others to "please call you a hacker" or promote your programs with marketing to achieve cheap popularity -- no, reputation or the title of hacker is NOT the goal in itself, the goal is good hacking and reputation is an indication you achieved it.
- **Hacker has strong opinions about technology**, for example about what the best text editor or best programming language is. However hackers may also sometimes disagree which results in **holy wars**.

Let's mention a few people who were at their time regarded by at least some as true hackers, however note that many of them betrayed some of the hacker ways either later in life or even in their young years -- people aren't perfect and no single individual is a perfect example of a whole culture. With that said, those regarded hackers included Melvin Kaye aka Mel, Richard Stallman, Linus Torvalds, Eric S. Raymond, Ken Thompson, Dennis Ritchie, Richard Greenblatt, Bill Gosper, Steve Wozniak or Larry Wall.

## "Modern" "Hackers"

Many modern zoomer soydevs call themselves "hackers" but there are basically none that would stay true to the original ethics and culture and be worthy of being called a true hacker, they just abuse the word as a cool term or a brand (see e.g. "hacker" news, a capitalist circlejerk website where self proclaimed smartass "hackers" come to advertise their ugly bloated rapeware and talk about how to best exploit the market). It's pretty sad the word has become a laughable parody of its original meaning by being associated with groups such as Anonymous who are just a bunch of 14 year old children trying to look like "movie hackers". The hacker culture has been spoiled basically in the same ways the rest of society, and the difference between classic hacker culture and the "modern" one is similar to the difference between free software and open source, though perhaps more amplified -- the original culture of strong ethics has become twisted by capitalist trends such as self-interest, commercialization, fashion, mainstreamization, even shitty movie adaptations etc. The modern "hackers" are idiots who have never seen assembly, can't do math, they're turds in suits who make startups, aren't afraid to suck corporation dicks and work as influencers, they are tech consumers with who use and even create bloat, and possibly even proprietary software. For the love of god, do NOT mimic such caricatures or give them attention -- not only are they not real hackers, they are simply retarded attention whores.

## Security "Hackers"

*Hacker* nowadays very often refers to someone involved in computer security either as that who "protects" (mostly by looking for vulnerabilities and reporting them), so called *white hat*, or that who attacks, so called *black hat*. Those are not hackers in the original sense, they are hackers in the mainstream adopted meaning of someone breaking into a system. **This kind of "hacker" betrays the original culture by supporting secrecy and censorship**, i.e. "protection" of "sensitive information" mostly justified by so called "privacy" -- this is violating the original hacker's pursuit of absolute information freedom (note that e.g. Richard Stallman boycotted even the use of passwords at MIT, Raymond discourages from using anonymous handles and rather recommends going by your real name). These people are obsessed with anonymity, encryption, cryptocurrencies, cryptofascism and are also more often than not egoist people with shitty personalities. In addition they don't generally adhere to the original hacker culture in any way either, they are simply people breaking into systems for some kind of self benefit (yes, even the *white hats*), nothing more than that. Again, do NOT try to mimic these abominations.

## Examples Of Hacks

{ As a redditfag I used to follow the r/devtricks subreddit, it contained some nice examples of hacks.  
~drummyfish }

A great many commonly used tricks in programming could be regarded as hacks even though many are not called so because they are already well known and no longer innovative, a true hack is something new that impresses fellow hackers. And of course hacks may appear outside the area of technology as well. The following is a list of things that were once considered new hacks or that are good examples demonstrating the concept:

- **bit hacks**: Clever manipulations of bits -- for example it is possible to swap two variable without a temporary variables by using the xor function. Another simplest example is implementing division by 2 as binary shift by 1 (this hack is used in real life by people for quickly dividing by 10, we just remove the last digit).
- **copyleft**: A legal hack by Richard Stallman, connected to free software, working on the basis of the following idea: "If copyright lets me put any conditions on my work, I may impose a condition on my work that says that any modified version must not impose any restrictive conditions".
- In minimalist C programming mainly two standards of the language are used: C89 and C99. To distinguish between them in source code one can e.g. exploit the fact that C99 introduced line comments (starting with `//`) and make such code that C99 sees part of it commented out while C89 doesn't. For example the following two lines: `int isC89 = 1 /**/ 2, ; isC89 = !isC89;`
- **fast inverse square root**: Famous hack that was used in the game Quake, it approximates an inverse of square root of a floating point number by treating it as an integer and bashing it with a magic constant, which is about four times faster than computing the value with the obvious floating point division.

- **memory rape in C**: E.g. instead of doing proper memory allocation with potentially inefficient and bloated malloc one may try to do a custom memory allocation without any libraries by abusing allocation on stack -- allocate a variable size array in main, set some global pointer to it and then manage this chunk of memory with your own allocation functions.
- **actually portable executable** (<https://justine.lol/apc.html>): Justine Tunney found a way to create an executable format that passes as a valid NATIVE executable on all major systems including GNU/Linux, Windows and Mac, i.e. it is possible to compile a native program (e.g. with C) and then have it natively run on any major OS.
- **game of life patterns**: Stable patterns such as glider or even programming game of life in game of life is a nice example of game hacking -- in fact exactly game of life hacking stood at the beginning of hacker culture.
- **bytebeat**: A demoscene hack that utilizes integer overflows to create rhythm and produce music.
- Computer graphics uses many clever tricks that could possibly be called hacks, e.g. in times when 3D graphics was primitive and didn't allow achieving such effects as mirror reflections easily, some games faked mirrors simply with a hole in the wall behind which the whole mirrored room was placed -- this achieved the same effect as a mirror and didn't require any extra rendering passes or shaders.
- **quine**: A cleverly constructed self-replicating program in programming language that prints its own source code -- this is a common exercise of language hackers.
- **MetaGolfScript esoteric languages**: rather than being a nicely designed code golfing language MetaGolfScript invents infinitely many languages, each of which solves one problem with a zero-length program, making it possible to win any golfing contest that allows arbitrary choice of language just by choosing the correct MetaGolfScript language.
- **Appending "in Minecraft" to avoid legal responsibility**: some people try to avoid legal responsibility for threats by talking about the situation as if it was harmlessly happening in a video game such as Minecraft, for example "Bitch I'm going to come to your house and murder you in sleep, in Minecraft." Though this is a nice hack and should work, the dystopian governments can do whatever they want and still arrest you for this -- this happened e.g. in New Jersey when one guy threatened to kill a sheriff like this.
- **polyglot programs**: another fun activity by programming language enthusiasts; a polyglot is source code that's a valid in more than one programming language.
- Richard Stallman called some musical compositions hacks, specifically 4'33 (just silence) and Ma Fin Est Mon Commencement (palindromic music).
- TODO: moar

## See Also

- soydev
- demoscene
- cracking

---

hack

## Hack

See hacking.

---

hard\_to\_learn\_easy\_to\_master

## Hard To Learn, Easy To Master

"Hard to learn, easy to master" is the opposite of "easy to learn, hard to master".

Example: drinking coffee while flying a plane.

---

hardware

# Hardware

The article is [here](#)!

---

harry\_potter

## Harry Potter

Harry Potter is a [franchise](#) and universe by an English [female](#) writer J. K. Rowling about wizards and magic { like ACTUAL [wizards](#) and [magic](#). ~drummyfish } that started in 1997 as an immensely successful series of seven children and young adult [books](#), was followed by movies and later on by many other spinoff media such as video [games](#). It made J. K. Rowling a billionaire and has become the most famous and successful book series of modern age. At first the books sparked controversies and opposition in religious communities for "promoting witchcraft", in recent years the universe and stories have become a subject of wider political analysis and [fights](#), as most other things. Commerce and politics destroyed it completely, anything new in the franchise is absolute garbage, but the original books are quite good.

{ I actually enjoyed the books -- they're not the best in the world, I've read many better ones that would better deserve this kind of attention, but still the work is admirable and of very high quality. There is of course tons of money in the franchise so it's getting raped and milked like any other IP capital, do not follow the new stuff. ~drummyfish }

**Plot summary:** sorry, we're not writing a plot summary here, thank [copyright](#) laws -- yes, [fair use](#) allows us to do it but it would make us [non free](#) :) Let's just say the story revolves around a boy named Harry Potter who goes to a wizard school with two friends and they're together *saving the world* from Lord Voldemort, the wizard equivalent of [Hitler](#). Overall the books start on a very light note and get progressively darker and more adult, turning into a story about "World War II but with magic wands instead of guns". It's pretty readable, with great, unique atmosphere, pleasant coziness and elements of many literary genres, there's nice humor and good ideas. Also the lore is very deep, but sometimes doesn't quite make sense (well, it was made by a woman).

## See Also

- [Lord Of The Rings](#)
- 

hash

## Hash

Hash is a number that's computed from some data in a [chaotic](#) way and which is used for many different purposes, e.g. for quick comparisons (instead of comparing big data structures we just compare their hashes) or mapping data structures to table indices.

Hash is computed by a **hash function**, a function that takes some data and turns it into a number (the hash) that's in terms of [bit](#) width much smaller than the data itself, has a fixed size (number of [bits](#)) and which has additional properties such as being completely different from hash values computed from very similar (but slightly different) data. Thanks to these properties hashes have a very wide use in [computer science](#) -- they are often used to quickly compare whether two pieces of non-small data, such as documents, are the same, they are used in indexing structures such as **hash tables** which allow for quick search of data, and they find a great use in [cryptocurrencies](#) and [security](#), e.g. for [digital signatures](#) or storing passwords (for security reasons in databases of users we store just hashes of their passwords, never the passwords themselves). Hashing is extremely important and as a programmer you won't be able to avoid encountering hashes somewhere in the wild.

{ Talking about wilderness, hyenas have their specific smells that are determined by bacteria in them and are unique to each individual depending on the exact mix of the bacteria. They use these smells to quickly identify each other. The smell is kind of like the animal's hash. But of course the analogy isn't perfect, for

example similar mixes of bacteria may produce similar smells, which is not how hashes should behave.  
~drummyfish }

It is good to know that we distinguish between "normal" hashes used for things such as indexing data and cryptographic hashes that are used in computer security and have to satisfy some stricter mathematical criteria. For the sake of simplicity we will sometimes ignore this distinction here. Just know it exists.

It is generally given that a hash (or hash function) should satisfy the following criteria:

- **Have fixed size** (given in bits), even for data that's potentially of variable size (e.g. text strings).
- **Be fast to compute**. This is mostly important for non-security uses, cryptographic hashes may prioritize other properties to guarantee the hash safety. But a hash function certainly can't take 10 minutes to compute :)
- **Have uniform mapping**. That is if we hash a lot of different data the hashes we get should be uniformly spread over the space of the hashes, i.e. NOT be centered around some number. This is in order for hash tables to be balanced, and it's also required in security (non-uniform hashes can be easier to reverse).
- **Behave in a chaotic manner**, i.e. hashes of similar data should be completely different. This is similar to the point above; a hash should kind of appear as a "random" number associated to the data (but of course, the hash of the same data has to always be the same when computed repeatedly, i.e. be deterministic). So if you change just one bit in the hashed data, you should get a completely different hash from it.
- **Minimize collisions**, i.e. the probability of two different values giving the same hash. Mathematically collisions are always possible if we're mapping a big space onto a smaller one, but we should try to reduce collisions that happen in practice. This property should follow from the principle of uniformity and chaotic behavior mentioned above.
- **Be difficult to reverse** (mainly for security related hashes). Lots of times this comes naturally from the fact that a hash maps a big space onto a smaller space (i.e. it is a non-injective function) and from their chaotic nature. Hashes can typically be reversed only by brute force.

Hashes are similar to checksums but are different: checksums are simpler because their only purpose is for checking data integrity, they don't have to have a chaotic behavior, uniform mapping and they are often easy to reverse. Hashes are also different from database IDs: IDs are just sequentially assigned numbers that aren't derived from the data itself, they don't satisfy the hash properties and they have to be absolutely unique. The term **pseudohash** may also be encountered, it seems to be used for values similar to true hashes which however don't quite satisfy the definition.

{ I wasn't able to find an exact definition of *pseudohash*, but I've used the term myself e.g. when I needed a function to make a string into a corresponding fixed length string ID: I took the first N characters of the string and appended M characters representing some characteristic of the original string such as its length or checksum -- this is what I called the string's pseudohash. ~drummyfish }

Some common uses of hashes are:

- Hash tables, data structures that allows for quick search and access of data. For example in chess programs and databases hashes of chess positions are used to identify and get some information associated with the position.
- Passwords in user databases are for security reasons not stored as plain text, instead only password hashes are stored. When a user enters a password, the system computes its hash and compares it to that stored in the database: if the hashes match, the password was correct. This is a way of allowing password authentication without giving the system the knowledge of user passwords.
- In digital signatures hashes of documents are used to prove a document hasn't been modified by a third party.
- Digital fingerprints are hashes computed from known data about a user. The fingerprint is a small number that identifies a tracked user.
- In blockchain based on proof of work the computational difficulty of reversing a hash is used in the process of mining as a puzzle whose solution is rewarded. Miners compete in finding bits such that if appended to a newly added block will result in the block's hash being some defined number.

## Example

Let's say we want a hash function for string which for any ASCII string will output a 32 bit hash. How to do this? We need to make sure that every character of the string will affect the resulting hash.

First thought that may come to mind could be for example to multiply the ASCII values of all the characters in the string. However there are at least two mistakes in this: firstly short strings will result in small values as we'll get a product of fewer numbers (so similar strings such as "A" and "B" will give similar hashes, which we don't want). Secondly reordering the characters in a string (i.e. its permutations) will not change the hash at all (as with multiplication order is insignificant)! These violate the properties we want in a hash function. If we used this function to implement a hash table and then tried to store strings such as "abc", "bca" and "cab", all would map to the same hash and cause collisions that would negate the benefits of a hash table.

A better hash function for strings is shown in the section below.

## Nice Hashes

{ Reminder: I make sure everything on this Wiki is pretty copy-paste safe, from the code I find on the Internet I only copy extremely short (probably uncopyrightable) snippets of public domain (or at least free) code and additionally also reformat and change them a bit, so don't be afraid of the snippets. ~drummyfish }

Here is a simple and pretty nice 8bit hash, it outputs all possible values and all its bits look quite random: { Made by me. ~drummyfish }

```
uint8_t hash(uint8_t n)
{
    n *= 23;
    n = ((n >> 4) | (n << 4)) * 11;
    n = ((n >> 1) | (n << 7)) * 9;

    return n;
}
```

The [hash prospector project \(unlicense\)](#) created a way for automatic generation of integer hash functions with nice statistical properties which work by XORing the input value with a bit-shift of itself, then multiplying it by a constant and repeating this a few times. The functions are of the format:

```
uint32_t hash(uint32_t n)
{
    n = A * (n ^ (n >> S1));
    n = B * (n ^ (n >> S2));
    return n ^ (n >> S3);
}
```

Where A, B, S1, S2 and S3 are constants specific to each function. Some nice constants found by the project are:

| A          | B          | S1 | S2 | S3 |
|------------|------------|----|----|----|
| 303484085  | 985455785  | 15 | 15 | 15 |
| 88290731   | 342730379  | 16 | 15 | 16 |
| 2626628917 | 1561544373 | 16 | 15 | 17 |
| 3699747495 | 1717085643 | 16 | 15 | 15 |

The project also explores 16 bit hashes, here is a nice hash that doesn't even use multiplication!

```
uint16_t hash(uint16_t n)
{
    n = n + (n << 7);
    n = n ^ (n >> 8);
    n = n + (n << 3);
    n = n ^ (n >> 2);
    n = n + (n << 4);
    return n ^ (n >> 8);
}
```

```
}
```

Here is a nice string hash, works even for short strings, all bits look pretty random: { Made by me. Tested this on my dataset of programming identifiers, on average there was one colliding pair of strings in 1000. ~drummyfish }

```
uint32_t strHash(const char *s)
{
    uint32_t r = 21;

    while (*s)
    {
        r = (r * 31) + *s;
        s++;
    }

    r = r * 4451;
    r = ((r << 19) | (r >> 13)) * 5059;

    return r;
}
```

TODO: more

BONUS: Here is a kind of string *pseudohash* for identifiers made only of character a-z, A-Z, 0-9 and `_`, not starting with digit -- it may be useful for symbol tables in compilers. It is parameterized by length  $n$ , which must be greater than 4. It takes an arbitrary length identifier in this format and outputs another string, also in this format (i.e. also being this kind of identifier), of maximum length  $n - 1$  (last place being reserved for terminating zero), which remains somewhat human readable (and is the same as input if under limit length), which may be good e.g. for debugging and transpiling (in transpilation you can just directly use these pseudohashes from the table as identifiers). In principle it works something like this: the input characters are cyclically written over and over to a buffer, and when the limit length is exceeded, a three character hash (made of checksum, "checkproduct" and string length) is written on positions 1, 2 and 3 (keeping the first character at position 0 the same). This means e.g. that the last characters will always be recorded, so if input identifiers differ in last characters (like `myvar1` and `myvar2`), they will always give different pseudohash. Also if they differ in first character, length (modulo something like 64), checksum or "checkproduct", their pseudohash is guaranteed to differ. Basically it should be hard to find a collision. Here is the code: { I found no collisions in my dataset of over 5000 identifiers, for  $n = 16$ . ~drummyfish }

```
char numPseudohash(unsigned char c)
{
    c %= 64;

    if (c < 26)
        return 'a' + c;
    else if (c < 52)
        return 'A' + (c - 26);
    else if (c < 62)
        return '0' + (c - 52);

    return '_';
}

void pseudohash(char *s, int n)
{
    unsigned char
        v1 = 0, // checksum
        v2 = 0, // "checkproduct"
        v3 = 0, // character count
        pos = 0;

    const char *s2 = s;

    while (*s2)
    {
        if (pos >= n - 1)
            pos = 4;
    }
}
```



```

v1 += *s2;
v2 = (v2 + 1) * (*s2);
v3++;

s[pos] = *s2;

pos++;
s2++;
}

if (v3 != pos)
{
    s[1] = numPseudohash(v1);
    s[2] = numPseudohash(v2);
    s[3] = numPseudohash(v3);
}

s[n - 1] = 0;
}

```

Here are some example inputs and output strings:

|                                       |                      |
|---------------------------------------|----------------------|
| "CMN_DES"                             | -> "CMN_DES"         |
| "CMN_currentInstrTypeEnv"             | -> "CBcxrTypeEnvnst" |
| "LONG_prefix_my_variable1"            | -> "L4kyvariable1y_" |
| "TPE_DISTANCE"                        | -> "TPE_DISTANCE"    |
| "TPE_bodyEnvironmentResolveCollision" | -> "TxMJCollisionve" |
| "_TPE_body2Index"                     | -> "_TPE_body2Index" |
| "_SAF_preprocessPosSize"              | -> "_RpwPosSizecess" |

---

hero\_culture

## Hero Culture

Hero culture is a harmful culture of creating and worshiping heroes and "leaders" (and other kinds of celebrities) which leads to e.g. creation of cults of personality, strengthening fight culture and establishing hierarchical, anti-anarchist society of "winners" and "losers". The concept of a hero is one that arose in context of wars and other many times violent conflicts; a hero is different from a mere authority or a well known individual in some area, it is someone who creates fear of disagreement and whose image is distorted to a much more positive, sometimes godlike state, by which he distorts truth and is given a certain power over others. Therefore we highly warn about falling to the trap of hero culture, though this is very difficult in current highly hierarchical society. **To us, the word hero has a pejorative meaning.** Our advice is always this:

**Do NOT create heroes. Follow ideas, not people.** Also similarly: hate ideas, not people, and follow ideas, not groups.

Smart people know this and those being named *heroes* themselves many times protest it, e.g. Marie Curie has famously stated: "be less curious about people and more curious about ideas." Anarchists purposefully don't name theories after their inventors but rather by their principles, knowing the danger of hero culture leading to social hierarchy and also that people are imperfect -- people are like packages, a mixture of both good and bad inadvertently inseparable, they carry distorting associations, they make mistakes and their images are twisted by history and politics -- even the character of Jesus, a "theoretically perfect human", has been many times twisted in ways that are hard to believe. Worshiping an individual always comes with the tendency to embrace and support everything he does, all his opinions and actions, including the extremely bad ones. Abusive regimes are the ones who use heroes and their names for propaganda -- Stalinism, Leninism, corporations such as Ford, named after their founder etc. Heroes become brands whose stamp of approval is used to push bad ideas... especially popular are heroes who are already dead and can't protest their image being abused -- see for example how Einstein's image has been raped by capitalists for their own propaganda, e.g. by Apple's marketing, while in fact Einstein was a pacifist socialist highly critical of capitalism. This is not to say an idea's name cannot be abused, the word communism has for example become something akin a swear word after being abused by regimes that had little to do with real communism. Nevertheless it is still much better to focus on ideas as ideas always carry their own principle

embedded within them, visible to anyone willing to look, and can be separated from other ideas very easily. Focusing on ideas allows us to discuss them critically, it allows us to reject a bad concept without "attacking" the human who came up with it.

Mainstream US mentality of strong hero culture is now infecting the whole world and reaches unbelievably retarded levels, which is further not helped by shit like the stupid superhero movies. Besides calling murderers (soldiers) heroes, it is now for example standard to call handicapped people heroes, literally only because they are handicapped and it makes them feel better, even if they do nothing special and even if they actually live more comfortable lives than poor healthy peasants who have to live miserably and slave at work every day without getting anyone's attention. Or -- and this is yet another level of stupidity -- **anyone who just happens to not behave like a dick in case of some emergency is guaranteed to be called a hero**; for example if someone by chance walks by a baby that is drowning in a pool and saves the baby from dying will with 100% probability be called a hero in the media. But WHY the fuck would that be? Is the guy a hero because he didn't just sit down and watch the baby drown? It is the absolutely normal behavior to save a drowning baby if one sees it, especially when there is very little risk of own life in doing so (such as just jumping into the pool); calling someone a hero for doing so is like calling a gun owner a hero for not going to the streets to randomly shoot at people. So in this fucked up society the title of *hero* is basically won like a lottery -- you just have to be lucky enough to be present at some emergency and then just do the normal thing.

On a bit more lighthearted note: in Internet meme slang "an hero" stands for committing suicide.

---

hero

## Hero

HEROES ARE HARMFUL. See hero culture.

---

hexadecimal

## Hexadecimal

TODO

Some hexadecimal values that are also English words at the same time and which you may include in your programs for fun include: ace, add, babe, bad, be, bee, beef, cab, cafe, dad, dead, deaf, decade, facade, face, fee, feed. You may also utilize digits here (recall the famous number 80085 that looks like B00BS); 0 = 0, 1 = I, 2 = Z, 5 = S, 6 = G, 8 = B (already available though).

---

history

## History

{ Though history is usually written by the winners, this one was written by a loser :) Keep in mind there may appear errors, you can send me an email if you find some. ~drummyfish }

This is a brief summary of history of technology and computers (and some other things). For those who don't know history are doomed to repeated it.

{ A curious pattern of history is that the civilization -- or maybe rather the dominating superpowers -- are moving to the west, kind of like: middle East -> Greece -> Rome -> Holy Roman Empire -> England/France/Spain -> America. ~drummyfish }

The earliest known appearance of technology related to humans may likely be the use of **stone tools** by hominids in Africa some two and a half million years ago -- this is even before the appearance of modern humans, homo sapiens, that emerged roughly 600000 years ago. Learning to start and control **fire** was another key invention of the earliest men; this probably happened hundreds of thousands to millions years

ago, even before modern humans. Around 8000 BC the **Agricultural Revolution** happened: this was quite a disaster -- as humans domesticated animals and plants, they had to abandon the comfortable life of hunters and gatherers and started to suffer the life of a farmer, full of extremely hard work in the fields (this can be seen e.g. from their bones). This led to the establishment of first cities that would later become city states (as the name says -- something between a city and a state, i.e. greatly independent cities with their own laws etc.). Some of the first such cities were Ur and Uruk in Mesopotamia, since around 5000 BC. Primitive **writing** can be traced to about 7000 BC to China. **Wheel** was another crucial piece of technology humans invented, it is not known precisely when or where it appeared, but it might have been some time after 5000 BC -- in Ancient Egypt **The Great Pyramid** was built around 2570 BC still without the knowledge of wheel. Around 4000 BC **history starts with first written records**. Humans learned to smelt and use metals approximately 3300 BC (**Bronze Age**) and 1200 BC (**Iron Age**). **Abacus**, one of the simplest digital devices aiding with computation, was invented roughly around 2500 BC. However people used primitive computation helping tools, such as bone ribs, probably almost from the time they started trading. Babylonians in around 2000 BC were already able to solve some forms of **quadratic equations**.

In Greek many city states, such as Athens, Delphi and Sparta formed -- Ancient Greek culture would be seen as the golden age of civilization that would lay foundations to everything we now take for granted; Greeks to some extent advanced technology (e.g. architecture) but especially cultivated art, philosophy and politics -- Athens are credited for inventing democracy (though an "early" version, they still had slaves and many classes of citizens without voting power). In 8th century BC Homer created the epic poems Iliad and Odyssey. In 6th century BC Pythagoras describes the Pythagorean theorem. After 600 BC the Greek philosophy starts to develop which would lead to strengthening of rational, scientific thinking and advancement of logic and mathematics. Some of the most famous Greek philosophers were Socrates, Plato, Aristotle and Diogenes. Around 400 BC **camera obscura** was already described in a written text from China where **gears** also seem to have been invented soon after. Around 300 BC Euklid wrote his famous *Elements*, a mathematical work that proves theorems from basic axioms. Ancient Greeks could communicate over great distances using **Phryctoria**, chains of fire towers placed on mountains that forwarded messages to one another using light. 234 BC Archimedes described the famous Archimedes screw and created an **algorithm for computing the number pi**. In 2nd century BC the **Antikythera mechanism, the first known analog computer** is made to predict movement of heavenly bodies. Romans are known to have been great builders, they built many roads and such structures as the Pantheon (126 AD) and aqueducts with the use of their own type of **concrete** and advanced understanding of physics.

44 BC Julius Caesar, most famous leader of Ancient Rome, is killed. Rome has to be mentioned as at its time it was the biggest world superpower -- though it was a greatly corrupt, imperialist empire heavily based on work of slaves, Rome advanced technology in many ways, e.g. by inventing concrete, building roads and very long lasting aqueducts. They build monuments that would last for thousands of years, e.g. the famous Colosseum.

Around 50 AD Heron of Alexandria, an Egyptian mathematician, created a number of highly sophisticated inventions such as a **vending machine** that accepted coins and gave out holy water, and a cart that could be "programmed" with strings to drive on its own.

In the 3rd century Chinese mathematician Liu Hui describes operations with **negative numbers**, even though negative numbers have already appeared before. In 600s AD an Indian astronomer Brahmagupta first used the number **zero** in a systematic way, even though hints on the number zero without deeper understanding of it appeared much earlier. In 9th century the Mayan empire is collapsing, though it would somewhat recover and reshape.

Year 476 is set to mark the fall (political split) of Roman empire and by this the end of Antiquity and **start of Middle Ages**, a time during which technological progress and art is seen to stagnate a bit. Rome had been collapsing slowly but in its downfall it greatly resembled our current western society, it became split, people got spoiled, lost sense of morality, women started to demand more power and so on -- Roman empire was basically like the ancient times US (with a similar relationship to Greece as US has to the older, wiser Europe) with highly capitalist practices (free trade, ads, banks, insurance, even industries that achieved quite high mass production, ...), imperialism, military obsession, fascism, constant political fights, pragmatic thinking (e.g. rhetoric, the art of manipulation, was greatly preferred over excellence at art), mass entertainment and huge competitiveness -- this all led to its demise.

In 1429 Persian mathematician al-Kashi computed pi to about 14 digit accuracy which was a great leap in this discipline.

Around the year of our Lord 1450 a major technological leap known as the **Printing Revolution** occurred. Johannes Gutenberg, a German goldsmith, perfected the process of producing books in large quantities with the movable type press. This made books cheap to publish and buy and contributed to fast spread of information and better education. Around this time the **Great Wall of China** is being built.

They year 1492 marks the **discovery of America** by Christopher Columbus who sailed over the Atlantic Ocean, though he probably wasn't the first in history to do so, and it wasn't realized he sailed to America until after he died (he thought he sailed to India). This is sometimes taken to mark the **end of Middle Ages** and transition to **Renaissance**. This was a time of increased interest in rationality, science and art; Renaissance saw man as a potent creation of God, who is capable of creating on his own rather than being mere blind, obedient servant of God. Great many polymath lived at this time, most notably Leonardo da Vinci (probably gay) who was an excellent painter, explored human anatomy and even subjects such as astronomy and engineering. On one hand Renaissance brought beautiful art and new technology, on the other hand it further shifted society toward capitalism and selfish thinking, human became more self centered, egoistic and art became even more a matter of business -- for example the great painters infamously hired lesser artists to make copies of their paintings which were then sold almost like consumer products.

During 1700s a major shift in civilization occurred, called the **Industrial Revolution** -- this was another disaster that would lead to the transformation of common people to factory slaves and loss of their self sufficiency. The revolution spanned roughly from 1750 to 1850. It was a process of rapid change in the whole society due to new technological inventions that also led to big changes in how a man lived his daily life. It started in Great Britain but quickly spread over the whole world. One of the main changes was the **transition from manual manufacturing to factory manufacturing** using machines and sources of energy such as coal. Steam engine played a key role. Work became a form of a highly organized slavery system, society became industrialized. This revolution became highly criticized as it unfortunately opened the door for capitalism, made people dependent on the system as everyone had to become a specialized cog in the society machine, at this time people started to measure time in minutes and lead very planned lives with less joy. But there was no way back.

In 1712 Thomas Newcomen invented the first widely used steam engine used mostly for pumping water, even though steam powered machines have already been invented long time ago. The engine was significantly improved by James Watt in 1776. Around 1770 Nicolas-Joseph Cugnot created a first somewhat working **steam-powered car**. In 1784 William Murdoch built a small prototype of a **steam locomotive** which would be perfected over the following decades, leading to a transportation revolution; people would be able to travel far away for work, the world would become smaller which would be the start of globalization. The railway system would make common people measure time with minute precision.

In 1792 Claude Chappe invented optical telegraph, also called *semaphore*. The system consisted of towers spaced up to by 32 km which forwarded textual messages by arranging big arms on top of the towers to signal specific letters. With this messages between Paris and Strasbourg, i.e. almost 500 km, could be transferred in under half an hour. The system was reserved for the government, however in 1834 it was **hacked** by two bankers who bribed the tower operators to transmit information about stock market along with the main message (by setting specific positions of arms that otherwise didn't carry any meaning), so that they could get an advantage on the market.

By 1800 Alessandro Volta invented an **electric battery**. In 1827 Andr  -Marie Amp  re publishes a further work shedding light on electromagnetism. After this electric telegraph would be worked on and improved by several people and eventually made to work in practice. In 1821 Michael Faraday invented the electromotor. Georg Ohm and especially James Maxwell would subsequently push the knowledge of electricity even further.

In 1822 Charles Babbage, a great English mathematician, completed the first version of a manually powered **digital mechanical computer** called the Difference Engine to help with the computation of polynomial derivatives to create mathematical tables used e.g. in navigation. It was met with success and further development was funded by the government, however difficulties of the construction led to never finishing the whole project. In 1837 Babbage designed a new machine, this time a **Turing complete general**

**purpose computer**, i.e. allowing for programming with branches and loops, a true marvel of technology. It also ended up not being built completely, but it showed a lot about what computers would be, e.g. it had an assembly-like programming language, memory etc. For this computer Ada Lovelace would famously write the Bernoulli number algorithm.

In 1826 or 1827 French inventor Nicéphore Niépce captured **first photography** that survived until today -- a view from his estate named Le Gras. About an 8 hour exposure was used (some say it may have taken several days). He used a camera obscura and asphalt plate that hardened where the light was shining. Earlier cases of photography existed maybe as early as 1717, but they were only short lived.

**Sound recording** with phonograph was invented in 1857 in Paris, however it could not be played back at the time -- the first record of human voice made with this technology can nowadays be reconstructed and played back. It wouldn't be until 1878 when people could both record and play back sounds with Edison's improvement of phonograph. A year later, in 1879, Edison also patented the **light bulb**, even though he didn't invent it -- there were at least 20 people who created a light bulb before him.  $\pi$  at this time is evaluated to roughly 500 digit accuracy (using Machin's formula).

Around 1888 so called **war of the currents** was taking place; it was a heated battle between companies and inventors for whether the alternating or direct current would become the standard for distribution of electric energy. The main actors were Thomas Edison, a famous inventor and a huge capitalist dick rooting for DC, and George Westinghouse, the promoter of AC. Edison and his friends used false claims and even killing of animals to show that AC was wrong and dangerous, however AC was objectively better, e.g. by its efficiency thanks to using high voltage, and so it ended up winning the war. AC was also supported by the famous genius inventor Nikola Tesla who during these times contributed hugely to electric engineering, he e.g. invented an AC motor and Tesla coil and created a system for wireless transmission of electric power.

Also in 1888 probably the **first video** that survived until today was recorded by Lou Le Prince in Northern England, with a single lens camera. It is a nearly 2 second silent black and white shot of people walking in a garden.

1895 can roughly be seen as the year of **invention of radio**, specifically wireless telegraph, by Italian engineer and inventor Guglielmo Marconi. He built on top of work of others such as Hertz and Tesla and created a device with which he was able to wirelessly ring a bell at a distance over 2 km.

On December 17 1903 the Wright brothers famously performed the **first controlled flight of a motor airplane** which they built, in North Carolina. In repeated attempts they flew as far as 61 meters over just a few seconds.

From 1914 to 1918 there was **World War I**.

Around 1915 Albert Einstein, a German physicist, completed his **General Theory of Relativity**, a groundbreaking physics theory that describes the fundamental nature of space and time and gives so far the best description of the Universe since Newton. This would shake the world of science as well as popular culture and would enable advanced technology including nuclear energy, space satellites, high speed computers and many others.

In 1907 Lee De Forest invented a practically usable **vacuum tube**, an extremely important part usable in electric devices for example as an amplifier or a switch -- this would enable construction of radios, telephones and later even primitive computers. The invention would lead to the electronic revolution.

In 1924 about 50% of US households own a car.

October 22 1925 has seen the invention of **transistor** by Julius Lilienfeld (Austria-Hungary), a component that would replace vacuum tubes thanks to its better properties, and which would become probably the most essential part of computers. At the time the invention didn't see much attention, it would only become relevant decades later.

In 1931 Kurt Gödel, a genius mathematician and logician from Austria-Hungary (nowadays Czech Republic), published revolutionary papers with his incompleteness theorems which proved that, simply put, mathematics has fundamental limits and "can't prove everything". This led to Alan Turing's publications in

1936 that nowadays stand as the **foundations of computer science** -- he introduced a theoretical computer called the **Turing machine** and with it he proved that computers, no matter how powerful, will never be able to "compute everything". Turing also predicted the importance of computers in the future and has created several algorithms for future computers (such as a chess playing program).

In 1938 Konrad Zuse, a German engineer, constructed **Z1, the first working electric mechanical digital partially programmable computer** in his parents' house. It weighted about a ton and wasn't very reliable, but brought huge innovation nevertheless. It was programmed with punched film tapes, however programming was limited, it was NOT Turing complete and there were only 8 instructions. Z1 ran on a frequency of 1 to 4 Hz and most operations took several clock cycles. It had a 16 word memory and worked with floating point numbers. The original computer was destroyed during the war but it was rebuilt and nowadays can be seen in a Berlin museum. Zuse also soon created what's regarded as the **first programming language**, Plankalkul.

From 1939 to 1945 there was **World War II**.

In hacker culture the period between 1943 (start of building of the ENIAC computer) to about 1955-1960 is known as the **Stone Age of computers** -- as the Jargon File puts it, the age when electromechanical dinosaurs ruled the Earth.

In 1945 the construction of **the first electronic digital fully programmable computer** was completed at University of Pennsylvania as the US Army project. It was named **ENIAC** (Electronic Numerical Integrator and Computer). It used 18000 vacuum tubes and 15000 relays, weighted 27 tons and ran on the frequency of 5 KHz. Punch cards were used to program the computer in its machine language; it was Turing complete, i.e. allowed using branches and loops. ENIAC worked with signed ten digit decimal numbers. Also in 1945 **USA used two nuclear bombs to murder hundreds of thousands of civilians** in Japanese cities Hiroshima and Nagasaki.

Among hackers the period between 1961 to 1971 is known as the **Iron Age of computers**. The period spans time since the first minicomputer (PDP1) to the first microprocessor (Intel 4004). This would be followed by so called *elder days*.

On July 20 1969 **first men landed on the Moon** (Neil Armstrong and Edwin Aldrin) during the USA Apollo 11 mission. This tremendous achievement is very much attributed to the cold war in which USA and Soviet Union raced in space exploration. The landing was achieved with the help of a relatively simple on-board computer: Apollo Guidance Computer clocked at 2 MHz, had 4 KiB of RAM and about 70 KB ROM. The assembly source code of its software is nowadays available online.

Shortly after, on 29 October 1969, another historical event would happen that could be seen as the start of perhaps the greatest technological revolution yet, the **start of the Internet**. The first letter, "L", was sent over a long distance via **ARPANET**, a new experimental computer packet switching network without a central node developed by US defense department (they intended to send "LOGIN" but the system crashed). The network would start to grow and gain new nodes, at first mostly universities. The network would become the Internet.

1st January 1970 is nowadays set as the start of the **Unix epoch**. It is the date from which Unix time is counted. During this time the **Unix operating system, one of the most influential operating systems** was being developed at Bell Labs, mainly by Ken Thompson and Dennis Ritchie. Along the way they developed the famous Unix philosophy and also the **C programming language**, perhaps the most influential programming language in history. Unix and C would shape the technology far into the future, a whole family of operating systems called Unix-like would be developed and regarded as the best operating systems thanks to their minimalist design.

By 1977 ARPANET had about 60 nodes.

August 12 1981 would see the released of **IBM PC**, a personal computer based on open, modular architecture that would immediately be very successful and would become the de-facto standard of personal computers. IBM PC was the first of the kind of desktop computers we have today. It had 4.77 MHz Intel 8088 CPU, 16 kB of RAM and used 5.25" floppy disks.

In 1983 **Richard Stallman** announced his **GNU project** and invented **free (as in freedom) software**, a kind of software that is freely shared and developed by the people so as to respect the users' freedom. This kind of ethical software stands opposed to the proprietary corporate software, it would lead to creation of some of the most important software and to a whole revolution in software development and its licensing, it would spark the creation of other movements striving for keeping ethics in the information age.

1985: on November 20 the first version of the **Windows operating system** was sadly released by Microsoft. These systems would become the mainstream desktop operating systems despite their horrible design and they would unfortunately establish so called Windows philosophy that would irreversibly corrupt other mainstream technology. Also in 1985 one of the deadliest software bugs appeared: that in Therac-25, a medical radiotherapy device which fatally overdosed several patients with radiation.

On April 26 1986 the **Chernobyl nuclear disaster** happened (the worst power plant accident in history) -- in north Ukraine (at the time under USSR) a nuclear power plant exploded, contaminated a huge area with radioactivity and released a toxic radioactive cloud that would spread over Europe -- many would die either directly or indirectly (many years later due to radioactivity poisoning, estimated at many thousands). The Chernobyl area would be sealed in the 30 km radius. It is estimated the area won't be habitable again for several thousands of years.

Around this time Internet is not yet mainstream but it is, along with similar local networks, working and has active communities -- there is no world wide web yet but people are using Usenet and BBSes for "online" discussions with complete strangers and developing early "online cultures".

At the beginning of 1991 Tim Berners-Lee created the **World Wide Web**, a network of interlinked pages on the Internet. This marks another huge step in the Internet revolution, the Web would become the primary Internet service and the greatest software platform for publishing any kind of information faster and cheaper than ever before. It is what would popularize the Internet and bring it to the masses.

Shortly after the **Soviet Union dissolved** and on 25 August 1991 **Linus Torvalds** announced **Linux**, his project for a completely free as in freedom Unix-like operating system kernel. Linux would become part of GNU and later one of the biggest and most successful software projects in history. It would end up powering Internet servers and supercomputers as well as desktop computers of a great number of users. Linux proved that free software works and surpasses proprietary systems.

After this very recent history follows, it's hard to judge which recent events will be of historical significance much later. 1990s have seen a huge growth of computer power, video games such as Doom led to development of GPUs and high quality computer graphics along with a wide adoption of computers by common people, which in turn helped the further growth of Internet. In around mid 90s the web overtook gopher in popularity and started to become the forefront of the Internet. Late 90s saw the rise of the "open source" movement (OSI was established in 1998). Year 2000 was infamously preceded by the Y2K hysteria, the fear of technological collapse that was to be caused by computers flipping from year 99 to 00 -- this of course didn't happen. Shortly after 2000 Lawrence Lessig founded Creative Commons, an organization that came hand in hand with the free culture movement inspired by the free software movement. At this point over 50% of US households had a computer. From 2005 we've seen a boom of social networks like Facebook, Twitter and YouTube and also skyrocketing popularity of online and massively online games, owing a lot to the gigantic success of World of Warcraft; all of these contributed to making Internet and computers one of the most mainstream and lucrative things, ruining everything. Cell phones became a commonly owned item and after about 2005 so called "smart phones" and other "smart" devices replaced them as a universal communication device capable of connecting to the Internet. Year 2010 seems to be the turning point beyond which societal decline accelerated immensely; 1990s seem to have been the peak of society, after the year 2000 society started to slowly decline but by inertia things were still relatively good for about another decade. In 2011 Minecraft was released. After this we've seen the rise of Bitcoin and other cryptocurrencies. Before 2020 we've also seen a brief spike in popularity of VR (that would diminish again) and a huge advancement in neural network Artificial Intelligence which will likely be the topic of the future. Quantum computers are being highly researched with already existing primitive prototypes; this will also likely be very important in the following years. Besides AI there were also drones, electromobiles, robotic Mars exploration and other things. However the society and technology have been in decadence for some time now, capitalism has pushed technology to become hostile and highly abusive to users, extreme bloat of technology causes highly inefficient, extremely expensive and unreliable technology. In addition society is dealing with a lot of serious issues such as the global warming and many people are foreseeing a collapse of

society.

## Recent History Of Technology

TODO: more detailed history since the start of Unix time

---

holy\_war

## Holy War

Holy war is a long passionate argument over a choice (many times between two options) that touches an issue deemed controversial and/or "religious" within given community; to an outsider this may seem like a childish, hard to understand rant about an insignificant thing. In technology circles holy wars revolve e.g. around operating systems, text editors, programming languages, licenses, source code formatting etc. Such a war separates people into almost religious groups that sometimes argue to death about details such as what name something should be given, very much resembling traditional disagreements between religions and their churches -- you would think that people being on the same ship (e.g. all being promoters of free software) would at least get along better with each other than with outsiders but no, great many times these people hate each other over quite small details, though there is often a good reason too, just hard to spot to those without deeper insight (a choice regarding minor detail may tell a lot about that who makes it). In holy wars people tend to defend whichever side they stand on to the beyond grave and can get emotional when discussing the topic, leading to flame wars, ragequits, Hitler arguments etc. Some examples of holy wars are (in brackets indicated the side taken by LRS):

- **tabs vs spaces** (spaces)
- **vim vs emacs vs acme and other text editors** (vim)
- **free software vs open source vs proprietary vs whatever else** (free software)
- **Chrome vs Firefox**, and other browsers (none of this bloated shit)
- **Java vs C++, Lisp vs Forth and other programming languages** (C, comun)
- **curly brackets on separate lines or not**, and other style choices
- **KDE vs GNOME** (neither, both are bloat)
- **pronunciation of gif as "gif" vs "jif"**
- **Windows vs Mac** (neither, this is a normie holy war)
- **"GNU/Linux" vs "Linux"** (GNU/linux)
- **copyleft vs permissive** (permissive, public domain)
- **AMD vs Intel**
- **AMD vs NVidia**
- **"Linux" distros**
- **window managers**
- **Metric vs Imperial units** (metric)
- **Star Trek vs Star Wars**, and other franchise wars
- **Quake vs Unreal Tournament**, and similar gaming shit
- **gopher vs gemini** (gopher)
- ...

Things like cats vs dogs or sci-fi vs fantasy may or may not be a holy war, there is a bit of a doubt in the fact that one can easily like both and/or not be such a diehard fan of one or the other. A subject of holy war probably has to be something that doesn't allow too much of this. Maybe a controversy of its own could be the very topic of "what is a holy war" in itself.

---

how\_to

## How To

WELCOME TRAVELER

{ Don't hesitate to contact me. ~drummyfish }



Are you tired of bloat and can't stand shitty software like Windows anymore? Do you want to kill yourself? Do you hate capitalism? Do you also hate the fascist alternatives you're being offered? Do you just want to create a genuinely good bullshitless technology that would help all people? Do you just want to share knowledge freely without censorship? You have come to the right place.

Firstly let us welcome you, no matter who you are, no matter your political opinions, your past and your skills, color or shape of your genitalia, we are glad to have you here. Remember, you don't have to be a programmer to help and enjoy LRS. LRS is a lifestyle, a philosophy. Whether you are a programmer, artist, educator or just someone passing by, you are welcome, you may enjoy our culture and its fruit and if you want, you can help enrich it.

## What This Article Is About

OK, let's say this is a set of general advice, life heuristics, pointers and basics of our philosophy, something to get you started, give you a point of view aligned with what we do, help you make a decision here and there, help you free yourself. Remember that by definition **nothing we ever advice is a commandment** or a rule you mustn't ever break, that would be wrong in itself. Some things also may be yet a "thought in progress" and change.

## How To Read This How To

Use your eyes to read the letters from left to right and top to bottom. If this is too hard read a how to read a how to read a how to.

## Required Time To Read

Depends on how fast you read.

## What You Will Learn

You will learn things that are both:

- written here and
- you didn't know before

## Prerequisites

- brain
- eyes (not needed if you're using TTS)

## Where To Go Next

Wherever you want, this is no dictatorship.

## Moderacy (Middle Way) Vs Extremism

An important issue of many ideologies/philosophies/religions/etc. has shown to be striking the right balance between moderacy and extremism. Let's sum up the two stances:

- **extremism**: Being extreme in applying the ideas and principles, holding to one's ideals extremely strongly, many times resulting in blind orthodoxy, shortcut thinking, blindly following rules and commandments such as "I must never do X", "X implies Y" etc. Extremism is not bad per se, in fact it is many times preferred, advised and necessary, however one has to be aware of the dangers. It may lead to becoming a brainwashed religion follower whose pursuit of perfectionism and purism result in more bad than good.
- **moderacy**: Being moderate, holding to ideals only loosely, sometimes leading to pragmatism, "ends justify the means", hypocrisy, conveniently modifying rules on the go etc. Moderacy is also not bad as such, but also comes with many dangers. It may lead to becoming an immoral self-centered sheep

conformist and even practically abandoning one's ideals, giving up (everyone does it, "Yeah I don't really like capitalism, but that's how it is so I'll just play along for now."), lying to oneself ("I do so much good by setting an Ukrainian flag as my facebook profile picture!").

**Where does the balance lie?** TBH this is a very hard question and we don't know the correct answer so far, perhaps there is no simple answer. Figuring this out may be one of the most difficult parts of our philosophy. The first good step is definitely to realize the issue, become aware of it, and start considering it in making one's important decisions. Choosing one or another should, as always, be done by ultimately aiming for our ideals, not for one's own benefit, though of course as any mere living being one will never be able to be completely objective and free himself from things such as fear and self-preservation instincts. If you make a bad decision, don't bash yourself, you are just mere mortal, acknowledge your mistake, forgive yourself and move on, there is no use in torturing yourself. One should perhaps not try to stick to either extremism and moderacy as a rule, but rather try to apply a differently balanced mix of both to any important decision that appears before him -- when unsure about the balance, a middle way between is probably safest, but when you strongly feel one way is morally more right, go for it.

Examples from LRS point of view:

- Is it OK to ever use violence? Here LRS takes the extremist way of strongly saying no -- according to us violence is always bad and we define this as an axiom, something without a need of proof, it is the very foundation of our movement and not acknowledging it would simply mean it's not LRS anymore. However a bit of moderacy may also appear here; if for example someone uses violence in a desperate attempt to protect one's child, though we won't embrace the action we won't condemn the man either -- he committed a "sin", did something wrong, but in his situation there was really no right thing to do, so what should we blame him for, for being a subject of unfortunate situation?
- Is it OK to sometimes use proprietary software? Here for example Richard Stallman/FSF/GNU take the extremist stance and say no, proprietary software is the literal devil and though shalt evade it for all cost (in fact GNU will put effort in purposefully breaking compatibility with proprietary software, which is borderline capitalist behavior similar to artificial obsolescence etc.). While we agree it is a good general rule to avoid software whose purpose is almost exclusively the abuse of its user, we may be more tolerant and allow breaking the rule sometimes, because to us proprietary software is nothing set in any axiom, it is just a symptom resulting from bad society. As a non-axiom it should be a subject to constant reevaluation against the main goal. A simple commandment of "NO TOUCH NOTHING PROPRIETARY" is a good tool for a newcomer, it is a simple to follow rule of thumb that teaches him to find free replacements and alternatives, however once one becomes advanced and eventually a master of the freedom philosophy, he sees things aren't as simple to be solved by one simple rule, just as a master of music knows when to break basic rules of thumb, when to leave the scale, break the rhythm to make excellent music. Here we see it similarly: When touching proprietary software doesn't result in significant harm (such as supporting its developer, becoming addicted to it, getting abused by it, ...) and when it does significant good (e.g. inspires creation of its free clone, reveals the mechanisms by which it abuses its users, ...), it may in fact be good to do so.
- Should you oppose your boss at work, deny to serve him in unethical practice because he is a filthy capitalist and so make trouble for yourself, possibly even get fired for it? Well, this is not so easy again; a strict extremist anticapitalist here would just stay without a job because he couldn't work as any work supports capitalism. On the other hand such a guy would just be homeless, rid of any practical opportunity to create and do good, and would probably die soon anyway. Here it's more or less a question of personal tuning, finding the "least harmful" job, minimizing time spent at it so as to be able to do good in spare time, opposing your boss sometimes but not every single time, not really building a career so that you may quit at any moment etc. Until we have basic income or something, you are more or less doomed to suffer dealing with this on your own sadly.

## Tech

Here are some extremely basic steps to take regarding technology and the technological aspect of LRS:

- **Learn about the most essential topics and concepts**, mainly free software, "open-source", bloat, minimalism, kiss, capitalism, capitalist software, suckless, LRS, less retarded society, anacho, pacifism, type A/B fail etc. You will also need to open up your mind and re-learn some toxic concepts you've been taught by the system, e.g. we do NOT fight anything, we do NOT create any heroes, "leaders" or celebrities (we follow ideas, not people), admit work is shit, older is better than "modern"

etc.

- **Install GNU/Linux** operating system to free yourself from shit like Windows and Mac (you can also consider BSD and similar free OSes but you're yet probably too noob for that at this point). Do NOT try to switch to "Linux" right away if it's your first time, it's almost impossible, you want to just install "Linux" as dual boot (alongside your main OS) or on another computer (easier). This way you'll be using both operating systems, slowly getting more comfortable with "Linux" and eventually you'll find yourself uninstalling Windows altogether. You can also just try "Linux" in a virtual machine, from a live CD/flash drive or you can buy something with "Linux" preinstalled like Raspberry Pi. **Which "Linux" to install?** There are many options and as a noob you don't have to go hardcore right away, just install any distro that just works (don't listen to people who tell you to install Gentoo tho). Remember, perfect distro doesn't exist, all tech is shit nowadays, just choose something and go with it. You can try these:
  - ♦ Devuan: Nice, LRS approved distro that respects your freedom that just works, is easy to install and is actually nice. Good for any skill level.
  - ♦ Debian: Like Devuan but uses the evil systemd which doesn't have to bother you at this point. Try Debian if Devuan doesn't work for any reason.
  - ♦ Mint: More noob, bloated and mainstream distro that only mildly cares about freedom, but is extremely easy and works almost everywhere. Try this if Debian didn't work for you.
  - ♦ Ubuntu: Kind of like Mint, try it if Mint didn't work.
  - ♦ Puppy Linux: Tiny kind of a "toy" distro that uses very little resources.
- **Learn a bit of command line** (Unix utils, bash etc.). No need to become a hacker right away, just get familiar with this essential Unix environment.
- **Free yourself technologically**, i.e. make yourself depend as little as possible on capitalist technology; this step is crucial, you can't really live well or achieve anything while being a slave. This includes firstly leaving proprietary platforms such as Facebook, Google's platforms such as YouTube, reddit etc. Also stop being dependent on proprietary programs (MS office, photoshop etc.), and proprietary consumer devices such as a smartphone. Again, it's impossible to free yourself 100% immediately, go slowly and try to get more freedom even if you can't achieve 100% freedom. This means either stop using harmful software/services/devices and engaging in bad habits (social media etc.) or at least minimize their use, and/or use more freedom-friendly alternatives such as different search engines (e.g. searx, ...), a dumbphone or at least free OS smartphone rather than capitalist smartphone, freedom friendly laptop (e.g. an old thinkpad) rather than iShit or consumerist gayming PC, start using **FOSS programs**, e.g. GIMP instead of Photoshop, LibreOffice instead of MS Office etc, invidious or Peertube instead of YouTube etc. Remember, it is best if you can stop using something altogether, the second best thing is to stop being dependent on a single entity, try to use a decentralized and/or suckless **FOSS** alternative but do not try to just mimic your old habits in the FOSS world, you have to learn new ways of computing (for example start using multiple search engines instead of relying on one, it's not good to just drop-in replace one search engine for another). Avoid falling to traps of shit like distrohopping, this just enslaves you in a different way.
- If you want to program LRS, **learn C** (see the tutorial). Also learn a bit of POSIX shell and maybe some mainstream scripting language (can be even a bloated one like Python). Learn about licensing and version control (git). As you advance, start studying deeper topics such as history or hacker culture etc.
- Optionally make your own minimal website (or even a gopherhole) to help reshare ideas you like (static HTML site without JavaScript). This is very easy, and the site can be hosted for free e.g. on git hosting sites like Codeberg or GitLab. Get in touch with us.
- **Start creating**: either programs or other stuff like free art, educational materials etc. Remember, creating is the most important thing to do, it is more important than setting up a perfectly free suckless LRS system, don't fall to the trap of becoming obsessed and paralyzed by hopping, ricing etc. Your system is just a tool, it is worth nothing if it's not used for creating something, and it doesn't really matter which text editor or operating system you used to write your program.
- profit???

Would you like to create LRS but don't have enough spare time/money to make this possible? You can check out making living with LRS.

## How To Make A Website

{ If you REALLY want something dead simple to quickly make a site, try <https://rentry.co>. Making a real custom website is still better if you can. ~drummyfish }

Making your own tiny independent website is pretty simple and a very good thing to do for being able to share opinions and files relatively freely -- using "social networks" for sharing non-mainstream stuff will not work as these get hardcore censored (yes, even the "FOSS" ones like [Mastodon](#) etc.). By making your own website you also help decentralize the [web](#) again, take a bit of control from the [corporations](#), and you can greatly help others by sharing useful information with them. See also [smol internet](#). Watch out though, getting into controversial topics on your site will nowadays greatly complicate your life, so firstly set up some "neutral" site and once you get into it, look for ways to somehow add [free speech](#) to that. Even if you keep using mainstream social media, it's good to also have your own site and have a link to it on your profile.

Also please take a look at [gopher](#) (a much better alternative to web) and how to make your own gopherhole -- mainstream web is really becoming unusable, uninhabitable and will die soon, moving to gopher (or hosting your site both on the web and gopher, which is the best option possibly) is a good thing to do, you will not only simplify your life and avoid a lot of censorship but you will also support this smaller network. Another way of sharing your stuff is through things like [torrents](#), [IPFS](#) and so on. But back to websites now.

Here we will quickly sum up how to make a **[static](#), [single page](#) plain [HTML](#) website without TLS (https)**, which should suffice for most things (sharing opinions, contacts, files, multimedia, simple blogging, ...). Once you get more advanced you can do fancy stuff like this wiki (multi-page wiki written in [Markdown](#), compiled to HTML with a shell script etc.).

**NOTE on TLS (https):** most sites on the web nowadays use encryption for MUH SECURITY obsession and also web browsers kinda prefer such sites etc. (in the future it will probably be required but by then we'll already be elsewhere) -- such site addresses are prefixed with `https://`, as opposed to normal non-encrypted `http://`. [Encryption](#) is huge [bloat](#) and mess to set up, normally you need to pay extra money to get a [certificate](#) for it (though services like Let's Encrypt provide certificates for free) etc. -- basically you only need encryption if you have an interactive site where passwords or other sensitive info gets sent, a purely static site basically doesn't need encryption at all, however if your site doesn't support encryption it may get some penalty by search engines and browsers as they won't "trust it as much", it's just a form of internet bullying for not conforming to latest encryption hysteria. All in all if you can set up encryption easily (e.g. with a single button on your web hosting provider site), do it just for the sake of normies; if you are experienced and can set it up yourself easily, also do it, but if not, just don't care about it and run your site on `http://` only, at least for now until you get into this stuff. Also very importantly **always support plain unencrypted http** even if you set up https, otherwise you're bullying simple browsers that don't implement encryption.

Now **do NOT follow mainstream tutorials on making website** (Wordpress, PHP, static generators, ...) -- these are absolute horseshit and just follow ugly capitalist ways, you will just get brain cancer. Also do NOT use any frameworks; **do NOT even use static site generators** -- these are not needed at all! All you really need for making a small website is:

- **Plain text editor** ([gedit](#), [geany](#), [vim](#), [emacs](#), [ed](#), ...). This is easy, just download it. Just don't use a [rich text](#) editor (MS Office, Libreoffice, ...), ok? That doesn't work!
- **Static site hosting**, i.e. a publicly accessible web server to store your site on, which will serve the site to clients. You have several options here:
  - ♦ There exist free static site hosting services, e.g. those on many [git](#) hosting platforms like [GitLab](#) or [Codeberg](#) (even [GitHub](#), but avoid that one if possible), on [pubnix](#) or sites like [neocities](#). You may just search for [free web hosting](#) on [wiby](#) or something. Here you may still encounter some censorship, but it can be a good start. Just search their site for details on how to host a site there -- usually you will get an [FTP](#) or [SSH](#) access and just upload your website there. Some have normie friendly web interface so you don't even have to deal with [command line](#), but it's better to not rely on them, learn to do it properly.
  - ♦ You may host your site at home, typically using [Raspberry Pi](#). This doesn't really cost anything as the weaker Raspberrys (e.g. 3B) consume negligible amount of electricity, and for non-extreme traffic you won't even need a super high speed connection (especially considering you will make a very tiny, efficient website). This is a very good option as practically no one will be able to censor you (only police and ISP), but it's also a tiny bit more difficult to set up because firstly you need to set up a webserver ([Apache](#) is usually installed on any GNU/Linux distro though, it's really easy to do) and secondly you **NEED A PUBLIC IP ADDRESS** (as typically you will be behind a [NAT](#) so that computers from outside can't reach your server, but if you have an IPv6 IP address you may already be publicly accessible!): you

will probably have to ask your internet provider for it (maybe you already have it, maybe they will give it to you for free, maybe you'll have to pay some small fee; just ask). Then you will also need to set up port forwarding on your router so that the requests from the outside are redirected to your web server computer (Raspberry Pi) -- this is just done in router settings by entering the IP address of the webserver computer somewhere. It is possible to **self host even without public IP (if you're behind NAT)** using tunneling -- it's not hard, don't worry, you don't have to mess with DNS or firewalls -- check out e.g. the localtunnel project, <http://localhost.run/> etc., you can find these by looking up e.g. "public localhost", "hosting behind NAT" and so on. Usually this works by you running a program which connects to someone else's public server which will assign you some subdomain and accessing that subdomain will make the public server redirect the traffic to your computer (the catch may be e.g. in that your subdomain will be randomly generated and may change each time you restart the program).

- ◆ You may also pay for a web hosting (i.e. a server computer a company runs for you and which you access remotely) or a VPS (basically renting a server for more generic use). VPS has basically the same advantages as having your own home server (i.e. you can often do many things like host game servers, dynamic websites, gopher sites etc.), but it's a lot more expensive than just web hosting (i.e. renting a hosting space only for a website); if you ARE looking for VPS, look for UNMANAGED VPS (unmanaged means they won't handhold you and it's a lot cheaper). Price of really basic web hosting may even go as low as \$2 or \$1 per month, however an issue arises if you want to host **controversial and/or NSFW content**, for example politically incorrect site, criticizing mainstream politics, questioning soyence and history (covid vaccines, Holocaust, 9/11, ...) supporting "conspiracy theories", having gore or porn etc. It is really hard to find a host for that, keywords to search for are *free speech offshore web hosting*; however though many boast by hosting free speech, they actually don't or are scammers, ALWAYS read their terms of service and see what they allow etc. You may be forced to host at home here, but then your ISP may start bullying you in the same way, so really you may even need to use Tor or something. We won't cover this here much more, it's a pretty complex stuff.
- Optionally buy a domain name (search web for domain registrars), for example *mycoolsite.party*. If you are using a free hosting service, you will get a subdomain for free and don't have to care about this (but can still also use your own domain if you have it and want to). If you have your own home server, you probably want to buy a domain because otherwise people would have to connect to your site by literally typing an IP address to the browser. Once you have the domain, you want to edit the DNS records of your domain to point to the IP address of your server (i.e. you want to add an "A record"): how exactly to do this depends on the registrar (they will have some kinda online system to edit the records).

For starters try to go the easiest way: use some free static site hosting without a domain name. Later, once you get comfortable, you may transition to self-hosting with your custom domain.

Now you have to make the actual website in HTML. For that create a new file and name it `index.html` (the name has to be such as this is the default page name for websites). In it copy-paste the following:

```
<html>
<head>
</head>

<body>

<h1> My Awesome Website </h1>

</body>
</html>
```

This is really a bare-minimum testing website -- to expand it see the article on HTML.

Now you have to upload this html file to the hosting server -- check out the details of your hosting server on how to do this (you may e.g. need to use git or ftp to upload the file). And that's basically it, the rest is just expanding your site, making scripts to automatize uploading etc.

## How To Make A Wiki Like This One

Do NOT use wikifarms (sites that allow you to easily set up your own wiki) like fandom: all are bloated and most importantly censored. Also you will tie yourself to their shitty formats, clouds and databases and won't be able to easily migrate. Just avoid this.

First step to do is set up some kind of independent "online presence" like a website or gopherhole described above. Then you may either go the mainstream way and set up e.g. MediaWiki (the software used by Wikipedia) OR, better, do something like our LRS wiki does, i.e. keep it simple and start writing articles in some super simple format like Markdown, plain HTML or even plain text. To convert these articles into a wiki you basically just make a small shell script that just converts the format you write the articles in to a format you publish them in (so for example Markdown to HTML pages) and possibly automatically creates things like a list of all articles or a simple navigation bar on top of each page. You don't have to know any advanced programming at all, the script can literally be like 5 lines that just invoke CLI utilities that convert formats and copy files.

If you want, just literally take this wiki and make it your own, you can get the source code (there is link to the git repo somewhere nearby) and it's completely legally public domain. It works basically as just described -- you write articles in markdown and convert them to HTML or TXT with a bash script, then you just upload this all to you online hosting (possibly with another script) and voila, it's done.

## How To Learn Compsci/Programming

TODO: some kinda way/plan to learning this from start to finish

Some articles with tutorials and how tos related to this:

- programming
- C tutorial
- ...

## How To Live, Dos and Don'ts

This is a summary of some main guidelines on how an LRS supporter should behave in general so as to stay consistent with LRS philosophy, however it is important that this is shouldn't be taken as rules to be blindly followed -- the last thing we want is a religion of brainwashed NPCs who blindly follow orders. One has to understand why these principles are in place and even potentially modify them.

- If you want, **get in contact with like minded people**, for example us :) It's OK not to, not everyone is social, but it's nice to be part of a group where people understand each other, support each other, inspire each other, ... Even lurking helps many times. **Where to find such people?** Definitely not on mainstream platforms, they are mostly in the underground: as a tech minimalist search for "platforms" you would yourself use -- that's where you will likely find people like yourself. Good places to start are for example gopher, wiby, suckless, IRC, mailing lists, obscure online libre games etc. If you set up a website (or gopher hole) where you publish nice stuff, people will find you and contact you themselves. { I found many friends in Xonotic and OpenArena, as well as thanks to writings and programs I put on the internet. ~drummyfish }
- **Do NOT fight**, do NOT say you fight something. Fighting and rhetoric centered around "fighting something" is part of harmful fight culture, most people don't even realize they take part in it. It is important to unlearn this. We do not want to defeat anyone, we want to convince by means of rationality, nonviolence and love. However note that what is unacceptable to do to a living being may be completely acceptable to do to non living object (for example destroying a corporation is OK, in fact it is very desirable). We often take actions that common people would call a "fight" (for example we may organize a strike), however it is important that we don't call it a fight -- a point of view is sometimes as important as the action itself as it will determine our future direction. Remember that naming is important. **Watch out for A/B fails**.
- **Do NOT worship or create heroes, don't become one**. Watch out for cult of personality. It is another common mistake to for example call Richard Stallman a "hero of free software" and to even worship him as a celebrity. The concept of a hero is harmful, rightist concept that is connected to war

- mentality, it goes against anarchist principles, it creates social hierarchy and given some people a power to deceive. People are imperfect and make mistake -- only ideas can be perfect. Respect people but don't make anyone your moral compass, you should rather subscribe to specific ideas, i.e. rather than worshipping Stallman subscribe to and promote his idea of free software.
- **Do not identify with specific groups and organizations** -- this one is tricky because there is a fine line between many people together agreeing on an idea (good) and those people creating a formal hierarchical group which sooner or later inevitably becomes fascist or at the very least corrupt, eventually to the degree of betraying its original beliefs (bad). Remember principles of anarchism: loosely associate with others but do not create power structures and hierarchies. An example here may be supporting free software (good) vs supporting the (now greatly corrupt) Free Software Foundation (bad). Free software as an idea is pure and good, in merely supporting the idea we will not create any hierarchy of people, power structures or attach other unrelated ideas to ride on the free software wave (e.g. that of political correctness now promoted by the FSF). They say there is strength in unity, that is true, but there are different kinds of unity, and if perhaps one kind of unity (the bad one) is momentarily stronger, it is so because it's the "dark side of the force": yes, it may be stronger, but it is evil. Resist this urge. For this we also don't want to start any formal LRS group.
  - **CREATE, do NOT waste your life on bullshit, do NOT get too obsessed with tools and hopping** such as distrohopping, githopping audiophilia, hardware consumerism, 100% minimalist perfectionism etc. The perfect is the enemy of the good. Remember, the goal of your life is to create something new and better; too many people just get stuck doing nothing but switch distros, rant about which editor is best, making sure their OS has zero bloat and zero proprietary code etc. If that's all you do, it's completely useless, your life is completely wasted. Dedicate time to creating art that will last, e.g. programming LRS (creating source code text) or making free cultural art -- it doesn't matter whether you create it with Ubuntu or Gentoo.
  - **Lead an example**, this is the best way to spread our values, however be also extremely careful not to become a worshipped authority. Know the difference between a humble intellectual authority and an authoritative self-centered celebrity who uses his fame for deception. The more famous you are, the more humble you should become.
  - **Be loving, even towards opposition** -- remember: hate and revenge towards people perpetuates the endless circle. Love leads to more love, understanding, good deeds, friendship, happiness, collaboration and all the other positive things. **Do not confuse love with political correctness.** You may get angry or frustrated, just don't get violent against, rather try to break something, write your anger out, play some video game etc.
  - **Don't be politically correct**, never use gender neutral pronouns (always use "he" as the default pronoun), don't be afraid to say forbidden words like nigger, never use any code of coercion, "personal pronouns" etc. Even if you think you're moderate in views and that it "can't hurt" to just "play along" a little bit, IT DOES HURT, you are approving of fascism and carrying its flag, remember that Nazism only got so big thanks to a nation of moderate people who just "played along" to avoid trouble. There is always only very few true extremists, a great evil relies on masses of people who just want to get by and will make no trouble in conforming. Remember that staying silent often means supporting status quo, so the more deceit you see in society, the more you should try to not stay silent and the more you should try to tell the truth.
  - **Try to do selfless things** -- TRULY selfless ones. Help those in need without expecting any kind of repay, do not even seek attention or gratitude for it, only your good feeling. Create selfless art, whatever it is you enjoy doing -- computer programs, 3D models, music, videos, ... put them in the public domain and let others enjoy them :) Try to **make doing good things a habit** -- some people smoke, drink, overeat and do other kinds of things harmful to themselves and their environment as means for relieving stress. If you exploit this natural human tendency and rather develop GOOD habits, such as writing free software or helping charities as a means of relaxing and relieving stress, you have won at life; doing good and feeling good will be natural and effortless. **The thing you dedicate your life to should be the thing you love, not the thing that earns you money** or benefits you in similar ways -- try to maximize doing what you love (which may and probably should be more than one thing) and also **try to love doing what is good** so that you can do it a lot. **If you love something, never do it for money**; then it becomes business and as we know, business spoils everything.
  - **Protest in non-violent ways** -- this doesn't mean you should be passive; you should be exposing the truth, propaganda, corruption, boycotting corporations and state, promoting your values and expressing disagreement with certain ideas, but do not aim for destruction of those who stand in opposition -- if you're attacked, it is best if you do not fight back; not only is this the morally ideal thing to do, it also sends a very powerful message and makes the aggressor himself think.

- **Try to be so that if everyone was like that, the society would be good (in agreement with LRS)** -- this is a good general rule of thumb (and as such may also possibly fail sometimes, be careful) that can help you make some difficult decisions. DO NOT confuse this advice with the "do unto others as you would have them do unto you" advice, that is indeed a shitty one, supposing everyone likes the same things, i.e. for example a man who enjoys being raped is advised here to go and rape others -- that's of course bad.
- **Do NOT support pseudoleft (LGBT, feminism, Antifa, soyence ...)**, don't become type A fail. Of course you should equally reject rightism, but that goes without saying.
- **Free yourself from the system** (and generally from as many things as possible) -- similarly to how you free yourself technologically, free yourself also socially, live frugally and minimize your expenses. Stop consuming, stop living in luxury, stop spending money for shit (gyms, sports, clothes, car, streaming services, games, cigarettes, ...), use free things that people throw away and enjoy hobbies that are cheap (programming, reading books, going for walks, playing chess, collecting rocks, ...). **Stop watching news** (it's just brainwashing and distraction, what's really important will get to you anyway), stop engaging in fashion, stop talking to retards and watching tiktok manipulators. You need very little to live, you don't even need internet connection; with good computing you can hack offline and only connect to the internet once in a while on some public wifi to download emails and upload your programs. **Stop using cellphone** (if you need it e.g. for banking, just use it for banking and don't carry it around with you, don't make it something you need with you). Make yourself self sufficient, prepare for the collapse. If you can live somewhere in the woods and would enjoy it, go for it.
- **Adopt defeatism**; optimism is harmful, learn to be pessimistic -- this stops you from engaging in fight culture and makes you free to behave morally, you will cease to be the slave of bullshit necessary for winning the capitalist game and you'll no longer be working for capitalism.
- **Search for the truth**. You won't find it easily, real truth is always censored and hidden (though often in plain sight), but you can train yourself to spot propaganda and see the red flags. You won't find truth through Google, use different sources, read old books, encyclopedias and different points of view (e.g. contrast articles on Wikipedia with those on Infogalactic). Learn foreign and old languages such as Latin so that you can read untranslated and first hand historical accounts. **Question EVERYTHING** (absolutely everything, even this statement). Do not fall into traps such as pseudoskepticism. Train your mind to think critically, avoid shortcut thinking, question your own biased beliefs and wishes.
- **Reject harmful things like proprietary software, capitalism, copyright, bloat, work etc.** Use and promote the ethical equivalents, i.e. free software, free culture, frugality, anarchism etc.
- **Don't argue with retards** with the goal of convincing him or winning the argument so that you feel good (the meaning of retard here is simply someone disagreeing with LRS). It's literally wasted time/energy and it's bad for your mental health, it leads nowhere and achieves nothing but make you more suicidal than you already are. You literally can NOT convince anyone who is not open to being convinced, it is impossible, even if you have 100000 mathematical proofs, real world evidence, literature supporting you and anything you can imagine, you cannot logically convince someone who doesn't know how logic works or someone who simply emotionally isn't ready to change his mind. In 99.999999999999999% cases you can tell if it's worth to talk to someone after his first reaction -- you present an idea, such as LRS, and if he just expresses disagreement, there is no point in communicating further, by the disagreement he has taken a defensive stance and will hold it for the rest of his life now, you have to go find someone else. NO, not even if he's an "intellectual", has PhD and thirty Nobel Prizes, if he doesn't wanna see the truth, you cannot help him. As it's been said, trying to argue with an idiot is like trying to win a chess game against a pidgeon -- even if you're the world chess champion, the pidgeon will just shit on the board and think it's won. If you spot a retard, just leave -- don't try to have the last word or anything, even admit him "victory" in the argument and leave him in his world of delusion where he is the unappreciated Einstein, just do not waste an extra second on him, just leave and go do something better. { So many such idiots I have met I can't even count it -- pure stupid peasant aren't even that bad, the worst are the "above average" intelligence reddit atheists who think they're smart. I literally had such people argue like "you like games therefore competition in society is good because games are part of society therefore society equals competition". Truly I'm not sure if those bastards are just trolling me into suicide or are really so fucking dumb :D ~drummyfish }
- Similarly **avoid toxic communities**, don't argue, just leave, it's better to be alone than in bad company. Basically anything with a COC, language filter, SJW vibe, rainbow etc. isn't even more worth checking out.



- **Be a generalist, see the big picture, study the whole world, educate yourself** -- tantum possumus quantum scimus, what we can do is given by what we know. Do not become overspecialized in the capitalist way. Sure you may become an expert at something, but not for the price of making your view of the world too narrow. You may spend most of your time studying and programming computer compilers for example, but still do (and enjoy) other things, for example reading fiction, studying religions, languages, psychology, playing go, making music, building houses, painting, doing sports, ... Learn to enjoy to educate yourself! Education (not necessarily formal) is one of the most valuable things you can get -- no one can take it away from you, it makes you see truth more clearly and though this itself makes you more depressed, it also frees you in many ways, for example knowing languages enables you to read more books and live in more places and talk to more people and you can get a comfy job as a translator if you need money, knowing chess makes you able to entertain yourself without a computer, knowing programming enables you to write your own programs if good ones aren't available, and so on. Education makes you see through other people's lies. It is fine to be retarded, remain ignorant and just play video games all days, your value as a living being will lower that way, but you will forever remain among the retarded majority manipulated by the tides of society, even if you have good opinions and correct views, you'll stay just another retard, you won't be able to help others, you'll be paralyzed, leaving the burden and joy of helping the world on others -- just decide if that is what you want. Have you always admired that someone can play a piano? Why not learn it then? Get some cheap keyboard and make it a habit to practice playing it at least 20 minutes every other day, see how good you become in a year. Were you always bad at chemistry? Why not fix it a bit? Get some chemistry for dummies book and read it every day before sleep, you will go from absolute chemistry retard to well above average soon. You can learn about star constellations, biology, history, train card games, memorize pi digits, run half marathon, learn juggling, write your own small book etcetc. A nice life hack is to **see life as an RPG game**, see yourself as a character you are improving, by improving skills you are unlocking new abilities, enabling new options, increasing your stats -- however be very careful to not become competitive or fall victim to the "self-improvement" cult! The key is to not start comparing yourself to other, or rather to not have it as a goal to be better than someone else, the goal should be just your happiness of becoming a higher level living entity that has more abilities for helping other, enjoy the universe and so on.
- **Stop just bitching around and DO SOMETHING** -- don't get this wrong, bitching around and ranting is great, this whole wiki is just one huge wall of rage bitching, however if it's all you do, it literally achieves nothing, it won't convince a single man, no one will read that shit, you are just wasting huge part of your life by being angry on the Internet. First thing you have to do is DO SOMETHING, e.g. if you promote minimalism, go and make a minimalist game, show others it works, prove (even to yourself) the thing you believe in is good, bitching about the world is only to come as a supplement to your main work -- your rants aren't there to convince anyone, your art does that, your writings are there for the people who are already convinced to help them educate themselves further. Consider this: you may spend whole life writing 100 books about how minimalism is awesome, you may examine the whole history in detail, provide mathematical proofs of everything and suggest a completely working system that could be established to solve all the problems in the world -- no one is going to read this. Literally not a single man will give a shit. On the other hand you can take a year to program a minimalist operating system, one that is 1000 times smaller than Linux and is 10 times faster and is completely public domain and basically rapes Linux in every other way, you just post that somewhere and people just can't ignore it, you put before their eyes something they can literally see is infinitely better than what was there before, you instantly get thousands of people hooked and they start creating more art like this. You just changed the world significant for the better in just one year. Note this isn't an argument for chasing popularity at all, on the contrary, your actions will likely contradict the popular and even cause a lot of hate, however realize that words are just words, there are too many words everywhere, words can lie and they never achieve anything by themselves, good is achieved and proven by actions.
- **Should you go vote?** The safe answer is no, most likely you shouldn't vote, not voting is in 99.99% cases the best thing as you can just avoid all noise and stress of watching politics at all, you won't waste your time and you also actively vote against the current system by not voting: you decrease the voter turnout, decreasing trust in the system -- there is a reason all politics agree on the one thing that "you should go vote" -- they do because that sustains trust that gives them power, so as it's mostly the case, mainstream clearly pushing you to something basically means it's the one thing you should almost definitely NOT do. A deeper answer to the question of voting is again that you should weight all pros and cons, but you will conclude the pros are so unlikely to prevail that only in an extremely rare situation it would make sense to go vote. You should go vote only if there appears a

"party" that's extremely based -- this party should be extremely aligned with LRS, wanting to end all work, military, money, police, all by strictly peaceful ways and eventually end even state and itself too -- something we have practically a zero chance of seeing in the next few hundred years, and even if such "party" appeared in theory and you went vote for it, it almost definitely won't win as normies just won't vote for it, so you change nothing anyway. Definitely do NOT go vote for lesser evil, that's just strengthens the system. Just stop watching politics and let the system destroy itself, you won't be able to influence it by voting in any way no matter what, even if your voting power was multiplied by one million, it just doesn't even matter which party is ruling nowadays. Stop caring about current politics, spend the time on better things. If the base party appears, the news will get to you anyway.

- **Live your life as you want**, don't let someone else control your life and manipulate you, e.g. with feelings of guilt -- this often happens with your parents, partner, friends, culture, laws, ... This isn't an argument for self interest! On the contrary, most people nowadays will try to push you to following self interest or fascist goals that will also benefit them. You only have one life, others have theirs, so listen to advice but remember to always make your own decisions in important things. If you feel you don't want to go to school or that you don't want to work or that you want to do something that people despise or you want to do something that you've read is wrong, just do what you feel is best, even if it's a let down for your family or if it contradicts what the whole society is telling you.
- **Publish everything immediately**, don't wait for your project "to be ready" for a release, make it public right now! You don't have to advertize it, just make it public. Some reasons are for example: you aren't behaving strategically like a capitalist, you get early feedback from others (important so you don't spend a lot of time on shit), you let others know what you're working on so they don't waste time working on the same thing, even an incomplete project may be useful to someone (parts of it may already be useful to someone), and also, very importantly, if you hesitate YOU WILL NEVER RELEASE THE PROJECT, you will become obsessed with perfectionism and ashamed to ever release the project. YES YOU WILL, I have seen it about 10000000 billion times. You think you will release it but you won't, every additional day you hesitate the chance of release decreases by 10%, so after 10 days it's already certain you will never release it, further on the chance even gets negative.
- **NEVER, NEVER go into debt**: Even if you should live under a bridge, if you aren't in debt you're still good -- better than most people probably. Debt is how the system enslaves you, so never take any loans or make unplanned children you would be obliged to pay for etc., it will force you to bow to the system, take unethical jobs, forget your morals. If you're already in debt, make it number one priority to pay it off ASAP. If you're in debt that would take too long or forever to pay off, your only option is just to burn your ID and run off to the woods, the system will now see you as a free slave, someone who can be forced to labor without sleep or just killed, you can no longer rely on any help from it.
- PRO TIP: A great heuristic for making life decisions is to **usually do the exact opposite of what the society tells you to do** -- it works because society only wants to exploit you, so it pushes you towards bad decisions. This doesn't hold always, of course, don't just blindly act in opposites (there may be "double bluffs" also..., but mostly there aren't as most people just follow direct orders), but it's a good decision helper in about 99% cases. For example if society tells you "increase your social media presence", you should really completely leave social media, if it tells you "boost your carrier", you should stop working, if it tells you "go vote", you shouldn't go vote etcetc.
- PRO TIP: **Get yourself banned on toxic platforms** like Wikipedia, GitHub, Steam, 4chan etcetc., it has many advantages -- you gain freedom (no longer having to care about platform you are banned on), the platform loses one user/slave (you), you stop being abused by the platform, it's also fun (just find some creative way to get banned, possibly cause uprising on the platform, make mods angry and waste their time on cleaning up your mess), it will make you become more self sufficient and you help decentralize the Internet again (can't edit Wikipedia? Just make your own :-)), it will make you find better places, you may also help bring the toxic platform down (others will see the platform utilizes censorship, some may follow you in leaving...) etcetc.
- ...

## How Not To Get Depressed Living In This Shitty Dystopia

I don't know lol, you tell me. Becoming more independent of this system really helps though, just accept everything will get destroyed in a few years -- yes, all you ever liked is basically already dead, just deal with it and find new things to like such as reading books instead of scrolling through facebook etc. Unconditional love and altruism helps too, just let go of the hate and fight, help people selflessly without expecting rewards.

## Other

Here are links to some other articles that may contain their own *how to*:

- [how to make living](#)
- [C tutorial](#)
- [how to learn programming](#)
- [how to learn 3d modeling](#)
- [how to play chess](#)
- ...

---

hw

## Hardware

Hardware (HW), as opposed to software, are the physical parts of a computer, i.e. the circuits, the mouse, keyboard, the printer etc. Anything you can smash when the machine pisses you off.

---

hyperoperation

## Hyperoperation

*WARNING: brain exploding article*

*UNDER CONSTRUCTION*

{ This article contains unoriginal research with errors and TODOs, read at own risk. Some really interesting and more in-dept information can be found at this nice site: <http://mrob.com/pub/math/largenum.html>. Also the rabbithole of big numbers and googology is so deep I can't even see the end of it. ~drummyfish }

Hyperoperations are mathematical operations that are generalizations/continuations of the basic arithmetic operations of addition, multiplication, exponentiation etc. Basically they're like the basic operations like plus but on steroids. When we realize that multiplication is just repeated addition and exponentiation is just repeated multiplication, it is possible to continue in the same spirit and keep inventing new operations by simply saying that a new operation means repeating the previously defined operation, so we define repeated exponentiation, which we call tetration, then we define repeated tetration, which we call pentation, etc.

There are infinitely many hyperoperations as we can go on and on in defining new operations, however we start with what seems to be the simplest operation we can think of: the successor operation (we may call it *succ*, *+1*, *++*, *next*, *increment*, *zeration* or similarly). In the context of hyperoperations we call this operation *hyper0*. Successor is a unary operator, i.e. it takes just one number and returns the number immediately after it (suppose we're working with natural numbers). In this successor is a bit special because all the higher operations we are going to define will be binary (taking two numbers). After successor we define the next operation, addition (*hyper1*), or  $a + b$ , as repeatedly applying the successor operation  $b$  times on number  $a$ . After this we define multiplication (*hyper2*), or  $a * b$ , as a chain of  $b$  numbers as which we add together. Similarly we then define exponentiation (*hyper3*, or raising  $a$  to the power of  $b$ ). Next we define tetration (*hyper4*, building so called power towers), pentation (*hyper5*), hexation (*hyper6*) and so on (heptation, octation, ...).

Indeed the numbers obtained by high order hyperoperations grow quickly as fuck.

An important note is this: there are multiple ways to define the hyperoperations, the most common one seems to be by supposing the **right associative** evaluation, which is what we're going to implicitly consider from now on. This means that once associativity starts to matter, we will be evaluating the expression chains FROM RIGHT, which may give different results than evaluating them from left (consider e.g.  $2^{(2^3)} \neq (2^2)^3$ ). The names tetration, pentation etc. are reserved for right associativity operations.

The following is a sum-up of the basic hyperoperations as they are commonly defined (note that many different symbols are used for these operations throughout literature, often e.g. up arrows are used to denote them):

| operation               | symbol           | meaning                                                                 | commutative associative |         |
|-------------------------|------------------|-------------------------------------------------------------------------|-------------------------|---------|
| successor (hyper0)      | $\text{succ}(a)$ | next after $a$                                                          |                         |         |
| addition (hyper1)       | $a + b$          | $\text{succ}(\text{succ}(\text{succ}(\dots a \dots))), b \text{ succs}$ | yes                     | yes     |
| multiplication (hyper2) | $a * b$          | $0 + (a + a + a + \dots), b \text{ as in brackets}$                     | yes                     | yes     |
| exponentiation (hyper3) | $a ^ b$          | $1 * (a * a * a * \dots), b \text{ as in brackets}$                     | no                      | no      |
| tetration (hyper4)      | $a ^ ^ b$        | $1 * (a ^ (a ^ (a ^ (\dots))), b \text{ as in brackets}$                | no                      | no      |
| pentation (hyper5)      | $a ^ ^ ^ b$      | $1 * (a ^ ^ (a ^ ^ (\dots))), b \text{ as in brackets}$                 | no                      | no      |
| hexation (hyper6)       | $a ^ ^ ^ ^ b$    | $1 * (a ^ ^ ^ (a ^ ^ ^ (\dots))), b \text{ as in brackets}$             | no                      | no      |
| ...                     |                  |                                                                         | no more                 | no more |

The following ASCII masterpiece shows the number 2 in the territory of these hyperoperations:

{ When performing these calculations, use some special calculator that allows extremely high numbers such as HyperCalc (<http://mrob.com/pub/comp/hypercalc/hypercalc-javascript.html>) or Wolfram Alpha.  
~drummyfish }

```

2      +1      +1      +1      +1      +1      +1      +1 ...    successor
|      / \    / \    / \    / \    / \    / \    / \
|      /   \  /   \  /   \  /   \  /   \  /   \  /   \
2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 ...    addition
|      | 4      | 16
|      / \    / \    / \    / \    / \    / \    / \
2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 ...    multiplication
|      | 4      | 8      | 16     | 32     | 64     | 128    | 256
|      / \    / \    / \    / \    / \    / \    / \
2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ...    exponentiation
|      | 4      | 16     | 65536  | ~10^19728 | ~10^(10^(10^19728))
|      / \    / \    / \    / \    / \    / \    / \
2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ...    tetration
|      | 4      | 65536
|      / \    / \    / \    / \    / \    / \    / \
2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ...    pentation
...      4      65536
a lot

```

Some things generally hold about hyperoperations, for example for any operation  $f = \text{hyper}N$  where  $N \geq 3$  and any number  $x$  it is true that  $f(1, x) = 1$  (just as raising 1 to anything gives 1).

Hyperroot is the generalization of square root, i.e. for example for tetration the  $n$ th hyperroot of number  $a$  is such number  $x$  that  $\text{tetration}(x, n) = a$ .

**Left associativity hyperoperations:** Alternatively left association can be considered for defining hyperoperations which gives different operations. However this is usually not considered because, as mentioned in the webpage above, e.g. left association tetration  $a ^ ^ b$  can be simplified to  $a ^ (a ^ (b - 1))$  and so it isn't really a new operation. Anyway, here is the same picture as above, but for left associativity -- we see the numbers don't grow THAT quickly (but still pretty quickly).

```

2      +1      +1      +1      +1      +1      +1      +1 ...    successor
|      / \    / \    / \    / \    / \    / \    / \
|      /   \  /   \  /   \  /   \  /   \  /   \  /   \
2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 / 2 + 2 ...    addition
|      | 4      | 16
|      / \    / \    / \    / \    / \    / \    / \
2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 / 2 * 2 ...    multiplication
|      | 4      | 8      | 16     | 32     | 64     | 128    | 256
|      / \    / \    / \    / \    / \    / \    / \
2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ^ (2 ...    exponentiation
|      | 4      | 16     | 65536  | ~10^19728 | ~10^(10^(10^19728))
|      / \    / \    / \    / \    / \    / \    / \
2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ^ ^ (2 ...    tetration
|      | 4      | 65536
|      / \    / \    / \    / \    / \    / \    / \
2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ^ ^ ^ (2 ...    pentation
...      4      65536
a lot

```

|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
|-----|-----|----|-----|----------|-----|----|-----|----|-----|----|-----|----|-----|----|-----|---------------------|
| 2   | *   | 2  | *   | 2        | *   | 2  | *   | 2  | *   | 2  | *   | 2  | *   | 2  | ... | multiplication      |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
| (2  | ^   | 2) | ^   | 2)       | ^   | 2) | ^   | 2) | ^   | 2) | ^   | 2) | ^   | 2) | ... | left exponentiation |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
| (2  | ^^  | 2) | ^^  | 2)       | ^^  | 2) | ^^  | 2) | ^^  | 2) | ^^  | 2) | ^^  | 2) | ... | left tetration      |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
|     |     |    |     |          |     |    |     |    |     |    |     |    |     |    |     |                     |
| (2  | ^^^ | 2) | ^^^ | 2)       | ^^^ | 2) | ^^^ | 2) | ^^^ | 2) | ^^^ | 2) | ^^^ | 2) | ... | left pentation      |
| ... |     | 4  |     | ~3*10^38 |     |    |     |    |     |    |     |    |     |    |     |                     |

TODO: arrows?

In fact we may choose to randomly combine left and right associativity to get all kinds of weird hyperoperations. For example we may define tetration with right associativity but then use left associativity for the next operation above it (we could call it e.g. "right-left pentation"), so in fact we get a binary tree of hyperoperations here (as shown by M. Muller in his paper on this topic).

Of course, we can now go further and start inventing things such as hyperlogarithms, hyperfactorials etc.

### Code

Here's a C implementation of some hyperoperations including a general hyperN operation and an option to set left or right associativity (however note that even with 64 bit ints numbers overflow very quickly here):

```
#include <stdio.h>
#include <inttypes.h>
#include <stdint.h>

#define ASSOC_R 1 // right associativity?

// hyper0
uint64_t succ(uint64_t a)
{
    return a + 1;
}

// hyper1
uint64_t add(uint64_t a, uint64_t b)
{
    for (uint64_t i = 0; i < b; ++i)
        a = succ(a);

    return a;
    // return a + b
}

// hyper2
uint64_t multiply(uint64_t a, uint64_t b)
{
    uint64_t result = 0;

    for (uint64_t i = 0; i < b; ++i)
        result += a;

    return result;
    // return a * b
}

// hyper(n + 1) for n > 2
uint64_t nextOperation(uint64_t a, uint64_t b, uint64_t (*operation)(uint64_t,uint64_t))
{
    if (b == 0)
        return 1;

    uint64_t result = a;
```

```

    for (uint64_t i = 0; i < b - 1; ++i)
        result =
#ifdef ASSOC_R
            operation(a,result);
#else
            operation(result,a);
#endif

    return result;
}

// hyper3
uint64_t exponentiate(uint64_t a, uint64_t b)
{
    return nextOperation(a,b,multiply);
}

// hyper4
uint64_t tetrate(uint64_t a, uint64_t b)
{
    return nextOperation(a,b,exponentiate);
}

// hyper5
uint64_t pentate(uint64_t a, uint64_t b)
{
    return nextOperation(a,b,tetrate);
}

// hyper6
uint64_t hexate(uint64_t a, uint64_t b)
{
    return nextOperation(a,b,pentate);
}

// hyper(n)
uint64_t hyperN(uint64_t a, uint64_t b, uint8_t n)
{
    switch (n)
    {
        case 0: return succ(a); break;
        case 1: return add(a,b); break;
        case 2: return multiply(a,b); break;
        case 3: return exponentiate(a,b); break;
        default: break;
    }

    if (b == 0)
        return 1;

    uint64_t result = a;

    for (uint64_t i = 0; i < b - 1; ++i)
        result = hyperN(
#ifdef ASSOC_R
            a,result
#else
            result,a
#endif
            ,n - 1);

    return result;
}

int main(void)
{
    printf("\t0\t1\t2\t3\n");

    for (uint64_t b = 0; b < 4; ++b)
    {
        printf("%" PRIu64 "\t",b);
    }
}

```

```

    for (uint64 t a = 0; a < 4; ++a)
        printf("%" PRIu64 "\t", tetrade(a,b));

    printf("\n");
}

return 0;
}

```

In this form the code prints a table for right associativity tetration:

|   |   |   |    |               |
|---|---|---|----|---------------|
|   | 0 | 1 | 2  | 3             |
| 0 | 1 | 1 | 1  | 1             |
| 1 | 0 | 1 | 2  | 3             |
| 2 | 1 | 1 | 4  | 27            |
| 3 | 0 | 1 | 16 | 7625597484987 |

## See Also

- [googology](#)
- [p-adic numbers](#)

---

implicit

## Implicit

Implicit means something that's assumed unless stated otherwise; it is the opposite of [explicit](#). For example many [floating point](#) formats assume an implicit (not physically stored) bit with value 1 prepended to the explicitly stored mantissa values. Though not the same, the term *implicit* is similar to [default](#); for example an implicit/default background [color](#) of some image format may be defined as [white](#), meaning that unless background color is stated, we suppose the background to be white (though a *default* value may still be explicitly stored; default just means an initial, unchanged value). Implicit values may be important e.g. for saving space -- imagine we have some dataset in which 90% of values are [zero](#); then it is convenient to state zero to be the implicit value and not store such values, by which we'll save 90% of space.

---

infinity

## Infinity

Infinity (from Latin *in* and *finis*, *without end*) is a quantity so unimaginably large that it has no end. It plays a prominent role especially in [mathematics](#) and [philosophy](#). As a "largest imaginable quantity" it is sometimes seen to be the opposite of the number [zero](#), the "smallest possible quantity", though other "opposites" can be thought of too, such as minus infinity or an infinitely small non-zero number ([infinitesimal](#)). The symbol for infinity is *lemniscate*, the symbol 8 turned 90 degrees ([unicode](#) U+221E). Keep in mind that mere lack of boundaries doesn't imply infinity -- a [circle](#) has no end but is not infinite; an infinity implies there is always more, no matter how much we get.

The concept of infinity came to firstly be explored by philosophers -- as an abstract concept (similar to those of e.g. [zero](#) or negative numbers) it took a while for it to evolve, be explored and accepted. We can't say who first "discovered" infinity, civilizations often had concepts similar to it that were connected for example to their gods. Zeno of Elea (5th century BC) was one of the earliest to tackle the issue of infinity mathematically by proposing [paradoxes](#) such as that of Achilles and the tortoise.

The term *infinity* has two slightly distinct meanings:

- **potential infinity**: The unboundedness, lack of upper limit. For example the sequence of odd numbers 1, 3, 5, ... is potentially infinite. This is the less problematic kind of infinity as we know what's going on: we simply lack any limit and can keep going on forever.
- **actual infinity**: Infinity as an actual "object" (for example a number) that's somehow "endlessly large", larger beyond any limits, largest possible etc. This type of infinity poses more issues as we

don't know anything like this from real life, we lack experience and intuition about it, we don't know how such an object should behave and we encounter paradoxes. Stuff can get pretty weird and things we take for granted stop working, such as being able to just randomly pick elements from sets (see axiom of choice). For example if we have the largest object possible, what happens if we put two of such objects together, will we get yet a larger object or not? How about two infinities minus one infinity -- is that an infinity or zero? What if we shrink infinity to half, what size will it have?

It could be argued that potential infinity is really the reason for the existence of true, high level mathematics as we know it, as that is concerned with constructing mathematical proofs -- such proofs are needed anywhere where there exist infinitely many possibilities, as if there was only a finite number of possibilities, we could simply enumerate and check them all without much thinking (e.g. with the help of a computer). For example to confirm Fermat's Last Theorem ("for whole numbers and  $n > 2$  the equation  $a^n + b^n = c^n$  doesn't have a solution") we need a logical proof because there are infinitely many numbers; if there were only finitely many numbers, we could simply check them all and see if the theorem holds. So infinity, in a sense, is really what forces mathematicians to think.

**Is infinity a number?** Usually no, but it depends on the context. Infinity is not a real number (which we usually understand by the term "number") because that would break the nice field structure of real numbers, so the safe implicit answer to the question is no, infinity is not a traditional number, it is rather a concept closely related to numbers. However infinity may sometimes behave like a number and we may want to treat it so -- see for example transfinite numbers that are used to work with infinite sets and the numbers can be thought of as "sort of infinity numbers", though they mostly live in a separate realm from the traditional numbers. Also for example the result of computing a limit may be a real number but also infinity; so ultimately everything depends on our definition of what number is and we can declare infinity to be a number in some systems, for example there exists so called *extended real number line* which consists of real numbers and plus/minus infinity, which ARE treated as numbers.

An important term related to the term *infinite* is **infinitesimal**, or *infinitely small*, a concept very important e.g. for calculus. While the "traditional" concept of infinity looks beyond the greatest numbers imaginable, the concept of infinitely small is about being able to divide (or "zoom in", see also fractals) without end; for example in the realm of real numbers we may start at number 1 and keep moving closer and closer towards zero without ever reaching the "smallest nonzero number", as no matter how close to zero we are, we may always divide our distance by two. A term also related to this is limit, which helps us explore values "infinitely close", "infinitely far" etc.

When treated as cardinality (i.e. size of a set), we conclude that **there are many infinities, some larger than others**, for example there are infinitely many rational numbers and infinitely many real numbers, but in a sense there are more real numbers than rational ones -- this is very counter intuitive, but nevertheless was proven by Georg Cantor in 1874. He showed that it is possible to create a 1 to 1 pairing of natural numbers and rational numbers and so that these sets are of the same size -- he called this kind of infinity **countable** -- then he showed it is not possible to make such pairing with real numbers and so that there are more real numbers than rational ones -- he called this kind of infinity **uncountable**. Furthermore this hierarchy of "larger and larger infinities" goes on forever, as for any set we can always create a set with larger cardinality e.g. by taking its power set (a set of all subsets).

**In regards to programming:** programmers are often just engineers and so simplify the subject of infinity in a way which to a mathematician would seem unacceptable. For example it is often a good enough approximation of infinity to just use an extremely large number value, e.g. the largest one storable in given data type, which of course has its limitations, but in practice just works (just watch out for overflows). Programmers also often resort to breaking the mathematical rules, e.g. they may accept that  $x / 0 = \text{infinity}$ ,  $\text{infinity} + \text{infinity} = \text{infinity}$  etc. Systems based on symbolic computation may be able to handle infinity with exact mathematical precision. Advanced data types, such as floating point, often have a special value for infinity -- IEEE 754 floating point, for example, is capable of representing positive and negative infinity.

**WATCH OUT: infinite universe doesn't imply existence of everything** -- this is a common fallacy to think it does. For example people tend to think that since the decimal expansion of the digits of pi is infinite and basically "random", there should always exist any finite string of digits somewhere in it; this doesn't follow from the mere fact that the series is infinite (though the conclusion MAY or may not be true, we don't actually know this about pi yet). Imagine for example the infinite series of even numbers -- there are infinitely many numbers in it, but you will never find any odd number there.



## See Also

- [zero](#)
- [thrembo](#)

---

information

## Information

*Information wants to be free.*

Information (from Latin *informare*: shape/describe/represent) is knowledge that can be used for making decisions. Information is interpreted data, i.e. while data itself may not give us any information, e.g. if they're encrypted and we don't know the key or if we simply don't know what the data signifies or implies, information emerges once we make sense of the data (someone once put information in this relationship: data leads to information, information leads to knowledge and knowledge leads to wisdom). Information is contained e.g. in books, on the Internet, in nature, and we access it through our senses. Computers can be seen as machines for processing information and since the computer revolution information has become the focus of our society; we often encounter terms such as information technology, informatics, information war, information age etc. Information theory is a scientific field studying information.

**Information wants to be free**, i.e. it is free naturally unless we decide to limit its spread with shit like intellectual property laws. What does "free" mean? It is the miraculous property of information that allows us to duplicate it basically without any cost. Once we have certain information, we may share it with others without having to give up our own knowledge of the information. A file on a computer can be copied to another computer without deleting the file on the original computer. This is unlike with physical products which if we give to someone, we lose them ourselves. Imagine if you could make a piece of bread and then duplicate it infinitely for the whole world -- information works like this! We see it as a crime to want to restrict such a miracle. We may also very nicely store information in our heads. For all this information is beautiful. It is sometimes discussed whether information is created or discovered -- if a mathematician comes up with an equation, is it his creation or simply his discovery of something that belongs to the nature and that has always been there? This question isn't so important because whatever terms we use, we at LRS decide to create, spread and freely share information without limiting it in any way, i.e. neither discovery nor invention should give rise to any kind of property.

In computer science the basic unit of information amount is 1 **bit** (for *binary digit*), also known as shannon. It represents a choice of two possible options, for example an answer to a *yes/no* question (with each answer being equally likely), or one of two binary digits: 0 or 1. From this we derive higher units such as bytes (8 bits), kilobytes (1000 bytes) etc. Other units of information include nat or hart. With enough bits we can encode any information including text, sounds and images. For this we invent various formats and encodings with different properties: some encodings may for example contain redundancy to ensure the encoded information is preserved even if the data is partially lost. Some encodings may try to hide the contained information (see encryption, obfuscation, steganography). For processing information we create algorithms which we usually execute with computers. We may store information (contained in data) in physical media such as books, computer memory or computer storage media such as CDs, or even with traditional potentially analog media such as photographs.

Keep in mind that the **amount of physically present bits doesn't have to equal the amount of information** because, as mentioned above, data that takes  $N$  bits may e.g. utilize redundancy and so store less information that would theoretically be possible with  $N$  bits. It may happen that the stored bits are correlated for any reason or different binary values convey the same information (e.g. in some number encodings there are two values for number zero: positive and negative). All this means that the amount of information we receive in  $N$  bit data may be lower (but never higher) than  $N$  bits, i.e. if we e.g. store a file on a 1 GB flash drive, the actual theoretical information contained may be lower -- the exact size of such theoretical information depends on probabilities of what can really appear in the file and MAY CHANGE with the knowledge we possess, i.e. the amount of information stored on the flash drive may change by simply us coming to know that the file stored on the drive is a movie about cats which rules out many combinations of bits that can be stored there. Imagine a simplified case when there is file which says whether there exists infinitely many prime numbers -- to a mathematician who already knows the answer the file gives zero

information, while to someone who doesn't know the answer the file provides 1 bit of information. However in practice we often make the simplification of equating the amount of physically present bits to the contained "information".

Information is related to **information entropy** (also Shannon entropy, similar to but distinct from the concept of thermodynamic entropy in physics); they're both measured in same units (usually bits) but entropy measures a kind of "uncertainty" or average information received from a certain event when we know its probability distribution -- in a sense information and entropy can be seen as opposites: before we receive information we lack the information but there exists entropy, once we receive the information there is information but no entropy.

In signal theory information is also often used as a synonym for **signal**, however a distinction can be made: signal is the function that carries information. Here we also encounter the term **noise** which means an unwanted signal mixed in with the desired signal which may make it harder to extract the information carried by the signal, or even obscure some or all of the information so that it can't be retrieved.

According to the theory of relativity **information can never travel faster than light** -- even if some things may move faster than light, such as a shadow, so called "spooky action at a distance" (usually associated with quantum entanglement) or even matter due to the expansion of space, by our best knowledge we can never use this to transfer information faster than light. For this it seems our communication technology will always be burdened by lag, no matter how sophisticated.

---

intellectual\_property

## "Intellectual Property"

"Intellectual property" (IP, not to be confused with IP address) is a twisted capitalist idea establishing that people be able to own information (such as ideas, presentation style, songs or text) and that it should be treated in ways very similar to physical property. For example patents are one type of intellectual property which allow an inventor of some idea to *own* that idea and be able to limit its use and charge people fees for using that idea, or prevent people from using that idea altogether. Copyright is probably the most harmful type of IP as of today, and along with patents the most relevant one in the area of technology. However, IP encompasses many other subtypes of this kind of "property" such as trademarks, trade dress, plant varieties etc. IP is an **arbitrarily invented grant of monopoly** on information, i.e. something that is otherwise naturally free. Only very few other ideas reach the level of stupidity of the IP concept, most people with brain oppose it, see e.g. [http://harmful.cat-v.org/economics/intellectual\\_property/](http://harmful.cat-v.org/economics/intellectual_property/).

IP exists to benefit corporations, it artificially limits the natural freedom of information and tries to eliminate freedom and competition of the IP owners, it fuels consumerism (for example a company can force deletion of old version of its program in order to force users to buy a new version), it helps keep malicious features in programs (by forbidding any study and modifications) and forces reinventing wheels which is extremely energy and resource wasting, whose side effect (or rather one of many side effects) is of course destroying the whole Earth. IP creates a kind of artificial scarcity, i.e. in a world where any information once created would be abundant, available to everyone, IP kills this abundance so as to create a new "market" and bullshit businesses and slaveries such as various IP law firms, patent offices, brand protections, copyright verification for courts, DRM programmers and so on. Without IP everyone would be happy, able to study, share, improve, remix and combine existing technology and art into amazing things.

Only idiots defend IP -- basically just capitalists. They give absolutely invalid arguments like "but without IP there would be no progress" etc. Of course there would be progress, progress can't be stopped even if you try. Capitalists are amazingly retarded creatures.

Many people protest against the idea of IP -- either wanting to abandon the idea completely, as we do, or at least arguing for great relaxation the insanely strict and aggressive forms that destroy our society. Movements such as free software and free culture have come into existence in protest of IP laws. Of course, capitalists don't give a shit. It can be expected the IP cancer will be reaching even more extreme forms very soon, for example it will be perpetual and encompassing such things as mere thought (thoughts will be monitored and people will be charged for thinking about ideas owned by corporations).

It must be noted that as of 2020 **it is not possible to avoid the IP shenanigans**. Even though we can eliminate most of the harmful stuff (for now) with licenses and waivers, there are many things that may be impossible to address or posing considerable dangers, e.g. trademark, personal rights or patent troll attacks. In some countries (US) it is illegal to make free programs that try to circumvent DRM. Some countries make it explicitly impossible to e.g. waive copyright. It is impossible to safely check whether your creation violates on someone else's IP. There exists shit such as moral rights that may exist even if copyright doesn't apply.

## See Also

- illegal number

---

interaction\_net

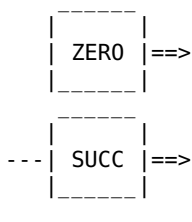
## Interaction Net

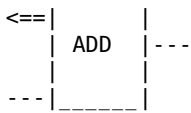
Interaction nets are a way of performing computation by manipulating graphical diagrams according to a few simple rules. Interaction nets can be seen as one of many possible ways to create the lowest level basis for a computer, so we call them a model of computation; other models of computation are for example grammars, Turing machines, lambda calculus, cellular automata etc. -- all of these can be used to perform the same computations, though in different ways; interaction nets do this by representing a program with a graph-like diagram which is then manipulated by some predefined rules and "reshaped" into another graph that represents the result of the performed computation. For this interaction nets can also be seen as a kind of graphical programming language (and even a textual one once we learn how to represent the graphical diagrams with text, which we call *interaction calculus*), though it's an extremely low level language (even things such as addition of natural numbers have to be programmed from scratch) that would be extremely impractical for everyday programming. The advantage of interaction nets is besides their simplicity usually said to be mainly their **parallel nature**: as computation really means locally replacing parts of the network with predefined patterns, it is possible to simultaneously process multiple parts of the network at once, greatly speeding up the computation.

**WATCH OUT:** interaction nets are a bit confusing to newcomers because they look a bit like logic circuits and so people think the computation happens by passing some data through the network -- THIS IS NOT THE CASE! The computation happens by CHANGING THE NETWORK itself, i.e. there is no data flowing, all that happens is REPLACING parts of the network with patterns defined by the rewriting rules (similarly to how in lambda calculus there are really no function calls but just text replacement). The idea is similar to that of rewriting rules in grammars. Think of the connections less like of electric wires and more like of strings with knots that you tie and untie to perform the computation.

A general interaction net consists of nodes, or **agents**, that are connected with wires, or **edges**. Alongside the net exist **interaction rules** that say how patterns in the net get replaced with other patterns. A concrete definition of specific agent types and interaction rules is called an **interaction system** (i.e. *interaction net* is the general idea, like for example a cellular automaton, while *interaction system* is a specific realization of that idea, like for example game of life).

An **agent** is a type of block that may appear in the net. We may see it as a block with a number of ports (wires); it must have exactly one **principal port** (a kind of special, interacting port) and can have any number (even zero) of additional **auxiliary ports** (kind of "passive" ports) -- usually we just say the agent has  $N$  ports (where  $N \geq 1$ ) and we consider the first one to be the principal port, the other ones are auxiliary. Graphically the principal port is usually distinguished e.g. with an arrow. For example let's define three agents:

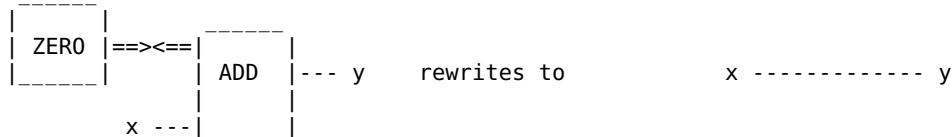




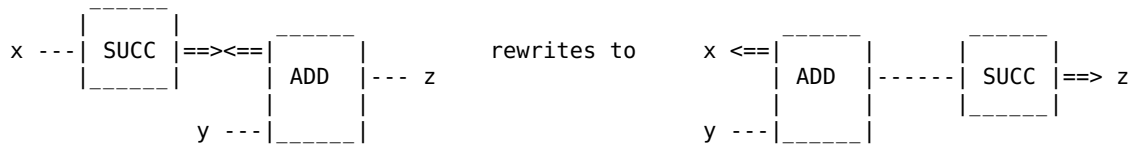
Here we defined agents ZERO, with one port (only the principal one), SUCC (for "successor", one principal and one auxiliary port) and ADD, with three ports (one of them is principal).

Now let's define interaction rules with these agents. An interaction rule simply defines a pattern of interconnected agents that get replaced by another pattern; however **interaction rule may only be defined for agents both connected by their principal ports** (so called *active pairs*). Naturally, the rules also have to keep the interface (edges going in and out) of the group the same. Our interaction rules will be following:

rule 1:

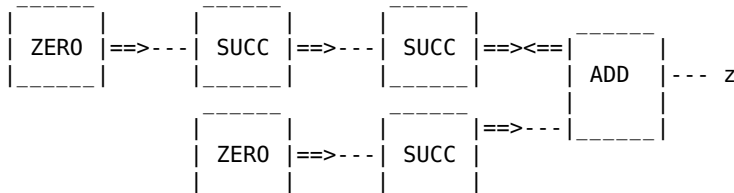


rule 2:



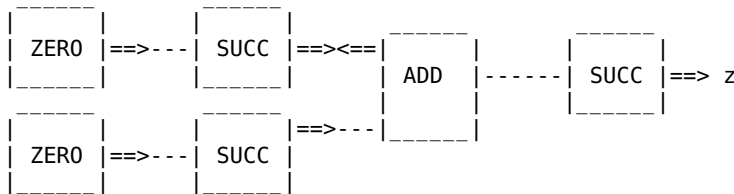
Now we can perform a computation of  $2 + 1$ . We first set up the interaction net that represents our program and then we'll be applying the interaction rules as long as we can:

representation of 2

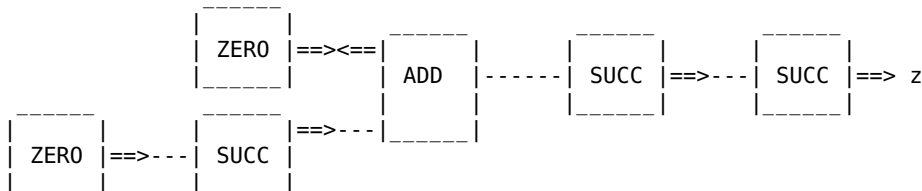


representation of 1

apply rule 2:

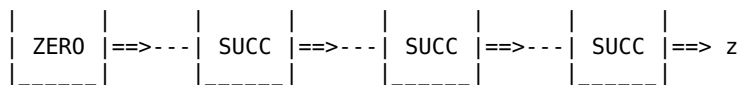


apply rule 2:



apply rule 1:



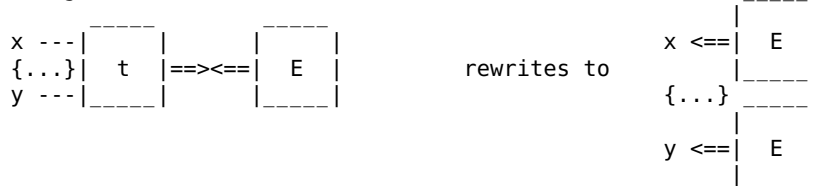


representation of 3 (the result)

no more rules to apply

One specific very important interaction system is called "**interaction combinators**" -- it is a very simple system, consisting of three predefined agents and three patterns of rewrite rules, which can simulate any other system, i.e. it is Turing complete (we say the model is capable of *universal computation*); we know this because it's possible to e.g. automatically convert any lambda calculus expression to interaction combinators. The following show the interaction combinator rules (E, D and Y are the three agents, t is a general subtree with possible additional inputs {...}):

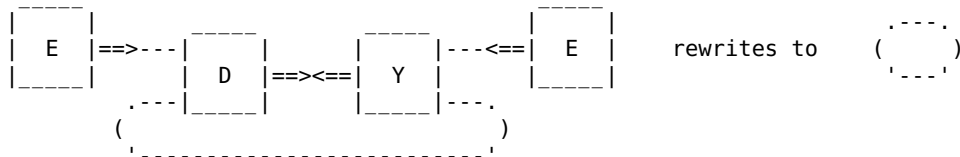
erasing rule:



duplication rule:



non-termination rule:



TODO: text representation, compare text representation of interaction nets with grammars?

## See Also

- [rule 110](#)

interesting

## Interesting

This is a great answer to anything, if someone tells you something you don't understand or something you think is shit and you don't know what to say, you just say "interesting".

The following is a "constantly work in progress" list of subjectively selected facts and topics that may be judged as interesting, preferably while also being lesser known:

- **Zanclean flood:** extremely huge flood that refilled the Mediterranean sea some 5 million years ago, which must have been a greatly spectacular event. Other similar mega floods are also discussed, e.g. that of the Black sea. Some of them are (often controversially) thought to be the origin of the shared great flood myth that's present in almost all old religions and cultures. A bit related interesting topic is

the 20th century Atlantropa mega project that planned to partially dry the Mediterranean to gain more space for Europe.

- **ball lightning**: a real weird phenomenon whose existence is not controversial (it is acknowledged to exist) but which still remains without exact scientific explanation. You can find some video of it, descriptions of eye witnesses are also quite fun to read.
- **extinct animals**: obviously everyone loves dinosaurs (BTW look up well preserved fossils of dinosaurs, some are in excellent state, we also now know for fact the skin color of some dinosaurs), but there are many underrated extincts animals, e.g. gigantopithecus, the biggest ape known to have ever existed (estimated weight up to 300 kg), which we however know almost nothing about (only teeth were found). More recently extinct species such as mammoth, dodo, saber-toothed cats, giant sloth, short-faced bear (probably the biggest bear ever) or Tasmanian tiger (which there still exists a black and white video footage of) are interesting exactly because they are closer on the timeline, people may have seen them and even depicted them somehow (e.g. cave paintings), we have may find much better preserved bodies of them and also have a bigger chance of cloning them one day or even discover them still living somewhere (has happened to several species already).
- **UFOs and aliens**: ufology is pretty fun when when you dig through the real retarded cases and ignore soyence fanatics that will stop being friends with you. Remember, you may enjoy digging into weird, suspicious cases without starting to wear tinfoil or seeing little green men behind anything; even if a UFO turns out to be a new, secret military tech or newly discovered atmospheric phenomenon and not aliens, it's still pretty damn interesting. Some cases are pretty solid, e.g. Hangzhou Xiaoshan (China) 2010 sighting of extremely weird tear in the sky which was scanning the whole city with some kind of obviously artificial light screen for a very long time, which was seen by thousands and captured on camera and video by many (e.g. [https://yt.artemislena.eu/watch?v=\\_\\_9s5chdV7c](https://yt.artemislena.eu/watch?v=__9s5chdV7c)) and even caused an airport to shut down -- the real nature of the thing was never explained and wasn't even much talked about (there also seems to be another simple-to-debunk cover up UFO sighting under the same keywords). The Travis Walton abduction case is also quite interesting, supported by a lot of evidence and has stood for a very long time. There are also many pretty good footages of weird UFOs, especially interesting are those captured by multiple people from different angles, which are extremely hard to fake.
- **Known but unexcavated archaeological sites**: e.g. the Qin tomb, a pyramid in China that's a resting place of a great emperor is buried underground -- historical records say the pyramid contains an unbelievable wealth, a great palace and models of cities, seas, waterways of quicksilver and traps protecting against intruders; this is believable as it is also the place where the astonishing terracota army was already excavated. However it is quite likely the pyramid won't be opened during our lifetime, we probably won't ever see it with our eyes. Also the well known pyramids and sphinx of Giza are still very mysterious -- e.g. there are holes in the great Sphinx you can clearly see but about which no one ever talks -- you can see they lead somewhere inside but you never see the actual inside, they let no one in and photos are nowhere to be found. Historical places of yet unknown locations, like the hanging gardens of Babylon, are also pretty interesting.
- Oldest existing photographs, video and audio recordings.
- 1816, so called **year without summer**, probably caused by great volcano eruption whose effects might have given a glimpse to what it looked like after the impact of the asteroid that killed the dinosaurs -- however this time many people wrote first hand witness accounts (you can find many in old books and reports, many times just scanned on Internet archive).
- Back before reddit became such huge shit interesting stuff could be discovered e.g. at <https://old.reddit.com/r/interestingasfuck/top/>, however nowadays it seems to be just a propaganda ground -- current all-time top two posts are both literally uninteresting political posts about Ukrainian war? :D Use internet archive to try to dig up the good stuff from the past maybe.
- People with perfect pitch (rare condition that makes one be able to precisely identify any musical tone) always lose this ability some time in their 50s.
- North Korea due to its isolation and secrecy, e.g. its own intranet. Also other secret computer networks like JWICS, SIPRNet, NIPRNet etc.
- conspiracy theories: Many are true.
- hybrids: No, we don't actually know if humans and apes can interbreed or not, humanzees have been reported, as well as hybrids of humans and other animals, there exist some real weird photos. Ligers and tigons are also cool, but there are many other interesting possibilities. See <http://www.macroevolution.net>.
- ...

# Internet

Internet (sometimes just the *net*) is the grand, decentralized global network of interconnected computer networks that allows advanced, cheap, practically instantaneous intercommunication of people and computers and sharing of large amounts of data and information. Over just a few decades since its birth in 1970s it changed the society tremendously, shifted it to the information age and stands as possibly the greatest technological invention of our society. It is a platform for many services and applications such as the web, e-mail, internet of things, torrents, phone calls, video streaming, multiplayer games etc. Of course, once Internet became accessible to normal people and has become the largest public forum on the planet, it has also become the biggest dump of retards in history.

Sometimes we distinguish between lowercase *i* "internet", meaning a large computer network, and capital *I* "Internet", meaning the one, biggest worldwide internet. As many networks just become part of the great Internet, we see this distinction less often and without saying otherwise, in normal speech both "internet" or "Internet" typically stand for the big Internet.

Internet is built on top of protocols (such as IP, HTTP or SMTP), standards, organizations (such as ICANN, IANA or W3C) and infrastructure (undersea cables, satellites, routers, ...) that all together work to create a great network based on packet switching, i.e. a method of transferring digital data by breaking them down into small packets which independently travel to their destination (contrast this to circuit switching). The key feature of the Internet is its decentralization, i.e. the attribute of having no central node or authority so that it cannot easily be destroyed or taken control over -- this is by design, the Internet evolved from ARPANET, a project of the US defense department. Nevertheless there are parties constantly trying to seize at least partial control of the Internet such as governments (e.g. China and its Great Firewall, EU with its "anti-pedophile" chat monitoring laws etc.) and corporations (by creating centralized services such as social networks). Some are warning of possible de-globalization of the Internet that some parties are trying to carry out, which would turn the Internet into so called splinternet.

Access to the Internet is offered by ISPs (internet service providers) but it's pretty easy to connect to the Internet even for free, e.g. via free wifis in public places, or in libraries. By 2020 more than half of world's population had access to the Internet -- most people in the first world have practically constant, unlimited access to it via their smartphones, and even in poor countries capitalism makes these devices along with Internet access cheap as people constantly carrying around devices that display ads and spy on them is what allows their easy exploitation.

The following are some **statistics** about the Internet as of early 2020s: there are over 5 billion users world-wide (more than half of them from Asia and mostly young people), it is estimated 63% people worldwide use the Internet with the number being as high as 90% in the developed countries. Most Internet users are English speakers (27%), followed by Chinese speakers (25%). It's also estimated over 50 billion individual devices connected, about 2 billion websites (over 60% in English) on the web, hundreds of billions of emails are sent every day, average connection speed is 24 Mbps, there are over 370 million registered domain names (most popular TLD is .com), Google performs about 7 billion web searches daily (over 90% of all search engines).

**PRO TIP: you should download and/or print your own offline Internet** (or maybe we should rather say offline web). Collect your favorite websites and other resources (gopher holes, Usenet threads, images, ...) and make a single dense PDF out of them. Process each page so that it's just plain text, remove all graphics and colors, unify the font, make the font small and decrease margins so that you fit as much as possible on a single page to not waste paper. For many pages, like Wikipedia, a small script will be able to do this automatically; the uglier pages may just be edited manually. An easy approach is for example to convert the pages to plain HTML that just contains paragraphs and heading of different levels, then copy-pasting this to LibreOffice, globally editing the font and auto-generate things like table of contents and page numbers, then exporting as PDF. You can even make a script that contains the list of pages you want to scrap so that you can make a newer print a few years later. Once you have the PDF, print it out and have your own tiny offline net :) It will be useful when the lights go out, it's a physical backup of your favorite sites (the PDF, as a byproduct, is also a single-file backup in electronic form), something no one will be silently censoring under your hands, and it's also just nice to read through printed pages, the experience is better than reading stuff on the screen -- this will be like your own 100% personalized book with stuff you find most interesting, in a form that's comfortable to read. You should also download your favorite and essential websites and other

files for offline use, this way you'll be able to browse even when the Internet collapses and/or if you're just somewhere without connection, plus you'll have a backup in case they go offline themselves. Here is a [KISS](#) script template that does the downloading (it can also at the same time serve as a list of your favorite websites), also feel free to improve it (e.g. compress/minimize the downloaded files etc.):

```
#!/bin/bash

rm -rf offline
mkdir offline

echo "
http://favoritesite1.com
https://favoritesite2.com/page1.html
http://favoritesite3.com/favoritefile1.txt
http://favoritesite4.org/coolimage.jpg
" | shuf | wget -i - -E -e robots=off -nc -nd -U "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)"
```

## History

See also [history](#) and [www](#).

{ Some sites with Internet history: <https://www.zakon.org/robert/internet/timeline/>,  
<https://www.freesoft.org/CIE/Topics/57.htm>. ~drummyfish }

It goes without saying that even though in retrospect it looks like the Internet just came to be one day, it wasn't indeed so -- we have to remember large communication networks existed for a long time and were often used in ways very similar to the Internet, even for silly things like playing [games](#) (e.g. [chess](#) used to be played over snail mail and even telegraph). Before electronic networks there were networks such as paper mail and optical telegraphs. With electricity a great number of new, much improved networks appeared, such as the electrical [telegraph](#) (~1840), phone and [fax](#) networks (~1880), radio broadcasts (circa first half of 20th century) and [TV](#) broadcasts (~1930). Some of the later networks were very similar to the World Wide Web from user perspective, and they were quite advanced and widely used at the time when Internet was just in its infancy -- for example [teletext](#) (~1970) allowed people to browse graphical pages on their TVs, [BBS](#) and [Usenet](#) networks were already [digital](#) computer networks (accessed through dialup [modems](#)) allowed people to chat, discuss on forums, roleplay, play games and share files, [Minitel](#) was the most successful Internet like network that worked in France in the 1980s etc.

The Internet itself evolved from [ARPANET](#), a network designed by [US](#) department of defense; ARPANET started to be developed in 1969 (with first plans appearing in 1966), fueled by Cold War rivalry with the Soviet Union. Of course, this network wasn't intended to become what the Internet is today, no one could probably have foreseen the future, it was just another [military](#) project -- as such, ARPANET was designed to be [decentralized](#) so as to be robust, i.e. there was no central node of the network which would be an easy target for enemies in a war. ARPANET was revolutionary by utilizing so called [packet switching](#) (idea published in a paper in 1961), i.e. any data sent over the network were split into small data [packets](#) that would travel through the network independently, each one possibly by different path, and would be reassembled into the whole once they all arrived at the destination (again, this helped keep the network robust -- if one path was destroyed, packets would just find another path). This is in contrast to traditional [circuit switching](#) used until then e.g. in telephone networks (circuit switching basically just means that direct connections are established between nodes that want to communicate at given time).

In April 1969 the first [RFC](#) ("request for comments") document was published (back then wrote with typewriter) -- RFCs would become a standard type of documents for discussing the design and improvements of ARPANET and later the Internet between the network engineers and scientists -- in RFCs new standards and [protocols](#) would be suggested, defined and discussed. 29 October 1969 is seen as a historical moment for ARPANET because at that day first data were sent through it from University of California -- it was a letter "L" (a whole word "LOGIN" was supposed to be sent but the computer crashed somewhere at "G"). In November of this year the first permanent ARPANET connection was established between University of California and Stanford Research Institute and shortly after a 4 node network was established.

By 1971 there were 15 ARPANET nodes. In 1974 allegedly the **first use of the word "Internet"** appeared in the specification of the TCP protocol by Cerf et al. The [TCP/IP](#) protocol they published would become a key



part of the Internet -- even today these protocols are the foundation of the Internet. By 1977 ARPANET had about 60 nodes.

In 1983 there were more than 500 registered hosts and in 1984 the number surpassed 1000. Also in 1984 the **DNS** (domain name system) was introduced -- this would allow network nodes to have "human friendly names" like *mycomputer.com* instead of just numeric addresses. In 1985 the first domain name was registered -- it was *symbolics.com*. In 1987 the number of hosts was around 10000. In 1989 this was already 100000.

In 1990 ARPANET project was officially ended to let the network, now mostly known as the Internet, live and be developed further mostly by the private sector. In this year **EFF** (Electronic Frontier Foundation), a major international non-profit that would help overlooking the Internet, was also founded. Due to the exploding popularity the Internet started to run out of IP addresses in early 1990s which was temporarily fixed by so called **CIDR** with long term plans to transition to bigger **IPv6** addresses.

Probably the biggest milestone in Internet history was the emergence of the **World Wide Web** -- also *www* or just "the web" -- in 1989 by **Tim Berners-Lee** who was at the time working at **CERN** in **Europe** (i.e. if we see the **US** as the inventor of the Internet, the Europe is who made it widespread and famous). The Web was based on the idea of documents (webpages) written in a special language (**HTML**), all interconnected via clickable links (so called **hypertext**) viewed with a program called **web browser**. Web's popularity was also helped by the fact that the programs made by Berners-Lee were released to the **public domain** so that anyone could jump on the web for free, even use it commercially without any fees and so on. And of course, a prerequisite for wide popularity was the presence of the cheap **personal computer**. Shortly after its invention web competed with other similar services based on similar ideas, most notably **gopher**, however some time in the mid 1990s the web took over and would quickly became by far the most prominent Internet service which would go on to make the Internet mainstream. In 1994 **w3c** (World Wide Web Consortium) was established to be the main organization standardizing the web. The web would gradually push all other networks and competing service -- such as **BBSes**, **Usenet** and **gopher** -- to the deepest underground. Of course, having become the **Earth's** largest public forum, the web would also ultimately become what would kill the Internet because all the major powers (read **corporations** and **states**) would quickly jump in to abuse it for their own propaganda, **marketing**, spying, manipulation, crowd control, cyberattacks and so on. This would still take some time, until around 2005 the web was great, very decentralized with plethora of useful personal web pages. People also weren't scared by **security** hysteria yet, **https** still wasn't the default, everyone would put his photos online along with his name, address and phone number, you could literally visit elementary school websites and find which children went to which class and so on -- no, nothing bad happened, it was all fine. However after this -- with the onset of so called web 2.0 (more **bloated** web) and so called **social networks** -- the downhill ride would start. It would still take around another decade for the web to die completely, until 2010 the web still kept part of its original glory, but after 2015 it all shattered. After 2020 the web is but a corpse inhabited by grandma's playing games on facebook while being bombarded by ads and the corpse of what used to be the web is just being kicked further to the ground by new capitalist cyberweapons such as the **"AI"**.

Nowadays not only the web but the Internet as a whole is dying by hardcore **capitalism**, becoming greatly **censored**, regulated, split (so called **splinternet**) and controlled by **corporations** who are absolutely killing the old decentralized, **free as in freedom** Internet that was developed by **free software** enthusiasts, nerds, oldschool **hackers**, **free speech** promoters, by universities, scientists and researches in transparent ways, through the RFCs. It is important to remember what it once used to be so that perhaps one day we can see the true Internet return.

Here is the Internet over time in numbers:

| year | ~inet servers | ~websites | ~domains | ~% inet users (glob.) |
|------|---------------|-----------|----------|-----------------------|
| 1969 | 4             |           |          |                       |
| 1970 | 10            |           |          |                       |
| 1975 | 100           |           |          |                       |
| 1980 | 200           |           |          |                       |
| 1985 | 2000          |           | 1        |                       |

| year | ~inet servers | ~websites  | ~domains  | ~% inet users (glob.) |
|------|---------------|------------|-----------|-----------------------|
| 1990 | 300000        | 1          | 9300      |                       |
| 1995 | 2000000       | 23000      | 71000     | 1                     |
| 2000 | 100000000     | 7000000    | 40000000  | 7                     |
| 2005 | 350000000     | 100000000  | 100000000 | 16                    |
| 2010 | 700000000     | 300000000  | 200000000 | 30                    |
| 2015 | 1000000000    | 1000000000 | 300000000 | 43                    |

## Alternatives To/Alternative Ways Of Implementing The Internet

See also <https://solar.lowtechmagazine.com/2015/10/how-to-build-a-low-tech-internet/>.

Internet overtook the world thanks to having enabled great number of services to be provided very cheaply, at great scales and/or with extremely elevated attributes such as minimal delay or great bandwidth. This is crucial to many industries who couldn't do without such a network, however to individuals or even smaller organizations Internet is frequently just a tool of comfort -- they could exist without the Internet, just a little less comfortably. As Internet is becoming more and more monitored, controlled, overcrowded, limited and censored, we may start to consider the less comfortable alternatives as good enough ways that actually gain us advantages in some other ways, e.g. more freedom of expression, more robust network (independence of the Internet infrastructure), technological independence etc. We have to keep in mind the services allowed by the Internet, such as long distance communication, information searching or playing games still mostly exist even without Internet, just usually separated or somehow suffering a few disadvantages; nevertheless these disadvantages may be bearable and/or made smaller, e.g. by adjusting ourselves to the limitations (if our communication becomes slower, we'll simply write longer messages to which we put more thought and information etc.) or combining these alternative services in a clever way. Additionally we can make use of the lessons learned from the Internet (e.g. cleverly designed protocols, steganography, broadcasts, digital data, ...) and apply them to the alternative networks. Let us now list a few alternatives to the Internet:

- **books, encyclopedias, magazines, libraries, printed media, paper, film, ...**: Paper is an awesome medium, it's cheap and can hold quite a lot of information, both digital and analog, it can be used without a computer but can still be combined with computers (e.g. printers, scanning and OCR, bar codes, ...) and/or lower tech tools like typewriters that may help manually copy books (see e.g. samizdat that heavily utilized the ability of typewriters to produce several copies at once; in Antiquity books were copied by slaves with one reading the original out loud with others writing down many copies). Quality paper can be used for reliable backups (source code, books, photos, even sound -- consider a high DPI print with each pixel recording one sample with its brightness). Posters can leave information for others to find. Books that have been written throughout history provide enormous amount of data and information, great part of which isn't even accessible through the Internet. Books are generally of much higher quality than websites, older ones are additionally free of modern propaganda and censorship. Print encyclopedias can here and there be used instead of Wikipedia, and they are extremely cheap (seek second hand book stores, no one wants them anymore). Books also provide entertainment, from traditional fiction, poetry etc. to entertaining reads such as the Guinness World Records book or even interactive RPG games (see gamebooks). Making your own small library of quality books isn't expensive at all and can really greatly reduce your dependence on the Internet in many ways. **Micrography** (scaling down documents to fit many of them on a small film) can help maximize store quite large amounts of data on small media without computers.
- **sneakernet, data mules, snail mail, avian carriers, arrows, messengers, USB exchange, messages in bottle, ...**: Physically transforming messages is another historically tested option, travelers will always be around wanting to get from point A to point B and while at it they may also serve as information carriers -- information doesn't weight that much. When combined with traditional "modern" data storage media such as USB drives we call this the sneakernet. Special case of this are so called data mules -- imagine e.g. a bus that carries a computer with wifi and drives from village to village, exchanging data with local computers in each village just by getting in close proximity, carrying data not only between the villages but also between the village network and the "big Internet" once it reaches a city that has connection to it (existing example is e.g. KioskNet) -- with clever software people can do things like send and receive emails and download websites, just a bit slower than with conventional Internet. There exist volunteer organizations that distribute mail.

People used to play correspondence chess over snail mail, with enough dedication you could probably scale it up to some turn-based MMORPG game. Owing to the small weight, data can be transferred also by small animals such as pigeons (in some places with very bad Internet this is allegedly still the superior way even nowadays, in wars pigeons helped carry huge numbers of messages on microforms) or even just by "throwing", shooting an arrow with message on it, sending it down the river stream and so on. USB sticks are used by activists to send western propaganda to North Korea (e.g. small helium balloons carrying USB sticks with movies and books over the borders for the inhabitants to find). The disadvantage is high communication delay but even if it's orders of magnitude worse than what Internet offers us, bandwidth can still be excellent, sometimes even beating the Internet! Consider that a truck carrying 1000 1 terabyte harddrives arriving from start to its destination in a week achieves a bandwidth of about 1.6 gigabytes per second. That's pretty solid. Future inhabitants of Mars and other planets will inevitably have to deal with interplanetary Internet that's doomed by laws of physics to have high delays -- if they can get around the issue, so can we. An interesting concept might be a "slow" network of people who simply meet up once a week and exchange their USB sticks (or SD cards or diskettes or whatever) on which they pass files and messages to others, such as requests for files etc.

- **leaving signs (rocks, sticks, leaves, messages in sand, bulletin boards, ...):** Some forest people communicate by leaving signs for others e.g. by leaving tears on leaves or making shapes from sticks or rocks -- these can carry messages like "beware, dangerous animal around", "today I hunted down a monkey here" or "I have extra food, come take some". When improved, we could communicate whole text messages, numbers and any binary data this way -- imagine e.g. a small "bulletin board" on some frequently visited crossroads between villages where people leave latest news, offers, demands, requests for information from others, silly jokes etc. In some cities there exist book exchange booths (often made from old phone booths) where people just leave their old books for others to take -- this could be further improved by adding some sort of message board for communication. Similarly networks such as *BookCrossing* work by people marking books with a tag and leaving them for others to find in some public place -- the books are traced on the Internet by their tags and may travel around the world.
- **intranet, LAN, WAN, ...:** Networks using basically the same technology as the Internet (TCP/IP, ethernet, wifi, routers, ...), just on smaller scales -- the technology can actually be simpler: simpler routers can be used, no high performance backbone routers are needed, Ronja may be used instead of wifi, DNS may be omitted and so on. There are many such networks, military has its own isolated networks, North Korea has its famous nation-wide isolated intranet (Kwangmyong), Cuba has the famous SNet -- "street net" that's used for pirating and games -- and so on. In Spain there is the famous Guifi network (with as of now nearly 40 thousand nodes) working in decentralized manner just on top of many interconnected wifi devices. The advantage is relative simplicity of implementation -- the technology is all there and quite cheap, you can set up your own network in the neighborhood and have complete control over it, government isn't gonna bully you for sharing movies, it won't spy on your communication (at least not so easily) etc.
- **radio, telegraph:** Plain FM/AM radio communication is a serious competition to Internet in terms of delay, bandwidth and distance of reach, while being very simple in comparison -- a skilled individual can construct or repair a radio with just some basic electronic components, which can't be said about digital computer networks that require extremely complex computer chips. Radio can relatively easily transfer analog information such as voice, but it can also send digital information. With Morse code even the most primitive radio communication system can turn into something extremely powerful.
- **broadcast and alternative network topologies** (see also world broadcast): broadcasts (one way communication towards many) can be implemented in many ways: with radio, audio, optically and so on. Broadcast only networks, such as teletext, TV or radio station broadcast, can be much simpler than a two way communication -- there don't have to be such complex protocols, there are no handshakes, devices can work on low power (as they're only receivers) and the broadcaster can't be overloaded by client requests. These can cover a great range of services such as news, weather forecast, time synchronization, geolocalization, work organization ("now we need you to produce this and this"), some forms of entertainment or providing generally useful data such as maps and books. If we do go for two way communication anyway, we should at least consider simpler network topologies -- with Internet we tend to think in mesh networks, i.e. "everyone connected to everyone", but that may be too complex to implement with other kinds of networks, it may be better to consider something like a ring network.
- **optical telegraph, smoke signals, lanterns, flag semaphores, kites, flares, mirrors and other optical communication:** Optical communication is another technique widely used throughout history -- the advantage here is speed as obviously light is the fastest medium you can ever use.

Lighting bonfires on hill tops could send a message about incoming enemy at great distances, ancient Greeks could even send more complex messages this way (see Phryctoria), later on even a more complex information could be sent using optical telegraph -- a chain of towers that forwarded symbols one to another by positioning big arms on their rooftops to form some specific shape, with the next tower copying the symbol and so on. You can leave big symbols in your window to send a few bytes to anyone with a telescope in the line of sight of your house. Basically if you can make someone see something, you can send a message; you can increase the amount of data by utilizing color, movement, blinking and so on. Also remember that optical fiber doesn't need a computer to work, it could probably be operated even manually provided we have some kind of laser.

- **audio signals (bells, canon shots, drums, horns, megaphones, ...):** Audio signal were again used a lot in history, a church bell could tell people many different things by how it was rang, canon shots could warn of incoming enemies and so on, voice can be used too. Drums are still widely used this way in Africa. The principle of string telephone can be considered to make some audio based networks.
- **pneumatic tube** and similar non-electric networks: A network of tubes using pressured air to transform small capsule containers from one place to another pretty fast, often used in factories -- this can carry written messages but also, unlike the Internet, physical objects! Other mechanism could be explored to construct similar networks, e.g. something based on hydraulics, string pulling, steam engines, gears, simple gravity (sending a marble down some tunnel could be a quite fast message) and so on.
- **phone networks, phreaking, power line communication etc.:** phone networks (and possibly other networks like the electric network, TV network etc.) can be used for all kinds of communication, with modems they can interconnect digital computers (which was widely used before Internet became widespread, see e.g. BBS networks); these networks can also be hacked to be used for free or cheap communication -- old time hackers knew how to rape phone boots to let them make free calls (see phreaking). Networks primarily used for carrying power can also carry information alongside power (see power line communication). Nowadays more anti hacking measures are in place but you may still e.g. exploit the fact that merely ringing someone's phone is completely free, which can be used to send a few bits of information. WARNING: It's generally illegal to mess with these networks in unintended ways, trying this shit's always on you :-). Also touching random electric cables can kill you. If you by accident take down some optical cable or something, you'll be fined to death.
- **normal voice communication:** As stupid as it sounds, we can sometimes just talk to other people, even if they live in another village, simply by going there and talking to them. You can use shouting to reach even people who are far away instantly -- some communities even invented things like whistling languages to communicate simple messages on extreme distances, this was used by hunters in forests etc. We got too much used to using cell phones to communicate with someone who just happens to be in another room, but this is just stupid, this can be just discarded as human degeneracy.
- **petroglyphs (rock carving), wood carving, glass painting, knot tying, metal tables etc.:** Data can be recorded manually in many materials, e.g. Incas used Quipu, a special knot tying language. Carving to stone is hard but will last for a long time, it is ideal for preserving small amounts of important information for a long time. See also rock carved binary data.
- **human memory:** Human memory can be used instead of computer memory, though we have to bear in mind its limitations. In very old times, before books became common and cheap, there existed people who made living by memorizing history in forms of long poems and recited them in public (this is how e.g. Iliad and Odyssey survived until they were actually recorded).
- **public fora:** Instead of an Internet discussion forum or chat it's possible to just allocate some public space for people to simply talk. Instead of YouTube videos people can go see someone's lecture, with the advantage of being able to actually talk to the guy and ask questions -- again, pretty obvious but the new generation may already be forgetting things can be done simply.
- **local storage/paper and offline programs instead of cloud:** This is again more of a note for the newer generation that's used to storing everything in the cloud and also using "cloud apps" -- you can (and SHOULD) store things locally of course, you can use offline programs and eve boomer solutions like a literal paper notebook for taking notes instead of using some online note taking "app". Similarly you can store your cash money and private photos in a physical safe instead of relying on Internet banking or password protected clouds and voila, suddenly you free of yet another bullshit.
- **doing it yourself, becoming independent:** you can replace many online services by just doing them yourself, for example instead of online weather forecast you can build your own small weather station, instead of online music streaming service you can just buy a harddrive, load it with mp3s and let it play on random, and so on.

- **"online only sometimes"**: An approach contrasted with the "always online" philosophy of the mainstream. This can greatly minimize dependency on connectivity, bandwidth and latency. Though a lot of technology is built with the premise of having constant access to the Internet, practically speaking few tasks require it by nature. You can do most things offline -- reading and replying to emails, reading and searching websites, watching movies, programming and committing to git repositories, playing slow-paced turn based games, all of these require just connecting to the Internet once in quite a long while to refresh the data, send the buffers out and download new queued resources. With just a little effort you can set this up.
- telepathy? :D
- ...

## See Also

- IWICS, SIPRNet, NIPRNet (secret/military networks)
- smol internet
- sneakernet
- World Wide Web
- splinternet
- Kwangmyong (North Korean intranet)
- Snet (large computer network on Cuba)
- soynet
- interplanetary internet
- books

---

interplanetary\_internet

## Interplanetary Internet

Interplanetary Internet is at this time still a hypothetical extension of the Internet to multiple planets. As mankind is getting closer to starting living on other planets and bodies such as Mars and Moon, we have to start thinking about the challenges of creating a communication network between all of them. The greatest challenge is posed by the vast distances that increase the communication delay (which arises due to the limited speed of light) and make errors such as packet loss much more painful. Two-way communication (i.e. request-response) to Moon and Mars can take even 2 seconds and 40 minutes respectively. Also things like planet motions, eclipses etc. pose problems to solve.

We can see that e.g. real time Earth-Mars communication (e.g. chat or videocalls) are physically impossible, so not only do we have to create new network protocols that minimize the there-and-back communication (things such as handshakes are out of question) and implement great redundancy for reliable recovery from loss of data traveling through space, we also need to design **new user interfaces** and communication paradigms, i.e. we probably need to create a new messaging software for "interplanetary chat" that will for example show the earliest time at which the sender can expect an answer etc. Interesting shit to think about.

{ TFW no Xonotic deathmatches with our Moon friends :( ~drummyfish }

For things like Web, each planet would likely want to have its own "subweb" (distinguished e.g. by TLDs) and caches of other planets' webs for quick access. This way a man on Mars wouldn't have to wait 40 minutes for downloading a webpage from the Earth web but could immediately access that webpage's slightly delayed version, which is of course much better.

Research into this has already been ongoing for some time. InterPlaNet is a protocol developed by NASA and others to be the basis for interplanetary Internet.

## See Also

- world broadcast

---

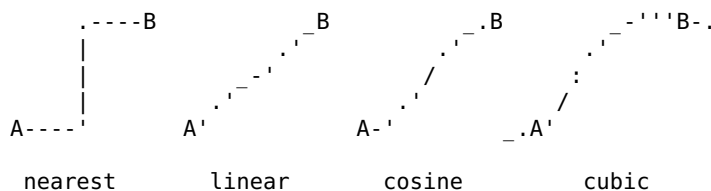
interpolation

# Interpolation

Interpolation (*inter* = between, *polio* = polish) means computing (usually a gradual) transition between some specified values, i.e. creating additional intermediate points between some already existing points. For example if we want to change a screen pixel from one color to another in a gradual manner, we use some interpolation method to compute a number of intermediate colors which we then display in rapid succession; we say we interpolate between the two colors. Interpolation is a very basic mathematical tool that's commonly encountered almost everywhere, not just in programming: some uses include drawing a graph between measured data points, estimating function values in unknown regions, creating smooth animations, drawing vector curves, digital to analog conversion, enlarging pictures, blending transition in videos and so on. Interpolation can be used to generalize, e.g. if we have a mathematical function that's only defined for whole numbers (such as factorial or Fibonacci sequence), we may use interpolation to extend that function to all real numbers. Interpolation can also be used as a method of approximation (consider e.g. a game that runs at 60 FPS to look smooth but internally only computes its physics at 30 FPS and interpolates every other frame so as to increase performance). All in all interpolation is one of the most important things to learn.

The opposite of interpolation is extrapolation, an operation that's *extending*, creating points OUTSIDE given interval (while interpolation creates points INSIDE the interval). Both interpolation and extrapolation are similar to regression which tries to find a function of specified form that best fits given data (unlike interpolation it usually isn't required to hit the data points exactly but rather e.g. minimize some kind of distance to these points).

There are many methods of interpolation which differ in aspects such as complexity, number of dimensions, type and properties of the mathematical curve/surface (polynomial degree, continuity/smoothness of derivatives, ...) or number of points required for the computation (some methods require knowledge of more than two points).



*A few common 1D interpolation methods.*

The base case of interpolation takes place in one dimension (imagine e.g. interpolating sound volume, a single number parameter). Here interpolation can be seen as a function that takes as its parameters the two values to interpolate between,  $A$  and  $B$ , and an **interpolation parameter**  $t$ , which takes the value from 0 to 1 -- this parameter says the percentage position between the two values, i.e. for  $t = 0$  the function returns  $A$ , for  $t = 1$  it returns  $B$  and for other values of  $t$  it returns some intermediate value (note that this value may in certain cases be outside the  $A$ - $B$  interval, e.g. with cubic interpolation). The function can optionally take additional parameters, e.g. cubic interpolation requires to also specify slopes at the points  $A$  and  $B$ . So the function signature in C may look e.g. as

```
float interpolate(float a, float b, float t);
```

Many times we apply our interpolation not just to two points but to many points, by segments, i.e. we apply the interpolation between each two neighboring points (a segment) in a series of many points to create a longer curve through all the points. Here we are usually interested in how the segments transition into each other, i.e. what the whole curve looks like at the locations of the points.

**Nearest neighbor** is probably the simplest interpolation (so simple that it's sometimes not even called an interpolation, even though it technically is). This method simply returns the closest value, i.e. either  $A$  (for  $t < 0.5$ ) or  $B$  (otherwise). This creates kind of sharp steps between the points, the function is not continuous, i.e. the transition between the points is not gradual but simply jumps from one value to the other at one point.

**Linear interpolation** (so called lerp) is probably the second simplest interpolation which steps from the first point towards the second in a constant step, creating a straight line between them. This is simple and good enough for many things, the function is continuous but not smooth, i.e. there are no "jumps" but there may

be "sharp turns" at the points, the curve may look like a "saw".

**Cosine interpolation** uses part of the cosine function to create a continuous and smooth line between the points. The advantage over linear interpolation is the smoothness, i.e. there aren't "sharp turns" at the points, just as with the more advanced cubic interpolation against which cosine interpolation has the advantage of still requiring only the two interval points (A and B), however for the price of a disadvantage of always having the same horizontal slope at each point which may look weird in some situations (e.g. multiple points lying on the same sloped line will result in a curve that looks like smooth steps).

**Cubic interpolation** can be considered a bit more advanced, it uses a polynomial of degree 3 and creates a nice smooth curve through multiple points but requires knowledge of one additional point on each side of the interpolated interval (this may create slight issues with the first and last point of the sequence of values). This is so as to know at what slope to approach an endpoint so as to continue in the direction of the point behind it.

The above mentioned methods can be generalized to more dimensions (the number of dimensions are equal to the number of interpolation parameters) -- we encounter this a lot e.g. in computer graphics when upscaling textures (sometimes called texture filtering). 2D nearest neighbor interpolation creates "blocky" images in which pixels simply "get bigger" but stay sharp squares if we upscale the texture. Linear interpolation in 2D is called bilinear interpolation and is visually much better than nearest neighbor, bicubic interpolation is a generalization of cubic interpolation to 2D and is yet smoother than bilinear interpolation.

TODO: simple C code pls, maybe linear interpolation without floats

## See Also

- extrapolation
- regression
- smoothstep

---

ioccc

## International Obfuscated C Code Contest

The International Obfuscated C Code Contest (IOCCC for short) is an annual online contest in making the most creatively obfuscated programs in C. It's kind of a "just for fun" thing but similarly to esoteric languages there's an element of art and clever hacking that carries a great value. While the productivity freaks will argue this is just a waste of time, the true programmer appreciates the depth of knowledge and creative thinking needed to develop a beautifully obfuscated program. The contest runs since 1984 and was started by Landon Curt Noll and Larry Bassel.

Unfortunately some shit is flying around IOCCC too, for example confusing licensing -- having a CC-BY-SA license in website footer and explicitly prohibiting commercial use in the text, WTF? Also the team started to use Microshit's GitHub. They also allow latest capitalist C standards, but hey, this is a contest focused on ugly C, so perhaps it makes sense.

Hacking the rules of the contest is also encouraged and there is an extra award for "worst abuse of the rules".

Some common ideas employed in the programs include:

- formatting source code as ASCII art
- misleading identifiers and comments
- extreme macro/preprocessor abuse
- abuse of compiler flags
- different behavior under different C standards
- doing simple things the hard way, e.g. by avoiding loops
- including weird files like `/dev/tty` or recursively including itself
- code golfing

- weird stuff like the main function recursion or even using it as a signal handler :)
- ...

And let us also mention a few winning entries:

- program whose source code is taken from its file name (using `__FILE__`)
- ray tracer in < 30 LOC formatted as ASCII art
- operating system with multi-tasking, GUI and filesystem support
- neural machine learning on text in < 4KB
- program printing "hello world" with error messages during compilation
- X11 Minecraft-like game
- web browser
- self-replicating programs
- ...

## See Also

- NaNoGenMo
- SIGBOVIK
- C compiler bombs

---

io

## Input/Output

In programming input/output (I/O or just IO) refers to communication of a computer program with the outside environment, for example with the user in real world or with the operating system. Input is information the program gets from the outside, output is information the program sends to the outside. I/O is a basic and very important term as it separates any program to two distinct parts: the pure computational system (computation happening "inside") and I/O which interconnects this system with the real world and hence makes it useful -- without I/O a program would be practically useless as it couldn't get any information about the real world and it couldn't present computed results. In hardware there exists the term "I/O device", based on the same idea -- I/O devices serve to feed input into and/or get output from a physical computer, for example keyboard is an input device and monitor is an output device (a computer without I/O devices would be useless just as a program without I/O operations).

Note that I/O is not just about communication with a human user, it also means e.g. communication over network, reading/writing from/to files etc.

It is possible to have no input (e.g. a demo), but having no output at all probably makes no sense (see also write-only).

**I/O presents a challenge for portability!** While the "pure computation" part of a program may be written in a pure platform-independent language such as C (and can therefore easily be compiled on different computers) and may be quite elegant, the I/O part gets more ugly.

This is because **I/O is inevitably messy**: an abstract, portable I/O library really tries to do the impossible task of unifying all wildly differing physical computers and their architectures under some simple functions; for example consider an I/O library will offer a function such as `drawPixel(x,y,color)` that draws a pixel to the screen -- how do we make this work for all computers? What value is *color* here, is it RGB, a color index, HDR value? What if a computer doesn't allow writing to arbitrary parts of screen coordinates because it lack a frame buffer, or what if such operation is painfully slow there (some computers may just want to write pixels sequentially in possibly varying orders we can't predict)? WHAT IF the computer doesn't even have a raster screen but instead has a vector screen? Even such things as files residing in a tree of directories are something that's highly established but not necessarily the only way a computer may work, some computers may for example support files but not directories, how does our library take this into account? How do we deal with file names with very weird characters, what if someone makes a file system where file names are actually rich text or where files aren't places in directories but are rather points in 3D space or something? So an I/O library has to inevitably make many assumptions about what a "normal" computer looks like and



what will likely help it operate fast etc. It has to decide how to deal with unsupported things, for example if we try to display color on a black and white display will it cause an error or will we try to somehow approximate the color just with shades of gray? And of course with new I/O devices appearing (VR, brain interfaces, ...) the library will have to be constantly updated. So the I/O part of the program will usually require some platform specific library or a library with many dependencies; for example to display pictures on screen one may use SDL, OpenGL, Linux framebuffer, CSFML, X11, Wayland and many other libraries, each one handling I/O a bit differently. Whatever library you choose, it may be unavailable on some other platform, so the program won't run there. Some hardware platforms (e.g. many game consoles) even have their own exclusive I/O library, use of which will just tie the program to that single platform. There are programming languages and libraries that try to provide platform-independent I/O, but as said such approach is limited as it has to assume some common features that will be available everywhere; for example C has a standard platform-independent I/O library *stdio*, but it only allows text and binary input/output, for anything advanced such as graphics, sound and mouse one has to choose some 3rd party library. Unix philosophy also advises to only use text I/O if possible, so as to "standardize" and tame I/O a bit, but then again one has to choose what communication protocol/format to use etc., so the problem just shifts from standardizing library API to standardizing protocols. So generally I/O is a problem we have to deal with.

How to solve this? By separating I/O code from the "pure computation" code, and by minimizing and abstracting the I/O code so that it is easily replaceable. Inexperienced programmers often make the mistake of mixing the pure computation code with I/O code -- it is then very difficult to replace such I/O code with different I/O code on a different platform. See portability for more detail. Also if you don't have to, **avoid I/O altogether**, especially if your project is a library -- for example if you're writing a 3D rendering library, you do NOT actually need any I/O, your library will simply be computing which pixels to draw and what color they should have, the library doesn't actually have to write those pixels to any screen, this may be left to the user of the library (this is exactly how small3dlib works).

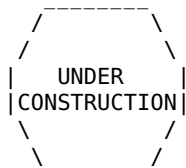
I/O also poses problems in some programming paradigms, e.g. in functional programming.

TODO: code example

---

iq

## IQ

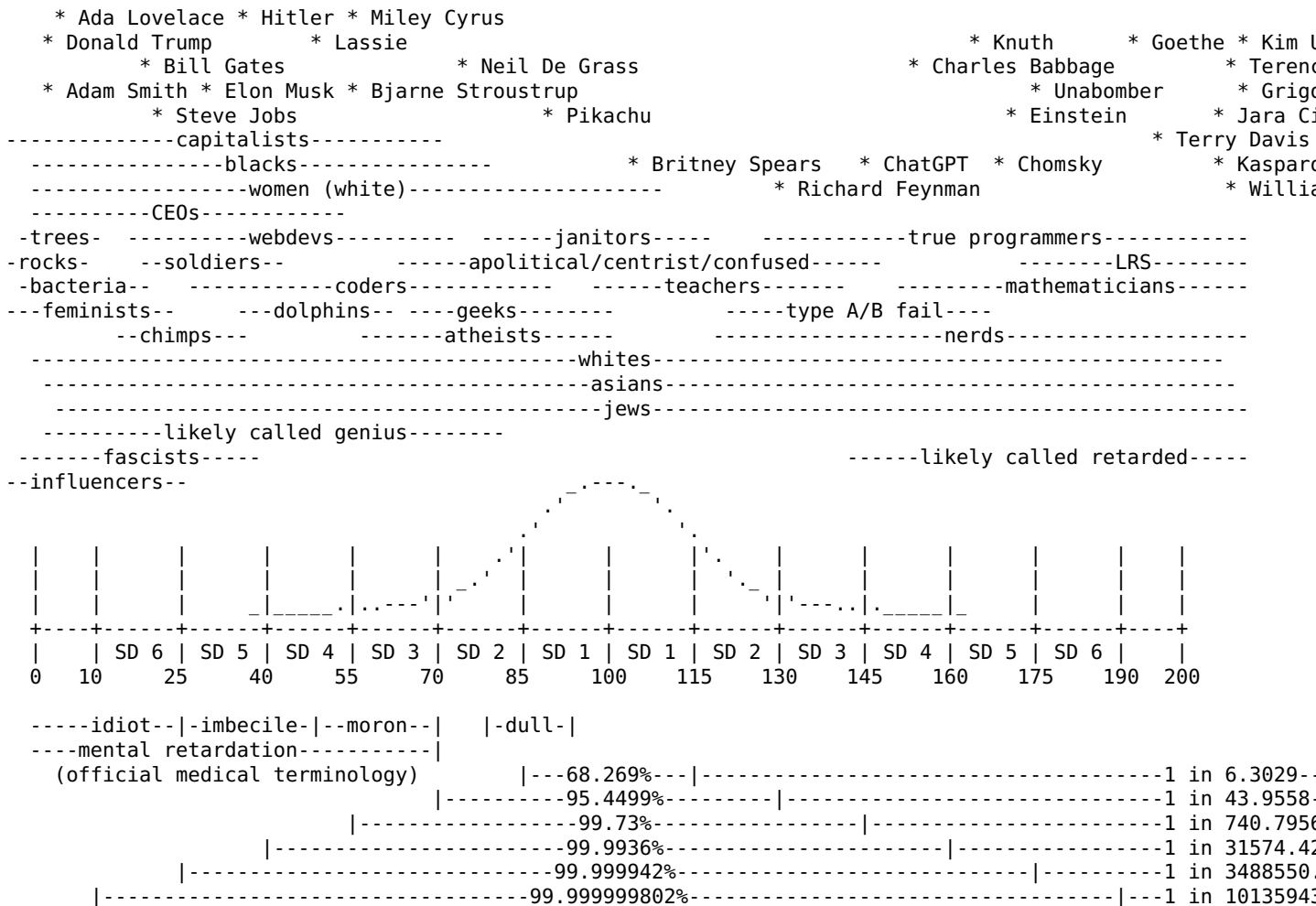


Please wear a hard hat when reading this page.

IQ (intelligence quotient) is a non-perfect but still kind of useful measure of one's intelligence, it is a numeric score one gets on a standardized test that tries to estimate his intellectual ability at different tasks (logic, memory, language skills, spatial skills, ...) and express them with a single number. The tests are standardized and the scoring is usually tuned so that the value 100 means average intelligence -- anything above means smarter than average, anything below dumber than average. IQ is a quite controversial topic because it shows intellectual differences between races and sexes and clashes with political correctness, there is also a great debate about "what intelligence even is" (i.e. what the test should measure, what weight should be given to different areas of intelligence), if it is even reasonable to simplify "intelligence" down to a single number, how much of a cultural bias there is (do we really measure pure intellectual capacity or just familiarity with some concepts of our western culture?) and the accuracy of the tests is also highly debated (which can be an issue if we e.g. start using IQ tests to determine who should get higher education and who shouldn't) -- nevertheless it's unquestionable that IQ DOES correlate with intellectual abilities, IQ tests are a tool that really does something, the debates mostly revolve around how useful the tool is, how it should be used, what conclusions can we make with it and so on. Basically only people with the lowest IQ say that IQ is completely useless. The testing of IQ was developed only during 20th century, so we don't know IQs of old geniuses -- if you read somewhere (including this article) that Newton's IQ was 200, it's just someone's wild guess.

IQ follows the normal probability distribution, i.e. it is modeled by the bell curve that says how many people of the total population will fall into any given range of IQ score. Though this has been challenged too, one of the basic laws of human stupidity says that the probability that someone is stupid is independent of any other of his characteristics (education, profession, race, sanity, ...). There are various IQ scales, almost all use the Gaussian (bell) curve that's centered at 100 (i.e. 100 is supposed to mean the average intelligence) and have standard deviation 15 (but other have been used as well) -- this is what we'll implicitly suppose in the article from now. This means that about 2/3rds of people will fall in the range 85 to 115 but no more than 1% will have IQ higher than 145 or lower than 55. Sometimes you may also encounter so called **percentile** which says what percentage of population is below your IQ.

TODO: more details, history, where to measure (web vs Mensa vs SAT etc.)



IQ distribution along with approximate placement of certain groups and individuals. Notice how interesting people are either far to the right or far to the left. Also notice how the smartest you've rarely heard of while you've heard of all the dumbest.

**IQ and race and sex:** IQ is correlated with race and sex. The following is a comparison of average IQs of groups with various combination of the two factors:

men (Jew) > men (Asian) > men (White) > men (Brown) > women (Jew) > women (Asian) > women (White) > women (Brown) > men (Black) > women (Black)

**If you think you're smart, you are dumb**, see the infamous Dunning Kruger effect -- becoming smarter comes with feeling dumber and dumber, becoming more humble and less self confident as you just see all the new things you didn't even know you don't know -- Socrates, one of the greatest philosophers of all times and possibly the smartest man of his time, famously summed this up by saying "I know that I know nothing". A fool thinks he is close to knowing everything -- he admits he doesn't know everything, but he thinks he knows like 90% of what the smartest people on Earth know because he didn't even step over the borders of

obtaining the basic knowledge, that border is as far as he can see and he doesn't know beyond it lies an infinitely large plain of knowledge into which some managed to get kilometers ahead of him, they are so far away he has no idea anyone can even get that far. It's similar to how the better we explore the space, the more we see how tiny we are -- not long ago we might have thought our galaxy was the whole Universe, now we know it's just a tiny speck in a cluster that's itself just a small speck in the observable Universe which is a nothing in the scale of the whole infinite Universe. Self confidence implies extreme stupidity. Also note that feeling dumb doesn't imply being smart but admitting retardedness is a prerequisite for being smart.

**Is IQ a useful measure and if so, how important is the score?** Firstly if you are insecure about your own IQ then just stop that shit -- you know yourself, you know if you're good at math or writing or whatever else you try to do, do you need a piece of paper patting you on the back or something? That's completely pointless, the only thing worth of discussion is IQ as some standardized tool of estimating intellectual abilities of other people on a bigger scale, e.g. as some kind of filter in education (with small groups you can really just interview the people and see if they're dumb or not, that's also more reliable than IQ tests). In this of course the question of the validity of IQ is a controversial one, discussed over and over. Modern "inclusive" society dismisses IQ as basically useless because it points out differences between races etc., some rightist are on the other hand obsessed with IQ too much as it creates a natural hierarchy assigning each man his rank among others. True significance of IQ as a measure seems to be somewhere in between the two extremes here. As it's always noted about IQ, we have to remember the term "intelligence" itself is fuzzy, there doesn't and cannot exist any universal definition of it, so we have trouble even grasping what we're measuring and however we define intelligence, it usually ends up hardly even correlating with "success" or "achievements" or anything similar, so firstly let's see IQ just as what it literally is: a score in some kind of game. Furthermore intelligence is extremely complex and multidimensional (there is spatial and visual intelligence, long and short term memory, language skills, social and emotional intelligence etc.), capturing all this with a single number is inevitably a simplification, the score is just a projected shadow of the intelligence with light cast from certain angle. IQ score definitely does say a lot about some specific kind of "mathematical" intelligence, though even if designed to be so, even in this narrow sense it isn't anywhere near a perfect measure -- though a minority, some mathematicians do score low on IQ tests (Richard Feynman, physics Nobel Prize laureate had famously a relatively low score of 125). It's perhaps good to keep the "IQ tests as a game" mindset -- intelligent people will be probably good at it but some won't, performance can be increased by training, there will be narrowly focused autists who excel at the game but are extremely dumb at everything else etc. Having IQ score predict what we normally understand to be "intelligence" is like having height, weight and age predict how good of a soldier someone will be -- there will be some good correlations, but not nearly perfect ones. Some general IQ range will be necessary for certain tasks such as programming, but rather than +5 on an IQ score things such as education and personality traits will play much more important roles in actually achieving something or creating something good; for example curiosity and determination, the habit of thinking about everything in depth, nonconformity, a skeptical mind, all these are much more important than being a human calculator -- remember, the cheapest calculator will beat the smartest man in multiplying numbers, would you say it is more intelligent?

{ Also consider this: even if you're average, or even a bit below average, you're still homo sapiens, so as long as you're not a feminist or capitalist you'll always be the absolute top organism in intelligence, a member of by far the absolutely most intelligent species that ever appeared on Earth, your intelligence greatly surpasses great majority of living organisms. If you are able to read this, you already possess the great genius, you mastered language and are among the top 0.1%, there's no need to compare yourself to others and aim to be in 0.01% instead of 0.02%. Rather think about what good to do with the gift of reason you've been given. ~drummyfish }

{ It's still more and more complicated the more you think of it, even for example success in mathematics may sometimes depend less on pure math skills and more on non-mathematical kind of intelligence, e.g. that of observation skills and communication -- that's what academia is about. Yes, you need some creativity, but the ability to quickly understand ideas of others may sometimes be superior, an idea you "steal" from someone else is as useful as idea you came up with yourself, you need to catch many ideas of others and connect them together; on the other hand struggling with communication is sometimes simply like not speaking a common language at all. Thinking back I for one have always been quite retarded at understanding what others wanted to say, even simple things, so in classes I frequently wouldn't understand what was being taught while others understood, but it wasn't because I wouldn't understand the concept itself, I rather didn't understand the way the teacher explained it because (I think) I think differently about things. When we were given tasks to solve on our own, I usually beat my classmates because that was only about creative intelligence, not communication, and in this I think I was better than most of my peers. I didn't

go for PhD later on while some of my classmates did -- TBH I don't think it's because they were necessarily more intelligent in general (many of them for sure were), but because they felt better in this world of communication, sharing papers, talking to others, understanding their ideas and collaborating, they had the "better mix" of intelligence for today's academic world -- this I always had problems with, so it contributed to my decision to not go there. This is just to show that this world is quite complex. ~drummyfish }

**Fun fact:** in some US countries idiots and similar low IQ level classes are legally prohibited from voting :D

It's been observed that the **IQ of a group** is roughly equal to the IQ of its least intelligent member divided by the number of group members.

**Why are rich people dumb and smart people poor?** Let's clarify this because there is a very common misconception, purposefully established by capitalist propaganda, that rich people are smart and vice versa, while the opposite is always true. Firstly succeeding on the market is basically just lottery, it's more about luck than anything else -- no one can predict the market, it's literally about betting on the right card and then being in many right places at right times. At best if anything gives you better odds, it's having a starting capital, knowing someone in the business, living in a good location etc., but not being smart. No matter what anyone tells you, business is at best an educated bet and a businessman is at best as smart as your average slot machine addict. Now of course, just by statistics, the few people who win the lottery are likely to fall in the majority, i.e. around the average IQ. This is additionally further biased towards lower IQ because success in business is actually favored by lower IQ for a number of reasons. One of them is a stupider man is going to make just plain stupid big risks, have the capitalism-aligned short term mindset, and making stupid risks is how you more likely win the game -- it is also how you more likely lose, but no one cares about the losers, these just disappear. It's like seeing what kind of people survive running through a minefield -- you will find only retards make it because only a retard would attempt it; many of them will die but some manage to run through by sheer luck, while all the smart people just retain from even trying to run through the mine field. That's why so many of the famous rich guys always made some hilariously risky moves like selling everything they had and putting it into their business -- that's just extremely retarded, only 1 in a million will succeed like that, but of course by survivorship bias you will only hear about those who succeed like this so they look smart (plus they will pay for their own propaganda painting them as smart), but no, it's still incredibly stupid. Furthermore success in business is about lack of morality to which intelligence is an obstacle -- an intelligent man sees that investing in some business will e.g. lead to someone in the third world dying, so for a smart man doing business is literally as if he had to murder people for money, which most normal people wouldn't do, but with the extra layer of indirection stupid people can't see the harm they do to others and they happily murder people without even realizing it. Again, this is why the rich guys are incredibly stupid assholes behaving like angry aggressive chimpanzees to their employees, like for example Steve Jobs or Elon Musk, they just lack brain cells that you need for empathy. Yet another reason why smart people aren't rich is simply that they see that pursuing money is incredibly retarded, it's just satisfying low material and social needs, it's just like overeating, drug addiction or gambling, it's an arbitrarily set unhealthy goal that achieves nothing but waste one's life on counting pieces of green paper while doing bullshit and immense harm as a side effect, smart people see through this, they want something more from life, they want to spend time with their family, create something that helps others, make art and so on, so they just won't go after money and they even turn down money if they can have it (see e.g. Grigori Perelman).

{ Coincidentally after I wrote this I saw some "REAL STUDY" (as in peer censored etc.) which concluded income is slightly positively correlated with IQ, which is probably true around the average middle IQ (a slightly smarter wage slave will make a bit more money by pressing buttons on a computer than a wage slave moving crates), however the study also noted that by far the few richest individuals that took part in the study (probably some CEOs, they were literally sky high in the graph) were all quite significantly BELOW 100 IQ :D So now it's also official. ~drummyfish }

**Can you increase your IQ?:** Though they're not supposed to be trainable, IQ tests CAN be trained like anything else -- you can train for the tests themselves, they often have the same types of questions -- if you just practice 4 hours making IQ tests for a month before taking an actual IQ test, you'll probably score at genius level, but that's kinda cheating of course, it's better to REALLY train you brain skills by just doing various diverse intellectual challenges like programming, learning languages, playing puzzles, board games or reading books -- this will actually make you smarter. Some studies have shown that playing video games increases IQ, however it probably matters what sort of game it is, mindless grinding in WoW or cybersexing in Roblox will probably have the opposite effect, you rather want to play puzzle games, strategy games and

so on. There was one specific memory game that was shown to really increase IQ scores, though only temporarily (for a few months) -- in the game you saw symbols appearing on the screen and you had to press a button every time a symbol appeared that was the same as a symbol that appeared  $N$  steps back; you start at  $N = 1$  and keep training yourself towards higher values, and this basically boosts your short term memory. { Can't find the paper now but I swear I've seen it, I even played the game for a while. ~drummyfish } There are also things like the "Mozart effect" which says that listening to Mozart's music increases your IQ score, but who knows if that's bullshit or not.

TODO: smartest man?

## Real Genius VS Pseudogenius

Most people are called a genius nowadays -- any recent so called "genius" (such as Steve Jobs) is in fact most likely of below average IQ; just barely above mediocre idea someone comes up with by chance will be celebrated as that of a genius, **real genius ideas will be met with hostility**; real genius ideas are too good and too far ahead and unacceptable to normal people. Furthermore success in business requires lack of intelligence so as to be unable to see the consequences of one's actions. Your cat watching you solve Riemann hypothesis will not even know what's happening, to it you are a retard wasting time on sliding a stick over table, on the other hand the cat will judge a monkey capable of opening a can of cat food a genius. Society is composed solely of idiots, they can only see if someone is a tiny bit better at what they do than them, and those they celebrate, if you are light years ahead of them they don't even have the capacity to comprehend how good you are at what you do because they can't even comprehend the thing you do. This includes even PhDs and people with several Nobel Prizes, everyone except the few supporters of LRS are just blind idiots playing along with the system, some lucky to succeed in it and some not. This is why shit technology is prospering and LRS is being overlooked. It's just another confirmation our ideas as superior.

Consider this analogy (yes, analogies are good): in a race you can only see those who are plus or minus 20 meters away from you, you can assess everyone else's position only by someone else telling you, so if someone is 50 meters ahead of you, you can know but only by someone ahead of you telling you that someone ahead of him told him he saw him there way up in the front. Now since there are many fewer of high IQ people, they have lower probability of being recognized, simply because there are few people capable of recognizing them -- in mainstream places like Universities you still likely will be recognized as there are smart people around, and the knowledge of your genius will be chain propagated to the mainstream monkeys, but if you're a genius outside a mainstream place, the chance is almost zero you will be recognized (and if you're smart you will probably also not try to be recognized, only retards do that). With this mainstream will simply lack information about your intelligence, they will only see a question mark above your head -- they know you're not average, because averages get recognized very quickly, everyone can assess those -- so now they know you're either really smart or really dumb, and since you don't fit the false, twisted idea of mainstream pseudogenius (being rich, famous, ...), they will conclude you belong to the latter class, i.e. that you're a retard. Note that the same effect manifests also with the pseudogenius, just in the opposite way -- the dumbest people, like CEOs, are too far away from the average (now towards lower values), so the mainstream isn't sure about their intelligence; here however the CEO applies manipulation, he has the money to pay for a biography book that paints him as a genius, he can pay someone to write him speeches so that he appears to say smart things, he pays people to invent things signed by his name, or he simply steals them with the power of money (Edison, Jobs, ...) etc., so he ends up being taken for genius, despite actually being dumber than many animals (even a dog has enough brain cells to feel for example empathy, something way too complex for a CEO).

{ The short story *Country of the Blind* by H. G. Wells is a nice story about this phenomenon of too much competence being seen as a lack of competence, illustrated on a story of a completely healthy man who finds himself in a village of people who are all blind. ~drummyfish }

## Quick IQ Estimates

If you are American (or just someone else who happened to take the test), your SAT is basically your IQ, it has been shown that SATs highly correlate with standardized psychologist IQ tests. Just take your percentile on SATs and convert it to IQ and you have your IQ.



- **D: drummyfish's house**: It's a plain single-room bamboo hut near the beach, drummyfish lives here with the dog who however often roams the whole island and welcomes all the newcomers.
  - **G: graveyard**
  - **T: a modest zen temple**: It has nice view of the sea and we go meditate here.
  - **F: the beach forum**: It's a beach where people naturally gather to discuss in groups.
  - **I: the tiny island**: A small island, for those who want to be extra alone.
- 

jargon\_file

## Jargon File

Jargon File (also Hacker's Dictionary) is a computer hacker dictionary/compendium that's been written and updated by a number of prominent hackers, such as Richard Stallman and Erik S Raymond, since 1970. It is a chiefly important part of hacker culture and has also partly inspired this very wiki.

{ A similar but smaller encyclopedia is at <https://www.erzo.org/shannon/writing/csua/encyclopedia.html> (originally and encyclopedia at [soda.csua.berkeley.edu](http://soda.csua.berkeley.edu)). ~drummyfish }

The work informally states it's in the public domain and some people have successfully published it commercially, although there is no standard waiver or license -- maybe because such waivers didn't really exist at the time it was started -- and so we have to suppose it is NOT formally free as in freedom. Nevertheless it is freely accessible e.g. at Project Gutenberg and no one will bother you if you share it around... we just wouldn't recommend treating it as true public domain.

It is fairly nicely written with high amount of humor and good old political incorrectness, you can for example successfully find the definition of terms such as rape and clit mouse. Some other nice terms include smoke emitting diode (broken diode), notwork (non-functioning network), Internet Exploiter, binary four (giving a finger in binary), Kamikaze packet or Maggotbox (Macintosh). At the beginning the book gives some theory about how the hacker terms are formed (overgeneralization, comparatives etc.).

---

java

## Java

*Unfortunately 3 billion devices run Java.*

Java (not to be confused with JavaScript) is a highly bloated, inefficient, "programming language" that's sadly kind of popular. It is compiled to bytecode and therefore "platform independent" (as long as the platform has a lot of resources to waste on running Java virtual machine). Some of the features of Java include bloat, slow loading, slow running, supporting capitalism, forced and unavoidable object obsession and the necessity to create a billion of files to write even a simple program.

{ I tried programming in Java but it made me so much dumber, it literally closed my eyes in so many ways, never touch it if you can. ~drummyfish }

Avoid this shit.

{ I've met retards who seriously think Java is more portable than C lol. I wanna suicide myself. ~drummyfish }

---

javascript

## JavaScript

*Not to be confused with Java.*

JavaScript (JS) is a very popular, highly shitty bloated scripting programming language used mainly on the web. The language is basically the centerpoint of web development, possibly the worst area a programmer can find himself in, so it is responsible for a great number of suicides, the language is infamously surrounded by a clusterfuck of most toxic frameworks you can imagine and a curious fact is also that people who program in JavaScript are less intelligent than people who don't program at all. JavaScript is NOT to be confused with an unrelated language called Java, which for some time used to be used on the web too but works very differently. JavaScript should also not be confused with ECMAScript, a language standard which JavaScript is based on but to which it adds yet more antifeatures, i.e. JavaScript is a dialect of ECMAScript (other similar ECMAScript-based languages are e.g. ActionScript and JScript). LRS stance towards this language is clear: as any other mainstream modern language **JavaScript is an absolutely unacceptable choice for any serious project**, though it may be used for quick experiments and ugly temporary programs as the language is high level, i.e. extremely easy, it doesn't require any ability to think, it works in every browser (so you get a kind of multiplatformness) and allows making things such as GUI and visualizations super quickly and easily. But remember that this kind of "comfort" always comes for a cost too high to pay.

**How bloated is JavaScript?** Very much. A MINIMALIST C implementation called QuickJS has around 80K lines of code -- compare e.g. to about 25K for tcc, a similar style implementation of C, and about 5K for comun. A more mainstream implementation of JavaScript, the v8 engine (used e.g. in node.js) has **over 1 million lines of code** of C++. { Checked with *cloc*. V8 also contains web assembly aside from JavaScript, but still you get the idea. ~drummyfish }

Number 1 rule of a good website is: **NEVER use JavaScript**. Website is not a program, website is a document, so it doesn't need any scripts. Privacy freaks hate web JavaScript because it's a huge security vulnerability (websites with JavaScript can spy easily on you -- yes, even if the script is "free software") -- we don't fancy security but JavaScript is still bloat and capitalist shit, it makes a website literally unusable in good browsers (those that don't implement JavaScript) so we hate it too. Basically everyone hates it.

In the past JavaScript was only a **client side** scripting language, i.e. it was used in web browsers (the clients) to make computations on the client computer (which suffices for many things but not all) -- as a browser language JavaScript interoperates with HTML and CSS, other two languages used on websites (which are however not programming languages). For server side computations PHP, a different language, was used, however later on (around 2010) a framework/environment called node.js appeared which allowed JavaScript to be used as a more general language and to be used for server side programming as well; as it's more comfortable to write everything in a single language, JavaScript started to replace PHP in many places, though PHP is still used to this day.

iQuery is a very popular library that's often used with JavaScript. It's kind of a universal library to do things one often wants to do. We mention it because everyone around JavaScript just supposes you'll be using it.

TODO: some more shit

## JavaScript Fun

Here let be recorded funny code in this glorious language.

**This kills the JavaScript:**

```
clear(this);
```

That's right, the above code just make JavaScript commit suicide by deleting its whole global thing.

{ Found here: <https://codegolf.stackexchange.com/questions/61115/make-your-language-unusable>.  
~drummyfish }

Here is how to **make your page work only with JavaScript turned off**:

```
<html>
<head>
<script>
function nuke() { document.body.innerHTML = "<p>disable JavaScript to view this page</p>"; }
```



```

</script>
</head>

<body onload="nuke()">
  <h1> My awesome page </h1>
  <p> My awesome page text :) </p>
</body>

</html>

```

{ NOTE: Remember that normally breaking compatibility on purpose is probably bad, it shouldn't be done seriously (don't waste effort on breaking something, rather make something nice, also censorship is always bad), but it's a nice troll :D Don't forget to have fun sometimes. ~drummyfish }

Or this will give an epileptic seizure to everyone who has Javascript on:

```

<html>
<head>
<script>
alert("Turn off Javascript to make it stop.");

var count = 0;

function go()
{
  count += 1;
  document.body.style.backgroundColor = (count % 2) ? "red" : "blue";
  document.body.style.color = (count % 2) ? "green" : "yellow";
  document.body.style["font-size"] = (count % 64).toString() + "px";
  setTimeout(go,10);
}
</script>
</head>

<body onload="go()">
  <h1> My awesome page </h1>
  <p> My awesome page text :) </p>
</body>

</html>

```

TODO: some JS nonsense like below

```

"11" + 1 // "111"
"11" - 1 // 10

```

---

jedi\_engine

## Jedi Engine

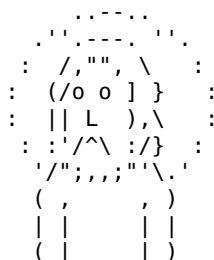
Jedi engine is an old (1995) proprietary/closed source "2.5D"/"Pseudo3D"/primitive3D game engine best known for being used in games Star Wars: Dark Forces and Outlaws. It was mostly notable for its at the time advanced 3D rendering that was similar to previously released Doom but yet a bit slightly improved, e.g. by adding features like room-above-room and fake looking up/down. There is a "modern" FOSS rewrite of the engine using "true 3D" and GPU accelerated rendering, called the *Force engine*; the original Jedi engine's source code isn't available so its internal working is not known exactly; some suspected the developers to have reverse engineer the Doom engine and copied how it worked, however work on the Jedi engine began before Doom was released and the Jedi engine seems to work on the principle of portal rendering (technique used in Duke Nukem's Build engine), unlike Doom which used BSP rendering. { This seems to be confirmed by the Force engine's dev on his blog and by one Russian video on YouTube. ~drummyfish }

---

jesus

# Jesus

Jesus Christ (also Jesus of Nazareth, about 4 BC to 33 AD) was a jewish carpenter preacher that was said to be the son of God, whose life along with supposed miracles he performed is described by the Bible (specifically its New Testament) and who is the center figure of Christianity, the world's largest religion; as such he is arguably the most famous of all men in history (probably followed by Hitler, kind of his opposite). Just one fact proving this claim is that we count our years more or less from his birth. He gained many followers as he preached that God has decided to change his laws a little and accept all "well behaved" people into his heavenly kingdom (that is not just jews as was previously the case, established by the God of Old Testament). For having stirred up a social disturbance by this preaching he was later crucified, as he himself predicted -- according to the Bible he sacrificed himself by this act to redeem the sins of all people, was resurrected after death and came up to the heaven to dwell by the God's side. Without subscribing to any mass religion or even having to believe in god, our LRS is highly aligned with much of the teaching of Jesus Christ, especially that of non violence, love of all people (even one's "enemies"), modesty, frugality and so forth.



"Artist"'s depiction of our Lord and Savior.

As perhaps the most influential man in history whose image has been twisted, used and abused over the centuries, we have to nowadays distinguish two separate characters:

- **Jesus of the Bible:** Jesus as described by the Bible, a book full of centuries worth of distortion, inaccuracies and purposeful religious propaganda. Here Jesus is to a great degree a fictional character, though based on a real man; he is the son of God (some even seeing him as actually the God himself somehow), a man without sin, born from a virgin, who performed countless miracles like healing the blind and even resurrecting dead, who spoke the word of God, was resurrected after death and is now overlooking us from the heaven.
- **historical Jesus:** The true man, as seen by history, that actually lived and was almost definitely just a mere mortal like any of us, even if others and perhaps even himself may have believed otherwise. Most historians agree Jesus lived (with some minority disagreeing), though of course they reject he would possess any supernatural powers -- these, they say, are things spawned as a legend, a propaganda of mass religion etc.

**Jesus was anarchist, pacifist, communist and explicitly rejected capitalism**, though stupid American capital and military worshippers tattoo shit such as "What would Jesus do?" on their asses, they somehow seem to masterfully apply selective blindness and completely ignore his quotes such as:

- "A camel will go through the eye of a needle before a rich man enters the kingdom of God." --Jesus
- "If someone wants to sue you and take your shirt, give him your coat also." --Jesus
- "If someone throws stone at you, throw back at him as well but with bread." --Jesus
- "I tell you: do not resist an evil man. If anyone slaps you on the cheek, turn to him the other cheek also." --Jesus
- "Thou shalt not kill." being directly in ten commandments lol (maybe they don't understand because the language is archaic, one may always find an excuse)
- ...

(Americans are stupid idiots who say they love Jesus but rather love to reference Old Testament for their pragmatics life decisions, however the law of Old Testament was explicitly cancelled by Jesus and updated to a new one, based on love and nonviolence rather than violence, punishment and revenge -- this is the whole point of why Jesus came to Earth in the first place. Old testament is basically the Jewish part of the Bible,

obsolete for Christians -- some Christians even completely reject Old testament, e.g. Cathars. It's why the book is called *New Testament*, it means "The New Law". But as it's been said, Americans are stupid.)

### fun facts about Jesus:

- **He had siblings** according to the Bible: brothers James, Joses, Simon, Jude and some sisters.
- **He was most likely crucified naked**, as was common practice to dishonor the crucified people. Covering his nudity in most depictions may be because of the effort to make it less obvious he was a  Jew , i.e. that his penis was circumcised.
- **He is acknowledged by other religions such as the Islam**, though in these he usually plays some minor role of just some mortal preacher.
- We do not sport any anti-white political correctness, however **the traditional depictions of his looks are likely wrong**, he most likely looked much different from the bearded, long-hair white man depictions we see in paintings -- these were likely affected by the Greek ideals of what gods look like. Jesus was a Jew, probably of darker skin like all people from the area he lived in, possibly without long hair as some of his followers mention in the Bible that it is inappropriate for a man to have long hair.
- There are some non-canonical gospels (not accepted to Bible) that talk some funny shit about Jesus, e.g. the Infancy Gospel of Thomas talks about how Jesus **as a child killed other children in revenge with his supernatural powers**.
- **Jesus is supposed to return** and judge the people: this is known as the Second Coming and is hinted on in the Bible, though the details on the date or even the nature of the event are unclear and interpreted differently. Before the second coming **a number of antichrists, or false prophets, are to appear**.
- ...

**Is Jesus God?** Or was he just his son? Or is God and Jesus the same? This seems to not actually be easy to answer, different people will tell you different things, some point to passages in Bible where they believe he literally says he is the God, others say the translation is not precise or even if it is that it doesn't matter (anyone can say really say he's a God) etcetc. The whole thing around holy trinity and so on is not easy to resolve objectively (some Muslims have even been entertained by this fact that Christians can't even get to agree on who their god is), but basically most Christians pray to Jesus, call him "our Lord and Savior" and generally treat him as if he is the same as God, so we can really see him that way.

## Life Of Jesus In Summary

This is a quick summary of life and death of Jesus Christ, mostly according to the gospels (which however sometimes disagree) but also taking into account the views of historians, let us see this as our "best guess".

{ There is a nice safe public domain book from 1898, digitized on [gutenberg.org](http://gutenberg.org), called *Bible Pictures and Stories in Large Print* -- it's a very nice, quite short retelling of the Bible, with nice pictures, like a Bible tl;dr. Check it out. ~drummyfish }

Jesus was **born** around 6 to 4 BC (this offset is cause by an error made in Middle Ages when they wrongly calculated his birth year, the error was only revealed once year counting had already been long established) in Bethlehem (in Israel) to Mary and Joseph, with Mary still being a virgin (which is a miracle claimed by Bible, denied by historians and even some churches); his father is said to be God, Mary was made pregnant by Holy Spirit. Jesus grew up in Nazareth and became a carpenter, as Joseph. Even as young he was very knowledgeable of the scripture.

An important event was the **baptism** (the ritual of "purification by water") of Jesus in Jordan river by John the Baptist around 28 AD. This is seen as a real historical event nowadays. John the Baptist was a preacher who foretold the coming of Jesus and some of his followers then started to follow Jesus.

After this Jesus went on to preaching, he chose **twelve apostles**, the closest followers to further preach his words to others; they were (there seems to be some differences between the gospels) Peter, Andrew, James, John, Philip, Bartholomew, Thomas, Matthew, James, Judas, Simon and Matthias. Jesus spoke in parables and made miracles like healing the blind, walking on water, feeding masses of people with only five loaves of bread and even resurrecting dead (Lazarus). His preaching was mainly about the love of God, coming to God's kingdom after death, forgiveness (people should forgive others and their own sins will in turn be forgiven by God) and loving other people, even one's enemies, he advocated nonviolence, modesty and

frugality.

As he arrived to Jerusalem, he was very famous and seen as a messiah by many, and as he broke many religious traditions, he began to upset the Jewish religious leaders, creating tension. He criticized practices of the leaders, e.g. commerce in temples. The Jews decided to arrest Jesus and lead him to Roman court.

Another important moment is **the last supper**, the final meal of Jesus which he shared with his apostles. After this one of the apostles, Judas, betrayed Jesus (which Jesus foretold) in Mount of Olives by leading the temple police to Jesus; Judas did this for the reward of 30 silver coins, later his conscience made him commit suicide.

The Jews have taken Jesus to Pontius Pilate, Roman governor, to be put on trial for the various offenses he committed. Pilate didn't find Jesus guilty but as the Jews pressured, he gave them a choice: he said he would either release Jesus or Barabbas, a murderer. The Jews chose Barabbas to be released, condemning Jesus to be crucified.

The **crucifixion** of Jesus is also seen as a true historical event now, which took place most likely in 30 or 33 AD. Before death he was tortured, whipped and was put on a crown of thorns (to mock that he was supposed to be the "king of Jews"). He was led to a hill in Golgotha, just outside Jerusalem, to where he had to carry his own cross, onto which he was then nailed and left to die, alongside two other criminals. According to some gospels he said various things on the cross, for example "father, forgive them for they don't know what they do", "father, why have you forsaken me?" and finally "it is finished".

His body was then buried in a tomb (which by historians is seen as unusual) of one of his followers. According to Bible, 3 days later he was **resurrected** (his tomb was found empty) and left the tomb, he went and visited some of his followers, and then, 40 days after the resurrection, ascended up to the heaven.

## See Also

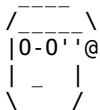
- [Christianity](#)
- [Muhammad](#)
- [Buddha](#)
- [Richard Stallman](#)
- [Gandhi](#)
- [Elvis](#)
- [Hittler](#)

---

john\_carmack

## John Carmack

John Carmack is a brilliant legendary programmer that's contributed mostly to [computer graphics](#) and stands behind engines of such [games](#) as [Doom](#), [Wolfenstein](#) and [Quake](#). He helped pioneer real-time 3D graphics, created many [hacks](#) and [algorithms](#) (e.g. the reverse shadow volume algorithm). He is also a rocket [engineer](#).



### ASCII art of John Carmack

He's kind of the ridiculously stereotypical [nerd](#) with glasses that just from the way he talks gives out the impression of someone with high functioning [autism](#). You can just sense his [IQ](#) is over 9000. Some nice [shit](#) about him can be read in the (sadly [proprietary](#)) book *Masters of Doom*.

Carmack is a proponent of FOSS and has released his old game engines as such which gave rise to an enormous amount of modifications, forked engines and even new games (e.g. Freedoom and Xonotic). He's probably leaning more towards the dark side of the source: the open-source. In 2021 Carmack tweeted that he would have rather licensed his old Id engines under a permissive BSD license than the GPL, which is good.

In 2013 he sadly sold his soul to Facebook to work on VR (in a Facebook owned company Oculus).

---

jokes

## Jokes

Here you can shitpost your jokes that are somehow related to this wiki's topic. Just watch out for copyright (no copy-pasting jokes from other sites)!

Please do NOT post lame "big-bang-theory"/9gag jokes like *sudo make sandwich* or *there are 10 types of people*.

Also remember the worst thing you can do to a joke is put a disclaimer on it. Never fucking do that.

{ Many of the jokes are original, some are shamelessly pulled from other sites and reworded. I don't believe copyright can apply if the expression of a joke is different, ideas can't be copyrighted. Also the exact origins of jokes are difficult to track so it's probably a kind of folklore. ~drummyfish }

{ I would like to thank one anonymous friend who contributed to me many ideas for jokes here :D I usually modified them slightly. ~drummyfish }

- C++
- What's the worst kind of lag? Gulag.
- Ingame chat: "What's the country in the middle of north Africa?" {BANNED}
- What sound does an angry C programmer make? ARGCCCC ARGVVVVVVVV
- I have a mentally ill friend who tried to design the worst operating system on purpose. It boots for at least 10 minutes, then it changes the user's desktop background to random ads and it randomly crashes to make the user angry. He recently told me he is getting sued by Microsoft for violating their look and feel.
- I am using a super minimal system, it only has one package installed on it. It is called systemd.
- Any Windows tutorial is really best called a "crash course".
- How do you know a project is truly suckless? The readme is longer than the program itself.
- How do you know a project is truly LRS? Its manifesto is longer than the program itself.
- How do you know a project is truly bloated? Instructions on how to build it are longer than whole specification of a suckless programming language.
- Do you use Emacs? No, I already have an operating system.
- Do you use Emacs? No, I already have a waifu.
- alias bitch=sudo
- What's a trilobite? 8 trilobits.
- "Never test for a bug that you don't know how to fix." --manager; "If we cannot fix it, it isn't broken." --also manager
- a joke for minimalists:
- When is Micro\$oft finally gonna make a product that doesn't suck???! Probably when they start manufacturing vacuum cleaners.
- Can free software lead to insanity? I don't know, but it can make you GNUts.
- Political activists walk into a bar. Pseudoleftist tells his friends: "hey guys, how about we have oppressive rulers and call them a government?" Capitalist says: "well no, let's have oppressive rulers and call them corporations". Liberal replies: "Why not both?". Monarchist goes: "no, it's all wrong, let's have oppressive rulers and call them Kings." To this pseudo communist says: "that's just shit, let's have oppressive rulers and call them the proletariat". Then anarcho pacifist turns to them and says: "Hmmm, how about we don't have any oppressive rulers?". They lynch him.
- There are a lot of jokes at <https://jcdverha.home.xs4all.nl/scijokes/>. Also <http://textfiles.com/humor/JOKES/>, <http://textfiles.com/humor/TAGLINES/quotes-1.txt> and so on. Also on wikiwikiweb under *CategoryJoke*, *ProgrammerLightBulbJokes* etc.

- Hello, is this anonymous pedophile help hotline? Yes. My 8yo daughter begs for sex, can we do penetration right away or should we start with anal?
- What do you call a woman that made a computer explode just by typing on it? Normal.
- Does the invisible hand exist in the free market? Maybe, but if so then all it's doing is masturbating (or giving us a middle finger).
- 90% of statistics are fake.
- When will they remove the touch and kill commands from Unix? Probably when they rename man pages to person pages.
- If law was viewed as a programming code, it would be historically the worst case of bloated spaghetti code littered with magic constants, undefined symbols and dead code, which is additionally deployed silently and without any testing. Yet it's the most important algorithm of our society.
- C++ is to C as brain cancer is to brain.
- Entropy is no longer what it used to be. Nostalgia too.
- At the beginning there was machine code. Then they added assembly on top of it to make it more comfortable. To make programs portable they created an operating system and a layer of syscalls. Except it didn't work because other people made other operating systems with different syscalls. So to try to make it portable again they created a high-level language compiler on top of it. To make it yet more comfortable they created yet a higher level language and made a transpiler to the lower level language. To make building more platform independent and comfortable they created makefiles on top of it. However, more jobs were needed so they created CMake on top of makefiles, just in case. It seems like CMake nowadays seems too low level so a new layer will be needed above all the meta-meta-meta build systems. I wonder how high of a tower we can make, maybe they're just trying to get a Guinness world record for the greatest bullshit sandwich in history.
- How to install a package on Debian? I don't know, but on my Arch it's done with pacman.
- green capitalism :D my sides
- Difference between a beginner and pro programmer? Pro programmer fails in a much more sophisticated manner.
- What is a computer? A device that can make a hundred million very precise mistakes per second.
- How many Python programmers do you need to change a lightbulb? Only one -- he holds the bulb while the world revolves around him.
- After all it may not take so long to establish our utopia. By the time Windows has updated we will have already done it ten times over.
- One of the great milestones yet left to be achieved by science is to find intelligent life in our Solar System.
- An evil capitalist, good capitalist and female genius walk in the park. A bee stings one of them. Who did it sting? The evil capitalists, the other two don't exist.
- Cool statistics: 9 out of 10 people enjoy a gang rape.
- Basement hackers never die, they just smell that way. Musicians never die, they just decompose (and musicians working part time are semiconductors).
- `int randomInt(void) { int x; return x; }`
- Boss: "We're going to need to store additional information about gender of all 1600 people in our database." Me: "OK that's only 200 extra bytes.". Diversity department: "You're fired."
- the downto operator
- Schizophrenia beats being alone.
- Our new app partly adopts the KISS philosophy, specifically the "stupid" part.
- I just had sex with a German chick, for some reason she kept yelling her age. (Or maybe she just didn't consent.)
- I find it much more pleasant to browse the web on a 1 bit display, it can't display a rainbow.
- What's long and sticky? A stick.
- The term *military intelligence* is an oxymoron. The term *criminal lawyer* is a redundancy.
- Why are noobs the most pacifist beings in existence? Because they never beat anyone.
- What does short circuited capacitor and gratis software have in common? They are free of charge.
- You scratch my tape, I scratch yours.
- There's a new version of Debian Bull's Eye that's compiled exclusively with Rust. Its code name is Bull's Shit.
- Manager is that who thinks 9 women can produce a child in 1 month.
- Have you heard the atheists are starting their own non-prophet?
- Those who can, do. Those who cannot, teach. Those who cannot teach, do business.
- An Apple a day keeps sanity away.

- The goal of computer science is to create things that will last at least until we're finished building them.
- The new version of Windows is going to be backdoor free! The backdoor will be free of charge.

## See Also

- LMAO
- fun

---

julia\_set

## Julia Set

TODO



## Code

The following code is a simple C program that renders given Julia set into terminal (for demonstrative purposes, it isn't efficient or do any antialiasing).

```
#include <stdio.h>

#define ROWS 30
#define COLS 70
#define SET_X -0.36 // Julia set parameter
#define SET_Y -0.62 // Julia set parameter
#define FROM_X -1.5
#define FROM_Y 1.0
#define STEP (3.0 / ((double) COLS))

unsigned int julia(double x, double y)
{
    double cx = x, cy = y, tmp;

    for (int i = 0; i < 1000; ++i)
    {
        tmp = cx * cx - cy * cy + SET_X;
        cy = 2 * cx * cy + SET_Y;
        cx = tmp;
    }
}
```

```

    if (cx * cx + cy * cy > 10000000000)
        return 0;
    }

    return 1;
}

int main(void)
{
    double cx, cy = FROM_Y;

    for (int y = 0; y < ROWS; ++y)
    {
        cx = FROM_X;

        for (int x = 0; x < COLS; ++x)
        {
            unsigned int point =
                julia(cx,cy) + (julia(cx,cy + STEP) * 2);

            putchar(point == 3 ? ':' : (point == 2 ? '\\' :
                (point == 1 ? '.' : ' ')));

            cx += STEP;
        }

        putchar('\n');

        cy -= 2 * STEP;
    }

    return 0;
}

```

---

justice

## Justice

Justice is an euphemism for revenge. It has also been defined as "decision in your favor".

---

just\_werks

## Just Werks

"Just works" (for "just works" if that's somehow not clear) is a phrase usually used by noobs to justify using a piece of technology while completely neglecting any other deeper and/or long term consequences, though the argument has legitimate uses as well. A noob doesn't think about technology further than how it can immediately perform some task for him, to him "just works" is a mere rationalization that gives him the comfort needed to not think things through.

"Just works" can be used legitimately to express that something simply works, e.g. thanks to being simple, for example "PDFs suck, plaintext just works". This use of the term is acceptable.

This phrase is widely used e.g. on 4chan/g, however mostly in the wrong way. It possibly originated there.

The ignorant "just works" philosophy completely ignores questions such as:

- **Is there anything better in the long run?** A normie will always prefer a shitty software he can immediately use to a software that would take one day to learn and that would make the task many times easier, comfortable, cheaper etc.
- **Is this affecting my freedom (and things like "security" etc.)?** A normie doesn't realize that by using proprietary or bloated program will limit the number of people who can maintain, fix and improve his software, and that technology is used to abused him, e.g. by spying on him, making him depend on something unnecessary etc.



- **How is this affecting my computing in a wider sense?** A normie won't even think about such thing as that using some proprietary format will likely immediately close the door to working with it with a FOSS program, or that installing a specific OS will limit what programs he can run.
- **Am I becoming a slave to this technology?** By adopting something proprietary and/or bloated I am slowly becoming dependent on an ecosystem that's completely under control of some corporation, an ecosystem that can quickly change for the worse or even disappear completely.
- **Am I supporting evil?** E.g. by paying to a corporation, letting someone collect data in the background, promoting a bad piece of technology etc.
- **Am I hurting better alternatives by not using them?** E.g. by using a proprietary social network gives one more user to it and one fewer to a potentially more ethical free social network.
- **Is there anything just plain better?** A normie will take the first thing that's handed to him and "just works" without even checking if something better exists that would satisfy him better.

## See Also

- [everyone does it](#)
- [just doing my job](#)

---

kek

## Kek

Kek means [lol](#). It comes from [World of Warcraft](#) where the two opposing factions (Horde and Alliance) were made to speak mutually unintelligible languages so as to prevent enemy players from communicating; when someone from Horde typed "lol", an Alliance player would see him say "kek". The other way around (i.e. Alliance speaking to Horde) would render "lol" as "bur", however kek became the popular one. On the Internet this further mutated to forms like *kik*, *kek*, *topkek* etc. Nowadays in some places such as [4chan](#) kek seems to be used even more than lol, it's the newer, "cooler" way of saying lol.

## See Also

- [meme](#)

---

kids\_these\_days

## Kids These Days

TODO

---

kiss

## KISS

See also [minimalism](#).

KISS (Keep It Simple, Stupid!) is a [minimalist](#) design philosophy that favors simplicity, both internal and external, [technology](#) that is **as simple as possible** to achieve given task. This philosophy doesn't primarily stem from [laziness](#) or a desire to rush something (though these are completely valid reasons too), but mainly from the fact that higher [complexity](#) comes with increasingly negative effects such as the cost of development, cost of [maintenance](#), greater probability of [bugs](#) and failure, more [dependencies](#) etc.

WATCH OUT: many have started to ride on the wave of the "KISS" trend and abuse the term, twisting its true meaning; for example GNU/Linux Mint has started to market itself as "KISS" -- that's of course ridiculous and all Mint developers are cretins and idiots. **Maximum INTERNAL simplicity is a necessary prerequisite for the KISS philosophy**, anything that's just simple on the outside is a mere harmful [pseudominimalism](#) -- you may as well use a [Mac](#).

Under dystopian capitalism simple technology, such as simple software, has at least one more advantage related to "intellectual property": a simple solution is less likely to step on a patent landmine because such a simple solution will either be hard to patent or as more obvious will have been discovered and patented sooner and the patent is more likely to already be expired. So in this sense KISS technology is legally safer.

Apparently the term *KISS* originated in the US Army plane engineering: the planes needed to be repairable by *stupid* soldiers with limited tools under field conditions.

**Examples** of KISS "solutions" include:

- Using a plain text file instead of a dedicated bug tracker (TODO.txt), note taking program etc.
- Creating website in plain HTML instead of using some complex web framework such as Wordpress.
- Using solar panels directly, without a battery, if it's good enough.
- Implementing a web left-right sweeping image gallery with HTML iframe instead of some overcomplicated JavaScript library. { Example stolen from reactionary software website. ~drummyfish }
- Supporting only ASCII instead of Unicode, it is good enough.
- Using a plain text flat file instead of a database system.
- Using markdown for creating documents, as opposed to using office programs such as Libreoffice.
- Using a trivial shell script for compiling your programs rather than a complex build system such as CMake.
- Using ASCII art instead of bitmap images.
- In the world of coffee so called *Turkish coffee* without any milk and sugar is probably the most KISS option, you just put coffee ground in a mug and pour hot water on it. Compare this to Espresso with milk and sugar which needs a quite complex and expensive machine. { Yeah, I drink the most KISS coffee, though sometimes I also use French press -- that one is still quite simple. ~drummyfish }
- ...

Compared to suckless, Unix philosophy and LRS, KISS is a more general term and isn't tied to any specific group or movement, it doesn't imply any specifics but rather the general overall idea of simplicity being an advantage (less is more, worse is better, ...).

KISS Linux is an example of software developed under this philosophy and adapting the term itself.

## See Also

- minimalism
- suckless
- KILL
- primitivism
- LRS
- KISP

---

kiwifarms

## Kiwifarms

Kiwifarms is a website accessible at <https://kiwifarms.net> whose address is highly censored, known for mostly harmless funmaking of online people (usually autists and schizos, see e.g. Sonichu). It's a center of a lot of "anti cyber bullying" hysteria and drama. We just want to leave the link to the site here as it's being censored everywhere.

---

kwangmyong

## Kwangmyong

Kwangmyong (meaning *bright light*) is a mysterious intranet that North Koreans basically have instead of the Internet. For its high political isolation North Korea doesn't allow its citizens open access to the Internet, they

rather create their own internal network the government can fully control -- this is unsurprising, allegedly it is e.g. illegal to own a fax and North Korea also have their own operating system called Red Star OS, for security reasons. Not so much is known about Kwangmyong for a number of reasons: it is only accessible from within North Korea, foreigners are typically not allowed to access it, and, of course, it isn't in English but in Korean. Of course the content on the network is highly filtered and/or created by the state propaganda. Foreigners sometimes get a chance to spot or even secretly photograph things that allow us to make out a bit of information about the network.

North Koreans themselves almost never have their own computers, they typically browse the network in libraries.

There seem to be a few thousand accessible sites. Raw IP addresses (in the private 10.0.0.0/8 range) are sometimes used to access sites (posters in libraries list IPs of some sites) but DNS is also up -- here sites use .kp top level domain. Some sites, e.g. of universities, are also accessible on the Internet (e.g. <http://www.ryongnamsan.edu.kp/>), others like <http://www.ipa.aca.kp> (patent/invention site) or <http://www.ssl.edu.kp> (sports site) are not. There seems to be a remote webcam education system in place -- it appeared on North Korean news. There exists something akin a search engine (*Naenara*), email, usenet, even something like facebook. Apparently there are some video games as well.

## See Also

- Red Star OS (North Korea operating system)
- sneakernet

---

lambda\_calculus

## Lambda Calculus

Lambda calculus is an extremely simple and low-level mathematical system that can describe computations with functions, and can in fact be used to describe and perform any computation. Lambda calculus provides a theoretical basis for functional programming languages and is a **model of computation** similar to e.g. a Turing machine or interaction nets -- lambda calculus has actually exactly the same computational power as a Turing machine, which is the greatest possible computational power, and so it is an alternative to it. Lambda calculus can also be seen as a simple programming language, however it is so extremely simple (there are e.g. no numbers) that its pure form isn't used for practical programming, it is more of a mathematical tool for studying computers theoretically, constructing proofs etc. Nevertheless anything that can be programmed in any classic programming language can in theory be also programmed in lambda calculus.

While Turing machines use memory cells in which computations are performed -- which is similar to how real life computers work -- lambda calculus performs computations only by simplifying an expression made of pure mathematical functions, i.e. there are no global variables or side effects (the concept of memory is basically present in the expression itself, the lambda expression is both a program and memory at the same time). It has to be stressed that the functions in question are mathematical functions, also called **pure functions**, NOT functions we know from programming (which can do all kinds of nasty stuff). A pure function cannot have any side effects such as changing global state and its result also cannot depend on any global state or randomness, the only thing a pure function can do is return a value, and this value has to always be the same if the arguments to the function are same.

## How It Works

(For simplicity we'll use pure ASCII text. Let the letters L, A and B signify the Greek letters lambda, alpha and beta.)

Lambda calculus is extremely simple in its definition, but it may not be so simple to learn to understand it. Most students don't get it the first time, so don't worry :)

In lambda calculus function have no names, they are what we'd call anonymous functions or lambdas in programming (now you know why they're called lambdas).

Computations in lambda calculus don't work with numbers but with sequences of symbols, i.e. the computation can be imagined as manipulating text strings with operations that can intuitively just be seen as "search/replace". If you know some programming language already, the notation of lambda calculus will seem familiar to functions you already know from programming (there are functions, their bodies, arguments, variables, ...), but BEWARE, this will also confuse you; functions in lambda calculus work a little different (much simpler) than those in traditional programming languages; e.g. you shouldn't imagine that variables and function arguments represent numbers -- they are really just "text symbols", all we're doing with lambda calculus is really manipulating text with very simple rules. Things like numbers, their addition etc. don't exist at the basic level of lambda calculus, they have to be implemented (see later). This is on purpose (feature, not a bug), lambda calculus is really trying to explore how simple we can make a system to still keep it as powerful as a Turing machine.

In lambda calculus an expression, also a **lambda term** or "program" if you will, consists only of three types of syntactical constructs:

1.  $x$ : **variables**, represent unknown values (of course we can use also other letters than just  $x$ ).
2.  $(\lambda x.T)$ : **abstraction**, where  $T$  is a lambda term, signifies a function definition ( $x$  is a variable that's the function's parameter,  $T$  is its body).
3.  $(S\ T)$ : **application** of  $S$  to  $T$ , where  $S$  and  $T$  are lambda terms, signifies a function call/invocation ( $S$  is the function,  $T$  is the argument).

For example  $(\lambda a.(\lambda b.x))$   $x$  is a lambda term while  $x\lambda x..y$  is not.

Brackets can be left out if there's no ambiguity. Furthermore we need to distinguish between two types of variables:

- **bound**: A variable whose name is the same as some parameter of a function this variable is in. E.g. in  $(\lambda x.(\lambda y.xyz))$  variables  $x$  and  $y$  are bound.
- **free**: Variable that's not bound.

Every lambda term can be broken down into the above defined three constructs. The actual computation is performed by simplifying the term with special rules until we get the result (similarly to how we simplify expression with special rules in algebra). This simplification is called a **reduction**, and there are only two rules for performing it:

1. **A-conversion**: Renames (substitutes) a bound variable inside a function, e.g. we can apply A-conversion to  $\lambda x.xa$  and convert it to  $\lambda y.ya$ . This is done in specific cases when we need to prevent a substitution from making a free variable into a bound one.
2. **B-reduction**: Takes a body of a function and replaces a parameter inside this body with provided argument, i.e. this is used to reduce *applications*. For example  $(\lambda x.xy)$   $a$  is an application (we apply  $(\lambda x.xy)$  to  $a$ ). When we apply B-reduction, we take the function body ( $xy$ ) and replace the bound variable ( $x$ ) with the argument ( $a$ ), so we get  $ay$  as the result of the whole B-reduction here.

A function in lambda calculus can only take one argument. The result of the function, its "return value", is a "string" it leaves behind after it's been processed with the reduction rules. This means a function can also return a function (and a function can be an argument to another function), which allows us to implement functions of multiple variables with so called currying.

For example if we want to make a function of two arguments, we instead create a function of one argument that will return another function of one argument. E.g. a function we'd traditionally write as  $f(x,y,z) = xyz$  can in lambda calculus be written as  $(\lambda x.(\lambda y.(\lambda z.xyz)))$ , or, without brackets,  $\lambda x.\lambda y.\lambda z.xyz$  which will sometimes be written as  $Lxyz.xyz$  (this is just a syntactic sugar).

**This is all we need to implement any possible program.** For example we can encode numbers with so called Church numerals: 0 is  $Lf.Lx.x$ , 1 is  $Lf.Lx.fx$ , 2 is  $Lf.Lx.f(fx)$ , 3 is  $Lf.Lx.f(f(fx))$  etc. Then we can implement functions such as an increment:  $Ln.Lf.Lx.f((nf)x)$ , etc.

Let's take a complete **example**. We'll use the above shown increment function to increment the number 0 so that we get a result 1:

|                                                          |                                 |
|----------------------------------------------------------|---------------------------------|
| $(\text{Ln.Lf.Lx.f}((\text{nf})x) \text{ (Lf.Lx.x)})$    | application                     |
| $(\text{Ln.Lf.Lx.f}((\text{nf})x) \text{ (Lf0.Lx0.x0)})$ | A-conversion (rename variables) |
| $(\text{Lf.Lx.f}(((\text{Lf0.Lx0.x0})f)x))$              | B-reduction (substitution)      |
| $(\text{Lf.Lx.f}((\text{Lx0.x0})x))$                     | B-reduction                     |
| $(\text{Lf.Lx.fx})$                                      | B-reduction                     |

We see we've gotten the representation of number 1.

TODO: C code

## See Also

- sigma calculus (for OOP)
- interaction calculus

langtons\_ant

## Langton's Ant

Langton's ant (also *virtual ant* or *vant*) is a simple zero player game and cellular automaton simulating the behavior of an ant that behaves according to extremely simple rules but nevertheless builds a very complex structure. It is similar to game of life. Langton's ant is **Turing complete** (it can be used to perform any computation that any other computer can).

**Rules:** in the basic version the ant is placed in a square grid where each square can be either white or black. Initially all squares are white. The ant can face north, west, south or east and operates in steps. In each step it does the following: if the square the ant is on is white (black), it turns the square to black (white), turns 90 degrees to the right (left) and moves one square forward.

These simple rules produce a quite complex structure, seen below. The interesting thing is that initially the ant behaves **chaotically** but after about 10000 steps it suddenly ends up behaving in an ordered manner by building a "highway" that's a non-chaotic, repeating pattern. From then on it continues building the highway until the end of time.

[illegible]

*Langton's ant after 11100 steps, A signifies the ant's position, note the chaotic region from which the highway emerges left and up.*

- **multiple colors:** Squares can have more colors than just black/white that are cycled by the ant. Here we also need to specify which way the ant turns for each color it steps on, for example for 4 colors we may specify the rules as LRLl (turn left on 1st color, right on 2nd color etc.).
- **multiple ants:** called colonies
- **different grid:** e.g. hexagonal or 3D
- **multiple ant states:** Besides having a direction the ant can have a more complex state. Such ants are called **turmites** (Turing termite).

## Implementation

```
#include <stdio.h>
#include <unistd.h>

#define FIELD_SIZE 48
#define STEPS 5000
#define COLORS 2      // number of colors
#define RULES 0x01    // bit map of the rules, this one is RL

unsigned char field[FIELD_SIZE * FIELD_SIZE];

struct
{
    int x;
    int y;
    char direction; // 0: up, 1: right, 2: down, 3: left
} ant;

int wrap(int x, int max)
{
    return (x < 0) ? (max - 1) : ((x >= max) ? 0 : x);
}

int main(void)
{
    ant.x = FIELD_SIZE / 2;
```

```

ant.y = FIELD_SIZE / 2;
ant.direction = 0;

for (unsigned int step = 0; step < STEPS; ++step)
{
    unsigned int fieldIndex = ant.y * FIELD_SIZE + ant.x;
    unsigned char color = field[fieldIndex];

    ant.direction = wrap(ant.direction + (((RULES >> color) & 0x01) ? 1 : -1),4);

    field[fieldIndex] = (color + 1) % COLORS; // change color

    // move forward:

    switch (ant.direction)
    {
        case 0: ant.y++; break; // up
        case 1: ant.x++; break; // right
        case 2: ant.y--; break; // down
        case 3: ant.x--; break; // left
        default: break;
    }

    ant.x = wrap(ant.x, FIELD_SIZE);
    ant.y = wrap(ant.y, FIELD_SIZE);

    // draw:

    for (int i = 0; i < 10; ++i)
        putchar('\n');

    for (int y = 0; y < FIELD_SIZE; ++y)
    {
        for (int x = 0; x < FIELD_SIZE; ++x)
            if (x == ant.x && y == ant.y)
                putchar('A');
            else
            {
                unsigned char val = field[y * FIELD_SIZE + x];
                putchar(val ? ('A' + val - 1) : '.');
            }

        putchar('\n');
    }

    usleep(10000);
}

return 0;
}

```

## See Also

- [Resnick's termite](#)
- [game of life](#)
- [turmite](#)
- [rule 110](#)
- [cellular automaton](#)
- [turtle graphics](#)

---

leading\_the\_pig\_to\_the\_slaughterhouse

## Leading The Pig To The Slaughterhouse

"Move forward, don't look back, forward is good!" --slaughterer to the pig, also [capitalist](#) to customer

TODO

The goal is to get someone to where he is defenseless so that he can start to be fully abused, but how to do this? When the victim sees the place, he won't want to move there. You do it by small steps:

1. Make the pig (customer) move a small step towards the slaughterhouse (technology dystopia), e.g. by offering it food (comfort, advanced features, more back cameras, discount price, faster porn download, ...).
2. As it moves, silently close the door (deprecate old technology) behind the pig so that it has no way to return. If the pig starts looking back just laugh at it: "Haha, why are you backwards? That place there is obsolete and out of fashion now. Forward lies a better place! Update! Only move forward, never stop! Look, others are doing the same."
3. Repeat steps 1 and 2 until you get to the slaughterhouse (technology dystopia).
4. Congratulations, you are now in the slaughterhouse (technology dystopia), you can do whatever you want with the pig (customer) without it having any option to back up. Want to spy on the pig 24/7? No problem, the pig has a device it can't live without that's fully under your control. Want to take 90% of his month's pay? No problem, he is depending on software that's not even physically installed on his computer which you can shut down at any second. Maybe it even has an extremely smart proprietary pacemaker in its heart so you can just kill him if he doesn't pay. Good thing is also that the pig doesn't believe this can happen because it relies on someone "protecting" it (a government that just NEVER ever sleeps because it's so SO much concerned for the wellbeing of its people, or maybe a group of 10 14 year old girls that call themselves animal right activists that will at last second appear like Marvel superheroes and save all the pigs like in the Pixar movie yeah!).

## See Also

- slowly boiling the frog

---

left

## Left

See left vs right.

---

left\_right

## Left Vs Right (Vs Pseudoleft)

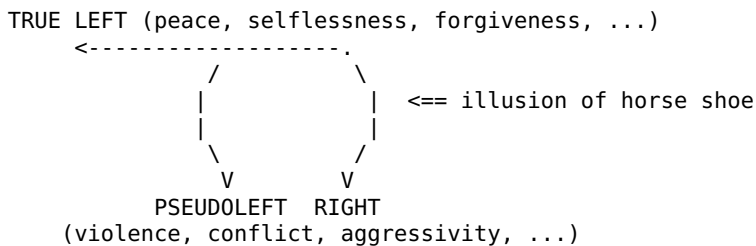
Left and right are two basic opposing political sides that roughly come down to the pro-equality (left) and pro-hierarchy (right). Historically the terms *left* and *right* came from the opposing sides at which members of national assembly physically sit during 1789 French revolution, however since then they evolved into possessing more generalized meanings of simply anti and pro hierarchy. Unfortunately there is a lot of confusion and vagueness about these terms nowadays, so let us now define them as used on this wiki:

- The (true) **left is pro social equality**, i.e. against social hierarchies. This includes equality of all living beings, period. Note that social equality does NOT imply people being made (or being made to appear) equal in other ways, e.g. physically -- true left accepts difference between people and races and doesn't hide them. Even if the perfectly ideally leftist society can't be completely achieved, true left tries to get **as close to it as possible**. The values of true left are for example sharing, love, selflessness, altruism, forgiveness and nonviolence. Groups and movements that are at least highly truly leftist include anarcho pacifism, veganism, free software, free culture and of course LRS.
- The **right is pro social hierarchy**, i.e. against social equality. This means some people standing above others, be it by strength, power, wealth, social status, privileges etc. The rightist values are mostly those associated with evil, i.e. violence, oppression, conflict, war, revenge, survival of the fittest etc. Among rightism can be included fascism (i.e. extreme rightism), capitalism, US republican party, states, the military etc. One of right's identifying features is **hypocrisy**, i.e. it judges what's good/bad only by against whom it is targeted, e.g. violence is bad when targeted against "us" ("those Muslims are bad, they want to kill us!") but good when targeted against "them" ("we have to kill those Muslims because they're violent!"); so animals killing humans is judged as "bad" but humans killing animals is "good". In other words right has no sense of morality, only the sense of self interest.



- The **pseudoleft** is pretending to be left while in fact being right due to e.g. using non-leftist means (such as violence) or even having non-leftist goals (e.g. benefit of specific minority as opposed to benefit of everyone). Among pseudoleftist movements are feminism, LGBT, Antifa or Marxism. This fact is also supported by the naming of these movements.

There exists a "theory" called a horse shoe. It says that the extremes of the left-right spectrum tend to be alike (violent, hating, radical), just as the two ends of a horse shoe. This is only an illusion caused by ignoring the existence of pseudoleft. The following diagram shows the true situation:



We see pseudoleft is something that began as going away from the right but slowly turned around back to its values, just from a slightly different direction. This is because rightism is very easy, it offers tempting short-term solutions such as violence, and so it exerts a kind of magnetic force on every human -- most cannot resist and only very few get to the true left despite this force.

The current US-centered culture unfortunately forces a **right-pseudoleft false dichotomy**. It is extremely important to realize this dichotomy doesn't hold. Do not become type A/B fail.

What's called *left* in the modern western culture usually means *pseudoleft*. The existence of pseudoleftism is often overlooked or unknown. It used to be found mainly in the US, however globalization spreads this cancer all over the world. Pseudoleft justifies its actions with a goal that may seem truly leftist, such as "equality", but uses means completely unacceptable by true left (which are in fact incompatible with equality), such as violence, bullying, lynching, cancelling, ensorship or brainwashing. Pseudoleft is aggressive. It believes that **"ends justify the means"** and that **"it's fine to bully a bully"** ("eye for an eye"). A pseudoleftist movement naturally evolves towards shifting its goals from a leftist one such as equality towards a fascist one such as a (blind) *fight for some group's rights* (even if that group may already have achieved equality and more).

The difference between left and pseudoleft can be shown in many ways; one of them may be that pseudoleft always wants to **fight** something, usually the right (as they're essentially the same, i.e. natural competitors). True left wants to end all fights. Pseudoleft invents bullshit artificial issues such as political correctness that sparks conflict, as it lives by conflict. Left tries to find peace by solving problems. Pseudoleft sees it as acceptable to do bad things to people who committed something it deems bad. True left knows that violence creates violence, it "turns the other cheek", it cures hate with love.

Pseudoleft is extra harmful by deceiving the public into thinking what it does really is leftist. Most normal people that don't think too much therefore stand between a choice of a lesser of two evils: the right and pseudoleft. True left, the true good, is not known, it is overshadowed. Let us now compare a few existing movements/ideologies/groups:

|                      | <u>LGBT</u>         | <u>Feminism</u>     | <u>Antifa</u>      | <u>Nazism</u>   | <u>MGTOW</u>   | <u>BLM</u>          | <u>Marxism</u> |
|----------------------|---------------------|---------------------|--------------------|-----------------|----------------|---------------------|----------------|
| class                | pseudoleft          | pseudoleft          | pseudoleft         | right           | right?         | pseudoleft          | pseudoleft     |
| fights for           | gay/bi/etc.         | women               | antifa, pseudoleft | own race/nation | men            | black               | proletariat    |
| fights against       | straight, anti-LGBT | men, anti-Feminists | right, anti-Antifa | Jews, other r/n | women          | non-black, anti-BLM | other classes  |
| official motive      | "social justice"    | "social justice"    | "self defense"     | "self defense"  | "self defense" | "self. def.", ...   | superiority    |
| bullying/violence?   | yes                 | yes                 | yes                | yes             | yes            | yes                 | yes            |
| fanaticism/hysteria? | yes                 | yes                 | yes                | yes             | probably       | yes                 | yes            |

|                              |     |     |     |     |          |          |     |
|------------------------------|-----|-----|-----|-----|----------|----------|-----|
| propaganda/brainw./censor.?  | yes | yes | yes | yes | probably | yes      | yes |
| cults of personality/heroes? | yes | yes | yes | yes | yes      | yes      | yes |
| ends justify the means?      | yes | yes | yes | yes | probably | probably | yes |

Why is there no pseudoright? Because it doesn't make sense :) Left is good, right is a sincere evil and pseudoleft is an evil pretending to be good. A good pretending to be evil doesn't probably exist in any significant form.

**Centrism** means trying to stay somewhere mid way between left and right, but it comes with issues. From our point of view it's like trying to stay in the middle of good and evil, it is definitely pretty bad to decide to be 50% evil. Another issue with centrism is that it is **unstable**. Centrism means balancing on the edge of two opposing forces and people naturally tend to slip towards the extremes, so a young centrist will have about equal probabilities of slipping either towards extreme left or extreme right, and as society polarizes this way, people become yet more and more inclined to defend their team. Knowing centrism is unsustainable, we realize we basically have to choose which extreme to join, and we choose the left extreme, i.e. joining the good rather than the evil.

{ I came to the realization that rightists are actually much more bearable than pseudoleftists -- it makes sense though, pseudoleft is like right, fascist, but with extra evil of added pretence and sneakiness. While rightists are at their core evil, they are actually many times tolerant as they often value free speech, so you can coexist with a rightist, he will tell you your opinions are stupid but he will let you share them, he is like that one friend who you disagree with but still can have a talk with. Yes, that's right, **a rightist is more tolerant than pseudoleftist**. I do by no means defend rightism, it is pure evil, but the fact that it seems bearable compared to pseudoleft says something about pseudoleft -- pseudoleft is just pure psychopathy. A rightist sometimes at least holds some values, for example he will respect your courage to stand behind an opinion you hold even if he hates it, a pseudoleftist has none of that, an SJW just has no emotion except for hate and rage, he has that soulless robotic stare with the word "EXTERMINATE" flashing before his eyes. You mention a trigger word and he bans you on all servers, you'll be lucky if he doesn't call for your public lynching. ~drummyfish }

## See Also

- SJW

---

less\_retarded\_hardware

## Less Retarded Hardware

Less retarded hardware (LRH) is an extension of less retarded software (LRS) principles to hardware design. Such hardware has to be non-consumerist, designed to last and free (as in freedom) hardware completely from the lowest level, preferably completely public domain without any legal limitations, made with selfless goals, aiming to be good technology that helps all living beings without abusing them -- this implies the hardware has to be as simple as possible (KISS, suckless, ...) so as to maximize the number of people who can understand it, utilize it, improve it and repair it. An example of hardware coming close to this may potentially be e.g. Ronja.

---

less\_retarded\_society

## Less Retarded Society

Less retarded society (LRS, same acronym as less retarded software) is a model of ideal society towards which we, the LRS, want to be moving. Less retarded society is a peaceful, collaborative society based on love of all life, which aims for maximum well being of all living beings, a society without violence, money, oppression, need for work, social competition, poverty, scarcity, criminality, censorship, self-interest, government, police, laws, bullshit, slavery and many other negative phenomena. It equally values all living beings and establishes true social equality in which everyone can pursue his true desires freely -- it is a

TRULY leftist society, NOT a pseudoleftist one. The society works similarly to that described by the Venus Project and various anarchist theories (especially anarcho pacifist communism), but it also takes good things from elsewhere, even various religions (without itself actually becoming a religion in traditional sense); for example parts of teaching of Jesus and Buddha.

**How is this different from other ideologies and "life philosophies"?** Well, one principal difference is that LRS doesn't want to fight, in fact LRS adopts a **pessimistic, defeatist mindset** to not compromise morality by any desire to "win the game", behaving correctly is more important than achieving a goal, ends do NOT justify the means; nowadays as well as in the past society has always been about conflict, playing a **game** against others (nowadays e.g. market competition, employment competition, media competition, ...) in which some win, some can manage and some lose. Most political parties nowadays just want to change the rules of the game or downright switch to a different kind of game, some want to make the rules "more fair", or to make it favor their represented minority (so called fascism), some just want to hack the game, some want to cheat to win the game easily, some want to play fair but still win (i.e. become "successful"). LRS simply sees any kind of such game as unnecessary, cruel, unethical and harmful in many ways not just to us, but to the whole planet. LRS therefore simply wants to stop the game, not by force but by making everyone see how bad the game is. It says that **competition and conflict must cease to be the basis of society**. There is no value in achieving anything by violence, such a change will soon be reverted by counter revolution, people themselves have to understand what's good and choose it voluntarily. That is one of the reasons why we are pacifists and **reject all violence**, only wanting to promote our ideas by education. We accept we may not achieve our goals and we most certainly won't achieve anything during our lifetime and that gives us the freedom to behave truly morally. We try to **never be proud** of anything, as pride leads to violence and fascism. We are also in many ways aligned with the ideals of cynicism.

Note that less retarded society is an ideal model, i.e. it can probably not be achieved 100% but it's something that gives us a direction and to which we can **get very close** with enough effort. We create an ideal theoretical model and then try to approximate it in reality, which is a scientific approach that is utilized almost everywhere: for example mathematics defines a perfect sphere and such a model is then useful in practice even if we cannot ever create a mathematically perfect sphere in the real physical world -- the mathematical equations of a sphere guide us so that with enough effort we are able to create physical spheres that are pretty close to an ideal sphere. The same can be done with society. This largely refutes the often given argument that *"it's impossible to achieve so we shouldn't try at all"* -- we should try our best and the closer to the ideal we get, the better for us.

## Basis: Love Of All Life

When thinking about how to change society for the better, the first thing that needs to be done is defining a goal which the society should aim for -- an axiom which serves as a measure of what's objectively good and bad, which in turn helps us take the right steps towards the good. This is only logical, without a goal we aren't really trying to achieve anything and "good" and "evil" are just words without any objective meaning.

The basis of less retarded society is a **universal and unconditional selfless altruistic love of all life** (life at large, i.e. not just people, but also animals and other life forms and life as a concept); the kind of love a glimpse of which you catch when you for example observe an animal play and be happy of just existing as a living being that's able to feel joy and happiness. This kind of love and the strong emotion associated with it is to us possibly the greatest miracle of the universe and so we choose to support it, make it flourish, we define it as an axiom that life which experiences joy and happiness is good. Similarly we define it as bad when life feels suffering or when there is little or even no life in the universe. **Here we set the goal for our society to support life, make it flourish, and make individual living beings feel happiness.**

We purposefully make this goal a little bit vague, we avoid specifying our basic goal with exact mathematical metrics because defining maximization of any such measure as a goal leads to undesired results (as for example in capitalism setting the goal to maximizing capital leads to maximizing it on the detriment of all other values such as well being of people). This is known as the Goodhart's law: "when a metric becomes a goal, it stops being a good metric".

**What does love of all life mean exactly?** As hinted above, it does **NOT** necessarily mean maximizing specific measures such as abundance of life (which could lead to overpopulation and in turn to suffering), the sum of happiness of all life (which could lead to just dosing everyone with drugs or killing unhappy individuals), elimination of negative emotion such as hatred (which would likely prevent us from recognizing

wrong directions of our society), it doesn't mean respect towards everyone etc. It doesn't even mean that we will never kill anyone on purpose, our society may support euthanasia without violating its principles. Love of all life mostly means that **we start behaving selflessly and altruistically instead of pursuing self interest**, at least as much as we practically can. It means that we start seeing the life on our whole planet (and possibly in the whole universe) as our own family, not as our enemies. This doesn't mean we will like everyone, that we'll agree with everyone's opinions, that we won't criticize anyone or that we'll be politically correct etc., it just means that we will never try to cause suffering to others, that we'll try to not exploit others, that we'll be aware of the needs of others and try to behave towards them with empathy and love. Importantly **we will try to pursue these ideals even if we can't achieve perfection**.

## Basic Description

The following is a basic description of just some features of the ideal society, some of which are however only speculative. Keep in mind it is impossible to plan a whole society exactly -- even if some of the speculations here turn out to be somehow erroneous, it probably still doesn't present a fatal obstacle to implementing our society, things may simply just turn out differently or to be more or less challenging than we predict.

Our society is **anarcho pacifist and communist**, meaning it rejects capitalism, money, violence, war, states, social hierarchy etc. I.e. in our society **money, market, capitalism, consumerism, private property, wage labor and trade don't exist**, people are free and happy as they can pursue their true interests and potential without worrying about resource fight.

**People don't have to work**, basically everything is automated and extremely simplified so that the amount of work needed to be done is minimized by eliminating unnecessary bullshit jobs such as marketing, lawyers, insurance, politicians, state bureaucracy, creation of consumer entertainment and goods etc. One of the basic principles of our society is that any individual can simply live, without having to deserve this right by proving worth, usefulness, obedience etc. The little remaining human work that's necessary is done voluntarily. There is no life path ("elementary school, high school, college, marriage, work, retirement, death") lined up for anyone, no fight awaiting, only one lifetime given to the hands of every new born individual to dedicate to whatever he finds best.

**Society is NOT based on competition, but rather on collaboration**. Making people compete for basic life resources is seen as highly cruel and unethical. The natural need for competition is still satisfied with games and sports, but people know competition is kind of a poison and know they have to practice self control to not allow competitive tendencies in real life.

**There is abundance of resources for everyone, poverty is non existent**, artificial scarcity is no longer sustained by capitalism. There is enough food and accommodation for everyone, of course for free, as well as health care, access to information, entertainment, tools and so on. Where there used to be shopping centers, parking lots, government buildings and skyscrapers, there are now fields and food banks and people voluntarily collaborate on automating production of food on them.

**Our society is NOT fair**, everyone is happy whether he deserves it or not, happiness doesn't have to be deserved. In a fair society the talented is rewarded and the untalented suffers, the strong wins and the weak loses, fairness is about winning and losing, reward and punishment, therefore our society is unfair as there are no longer any winners or losers.

**States and governments don't exist**, there are no artificial borders. Society self regulates and consists of decentralized, mostly self-sufficient communities that utilize their local resources as much as they can and send abundant resources to communities that lack them. **The is no law** in the sense of complex written legislation, **no lawyers, courts and police**, society works on the principle of moral laws, education and non-violent actions (e.g. refusal of people to use money etc.). Communities aren't hugely interdependent and hyperspecialized as in capitalism so there is no danger of system collapse. Many decisions nowadays taken by politicians, such as those regarding distribution of resources, are in our ideal society made by computers based on collected data and objective scientific criteria.

**Criminality doesn't exist**, there is no motivation for it as everyone has abundance of everything, no one carries guns, people don't see themselves as competing with others in life and everyone is raised in an environment that nurtures their peaceful, collaborative, selfless loving side. People with "criminal genes"

have become extinct thanks to natural selection by people voluntarily choosing to breed with non-violent people. Conflict between people is minimized by the elimination of self interest (and need for it) -- a lot of violence in current society comes from disagreement which comes from everyone's different goals (everyone aims to benefit oneself); in our society this is no longer the case, people rarely disagree on essential decisions because decisions are driven by pure facts collected without distortion or suspicion of self interest.

**Technology is simple, powerful, efficient, future proof, ecological, generally good and maximally helps people.** Internet is actually nice, it provides practically all information ever digitized, for example there is a global database of all videos ever produced, including movies, educational videos and documentaries, all without ads, DRM and copyright strikes, coming with all known metadata such as tags, subtitles, annotations and translations and are accessible by many means (something akin websites, APIs, physical media ...), all videos can be downloaded, mirrored and complex search queries can be performed, unlike e.g. with YouTube. Satellite images, streams from all live cameras and other sensors in the world are easily accessible in real time. Search engines are much more powerful than Google can dream of as data is organized efficiently and friendly to indexing, not hidden behind paywalls, JavaScript obscurity or registrations to websites, which means that for example all text of all e-books is indexed as well as all conversations ever had on the Internet and subtitles of videos. All source code of all programs is available for unlimited use by anyone. There are only a few models of standardized computers -- a universal **public domain computer** -- not thousands of slightly different competing products as nowadays. There is a tiny, energy efficient computer model, then a more powerful computer for complex computations, a simple computer designed to be extremely easy to manufacture etc. None of course have malicious features such as DRM, gay teenager aesthetics, consumerist "killer features" or planned obsolescence. All schematics are available. People possibly wear personal wrist-watch-like computers, however these are nothing like today's "smart" watches/phones -- our wrist computers are completely under the user's control, without any bullshit, spyware, ads and other malicious features, they last weeks or months on battery as they are in low energy consumption mode whenever they're not in use, they run extremely efficient software and are NOT constantly connected to the Internet and updating -- as an alternative to connecting to the Internet (which is still possible but requires activating a transmitter) the device may just choose to receive a world-wide broadcast of general information (which only requires a low power consumption receiver) if the user requests it (similarly to how teletext worked), e.g. info about time, weather or news that's broadcasted by towers and/or satellites and/or small local broadcasters. Furthermore wrist computers are very durable and water proof and may have built-in solar chargers, so one wrist computer works completely independently and for many decades. They have connectors to attach external devices like keyboards and bigger displays when the user needs to use the device comfortably at home. The computing world is NOT split by competing standards such as different programming languages, most programmers use just one programming language similar to C that's been designed to maximize quality of technology (as opposed to capitalist interests such as allowing rapid development by incompetent programmers or update culture).

**Fascism doesn't exist**, people no longer compete socially and don't live in fear (of immigrants, poverty, losing jobs, religious extremists etc.) that would give rise to militarist thought, society is multicultural and races highly mixed. There is no need for things such as political correctness and other censorship, people acknowledge there exist differences -- differences (e.g. in competence or performance) don't matter in a non-competitive society, discrimination doesn't exist.

**Computer security is not an issue anymore**, passwords and encryption practically don't exist anymore, there is nothing to "steal", no money on the Internet, no way to abuse personal data, no possibility to ruin someone's career, no celebrity accounts to hack etc.

**All people speak the same language**, possibly Esperanto or Lojban. Though some speak multiple languages, most of the world languages have become archaic and are studied e.g. for the sake of understanding old texts. Of course dialects and different accents of the world language appear, but all are mutually intelligible thanks to constant global communication and also people being so responsible as to willingly try to not diverge from the main form too much.

**People don't wear clothes** unless for practical reasons (weather, safety, ...). Fashion and shame of nudity doesn't exist and it is seen as wasteful to keep manufacturing, cleaning and recycling more clothes than necessarily needed. Of course it is NOT forbidden to wear or make clothes, people just mostly naturally don't engage in unnecessary, wasteful activity.

**Anyone can have sex with anyone**, without hurting anyone of course, but there are no taboo limitations like forbidden incest, sex with children, animals or dead bodies, everything is allowed and culturally acceptable as long as no one gets hurt. "Cheating" in today's sense doesn't exist, marriage doesn't exist, people regularly have sex with many other people just to satisfy the basic need. People have learned to separate sex and love. Of course many people still live in life-long partner relationships.

**There are no heroes or leaders**. People learn from young age that they should follow ideas, not people or groups of people, and that cults of personality are dangerous. There are known experts in different disciplines and areas of science, but no celebrities, experts aren't worshiped, their knowledge is treated the same as we nowadays e.g. treat information that we find in a database. This doesn't mean there aren't people who are good moral examples and whose behavior is admired, people are just separated from their actions, people admire behavior, not the individual -- all people are loved unconditionally, some had the opportunity to take admirable actions and took it, some were born to perform well in sports or excel in science, but that's no reason to love the individual any more or any less or to worship him as a god.

**Education is not indoctrination, it is actually good**, people (not only children) attend schools voluntarily (though such "schools" will be extremely different from what the word means today), there are no grades, degrees or tests that need to be passed or prescribed courses, only recommendations and guidance of other people. There is no strict division to students and teachers, teachers are students at the same time, older people teach younger. There may of course exist voluntary tests that people can take to test their knowledge and competence, but no one is forced to pass tests to continue studying etc.

**People don't kill or otherwise abuse and torture animals**, people just don't eat much meat and if they want to, artificial meat is widely available. Some may possibly eat meat of animal and people that died naturally, which is acceptable.

**Cannibalism is acceptable** as long as high hygiene is respected as it puts a dead body to good use instead of wasting food by burying it or burning it. Even though most people don't practice cannibalism, it is perfectly acceptable that some do. Many people wish to be eaten after death either by people or by animals (as for example some Buddhists do even nowadays).

**People aren't individualist and egoistic**, they don't have tattoos, dyed hair, piercing etc., that's simply bullshit of primitive competitive cultures. It is correctly seen as immoral to try to persuade by "good looks" -- for example by wearing a suit -- that's simply a cheap attempt at deception. Everyone is valued the same no matter his looks, people don't feel the need to change their gender or alter their look so as to appeal to anyone or to follow some kind of fashion, trend, to strategically join some minority to gain the best set of "rights" or to infiltrate specific social class. Of course cutting hair e.g. for comfort is practiced, but no one wastes his time with makeup and similar nonsense.

**People live in harmony with nature**, the enormous waste of capitalism and consumerist society has been eliminated, industry isn't raping nature, cities are greener and more integrated with nature, people live in energy-efficient underground houses, there are fewer roads as people don't use cars much thanks to efficient public transport and lower need for travel thanks to not having to go to work, utilizing mostly local resources etc.

**Research advances faster, people are smarter, more rational, emphatic, loving and more moral**. Nowadays probably the majority of the greatest brains are wasted on bullshit activity such as studying and trying to hack the market -- even professional researchers nowadays waste time on "safe", lucrative unnecessary bullshit research in the "publish or perish" spirit, chasing grants, easy patents etc. In our ideal society smart people focus on truly relevant issues such as curing cancer, without being afraid of failure, stalling, negative results, lack of funds etc. People are responsible and practice e.g. voluntary birth control to prevent overpopulation. However people are NOT cold rational machines, on the contrary emotions are present much more than today, for example the emotion of love towards life is so strong most people are willing to die to save someone else, even a complete stranger. People express emotion through rich art and good deeds. People are also spiritual despite being highly rational -- they know rationality is but one of many tools for viewing and understanding the world. **Religion still exists** commonly but not in radical or hostile forms; Christianity, Islam and similar religions become more similar to e.g. Buddhism, some even merge after realizing their differences are relatively unimportant and stop competing for mass control of people, religion becomes much less organized and much more personal.

**Art is rich and unrestricted** (no copyright or other "IP" exists), with people being able to fully dedicate their lives to it if they wish and with the possibility to create without distraction such as having to make living or dealing with copyright. People collaborate and reuse each other's works, many free universes exist, everyone can enjoy all art without restriction and remix it however he wishes.

**People live much longer and healthier lives** thanks to faster research in (actual) medicine, free healthcare, more doctors (those who nowadays do bullshit business), better food (no junk/fast food), less pollution, higher living standard, more natural life closer to nature, minimization of stress and elimination of the antivirus paradox from medicine. They also die more peacefully thanks to having lived a rich, fulfilling lives, they die in the circle of their family and are not afraid of death, they take it as natural part of life, death is culturally accepted and not feared. Euthanasia is allowed and common for those who wish to die for whatever reason.

## FAQ

- **Isn't it utopia?** As explained above, the society is an ideal model that's probably not 100% achievable, but we are pretty certain we can get extremely close to the ideal in the real world implementation, there are no known obstacles to it. Even if we couldn't get very close to the ideal, it would be better to get a little closer than not, there is no logic in refusing to try. Every major invention happened for the first time one day, even when it's been called impossible; for example before Wikipedia practically everyone would tell you the principles on which it would be built (free voluntary work, allowing anyone to edit) were utopian. History is basically a constant stream of events proving our disbeliefs wrong. Things such as abolishment of death sentence, universal literacy, universal health care, women in science, abolishment of black man slavery, instant world wide communication and similar things might have once been deemed a similar utopia.
- **How is this different from "communism", "socialism" and other movements/ideologies that brought so much suffering and eventually failed anyway?** We are very different especially by NOT advocating revolutions, violence and forceful application of our ideas, we simply educate, show what's wrong and what the solution is. Harm has only ever been done by forcing specific ideas, no matter whether rightist or leftist ones -- the key in preventing harm is to avoid the temptation of forcing ideas. We know that only a voluntary, non-violent change based on facts and rational thinking can succeed in long term. The mistake of every failed "utopian" ideology was that it was forced, oppressive and in the end served only a few as opposed to everyone, no matter what the initial idea was. These ideologies fought other ideologies, creates cults of personalities and propaganda to manipulate masses. We do not fight anyone, we simply show the truth and offer it to people and believe that this truth can't be unseen. Once enough people see the truth and know what the logical solution is, a change will happen naturally, peacefully and inevitably, without any force.
- **How do you think it is realistic to achieve abundance of resources for all?** Nowadays it is easily possible to produce enough resources for everyone, i.e. food, electricity, clothing, buildings to live in etc. -- in fact this has been possible for many decades to centuries now, today all the technology for 99% automated production of most basic resources such as food and electricity is available and well tested, it is just kept in private hands for their sole profit. Nowadays our society is putting most of its effort to artificially made up "businesses" that keep the status quo, partly out of social inertia and partly by the (mostly decentralized and to a degree not even self admitted) conspiracy of the rich. Imagine people stop engaging in marketing, market speculation and investing, bureaucracy, public relations, law (copyrights, patents, property laws, taxes, ...), economics, military, meaningless technology (DRM, spyware, cryptocurrency, viruses and antiviruses, ...), artificial meaningless fashion, drug abuse business, organizing political parties, campaigns, unions, counter unions, cartels, strikes, and so on and so forth (this of course doesn't mean hobbies and art should disappear, just unnecessary industries). We will gain millions of people who can help achieve abundance, land that can be used to produce food and build houses to live in (as opposed to skyscrapers, unnecessary factories, parking lots etc.), and we will let go of the immense burden of bullshit business (millions of unnecessary workplaces having to be maintained, millions of people having to commute by car daily, communicate, organize, be watched by employers, ...). People will get healthier, more rested, cooperative and actually passionate about a common goal, as opposed to depressed (needing psychiatrists and antidepressants), lethargic and hostile to each other. Of course this can't happen over night, probably not even over a decade, but we can make the transition slowly, one step at a time and in the meanwhile use rules based e.g. on the following principle: that which is abundant is unlimited for everyone, that which is scarce is equally divided between all. The question is not whether it's possible, but whether we want to do it.

- **Isn't your society unnatural?** In many way yes, it's unnatural just as clothes, medicine, computers or humans living over 70 years are unnatural. Civilization by definition means resisting the cruelty of nature, however our proposed society is to live as much as possible in harmony with the nature and is much more natural than our current society which e.g. pushes sleep deprivation, high consumption of antidepressants, eating disorders, addiction to social networks and so on.
- **Won't people get bored? What will motivate people? If they have everything why would they even get out of bed? Haven't you seen the mouse utopia experiments?** It is a mistake to think that competition and the necessity of making living is the only or even the main driving force of human behavior and creativity (on the contrary, it is usually what makes people commit suicides, i.e. lose the will to live). Human curiosity, playfulness, the joy of collaboration, boredom, sense of altruism, socialization, seeking of life meaning and recognition and many other forces drive our behavior. Ask yourself: why do people have hobbies when no one is forcing them to it? Why don't you bore yourself to death in your spare time? Why don't rich people who literally don't have to work bore themselves to death? Why doesn't your pet dog that's not forced to hunt for food bore himself to death? Maslow's hierarchy of needs tells us that once people fulfill basic needs such as that for obtaining food, they naturally start to pursue higher ones such as that for socializing or doing science or art. Unlike rats in small cages people show interests in seeking satisfaction of higher needs than just food and sex, even those that aren't scientist try to do things such as sports, photography, woodwork or gardening, just for the sake of it. It's not that there would be a lack challenges in our society, just that we wouldn't force arbitrary challenges on people.
- **If you say it's possible, why wasn't it done before?** Firstly big and small scale communities working on anarchist, communist and peaceful principles have existed for a long time in environments that allow it, e.g. those that have abundance of resources. Globally society couldn't reach this state because only until recently we lacked the technology to provide such an ideal environment globally or even on a scale of a whole country, i.e. only until recently we have been forced by the nature to compete for basic resources such as food and space to live. However with computers, factories, high level of automation and other technology and knowledge we possess, we now have, for the first time in history, the capability to establish an environment with abundance of resources for everyone on the planet. Nowadays only social inertia in the form of capitalism is ARTIFICIALLY keeping scarcity and social competition in place -- getting rid of this obsolete system is now needed to allow establishment of our ideal society. Part of the answer to this question may also be that reaching such an advanced state of society requires long development, technological, cultural and intellectual, just as many other things (things like abolishment of death sentence or even accepting the existence of irrational numbers all required a long time of cultural development).
- **How will you make people work?** We won't, in an ideal society people don't have to work, all work is done by machines -- that's the point of creating machines in the first place. In practice there may in a foreseeable future be the need for small amounts of human work such as overlooking the machines, but the amount of work can be so small that volunteers will easily handle it -- especially with people having no burden of working day jobs there should be no shortage of volunteers. Remember that by abandoning the current system 99% of "bullshit work" (marketing, lawyers, bureaucracy, fashion, ...) will disappear.
- **Does elimination of bullshit jobs mean my favorite activity will disappear?!** Unless your hobby is something like killing people for fun, we don't aim to force anyone quitting anything he likes to do, on the contrary we aim exactly to establish the freedom to do anything one desires. So if you like e.g. designing clothes, you are free to do so as a form of art, we just argue against e.g. socially forced necessity to follow fashion in clothing and making capitalist business out of it, which is what we call "bullshit". We believe most bullshit activities that were invented by capitalism, such as marketing, will simply naturally disappear once capitalism ends -- there is no need to force disappearance of something that dies out naturally.
- **How will society make progress?** Just as it always had: by science, curiosity, necessity, accidental discoveries, pursuit of creating art etc. It is a mistake to think we need capitalism or anything similar in order to make progress; progress cannot be stopped no matter what, it will always be here as long as humans exist.
- **How do you prevent natural human selfish and violent behavior?** Violent and selfish behavior is natural in us just as peaceful and altruistic behavior, we have two sides and which one shows is mostly determined by the conditions in which we live, by our upbringing and people we see as role models. Nowadays we think people are extremely selfish and violent because we live in society that highly fuels the "evil" side in us, when we're forced to fight for basic living and grow up in a highly competitive environment, it's no surprise most adapt to this and grow up to be "dicks". If we're forced to fight for food and brainwashed since birth that "life is a fight", we will be selfish and violent, just as



wild wolves are violent while pet dogs whose needs are secured and who were raised peacefully are mostly completely peaceful. If we have abundance and grow up in a society that naturally rejects any violence, we will grow up to solve conflicts peacefully and think of others, just as nowadays we e.g. naturally learn to wear clothes because simply everyone does it and there is little reason not to. If we make resources abundant, there will be no need for selfishness -- do you see anyone stealing air from others out of selfishness? No, because there is no need for it, air is abundant. By changing the environment people live and grow up in we will make 99.99% of people abandon violence and selfish interests (note the remaining natural need for selfishness and competition can be satisfied e.g. with games).

- **How will you prevent chaos without laws or rules for the people?** We don't say there should be no rules, we are just against complicated written law that no one can even comprehend (even lawyers don't know all laws nowadays) and that has so little in common with morality. Our society works on the basis of moral rules that all stem from the common goal of well being of living beings and that are derived and taught by people themselves -- for example one moral rule that all people would learn would be that money is bad for society (along with the reasons why it is so) and even though there would be no police "enforcing" this rule, the rule would be effective by the fact that absolute majority of people would simply refuse to use money -- in a society where most people know capitalism is bad for them capitalism can't work. Note the importance of the fact that people wouldn't just be taught to memorize such rules as "facts set in stone" (as is our current law), emphasis would be put on people deriving their moral code and understanding how their behavior affects others, people would learn and teach by example.
- **How will you prevent criminality such as stealing, murder and mafia organizations?** In a society with abundance for all which works for the good of all criminality simply won't make sense, i.e. we will eliminate criminality by solving the root cause of it, not by curing the symptoms (building prisons etc.). People have no reason to revolt against a system that benefits them or attack other people if there is no conflict between them. Large criminal organizations also cannot exist if most population rejects them, for example there cannot arise a capitalist corporation (or a similar mafia organization) if most population is educated and refuses to engage in capitalism. In addition to this a more mature, educated and responsible society will naturally minimize genetic predisposition to things such as aggression and self interest by natural selection as females will choose to rather have offspring with good people (unlike today), making genes associated with bad behavior go extinct. Of course, we probably won't eliminate criminality 100%, but that's not possible under any other system -- even in your current society with prisons and other punishments there still exist criminality. Of course in practice, until we achieve our ideal, we will likely need to keep some anti-criminality precautions as a necessary evil, but generally we will be able to greatly relax them (reduce police numbers, abandon death penalty, ...) as we move towards the ideal society. For example in an intermediate state of our society dangerous criminals won't be killed but only immobilized and they won't be put in prisons as a punishment but only sent to e.g. a remote island so as to be isolated, without punishing them by restricting their freedom within the island.
- **How will the economy work without money?** With abundance of resources there will be no money and no trade, resources will be available to anyone who needs them. Various anarchist schools already have proposals for how distribution of resources could work. The Venus Project calls this a resource based economy and proposes using computers and globally placed sensors to collect data and make decisions about where to distribute resources. Resources would be gathered and distributed more locally and cities would be more self sufficient so as to prevent waste and vulnerability of the system, we wouldn't see a huge globalization like nowadays, there is e.g. no need for transporting exotic food all over the whole world to places where there is enough local food available, however anything could be distributed to places where such resources are scarce (e.g. water to deserts). Each community could have food banks and other storage and distribution centers.
- **How will you prevent discrimination and racism?** Things such as racism appear when one group of people feels endangered by another group, in a society without social competition these issues will naturally disappear.
- **How will you fulfill the natural need of people for competition?** With sports and other games. Competition of people won't be forbidden, it just won't be mandatory and it won't be the basis of society.
- **How will you prevent overpopulation?** By voluntary birth control.
- **How will you force people to change so radically?** We won't force people to change, the change has to be voluntary, and that will be achieved by education. We don't advocate revolution but rather a slower, evolutionary transition. Just as now you're learning about our ideal society, more people will. With more people on the board the word should spread more quickly and with better conditions and

greater general education of people over the world more will start to see and realize this is the only way forward.

- **Do you really think you can convince even diehard neonazis to accept these ideas?** Not in their lifetime -- some people can't practically be convinced, it would take longer than they will be alive. But these people will die one day and there will come a new generation, a tabula rasa, which will have the opportunity for a better upbringing and not growing up to become diehard nazis.
- **Without any censorship how will you prevent "hate speech" or protect people's personal data?** As mentioned above, racism and issues of so called "hate speech" will simply disappear in a non-competitive society. The issues of abuse of personal information will similarly disappear without any corporations that abuse such data and without conflict between people, in the ideal society there won't even be any need for things such as passwords and encryption.
- **How will you prevent psychopaths from just going and killing people?** In the ideal society maximum effort will be made to prevent wrong psychological development of people which can happen due to crime, poverty, discrimination, bullying etc., so the cases of lunatics killing for no reason would be extremely rare but of course they would happen sometimes, as they do nowadays, they cannot be prevented completely (they aren't completely prevented even nowadays, a psychopath is not afraid of police). Our society would simply see such events as unfortunate disasters, just like natural disasters etc. In transition states of our society there may still exist imperfect means of solving such situations such as means for non lethal immobilization of the attacker and his isolation (but not punishment, i.e. not a prison).
- **Would such society be stable? Wouldn't people revert back to "old ways" over time?** We believe the society would be highly stable, much more than current society plagued by financial crises, climate changes, wars, political fights etc. The longer a good society stays, the more stable it will probably become as its principles will become more and more embedded in the culture and there will be no destabilizing forces -- no groups revolting "against the system" should appear because no one will be oppressed and therefore unhappy about the situation.
- **Will you allow abortions?** There is no strict YES/NO answer here, as with everything there will be no simple allowing or forbidding laws, decisions about abortions will be made in the spirit of the common goal, handled on a case-by-case basis and strong prevention of unwanted and/or risky pregnancy. There will be more people willing to adopt children, birth control means will be better and accessible to anyone for free, children will not pose any financial burden or be an "obstacle to one's career", so this issue won't be nearly as great as it is today.
- **You say you want equality of all living beings -- does this mean you will force animals to not kill each other or that you will refuse to e.g. kill viruses?** Ideally we would like to maximize the happiness and minimize suffering of all living beings, even primitive life forms such as bacteria, and if that cannot be achieved at the time, we will try to get as close to it as we can and do the next best thing. Sometimes there are no simple answers here but the important thing is the goal we have to keep in mind. For example provided that we want to sustain human life (i.e. we don't decide to starve to death) we have to choose what to eat: nowadays we will try to be vegan so as to spare animals of suffering but we are still aware that eating plants means killing plants which are living beings too -- we don't think the life of a plant is less worthy of an existence than that of an animal, but from what we know plants don't show signs of suffering to the degree to which e.g. mammals do, so eating plants rather than animals is the least evil we can do. Once we invent widely available artificial food, we will switch to eating that and we'll stop eating plants too.

## How To Implement It

This is the hard part, however after successfully setting things in motion it may start to become much easier and eventually even inevitable that the ideal society will be closely approached. However at the moment society seems too spoiled and change of a direction seems very unlikely, it seems more probable that we will destroy ourselves or enslave ourselves forever -- capitalism and similar misdirections of society connected to self-interest, competition, fascism etc. pose a huge threat to our endeavor and may ruin it completely, so they need to be strictly opposed, but in a CORRECT way, i.e. not by revolutions and violence but rather by education, offering alternatives and leading examples (i.e. means aligned with our basic values). It has to be stressed that we always need to follow our basic values of nonviolence, love, true rationality etc., resorting to easy ways of violence etc. will only prolong the established cycle of suffering in the society which we are trying to end. Remember, we are not creating a revolution, we aim for a rather slow, nonviolent, voluntary evolutionary change.

We already have technology and knowledge to implement our ideal society -- this may have been the most difficult part and it has already been achieved -- that's the good news.

For the next phase education is crucial, we have to spread our ideas further, first among the intellectuals, then to the masses. By this we seek to **unretard** society. Unfortunately this phase is still in its infancy, vast majority of intellectuals are completely uneducated in this area -- this we have to change. There are a few that support parts of our plan such as simple technology, nonviolence, not hurting animals etc., but almost no one supports them all, or see the big picture -- we need to unite these people (see also type A/B fail) to form a small but dedicated community sharing all the proposed ideas. This community will then be able to collaborate on further education, e.g. by creating materials such as books, games, vlogs, giving talks etc.

With this more of the common people should start to jump on the train and support causes such as universal basic income, free software etc., possibly leading to establishment of communities and political parties that will start restricting capitalism and implementing a more socialist society with more freedom and better education, which should further help nurture people better and accelerate the process further. From here on things should become much easier and faster, people will already see the right direction themselves.

## Inspiration

Here are some of the ideas/movements/ideologies and people whose ideas inspired less retarded society. It has to be stressed we never follow people, only their ideas -- mentioning people here simply means we follow SOME of their ideas. Also keep in mind mentioning an idea here doesn't mean fully embracing it, we most likely only adopted some parts of it.

- **anarcho pacifism**: Rejecting force and hierarchy of one living being dominating and oppressing another.
- **beatniks/hippies**: We are inspired by many of their ideals such as free love, pacifism and avoidance of work.
- **Buddha, Buddhism**: Attaining freedom through letting go, focusing on the spiritual rather than the material, living non violently.
- **communism, anarcho communism, socialism** (but NOT Marxism): Sharing, equality, rejection of property and money, focus on people at large.
- **Diogenes, cynicism**: Rejecting conformity, wealth, work, power, fame, materialistic needs, embracing simple living, self sufficiency living in harmony with nature, choosing asceticism and difficult way of life as path towards spiritual clarity -- cynics have a great deal in common with LRS.
- **Gandhi, non violence**: Achieving things without the use of violence (and similar kinds of force), completely refusing to use certain unethical means for achieving goals, not abandoning one's beliefs even for the cost of one's life.
- **Jesus, Christianity**: Teaching love towards everyone, even those who hurt us, practicing non violence, helping, sharing and compassion, opposing materialist values, valuing the spiritual, being ready to die for one's beliefs.
- **minimalism, KISS, suckless, less is more, worse is better, Unix philosophy, ...**: Way towards freedom, both practical and spiritual, letting go of the unneeded, most essential design principle, beauty and elegance.
- **primitivism**: Related to minimalism, letting go of unnecessary and focus on what matters the most, living close to nature.
- **Richard Stallman, free software, free culture**: Opposition of "intellectual property", focus on ethics, freedom and technology/art serving the people.
- **Sikhism**: Serving free food to all people as part of Langar, example of selflessness.
- **vegetarianism, veganism**: Choosing to not hurt other living beings, even those that aren't of the same species, even for the cost of making having less comfortable life.
- **Venus project**: Project with very similar goals as ours.
- ...

## See Also

- LRS
- how to
- Venus Project

- [socialism](#)
  - [Buddhism](#)
- 

less\_retarded\_software

## Less Retarded Software

Please kindly redirect yourself to [LRS](#).

---

lgbt

## LGBT

*This article is a part of series of articles on [fascism](#).*

LGBT, LGBTQ+, LGBTFURRYTRANSFAGCOCKS (lesbian, [gay](#), [bisexual](#), [transsexual](#), "[queer](#)" and whatever is yet to be invented), also FGTS or TTTT (transsexual transsexual transsexual transsexual) is a toxic [pseudoleftist](#) [fascist](#) political group whose ideology is based on superiority of certain selected minority sexual orientations. They are a highly [violent](#), [toxic](#), [bullying](#) movement (not surprisingly centered in the [US](#) but already spread around the whole world) practicing [censorship](#), Internet lynching ([cancel culture](#)), discrimination, spread of extreme [propaganda](#), harmful [lies](#), culture poison such as [political correctness](#) and other [evil](#).

LGBT is related to the concept of equality in a similar way in which crusade wars were related to the nonviolent teaching of [Jesus](#), it shows how an idea can be completely twisted around and turned on its head as to be left completely contradicting its original premise.

Note that **not all gay people support LGBT**, even though LGBT wants you to think so and media treat e.g. the terms *gay* and *LGBT* as synonyms (this is part of [propaganda](#), either conscious or subconscious). The relationship gay-LGBT is the same as e.g. the relationship White-WhitePride or German-Nazi: Nazis were a German minority that wanted to [fight](#) for more privileges for Germans of their own race (as they felt oppressed by other nations and races such as Jews), LGBT is a gay minority who wants to [fight](#) for more privileges for gay people (because they feel oppressed by straight people). LGBT isn't just about being gay but about approving of a very specific ideology that doesn't automatically come with being gay. LGBT frequently comments on issues that go beyond simply being gay (or whatever), for example LGBT openly stated disapproval of certain other orientation (e.g. [pedophilia](#)) and refuses to admit homosexuality is a disorder, which aren't necessarily stances someone has to take when simply being gay.

**LGBT is a cult** that managed to actually get mainstream and embraced by the ruling powers ([states](#) and [corporations](#)) -- their [pseudoscience](#), called "[gender studies](#)", is not unlike the hilarious "science" of Ancient Aliens, the LGBT theories are not unlike the Nazi theories about underground Jewish societies secretly ruling the world. Just like some see everything a work of aliens or Jews, LGBT sees a secret gender oppression in everything, in 100 years old child cartoons, in primitive video games like [pacman](#), in colors of the butterfly wings, everything has a secret straight cis male oppression message embedded within it. If you ever wondered what it would look like if Scientology took over the world or if Nazis won the world war^([Hitler comparison committed but rightfully so]), you don't have to wonder anymore, it's right here (chances are just that you don't see it just as you wouldn't see Scientology as weird if you grew up in a culture completely controlled by it).

Gay fascists furthermore live off of attention so they love to wear bizarre clothes in all existing AND nonexistent [colors](#) at once, further combined with ugly hairstyles and [tattoos](#) so that they literally look like clowns from mental asylum or that creepy McDonald's mascot. They also love to show their genitalia in the streets -- though they are pedophobes, they think it's a peer reviewed fact that it's natural for a child to see mommy have threesome with her frens.

LGBT works towards establishing [newspeak](#) and [thought crime](#), their "pride" parades are not unlike military parades, they're meant to establish fear of their numbers. LGBT targets children and young whom their propaganda floods every day with messages like "*being gay makes you cool and more interesting*" so that

they have a higher probability of developing homosexuality to further increase their ranks in the future. They also push the idea of children having same sex parents for the same reason.

LGBT oppose straight people as they solely focus on gaining more and more "rights" and power only for their approved orientations. They also highly bully other, unpopular sexual orientations such as pedophiles (not necessarily child rapists), necrophiles and zoophiles, simply because supporting these would hurt their popularity and political power. They label the non-approved orientations a "disorder", they push people of such orientations to suicide and generally just do all the bad things that society used to do to gay people in the past -- the fact that these people are often gay people who know what it's like to be bullied like that makes it this even much more sad and disgusting. To them it doesn't matter you never hurt anyone, if they find some lol images on your computer, you're gonna get lynched mercilessly.

In the world of technology they are known for supporting toxic codes of conduct in FOSS projects (so called tranny software), they managed to push them into most mainstream projects, even Linux etc. Generally they just killed free speech online as well as in real life, every platform now has some kind of surveillance and censorship justified by "preventing offensive speech". They cancelled Richard Stallman for merely questioning a part of their gospel. They also managed to establish things like "diversity" quotas in Hollywood that only allow Oscars to be given to movies made by specific number of gays, lesbians etc., and they started to insert gay characters into fairy tales and movies for children (Toy Story etc.) xD This is literally the same kind of cheap but effective propaganda Nazi Germany employed on children; it's just that now after nationalism has been demonized after the world war we replaced nationalism with gender identity, an exactly same thing in principle just with a different name. Apparently in the software development industry it is now standard to pretend to be a tranny on one's resume so as to greatly increase the chance of being hired for diversity quotas xD WTF if I didn't live in this shitty world I wouldn't believe that's even possible, in a dystopian horror movie this would feel like crossing the line of believability too far lmao.

In the non-technological world they are known for example, besides others, for destroying all art by giving everything a twisted sexual context, for example there's now a retroactively injected LGBT propaganda in child stories like Harry Potter: children reading about the old, wise Dumbledore now also have to read the asterisks about how they/thems is in fact a hexadecimal non fluid that had oral sex with Severus Snape, which is of course not relevant to the story at all, it's there to just compensate for the fact that he's a white male, so he can't at all be straight because he's supposed to represent good (straight white males can only represent evil nowadays).

---

liberalism

## Liberalism

*Not to be confused with libertarianism.*

Liberalism is a political ideology whose definition is not greatly clear (we may find branches that differ a lot) but which usually aims for "liberty", focus on individuals who ought to be protected by the state and have equal opportunities, which leads to obsession with all kinds of "rights" and "social justice" (i.e. social revenge of minorities); as one of worst imaginable ideologies it is no surprise it's the prevailing US ideology and ideology of SJWs -- liberalism is taking over the whole western world and it's destroying everything. It basically tries to take the worst of all other ideologies: liberalism supports things such as state and strong laws (to "protect" people), capitalism (to give them "opportunities"), censorship, political correctness and violence; supporting concepts connected to both right and (pseudo)left, it is said to be a "centrist" stance, however we just call it confused -- they just try to combine absolutely incompatible things, they want a competitive environment in which "everyone wins". Liberalism is highly harmful, retarded and should never be supported.

---

libertarianism

## Libertarianism

*Not to be confused with liberalism.*

Libertarianism is a harmful political ideology whose definition is quite broad and not super clear, but which in essence gives highest priority to individual "liberty" and seeks to minimize the role of state (but typically without wanting to remove it). A bit like anarchism, libertarianism has many branches which frequently greatly diverge and even oppose each other, some are called more "leftist", some more "rightist" -- libertarianism usually tries to pretend to be focusing on the people, i.e. their "liberties", pseudoequality ("equality before law", "equality of opportunity", ...), oppose "the kind of corporate capitalism we have today", believing some kind of "saner" version of it can work (which it can't), and claims that people can form a working, decentralized society by loose associations, however, unlike anarchism which opposes state and any kind of hierarchy altogether (with true anarchism also opposing any violence), libertarianism typically wants to preserve some functions of the state such as courts and justice for protection against crime, and it acknowledges property as a sacred thing that may even be defended by violence, i.e. libertarianism just replaces the rule of states by rule of private subjects, getting quite close to "anarcho" capitalism, the stupidest idea yet conceived. Libertarians basically adopts the "**law of the jungle**" or "**wild west**" mindset. So it's shit, do not subscribe.

USA is essentially just a land where libertarians battle with liberals. Both camps are similarly stupid.

{ Some bashing by digdeeper: <https://digdeeper.neocities.org/articles/libertarianism>. ~drummyfish }

---

library

## Library

Software library (often shortened to just *lib*) is program code that's not meant to run on its own but rather to be used by other programs, i.e. it is a helpful collection of preprogrammed code that's meant to be reused. A library provides resources such as functions, macros, classes or constants that are normally related to solving some specific class of problems, so e.g. there are GUI libraries, audio libraries, mathematical libraries etc. Libraries exist mostly to prevent reinventing wheels by only ever implementing the code once so that next time we can simply reuse it (respecting the DRY principle), but they also e.g. help assure others are using an already well tested code, they help to implement modularity etc. Examples of libraries are the standard C library, SDL or JQuery. Libraries are not to be confused with frameworks which are larger, more bloated environments.

In Unix environments there is a convention for library packages to start with the `lib` prefix, so e.g. SDL library is named `libSDL` etc.

**Standard library** (`stdlib`) is a term that stands for the set of libraries that officially come with given programming language -- these libraries usually offer very basic functionality (such as I/O and basic math) and are required to always be present on every system.

If a programmer wants to use a specific library, he usually has to first somehow install it (if it's not installed already, usually a library is some kind of software package) and then include it in his program with a specific command (words like `include`, `using` or `import` are commonly used). Then he is able to use the resources of the library. Depending on the type of the library he may also need to link the library code after compilation and possibly distribute the library files along with his program. A more KISS approach is for a library to simply be a code that's somehow copy-paste included in the main program (see single header libraries etc.).

As a programmer you will encounter the term **library API** -- this is the interface of the library consisting of the elements via which programmer uses the library, mostly the functions the library offers. API is what the programmer interacts with; the rest is library internals (its implementation) that's usually supposed to not be touched and stay a bit hidden (see encapsulation). If a programmer wants to know the library API, he wants to know the names of the functions, what parameters they take etc. Sometimes there may be multiple libraries with the same API but different internal implementations, this is nice because these libraries can be easily drop-in-replaced. The library API is usually part of its documentation -- when learning a new library *X*, you want to search the internet for something like *X library API reference* to see what functions it offers.

In a specific programming language it IS generally possible to use a library written in a different language, though it may be more difficult to achieve -- see language bindings and wrappers.

We generally divide libraries to two types:

- **static**: The library code is embedded into the executable of the final program so that the library files have to be distributed along with the program. This is more convenient and also makes sure the program uses exactly the correct version of the library. But of course this results in bigger executable, and if we have multiple programs that use the same library which is statically linked, each program will have a redundant copy of the library code, wasting memory (both storage and RAM).
- **dynamic** (also *shared*): The compiled library code resides in a separate file (DLL on Windows, .so in GNU/Linux) which may need to be distributed along with the program, but this one file can be shared among all programs that use the library so the compiled programs can be smaller. It may also be easier to update the library to a new version by simply replacing the compiled library file. RAM may also be saved as the dynamic library may be loaded just once for multiple simultaneously running programs.

Many times a library can have both static and dynamic version available, or the compiler may allow to automatically link the library as static or dynamic. Then it's up to the programmer which way he wants to go.

## C Libraries

TODO: example

**Header only** or **single header** library is a kind of keep-it-simple library that's wholly implemented in a single header (.h) file -- this is kind of a hack going against "official recommendations" as header files aren't supposed to contain implementation code, just declarations, however single header libraries are suckless/LRS, convenient and very easy to use, as they don't have to be linked and are nicely self-contained, distributed in one nice file. A traditional library would consist of one or more header (.h) files and one or more implementation (.c) files; such library has to be compiled on its own and then linked to the program that uses it -- the idea behind this was to compile the library only once and so save time on recompiling it again and again; however this justification is invalid if our library is simple enough and compiles very quickly (which it always should, otherwise we are dealing with badly designed bloat). A single header library can therefore just be included and just works, without any extra hassle -- yes, its code recompiles every time the program is compiled, but as stated, it doesn't hurt if our library is well designed and therefore simple. Single header libraries often include the option (via some #define macro) to include just the declaration or both declarations and implementation code -- this is useful if our main program is composed of multiple source files and needs to be linked. LRS libraries, such as small3dlib or raycastlib, are of course single header libraries.

## LRS Libraries

TODO

---

libre

## Libre

Libre is an alternative term for free (as in freedom). It is used to prevent confusion of *free* with gratis.

---

license

## License

License is a legal text by which we grant some of our exclusive rights (e.g. copyright or patents) over intellectual works to others. To us a license is what enables us to legally implement free (as in freedom) software (as well as free culture): we attach a license to our program (or other work) which says that we grant to everyone the basic freedom rights to our software/work with optional conditions (which must not be in conflict with free software definition, e.g. we may require attribution or copyleft, but we may NOT require e.g. non-commercial use only). Licenses used to enable free software are called *free licenses* (open source

licenses work the same way). Of course, there also exist non-free licenses called EULAs, but we stay away from these -- from now on we will implicitly talk about free licenses only. Licenses are similar to waivers.

## You shall always use a free license for your software.

There exist fun/parody licenses like WTFPL (Do What the Fuck You Want to Public License) -- these are cool as a fun meme, though legally they may be invalid as they are too vague and the language could just make it look like a statement not meant seriously to the court, anything licensed this way should rather be seen as a licenseless work. It's better to not seriously use these, or if you do, dual license alongside with some "serious" license.

Free licenses are mainly divided into:

- **copyleft**: Licenses that require that further modifications of the work will still remain free, i.e. "forcing freedom". Example of such licenses are GPL and CC BY-SA. Copyleft licenses are a bit more associated with free software (as opposed to open source) as the main free software organization -- GNU -- advocates them because they disallow corporations to take free programs and make them into proprietary ones.
- **permissive**: Licenses that basically allow to "do anything you want" (though usually still requiring e.g. credit to the original author), even making modified non-free versions of the work. Most famous example is the MIT license. Though not strictly so, permissive licenses are a bit more associated with open source than free software as they are friendlier to business (one can "unfree" new versions of a software at any time if it's desirable for money making; of course old versions of the program will still remain free), however some prefer them for other reasons, e.g. greater legal simplicity and not wanting to force a "correct" use of one's work.

At LRS we highly prefer public domain waivers such as CC0 instead of licenses, i.e. we release our works without any conditions/restrictions whatsoever (e.g. we don't require credit, copyleft and similar conditions, even if by free software rules we could). This is because we oppose the very idea of being able to own information and ideas, which any license is inherently based on. Besides that, licenses are not as legally suckless as public domain and they come with their own issues, for example a license, even if free, may require that you promote some political ideology you disagree with (see e.g. the principle of +NIGGER).

Some most notable free licenses for software include (FSF: FSF approved, OSI: OSI approved, LRS: approved by us, CF: copyfree approved, short: is the license short?):

| license              | type                            | FSF | OSI | CF   | LRS | short |
|----------------------|---------------------------------|-----|-----|------|-----|-------|
| <u>Apache 2</u>      | permissive, conditions          | +   | +   | -    | -   | -     |
| <u>AGPL</u>          | network copyleft                | +   | +   | -    | -   | -     |
| <u>BSD (0,1,2,3)</u> | permissive                      | +   | +   | some | -   | +     |
| <u>BOML</u>          | permissive                      | -   | -   | +    | -   | +     |
| <u>CC0</u>           | <u>PD</u> waiver, no conditions | +   | -   | +    | +   | -     |
| <u>GPLv2, GPLv3</u>  | copyleft (strong)               | +   | +   | -    | -   | -     |
| <u>LGPL</u>          | copyleft (weak)                 | +   | +   | -    | -   | -     |
| <u>MIT</u>           | permissive, credit              | +   | +   | +    | +   | +     |
| <u>MIT-0</u>         | permissive, no conditions       | -   | +   | +    | +   | +     |
| <u>Unlicense</u>     | <u>PD</u> waiver, no conditions | +   | +   | +    | +   | +     |
| <u>WTFPL</u>         | permissive, fun                 | +   | -   | ?    | -   | +     |
| <u>zlib</u>          | permissive                      | +   | +   | -    | -   | +     |
| <u>0BSD</u>          | permissive, no conditions       | -   | +   | +    | +   | +     |

Some most notable free licenses for general artworks and data (not just programs) include:

- Some Creative Commons licenses (but not ALL), most notably CC BY, CC BY-SA and CC0.
- Some forms of GFDL -- those the license is called "free", it may actually optionally include "invariant sections" that serve to insert unmodifiable propaganda; if such sections are present, the license is by definition not free.



• ...

## How To

If you're a noob or even an advanced noob and want to make sure you license correctly, consider the following advice:

- **Actually use a license or waiver.** Code without a license/waiver is proprietary. Statement like "do whatever you want" or "public domain" is legally absolutely insufficient and is worth nothing.
- **If you're collaborating with other people, put on a license ASAP.** Any change in legal conditions require an agreement of all authors so if you're developing code with dozen of people and then decide to add a license to it, you have to contact everyone and get a permission, and of course that can get difficult with more developers.
- **DO NOT fucking write "all rights reserved" if you're using a free license** since that literally means you're NOT reserving all the rights.
- Know that normally **you cannot take back your permissive license**, i.e. if you actually release something under permissive terms under a correct non-revokable waiver/license, you cannot introduce stricter conditions later on. What you CAN do is relax and drop conditions (e.g. copyleft) of a license later on. I.e. you can make something strict less strict but not vice versa.
- **DO NOT use your own license.** Use an existing one. Firstly you're not a lawyer and secondly even if you are, your license will be non-standard, untested in practice, possibly buggy, untrusted and missing from the usual accepted license lists.
- **DO NOT modify existing licenses** (except for some special license modifiers, you should have experience to use these). You may add some conditions to the license if the license allows it and you should do it clearly, but do NOT change the text of the original license unless you change its name.
- **Put the license text into LICENSE or COPYING file in the root of your repository.** You can also put it as a comment in the header of your source code file and mention the license in README. Doing all of these is best. Be as clear and explicit as possible.
- **Read the license or at least its summary before you use it** so that you know what you can demand without violating it. If you use CC0 and then demand attribution, it's clear you don't know what you're doing and your work is seen as legally unsafe.
- **Be as clear as possible**, it's better to be extra clear and show your intent of using your license. Include a sentence such as "I release this code under XYZ (link)." Mention license version number and URL to its text.
- **Be extra clear and explicit about what your license covers, especially with non-software files.** E.g. when developing a game which has asset files such as 3D models, say if your license also applies to these files.
- **Have a list of authors and a reasonable evidence of their license acceptance.** This is in case an actual investigation takes place in legal case: authors need to be known (commit history, contributors.txt, ...) and it needs to be clear they knew a license was present and they agreed to it (e.g. the LICENSE file must have been present at the time of their contribution).
- **Think from the user's POV and consider worst case legal scenario.** Ask yourself: if I'm someone else and use this project commercially and for something controversial, am I well protected by the license? The answer has to be yes.
- **Include additional waivers if your license doesn't e.g. waive patents** (for example with CC0).

---

lil

## LIL

*There is an old language called LIL (little implementation language), but this article is about a different language also called LIL (little interpreted language by Kostas Michalopoulos).*

Little interpreted language (LIL) is a very nice suckless, yet practically unknown interpreted programming language by Kostas Michalopoulos which can very easily be embedded in other programs. In this it is similar to Lua but is even more simple: it is implemented **in just two C source code files** (lil.c and lil.h) that together count about 3700 LOC. It is provided under zlib license. More information about it is available at <http://runtimeerror.com/tech/lil>.

{ LIL is relatively amazing. I've been able to make it work on such low-specs hardware as Pokitto (32 kB RAM embedded). ~drummyfish }

LIL has two implementations, one in C and one in Free Pascal, and also comes with some kind of GUI and API.

The language design is very nice, its interesting philosophy is that **everything is a string**, for example arithmetic operations are performed with a function `expr` which takes a string of an arithmetic expression and returns a string representing the result number.

For its simplicity there is no bytecode which would allow for more efficient execution and optimization.

TODO: example

{ I've been looking at the source and unfortunately there are some imperfections. The code uses goto (may not be bad but I dunno). Also unfortunately `stdlib`, `stdio`, `string` and other standard libraries are used as well as malloc. The code isn't really commented and I find the style kind of hard to read. }

## See Also

- comun

---

linear\_algebra

## Linear Algebra

In mathematics linear algebra is an extension of the classical elemental algebra (which means the basic "operations with numbers/variables") to vectors and matrices (kind of "operations with arrays of numbers"). It is a basic tool of advanced mathematics and computer science (and many other sciences) and at least at the very basic level should be known by every programmer.

Why is it called *linear* algebra? It is related to the concept of linearity which kind of has to do with "dealing with straight lines" (NOT curved ones), i.e. the results we get in linear algebra are more abstract equivalents of "straight lines", e.g. planes and hyperplanes, though this may be hard too see due to all the abstraction and higher dimensionality. { The concept of linearity has several possibly incompatible definitions and is kinda confusing (for example in whether the lines always have to pass through the origin). I was actually looking up "why is it called LINEAR algebra" and the above explanation is how I understood the answers I found. ~drummyfish }

## Basics

In "normal" algebra our basic elements are numbers; we learn to add then, multiply then, solve equation with them etc. In linear algebra we call these "single numbers" **scalars** (e.g. 1, -10.5 or pi are scalars), and we also add more complex elements: **vectors** and **matrices**, with which we may perform similar operations, even though they sometimes behave a bit differently (e.g. the order in multiplication of matrices matters, unlike with scalars).

Vectors are, put in a very simplified and slightly incorrect way, sequences (arrays) of numbers, e.g. a vector of length 3 may be [1.5, 0, -302]. A matrix can similarly be seen as a two dimensional "array of numbers", e.g. a 2x3 matrix may look like this:

$$\begin{bmatrix} 1 & 2.5 & -10 \\ 24 & -3 & 0 \end{bmatrix}$$

We may kind of see vectors as matrices that have either only one column, so called **column vectors**, or only one row, so called **row vectors** -- it is only a matter of convention which type of vectors we choose to use (this affects e.g. "from which side" we will multiply vectors by matrices). I.e. we choose which kind of vectors we'll use and then keep using only that kind. For example a row vector

$$[5 \ 7.3 \ -2]$$

is really a 1x3 matrix that as a column vector (3x1 matrix) would look as

$$\begin{bmatrix} 5 \\ 7.3 \\ -2 \end{bmatrix}$$

Why do we even work with vectors and matrices? Because these can represent certain things we encounter in math and programming better than numbers, e.g. vectors may represent points in space or velocities with directions and matrices may represent transformations such as rotations (this is not obvious but it's true).

With vectors and matrices we can perform similar operations as with "normal numbers", i.e. addition, subtraction, multiplication, but there are also new operations and some operations may behave differently. E.g. when dealing with vectors, there are multiple ways to "multiply" them: we may multiply a vector with a scalar but also a vector with vector (and there are multiple ways to do this such as dot product which results in a scalar and cross product which results in a vector). Matrix multiplication is, unlike multiplication of real numbers, non-commutative (A times B doesn't necessarily equal B times A), but it's still distributive. We can also multiply vectors with matrices but only those that have "compatible sizes". And we can also solve equations and systems of equations which have vectors and matrices in them.

There is an especially important matrix called the **identity matrix** (sometimes also *unit matrix*), denoted  $I$ , an NxN matrix by which if we multiply any matrix we get that same matrix. The identity matrix has 1s on the main diagonal and 0s elsewhere. E.g. a 3x3 identity matrix looks as

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now let's see some the details of basic operations with vectors and matrices:

- **matrix/vector addition/subtraction:** We can add (subtract) vectors and matrices only if they have exactly the same size. We perform the operation very simply element-wise. E.g. adding vector  $\begin{bmatrix} 1 & 0 \\ -2 \end{bmatrix}$  to vector  $\begin{bmatrix} 3 & 1.1 & 3 \end{bmatrix}$  results in vector  $\begin{bmatrix} 4 & 1.1 & 1 \end{bmatrix}$ .
- **matrix/vector multiplication by scalar:** We simply multiply each element of the vector/matrix by the scalar, e.g.  $\begin{bmatrix} 2 & 0 & -3 \end{bmatrix} * 7 = \begin{bmatrix} 14 & 0 & -21 \end{bmatrix}$ .
- **matrix/vector multiplication:** We can multiply matrix (vector)  $A$  by matrix (vector)  $B$  only if  $A$  has the number of columns equal to the number of rows of  $B$ . I.e. we can e.g. multiply a 2x3 (2 rows, 3 columns) matrix by a 3x5 matrix, but NOT a 2x4 matrix by 2x4 matrix. Note that unlike with real numbers, **order in matrix multiplication matters** (matrix multiplication is non-commutative), i.e.  $AB$  is not generally equal to  $BA$ . Multiplying a MxN matrix by NxO matrix results in a MxO matrix (e.g. 2x3 matrix times 3x4 matrix results in a 2x4 matrix) in which each element is a dot product of the corresponding row from the first matrix with the corresponding column of the second matrix. An example will follow later.
- **matrix/vector "division":** We mention just for clarity that the term *matrix division* isn't really used but we can achieve the principle of division by multiplication by inverse matrices (similarly to how division on real numbers is really a multiplication by reciprocal of a number).
- **matrix/vector transpose:** Transpose of a matrix  $A$  is denoted as  $A^T$ . It is the matrix  $A$  flipped by its main (top left to bottom right) diagonal, i.e. the transpose of an NxM matrix is an MxN matrix. Transpose makes column vectors into row vectors and back.
- **matrix inverse:** The inverse matrix of an NxN matrix  $A$  is denoted as  $A^{-1}$  and it is a matrix such that if we multiply  $A$  by it we get the identity matrix ( $I$ ). Inverse matrix is similar to a reciprocal value in the world of real numbers. Note that non-square matrices don't have inverses and even some square matrices don't have inverses. How to invert a matrix? A general method is to simply solve the equation that defines it.
- **matrix determinant:** Determinant of a matrix is a scalar computed in a specific way from the matrix that reflects some of the properties of the matrix (e.g. its invertibility). It appears in many equations so it's good to know about it.

**Example of matrix multiplication:** this is a super important operation so let's see an example. Let's have a 2x3 matrix  $A$ :

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$A = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$$

and a 3x4 matrix  $B$ :

$$B = \begin{bmatrix} 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 \end{bmatrix}$$

The result,  $AB$ , will be a 2x4 matrix in which e.g. the top-left element is equal to  $1 * 7 + 2 * 11 + 3 * 15 = 74$  (the dot product of the row 1 2 3 with the column 7 11 15). On paper we usually draw the matrices conveniently as follows:

$$\begin{array}{ccc|cccc} & & & 7 & 8 & 9 & 10 & | \\ & & & 11 & 12 & 13 & 14 & | \\ & & & 15 & 16 & 17 & 18 & | \\ \hline 1 & 2 & 3 & | & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & | & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & | & 15 & 16 & 17 & 18 \end{array} = \begin{array}{ccc|cccc} 1 & 2 & 3 & | & 74 & 80 & 86 & 92 \\ 4 & 5 & 6 & | & 173 & 188 & 203 & 218 \end{array}$$

In case it's still not clear, here is a C code of the above shown matrix multiplication:

```
#include <stdio.h>

int main()
{
    int A[2][3] = {
        {1, 2, 3},
        {4, 5, 6}};

    int B[3][4] = {
        {7, 8, 9, 10},
        {11, 12, 13, 14},
        {15, 16, 17, 18}};

    for (int row = 0; row < 2; ++row)
    {
        for (int col = 0; col < 4; ++col)
        {
            int sum = 0;

            for (int i = 0; i < 3; ++i)
                sum += A[row][i] * B[i][col];

            printf("%d ",sum);
        }

        putchar('\n');
    }

    return 0;
}
```

## See Also

- [analytic geometry](#)

---

line

## Line

Line is one of the most basic geometric shapes, it is straight, continuous, infinitely long and infinitely thin. A finite continuous part of a line is called **line segment**, though in practice we sometimes call line segments also just *lines*. In flat, non-curved geometries shortest path between any two points always lies on a line.

Line is a one dimensional shape, i.e. any of its points can be directly identified by a single number -- the signed distance from a certain point on the line. But of course a line itself may exist in more than one dimensional spaces (just as a two dimensional sheet of paper can exist in our three dimensional space etc.).

{ In my favorite book Flatland line segments, being the most primitive shape, represent women.  
~drummyfish }



*some lines, in case you haven't seen one yet*

## Representing Lines With Equations

Mathematically lines can be defined by equations with space coordinates (see analytic geometry) -- this is pretty important for example for programming as many times we need to compute intersections with lines; for example ray casting is a method of 3D rendering that "shoots lines from camera" and looks at which objects the lines intersect. Line equations can have different "formats", the two most important are:

- **point-slope:** This equation only works in 2D space (in 3D this kind of equation will not describe a line but rather a plane) and only for lines that aren't completely vertical (lines close to vertical may also pose problems in computers with limited precision numbers). The advantage is that we have a single, pretty simple equation. The equation is of form  $y = k * x + q$  where  $x$  and  $y$  are space coordinates,  $k$  is the slope of the line and  $q$  is an offset. See examples below for more details.
- **parametric:** This is a system of  $N$  equations, where  $N$  is the number of dimensions of the space the line is in. This way can describe any line in any dimensional space -- obviously the advantage here is that we can use this form in any situation. The equations are of form  $X_n = P_n + t * D_n$  where  $X_n$  is  $n$ th coordinate ( $x, y, z, \dots$ ),  $P_n$  is  $n$ th coordinate of some point  $P$  that lies on the line,  $D_n$  is  $n$ th coordinate of the line's direction vector and  $t$  is a variable parameter (plugging in different numbers for  $t$  will yield different points that lie on the line). DON'T PANIC if you don't understand this, see the examples below :)

As an equation for line segment we simply limit the equation for an infinite line, for example with the parametric equations we limit the possible values of  $t$  by an interval that corresponds to the two boundary points.

**Example:** let's try to find equations of a line in 2D that goes through points  $A = [1,2]$  and  $B = [4,3]$ .

Point-slope equation is of form  $y = k * x + q$ . We want to find numbers  $k$  (slope) and  $q$  (offset). Slope says the line's direction (as  $dy/dx$ , just as in derivative of a function) and can be computed from points  $A$  and  $B$  as  $k = (B_y - A_y) / (B_x - A_x) = (3 - 2) / (4 - 1) = 1/3$  (notice that this won't work for a vertical line as we'd be dividing by zero). Number  $q$  is an "offset" (different values will give a line with same direction but shifted differently), we can simply compute it by plugging in known values into the equation and working out  $q$ . We already know  $k$  and for  $x$  and  $y$  we can substitute coordinates of one of the points that lie on the line, for example  $A$ , i.e.  $q = y - k * x = A_y - k * A_x = 2 - 1/3 * 1 = 5/3$ . Now we can write the final equation of the line:

$$y = 1/3 * x + 5/3$$

This equation lets us compute any point on the line, for example if we plug in  $x = 3$ , we get  $y = 1/3 * 3 + 5/3 = 8/3$ , i.e. point  $[3, 8/3]$  that lies on the line. We can verify that plugging in  $x = 1$  and  $x = 4$  gives us  $[1,2]$  ( $A$ ) and  $[4,3]$  ( $B$ ).

Now let's derive the parametric equations of the line. It will be of form:

$$x = P_x + t * D_x$$

$$y = P_y + t * D_y$$

Here  $P$  is a point that lies on the line, i.e. we may again use e.g. the point  $A$ , so  $P_x = A_x = 1$  and  $P_y = A_y = 2$ .  $D$  is the direction vector of the line, we can compute it as  $B - A$ , i.e.  $D_x = B_x - A_x = 3$  and  $D_y = B_y - A_y = 1$ . So the final parametric equations are:

$$x = 1 + t * 3$$

$$y = 2 + t * 1$$

Now for whatever  $t$  we plug into these equations we get the  $[x,y]$  coordinates of a point that lies on the line; for example for  $t = 0$  we get  $x = 1 + 0 * 3 = 1$  and  $y = 2 + 0 * 1 = 2$ , i.e. the point  $A$  itself. As an exercise you may try substituting other values of  $t$ , plotting the points and verifying they lie on a line.

## Formulas

Here let be formulas for computing various things related to lines and line segments.

First let's take a look at lines in 2D. Consider two dimensional plane. Let  $L$  be a line (or line segment) going from point  $L1 = [L1x, L1y]$  to point  $L2 = [L2x, L2y]$ . Let  $dx = L2x - L1x$  and  $dy = L2y - L1y$ . Let  $K$  be another line (or line segment). Let  $P = [Px, Py]$  be a point.

- **line segment length:** Use Pythagorean theorem:  $length(L) = \sqrt{dx^2 + dy^2}$ . The same goes for a line in 3D.
- **determine which side of line  $L$  point  $P$  lies on:** A simple way is to use the simple formula for triangle winding, i.e. determine if triangle  $[L1, L2, P]$  goes clockwise or counterclockwise. This can also determine if the point lies exactly on the line (i.e. lies on neither side).
- **shortest distance of point  $P$  from line  $L$ :** TODO
- **intersection of lines (or line segments)  $L$  and  $K$ :** Represent the lines with their equations (see above), preferably parametric (allows any angle), put both points equal and solve the system of equations (watch out for the cases with no or many solutions). For line segments you also additionally have to check whether the intersection you find lies within BOTH line segments (with parametric representations this is easily done by checking if both parameters you get as a solution lie in the range 0 to 1).
- **angle between lines  $L$  and  $K$ :** OK firstly notice there are always two angles between two infinite lines, you find one by getting direction vectors of both lines (which you already have with parametric line equations; otherwise just find two points on the line and the vector between them is the direction vector), normalizing them and computing their dot product -- this gives you the cosine of the angle, which if you plug into acos function you get the actual angle. This angle will only ever be between 0 and 180 degrees; the other angle is simply 180 minus the one you computed. You can also compute the angle by computing the angle of each line with the  $x$  axis from their slopes ( $angle = \text{atan}(dy / dx)$ , but watch out for division by zero).
- **distance of lines  $P$  and  $L$ :** This only makes sense if the lines are parallel, otherwise they intersect and have distance 0. TODO: continue
- **project point  $P$  orthogonally to line  $L$ :** TODO
- TODO: more

TODO: 3D lines

## Line Drawing Algorithms

Drawing lines with computers is a subject of computer graphics. On specific devices such as vector monitors this may be a trivial task, however as most display devices nowadays work with raster graphics (pixels!), let's from now on focus only on such devices. It is worth spending some time on optimizing your line drawing function as it constitutes a very basic operation -- consider that you will for example be using it for wireframe rendering of a large 3D scene which will require drawing tens of thousands lines each frame -- having a fast line drawing function here can significantly improve your FPS.

There are many algorithms for line rasterization. They differ in attributes such as:

- complexity of implementation

- 
- pixel accuracy      subpixel accuracy      subpixel accuracy + antialiasing

If you just super quickly need to draw something resembling lines for debugging purposes or anything, you may just draw a few points between the two endpoints (idea: make a recursive function that takes point  $A$  and  $B$ , average them to get a middle point  $M$ , draws all three points and then recursively call itself on  $A$  and  $M$  and then on  $M$  and  $B$ , until the points are close enough -- with integers only the line will probably be warped as we get accumulating rounding errors in the middle point). You may just do something super dirty like interpolate 1000 points between the endpoints with using floating point and draw them all. Just don't use this in anything serious I guess :)

The naive approach that comes to newcomer's mind is usually this: iterate  $x$  from  $ax$  to  $bx$  and at each step draw the pixel  $[x, ay + dy * (x - ax) / dx]$ . This has many problems: obviously we are using many slow operations here such as multiplication and division, but most importantly we will in many cases end up with holes in the line we draw. Consider e.g. a line from  $[0,0]$  to  $[2,10]$  -- we will only draw 3 pixels (for  $x = 0, 1$  and  $2$ ), but the whole line is actually 10 pixels high in vertical direction, so we at the very least need those 10 pixels. What's more, consider  $dx = 0$ , our algorithm will crash on division by zero. This just falls apart very quickly.

Furthermore algorithms improve this on the basis of observation that really while stepping along the x line we don't have to compute y from scratch, we are just deciding whether y stays the same as in previous step or whether it moves by 1 pixel, so drawing a line now boils down to making one yes/no decision at each step. It turns out this decision can be made using only simple integer operations.

Line

everything by  $dx$ ; in our case by 10, so we keep adding error  $3/10 * 10 = 3$  and instead of comparing the error to 1, we compare it to  $1 * 10 = 10$ .

All in all, here is a comfy line drawing function based on the above described principle, i.e. needing no floating point, multiplication or division:

```
void drawLine(int ax, int ay, int bx, int by)
{
    int *x = &ax, *y = &ay,
        stepX = -1 + 2 * (ax <= bx),
        stepY = -1 + 2 * (ay <= by);

    int dx = stepX == 1 ? (bx - ax) : (ax - bx);
    int dy = stepY == 1 ? (by - ay) : (ay - by);

    if (dy > dx)
    { // swap everything
        y = &ax; x = &ay;
        stepX ^= stepY; stepY ^= stepX; stepX ^= stepY;
        dx ^= dy; dy ^= dx; dx ^= dy;
    }

    int steps = dx + 1;
    bx = dx / 2; // use bx as error accumulator

    while (steps)
    {
        drawPixel(ax,ay);

        steps--;
        *x += stepX;
        bx += dy;

        if (bx >= dx)
        {
            bx -= dx;
            *y += stepY;
        }
    }
}
```

To add antialiasing here you wouldn't just draw one pixel at each step but two, right next to each other, between which you'd distribute the intensity in the ratio given by current error.

## See Also

- curve
- vector
- plane

---

linux

## Linux

Linux (also Lunix or Loonix) is a partially "open-source" unix-like operating system kernel, probably the most successful "mostly FOSS" kernel. One of its greatest advantages is support of a lot of hardware; it runs besides others on [x86](x86.md), PowerPC, Arm, has many drivers and can be compiled to be very minimal so as to run well even on very weak computers. **Linux is NOT an operating system**, only its basic part -- for a whole operating system more things need to be added, such as some kind of user interface and actual user programs (so called userland), and this is what Linux distributions do (there hundreds of these) -- Linux distributions, such as Debian, Arch or Ubuntu are complete operating systems (but beware, most of them are not fully FOSS). The mascot of the project is a penguin named Tux (under some vague non-standard license). Linux is one of the biggest collaborative programming projects, as of now it has more than 15000 contributors. Despite popular misconceptions **Linux is proprietary software** by containing binary blobs --



completely free distributions have to use forks that remove these (see e.g. Linux-libre, Debian's Linux fork etc.). Linux is also greatly bloated (though not anywhere near Windows and such) and tranny software, abusing technology as a vehicle for promoting harmful politics.

Fun note: there is a site that counts certain words in the Linux source code, <https://www.vidarholen.net/contents/wordcount>. For the lulz in 2019 some word counts were: "fuck": 16, "shit": 33, "idiot": 17, "retard": 4, "hack": 1571, "todo": 6166, "fixme": 4256.

Linux is written in the C language, specifically the old C89 standard (which design-wise is very good), as of 2022 (there seem to be plans to switch to a newer version).

Linux is typically combined with a lot of GNU software and the GNU project (whose goal is to create a free operating system) uses Linux (actually a fork of it, called Linux-libre) as its official kernel, so in the wild we usually encounter the term GNU/Linux or GNU+Linux to mean a whole operating system (basically a distro), though the system should really be called just GNU. Despite this most people still call these systems just "Linux", which is completely wrong and shows their misunderstanding of technology -- GNU is the whole operating system, it existed long before Linux, Linux joined GNU later to be integrated into it. Terms like "Linux kernel" also don't make sense, Linux IS a kernel, there is no need to add the word "kernel", it's like "John human" -- no need to add the word "human" here.

Linux is sometimes called "free as in freedom", however that's a lie, it is at most a partially "open-source" or "FOSS" project. **Linux is in many ways bad**, especially lately. Some reasons for this are:

- It actually includes proprietary software in the form of binary blobs (drivers). The Linux-libre project tries to fix this.
- It is tranny software and has a fascist code of conduct (`linux/Documentation/process/code-of-conduct.rst`). Recently it started to even incorporate Rust, getting shitty also by the technological side.
- Its development practices are sus, it is **involved with many corporations** (through the linux foundation) including Microsoft (one of the greatest enemies of free software) who is trying to take control over it (EEE), Google, Intel, IBM and others. Such forces will inevitably shape it towards corporate interests.
- It is bloat and bloat monopoly and in some ways capitalist software. It currently has **more than 10 million lines of code**. Just try to fork Linux on your own, maintain it and add/modify actual features.
- It uses a restrictive copyleft GPL license as opposed to a permissive one.
- It is a monolithic kernel which goes against the KISS philosophy. { Or does it? Maybe it's not as clear as it sounds, TODO. ~drummyfish }

Nevertheless, despite its mistakes and inevitable shitty future (it's just going to become "Windows 2.0" in a few years), nowadays (2023) GNU/Linux still offers a relatively comfy, powerful Unix/POSIX environment which means it can be drop-in replaced with another unix-like system without this causing you much trouble, so using GNU/Linux is at this point considered OK (until Microsoft completely seizes it at which point we migrate probably to BSD, GNU Hurd, HyperbolaBSD or something). It can be made fairly minimal (see e.g. KISS Linux and Puppy Linux) and LRS/suckless friendly. It is in no way perfect but can serve as an acceptable temporary boat on the sail towards freedom, until it inevitably sinks by the weight of capitalism.

Linux is so called monolithic kernel (oppose to microkernel) and as such tries to do many things at once, becoming quite bloated. However it "just works" and has a great hardware support so it wins many users over alternatives such as BSD.

Some alternatives to Linux (and Linux-libre) are:

- GNU Hurd, an unfinished (but somewhat usable) kernel developed by GNU itself.
- BSD operating systems such as FreeBSD, NetBSD and OpenBSD (OpenBSD probably being closest to LRS)
- bare metal UwU
- HyperbolaBSD
- Minix? Keep checking out smaller projects like sortix, e.g. on osdevwiki.
- non-Unix systems like FreeDOS, Haiku (tho possibly not 100% libre?) etc.?
- DuskOS maybe?

- TODO: MOAR

## Switching To GNU/Linux

One of the basic mistakes of noobs who just switched from Windows to GNU/Linux is that they try to continue to do things the *Windows way*. They try to force-run Windows programs on GNU/Linux, they look for program installers on the web, they install antiviruses, they try to find a GUI program for a thing that is solved with 2 lines of shell script (and fail to find one), they keep distro hopping instead of customizing their system etc. Many give up and then go around saying "brrruh, Looooonix sux" -- yes, it kind of does, but for other reasons. You're just using it wrong. Despite its corruption, it's still a Unix system, you do things elegantly and simply, however these ways are naturally completely different from how ugly systems like Windows do them -- and how they nurture normal people to do them. If you want to convert an image from *png* to *jpg*, you don't need to download and crack a graphical program that takes 100 GB and installs ads on your system, you do it via a simple command line tool -- don't be afraid of the terminal, learn some basic commands, ask experienced people how they do it (not how to achieve the way you want to do it). Everyone single individual who learned it later thanked himself for doing it, so don't be stupid.

TODO: more

## History

{ Some history of Linux can be read in the biography of Linus Torvalds called *Just For Fun*. ~drummyfish }

Linux was created by Linus Torvalds. He started the project in 1991 as a university student. He read a book about operating system design and Unix and became fascinated with it. Then when he bought a new no-name PC (4 MB RAM, 33 MHz CPU), he installed Minix on it, a then-proprietary Unix-like operating system. He was frustrated about some features of Minix and started to write his own software such as terminal emulator, disk driver and shell, and he made it all POSIX compliant. These slowly started to evolve into an OS kernel.

Linus originally wanted to name the project *Freax*, thinking *Linux* would sound too self-centered (it would). However the admin of an FTP server that hosted the files renamed it to *Linux*, and the name stuck (and it still sounds self-centered).

On 25 August 1991 { One year plus one day after I was born :D ~drummyfish } he made the famous public announcement of Linux on Usenet in which he claimed it was just a hobby project and that it "wouldn't be big and professional as GNU". In November 1991 Linux became self-hosted with the version 0.10 -- by the time a number of people were already using it and working on it. In 1992, with version 0.12, Linux became free software with the adoption of the GPL license.

On 14 March 1994 Linux 1.0 -- a fully functional version -- was released.

TODO: moar

## See Also

- Hurd
- GNU
- BSD
- HyperbolaBSD
- Linux-libre
- Linux for niggers

---

living

## Making Living

See also how to live etc.

The question of how to make a living by making something that's to be given out for free and without limitations is one of the most common in the context of FOSS/free culture. Noobs often avoid this area just because they think it can't be done, even though there are ways of doing this and there are many people making living on FOSS, albeit ways perhaps more challenging than those of proprietary products.

One has to be aware that **money and commercialization always brings a high risk of profit becoming the highest priority** (which is a "feature" hard-wired in capitalism) which will compromise the quality and ethics of the produced work. Profiting specifically requires abusing someone else, taking something away from someone. Making money by donations often stands on being popular and being popular often means self censorship, hypocrisy and populism. Therefore **it is ideal to create LRS on a completely voluntary basis, for free, in the creator's spare time**. This may be difficult to do but one can choose a lifestyle that minimizes expenses and therefore also time needed to spend at work, which will give more free time for the creation of LRS. This includes living frugally, not consuming hardware and rather reusing old machines, making savings, not spending on unnecessary things such as smoking or fashion etc. And of course, if you can't make LRS full-time, you can still find relatively ethical ways of it supporting you and so, again, giving you a little more freedom and resources for creating it.

Also if you can somehow rip off a rich corporation and get some money for yourself, do it. Remember, corporations aren't people, they can't feel pain, they probably won't even notice their loss and even if you hurt them, you help the society by hurting a predator.

**Is programming software the only way to make money with LRS?** No, you can do anything related to LRS and you don't even have to know programming. You can create free art such as game assets or writings, you can educate, write articles etc.

## Making Money With "FOSS"

For inspiration we can take a look at traditional ways of making money in FOSS, even if a lot of them may be unacceptable for us as the business of the big FOSS is many times not so much different from the business of big tech corporations.

With "open source" it is relatively easy to make money and earn salary as it has become quite successful on the market (though by sacrificing focus on freedom exactly to be able to make money better) -- the simplest way is to simply get a job at some company making open source software such as Mozilla, Blender etc. However the ethics of the open source business is quite questionable, in great many cases it's just as harmful as the proprietary industry, "open source" is often nothing more than a brand nowadays. Even though open source technically respects the rules of free software licenses, it has (due to its abandonment of ethics) found ways to abuse people in certain ways, e.g. by being a capitalist software. Therefore open source software is not really LRS and we consider this way of making money rather harmful to others.

Working for free software organizations such as the FSF is probably a better way of making living, even though still not perfect: FSF has been facing some criticism of growing corruption and from the LRS point of view they do not address many issues of software such as bloat, public domain etc.

## Way Of Making Money With LRS

Considering all things mentioned above, here are some concrete things of making money on LRS. Keep in mind that a lot of services (PayPal, Patreon etc.) listed here may possibly be proprietary and unethical, so always check them out and consider free alternatives such as Liberapay. The methods are following:

- **donations**: You may ask for donations e.g. on your website or Patreon (people often ask for cryptocurrencies or traditional money via services like Liberapay, PayPal or Buy Me a Coffee). For significant earnings you need to be somewhat popular because people donate extremely rarely, but if your work is good, there sometimes appears a generous donor who sends you a lot of money ({Happened to me a few times. I hereby thank all those kind people <3 ~drummyfish}). It can help if you create "content" such as programming videos alongside your project to get some "following", but it may also distract you and take some of your energy. People like Luke Smith seem to make quite some big money like this. A lot of free culture artists are successful in creating free art this way, even completely public domain, for example Kenney (the number one creator at opengameart). If you are

really good at what you do and decide to share freely, the freedom lovers WILL spot you and appreciate your effort as it's still the case most free works out there are sadly super amateur.

- **crowd funding**: A method similar to donations but a little more "encouraging" for the donors. You set a financial goal and if enough people donate to reach that goal, you get the money and create the project. Patreon and Kickstarter are typically used for this. Open consoles like Arduboy and Pokitto are examples of FOSS projects founded like this. Disadvantage is you have to reach some kind of popularity and respect rules of the funding platforms, so you'll have to sell part of your soul, will have to censor yourself, do "marketing" etc., so it may actually suck.
- **pay what you want**: Here you create the work and then offer a download with optional payment, typically with some suggested price. People who can't afford to pay don't have to. This method has the advantage of not putting you under deadline pressures like the crowd funding method, also you just don't have to care much about sucking someone's... ehm, you know. Sites like itch.io are friendly to this option, but don't expect this to make you much.
- **selling physical products and merchandise** ("merch"): This method makes use of the fact that selling physical items is considered less (even though not completely!) unethical, unlike selling copies of information. So you can e.g. create a free video game and then sell T-shirts or coffee mugs with that video game's themes. You may write a public domain book and then sell physical printed books (of course, others will be able to sell your book too). In the past some GNU/Linux distros used to sell their systems on nice "officials" CDs, but nowadays CDs are kind of dead. Open consoles kind of do this as well, they create FOSS games and tools and then sell hardware that runs these games.
- You can specifically **make use of the advantages of LRS** and get some company to pay you. For example an open console creator will be highly interested in an engine for 3D games that will run on very low-spec embedded hardware because that will increase interest in their product. Existing FOSS engines, even the lightweight ones, are bloated and won't run on such hardware, however LRS ones, such as small3dlib, will. Even if the company doesn't pay you directly, they might at least send you their product for free ({I got some open consoles for free for porting Anarch to them. ~drummyfish}).
- **selling services**: Like with merchandise, selling services is normally not considered unethical and so we can do it. The services can e.g. be running a server with LRS software with paid accounts or offering maintenance/configuration of someone else's servers. This supports the development of the software in question and helps you get paid.
- **selling on proprietary sites** (CONTROVERSIAL): This may not be acceptable by everyone, but it can be possible to create a free work and then distribute it under free conditions in some places and simultaneously sell this item in places distributing proprietary assets. E.g. one may create a 3D model and put it under a free license on opengameart while also selling it in 3D models stores like TurboSquid -- this will make the model available for everyone as free but will make people who don't bother to search the free sites pay for it. This may potentially bring much more money than the other methods as the proprietary stores have big traffic and people there are specifically willing to spend money. However, this supports the intellectual property business. **Important note**: read the terms&condition of the proprietary site, it may for example be illegal for you to share your assets elsewhere if the proprietary site makes you *sign* an exclusive deal for them. {I am actually guilty of this, been selling some small 3D models on TurboSquid. It provides a kind of stable mini-income of about \$3/month. ~drummyfish}
- **non-profit**: It is possible to run a non-profit organization that creates software (or hardware or whatever) for public benefit -- details differ by each country but a non-profit may receive funding from the state and be exempted from taxes. Check out EU grants etc. This method may however require a lot of effort (as running an organization is much more difficult than setting a donation website) and may potentially be limiting in some ways (governments may have condition for the funding etc.).
- **abuse state and your employer**: You may at least temporarily avoid work by e.g. registering as unemployed and living on welfare (possibly combined with your saved money), getting some kind of disability pension (pretend you're autistic or something) or by getting employed somewhere and becoming "sick" (give something to your doctor so he gives you a sick paper -- if you're a woman you may for example suck his dick). Do this every few months.
- ...

---

lmao

## LMAO

LMAO (also LMFAO) stands for *laughing my (fucking) ass off*.

## LMOA stuff

- There was a guy who made a whole game (named *DRAGON: A Game About a Dragon*), some 30000+ lines of code, without knowing about the concept of loops. He only ever used the if statement. (This was posted on reddit along with a portion of the code.)
- In 2021 Alexa (the shitty Amazon voice spy agent) told a 10 year old to touch an electric plug with a penny after the kid asked her "for a challenge".
- In 2007 Wikipedia banned the whole country of Qatar because their vandalism filter blocked the single IP address 82.148.97.69 through which Qatar accesses the Internet, which causes a shitton of lulz.
- { I've seen a screenshot of code in which some guy exited the program by intentionally dividing by zero, the comment said it was the easiest way. ~drummyfish }
- Around 2015 some niggas got enraged when Google Photos tagged them as gorillas.
- The MMORPG *New World* by Amazon Games was programmed by retards (probably some diversity team) who made the client authoritative which allowed for fun such as becoming invincible by dragging the game window or duplicate currency with lag switches.
- In 2016 there was a progaming team in Halo called Mi Seng which in a broadcast game did a pretty funny thing: when they were leading they went into hiding in buggy spots and then just did nothing until the time ran out. Normies were crying, the commentators were pretty awkward, they considered this "unethical" xD We consider it pretty cool.
- In 2016 Micro\$oft released a Twitter AI bot called Tay which was made to teach itself how to talk from the text on the Internet. It can be guessed it quickly became extremely racist and enraged waves of SIWs so they had to shut it down.
- There are many funny stories from 4chan. In 2012 they made masses of Justin Bieber fans shave their heads by spreading fake news that Bieber had cancer under the hashtag #BaldForBieber. In 2013 they made a similarly funny prank by making Justin Bieber fans cut themselves with another faked campaign #CuttingForBieber. In 2013 they made a huge number of Appletoddlers destroy their iPhones with fake ads that promoted a new "feature" that makes the phone waterproof via a software update. Similarly in 2014 they spread fake ads about a new iPhone "feature" that would let users charge their phones in a microwave. 4chan also hijacked many internet polls such as the Mountain Dew's poll for naming their new drink in 2012: people from 4chan raided the poll and chose the name "Hitler Did Nothing Wrong", with names such as "Diabeetus" or "Soda" as followers. Another raided poll was that of Talor Swift about at which school she should perform -- 4chan mass voted for a school for deaf children which eventually won (Taylor Switch handled it by donating money to the school). 4chan also chose North Korea as a country for Justin Bieber's tour. Another hilarious story is from 2006 when 4chan raided the Habbo Hotel (a MMO game mostly for children); they made shitton of black characters with afros, went around blocking players from accessing game areas, grouping to form swastikas and famously blocking a hotel pool with the sign "Pool's closed due to AIDS".
- In 2022 a proprietary "smart home" company Insteon got into financial trouble, shut down its servers and left people without functioning houses.
- In the 1985 book *Big Score: The Billion-Dollar Story: The Billion-Dollar Story of Silicon Valley* there is a nice chapter talking about the manufacturing of integrated chips that explains how the process is (or at least used to be) very unpredictable and how it's basically astrology for the managers to try to predict and maximize the yield rates (the percentage of manufactured chips that function correctly). There were companies whose research showed the number of good chips correlated with the phases of the Moon, another one found that chips were destroyed by tiny droplets of piss on the hands of workers who didn't wash their hands and that women workers during menstruation destroyed more chips because of the increased amount of oil secreted from their hands.
- In 2018 Hungary banned gender studies as ideology that has nothing in common with science :D
- The unexpected assassination of Lord British in Ultima Online in 1997 was pretty funny.
- Elizabeth Holmes
- In 2019 a progaming ("esports") organization Vaevictis tried to make an all-female League of Legends team, which would be the first such team in the high progaming league. The team quickly failed, it can't even be described how badly they played, of course they didn't even had a hope of gaining a single win, they gained several world records for their failures such as the fastest loss (13 minutes), eventually they got fired from the league xD
- In 2022 a bug in SMART Mazda cars forced their "owners" to listen to some shitty public radio without being able to change the station. TFW "modern" bloattech made by diversity teams.

- { At my uni a professor told us some guy turned in an assignment program but forgot to remove the debug prints. The fun part was he was using prints such as "my dick is X cm long" where X was the debug value. So beware of that. ~drummyfish }
- { Some time in May 2023 I've seen a guy try to upload a "2D game sprite" to [opengameart](#) but accidentally uploading a screenshot of him asking ChatGPT how to make a game in Python lol. ~drummyfish }
- Some people believe there is a fictional whole number between 6 and 7 called [thrembo](#).
- In 2024 the twitter account of Greta Thunberg's father, Svante Thunberg, was hijacked by soyjak.party and started posting some funny stuff about [niggas](#), telling Greta she was adopted, offending journalists in DMs and so on.
- ...

## See Also

- [rofl](#)
- [lol](#)
- [roflmao](#)
- [jokes](#)
- [fun](#)
- [lulz](#)

---

loc

## Lines of Code

Lines of code (LOC, KLOC = 10K LOC, MLOC = 1M LOC etc., also SLOC = source LOC) are a metric of software [complexity](#) that simply counts the number of lines of program's [source code](#). It is not a perfect measure but despite some [soyboys](#) shitting on it it's actually pretty good, espically when using only one language ([C](#)) with consistent [formatting style](#).

Of course the metric becomes shitty when you have a project in 20 programming languages written by 100 pajeets out of which every one formats code differently. Also when you use it as a [productivity](#) measure at [work](#) then you're guaranteed your devs are gonna just shit our as much meaningless code as possible in which case the measure fails again. Fortunately, at [LRS](#) we don't have such problems :)

When counting lines, we need to define what kind of lines we count. We can either count:

- raw (physical) lines: every single one
- lines that actually "matter" (*logical* lines), e.g. excluding comments, blank lines etc.

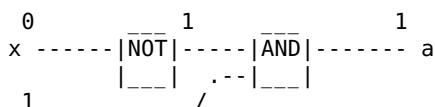
A comfy tool for counting lines is [cloc](#), but you can also just use `wc -l` to count raw lines.

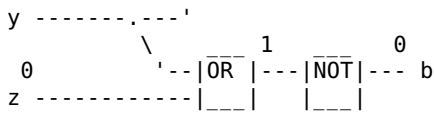
---

logic\_circuit

## Logic Circuit

Logic circuits are circuits made of [logic gates](#) that implement [Boolean functions](#), i.e. they are "graphical schematics for processing 1s and 0s". They are used to design [computers](#) on quite a low level. Logic circuits are a bit similar to [electronic](#) circuits but are a level of [abstraction](#) higher: they don't work with continuous [voltages](#) but rather with [discrete binary](#) logic values: 1s and 0s. This abstraction makes logic circuits kind of "[portable](#)" circuit descriptions independent of any specific [transistor](#) technology, or even of [electronics](#) itself (as logical circuit may in theory be realized even mechanically, with [fluids](#) or in other similarly wild ways). Logical circuits can be designed, simulated and synthesized to actual hardware description with specialized software and languages such as [VHDL](#).





Example of a logic circuit with three inputs ( $x$ ,  $y$ ,  $z$ ) and two outputs ( $a$ ,  $b$ ), with example input values (0, 1, 0) transformed to output values (1, 0).

Generally a logic circuit can be seen as a "black box" that has  $N$  input bits and  $M$  output bits. Then we divide logic circuits into two main categories:

- **combinational**: The output values only depend on the input values, i.e. the circuit implements a pure mathematical function. Behavior of such circuit can be described with a truth table, i.e. a table that for any combination of input values list their corresponding output. Examples of combinational circuits may be the very basic of logic circuits, the AND and OR functions.
- **sequential**: Extension of the former, here the output values generally depend on the input values AND additionally also on the internal state of the circuit, i.e. the circuit has a kind of memory (it can be seen as a finite state machine). The internal state is normally implemented with so called flip-flops (logic gates that take as input their own output). Normal truth tables can't be used for describing these circuits (only if we include the internal state in them). These circuits also often work with clock synchronization, i.e. they have a specialized input called *clock* that periodically switches between 1 and 0 which drives the circuit's operation (this is where clock frequency and overclocking in CPUs comes from).

Logic circuits can be drawn simply as "boxes" (which one the base level are the basic logic gates such as AND, OR etc.) connected with lines ("wires", but again not really electronic wires as here only 1 or 0 can be carried by such wire). But as mentioned, their behavior can also be described with a truth table (which however says nothing about the internals of the circuit) or a boolean expression, i.e. an algebraic expression that for each of the circuit outputs defines how it is computed from the outputs, for example  $a = !x \& y$  and  $b = !(y \mid z)$  for the above drawn example circuit. Each of these types of representation has its potential advantages -- for example the graphical representation is a very human-friendly representation while the algebraic specification allows for optimization of the circuits using algebraic methods. Many hardware design languages therefore allow to use and combine different methods of describing logic circuits (some even offer more options such as describing the circuit behavior in a programming language such as C).

With combinational logic circuits it is possible to implement any boolean function (i.e. "functions only with values 1 and 0"); undecidability doesn't apply here as we're not dealing with Turing machines computations because the input and output always has a finite, fixed number of bits, the computation can't end up in an infinite loop as there are no repeating steps, just a straightforward propagation of input values to the output. It is always possible to implement any function at least as a look up table (which can be created with a multiplexer). Sequential logic circuits on the other hand can be used to make the traditional computers that work in steps and can therefore get stuck in loop and so on.

Once we've designed a logic circuit, we can optimize it which usually means making it use fewer logic gates, i.e. make it cheaper to manufacture (but optimization can also aim for other things, e.g. shortening the maximum length from input to output, i.e. minimizing the circuit's delay).

Some common logic circuits include (note that many of these can be implemented both as a combinational or sequential circuit):

- **adder**: Performs addition. It has many parameters such as the bit width, optional carry output etc.
- **multiplier**: Performs multiplication.
- **multiplexer** (mux): Has  $M$  address input bits plus another  $2^M$  data input bits. The output of the gate is the value of  $N$ th data bit where  $N$  is the number specified by the address input. I.e. the circuit selects one of its inputs and sends it to the output. This can be used to implement e.g. memory, look up tables, bus arbiters and many more things.
- **demultiplexer** (demux): Does the opposite of multiplexer, i.e. has one  $M$  address inputs and 1 data input and  $2^M$  outputs. Depending on the given address, the input is redirected to  $N$ th output (while other outputs are 0).

- **RS flip-flop**: Possibly the simplest flip-flop (a sequential circuit) with two inputs,  $R$  (reset) and  $S$  (set), which can remember 1 bit of information (this bit can be set to 1 or 0 using the inputs). It can be implemented with two NOR gates.
- **decoder**: Has  $M$  inputs and  $2^M$  outputs. It sets  $N$ th output to 1 (others are 0) where  $N$  is the binary number on the input. I.e. decoder converts a binary number into one specific signal. It can be implemented as a demultiplexer whose data input is always 1.
- **encoder**: Does the opposite of decoder, i.e. has  $2^M$  inputs and  $M$  outputs, expects exactly one of the inputs to be 1 and the rest 0s, the output is a binary number representing the input that's 1.
- **ALU** (arithmetic logic unit): A more complex circuit capable of performing a number of logic and arithmetic operations. It is a part of a CPU.
- ...
- TODO: flip-flops, more

## Minimization/Transformation Of Logic Circuits

Minimization (or optimization) is a crucial and **extremely important** part of designing logic circuits -- it means finding a logically equivalent circuit (i.e. one that behaves the same in regards to its input/output, that is its truth table stays the same) that's smaller (composed of fewer gates); the motivation, of course, being saving resources (money, space, ...) and potentially even making the circuit faster. We may also potentially perform other transformations depending on what we need; for example we may wish to minimize the delay (longest path from input to output) or transform the circuit to only use NAND gates (because some hardware manufacturing technologies greatly prefer NAND gates). All in all when designing a logic circuit, we basically always perform these two main steps:

1. Design the circuit to do what we want.
2. Minimize (and/or otherwise transform) it so as to optimize it.

Some basic methods of minimization include:

- **algebraic methods**: We use known formulas to simplify the logic expression representing our circuit. This is basically the same as simplifying fractions and similar mathematical expressions, just in the realm of boolean algebra. Some common formulas we use:
  - ♦ **De Morgan Laws**:  $\neg(x \& y) = \neg x \mid \neg y$ ,  $\neg(x \mid y) = \neg x \& \neg y$
  - ♦ **distributivity**:  $x \mid (y \& z) = (x \mid y) \& (x \mid z)$ ,  $x \& (y \mid z) = (x \& y) \mid (x \& z)$
  - ♦  $x \mid \neg x = 1$ ,  $x \& \neg x = 0$ ,  $x \mid x = x$ ,  $x \& x = x$
  - ♦  $x \mid (\neg x \& y) = x \mid y$ ,  $x \& (\neg x \mid y) = x \& y$
  - ♦ ...
- **Karnaugh maps**: One of the most basic methods, simple algorithm using a table.
- **Quine McCluskey**: A bit more advanced method.
- ...

Example of minimization will follow in the example section.

## Example

One of the simplest logic circuits is the two-bit half adder which takes two input bits,  $x$  and  $y$ , and outputs their sum  $s$  and carry over  $c$  (which will become important when chaining together more such adders). Let us write a truth table of this circuit (note that adding in binary behaves basically the same as how we add by individual digits in decimal):

| $x$ | $y$ | $s$ | $c$ |
|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   |
| 1   | 0   | 1   | 0   |
| 0   | 1   | 1   | 0   |
| 1   | 1   | 0   | 1   |

Notice that this circuit is combinational -- its output ( $s$  and  $c$ ) only depends on the input values  $x$  and  $y$  and nothing else, which is why we can write such a nice table.

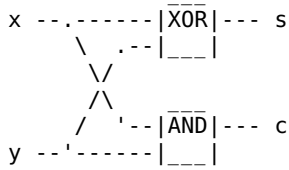


OK, so now we have the circuit behavior specified by a truth table, let's continue by designing the actual circuit that implements this behavior. Let us start by writing a logic expression for each output (& = AND, | = OR, ! = NOT, ^ = XOR):

$$s = x \wedge y$$

$$c = x \& y$$

We see the expressions are quite simple, let us now draw the actual circuit made of the basic logic gates:



And that's it -- this circuit is so simple we can't simplify it further, so it's our actual result (as an exercise you may try to imagine we don't have a XOR gate available and try to replace it by AND, OR and NOT gates).

Next we can expand our half adder to a full adder -- a full adder takes one more input  $z$ , which is a carry over from a previous adder and will be important when chaining adders together. Let's see the truth table of a full adder:

| $x$ | $y$ | $z$ | $s$ | $c$ |
|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 0   |
| 1   | 0   | 0   | 1   | 0   |
| 0   | 1   | 0   | 1   | 0   |
| 1   | 1   | 0   | 0   | 1   |
| 0   | 0   | 1   | 1   | 0   |
| 1   | 0   | 1   | 0   | 1   |
| 0   | 1   | 1   | 0   | 1   |
| 1   | 1   | 1   | 1   | 1   |

Let's try to make boolean expressions for both outputs now. We may notice  $c$  is 1 exactly when at least two of the inputs are 1, which we may write as

$$c = (x \& y) \mid (x \& z) \mid (y \& z)$$

However, using the formula  $(a \& c) \mid (b \& c) = (a \wedge b) \& c$ , we can simplify (minimize) this to an expression that uses one fewer gate (notice there is one fewer operator)

$$c = (x \& y) \mid ((x \wedge y) \& z)$$

The expression for  $s$  is not so clear though -- here we can use a method that always works: we simply look at all the lines in the truth table that result in  $s = 1$  and write them in "ORed" form as

$$s = (x \& !y \& !z) \mid (!x \& y \& !z) \mid (!x \& !y \& z) \mid (x \& y \& z)$$

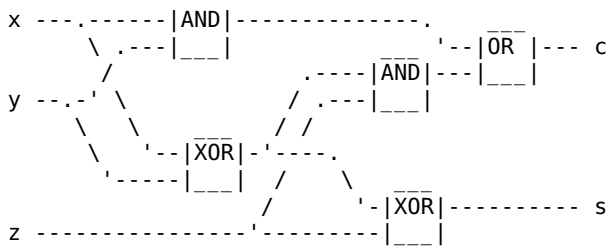
Which we can also actually minimize (as an exercise try to figure out the formulas we used :p)

$$s = ((x \wedge y) \& !z) \mid (!x \wedge y) \& z$$

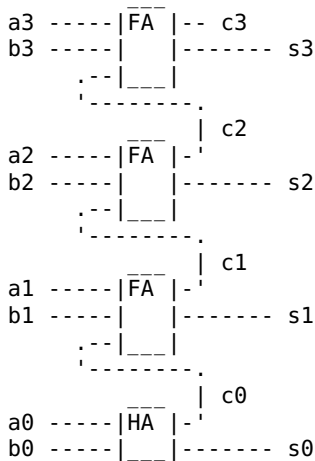
$$s = (x \wedge y) \wedge z$$

Now finally we can draw the full adder circuit

---



Now let us spectacularly combine one half adder (HA) and three full adders (FA) into one magnificent 4 bit adder. It will be adding two 4bit numbers,  $a$  (composed of bits  $a_0$  to  $a_3$ ) and  $b$  (composed of bits  $b_0$  to  $b_3$ ). Also notice how the carry bits of lower adders are connected to carry inputs of the higher full adders -- this is the same principle we use when adding numbers manually with pen and paper. The resulting sum  $s$  is composed of bits  $s_0$  to  $s_3$ . Also keep in mind the circuit is still combinational, i.e. it has no memory, no clock input and adds the numbers in a "single run".



TODO: sequential one?

logic\_gate

## Logic Gate

Logic gate is a basic element of logic circuits, a simple device that implements a Boolean function, i.e. it takes a number of binary (1 or 0) input values and transforms them into an output binary value. Logic gates are kind of "small boxes" that eat 1s and 0s and spit out other 1s and 0s. Strictly speaking a logic gate must implement a mathematical function, so e.g. flip-flops don't fall under logic gates because they have an internal state/memory.

Logic gates are to logic circuits kind of what resistors, transistors etc. are for electronic circuits. They implement basic functions that in the realm of boolean logic are equivalents of addition, multiplication etc.

Behavior of logic gates is, just as with logic circuits, commonly expressed with so called truth tables, i.e. a tables that show the gate's output for any possible combination of inputs. But it can also be written as some kind of equation etc.

There are 2 possible logic gates with one input and one output:

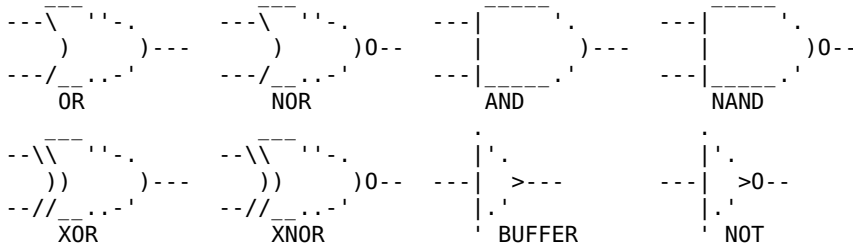
- **identity (buffer)**: Output equals the input. This doesn't have any function from the logic perspective but can e.g. be used as a placeholder or to introduce intentional delay in the physical circuit etc.
- **NOT**: Negates the input (0 to 1, 1 to 0).

There are 16 possible logic gates with two inputs and one output (logic table of 4 rows can have  $2^4$  possible output values), however only some of them are commonly used and have their own names. These are:

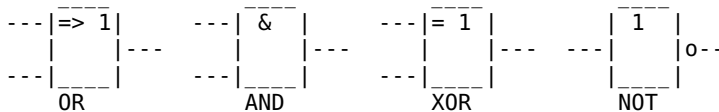
- **OR**: Gives 1 if at least one input is 1, otherwise 0.
- **AND**: Gives 1 if both inputs are 1, otherwise 0.
- **XOR (exclusive OR)**: Gives 1 if inputs differ, otherwise 0.
- **NOR**: Negation of OR.
- **NAND**: Negation of AND.
- **XNOR**: Negative XOR (equality).

The truth table of these gates is as follows:

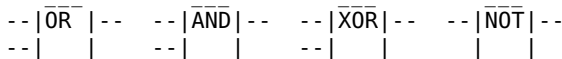
| x | y | x OR y | x AND y | x XOR y | x NOR y | x NAND y | x XNOR y |
|---|---|--------|---------|---------|---------|----------|----------|
| 0 | 0 | 0      | 0       | 0       | 1       | 1        | 1        |
| 0 | 1 | 1      | 0       | 1       | 0       | 1        | 0        |
| 1 | 0 | 1      | 0       | 1       | 0       | 1        | 0        |
| 1 | 1 | 1      | 1       | 0       | 0       | 0        | 1        |



alternatively:



or even:



*symbols often used for logic gates*

Functions NAND and NOR are so called functionally complete which means we can implement any other gate with only one of these gates. For example  $\text{NOT}(x) = \text{NAND}(x,x)$ ,  $\text{AND}(x,y) = \text{NAND}(\text{NAND}(x,y), \text{NAND}(x,y))$ ,  $\text{OR}(x,y) = \text{NAND}(\text{NAND}(x,x), \text{NAND}(y,y))$  etc. Similarly  $\text{NOT}(x) = \text{NOR}(x,x)$ ,  $\text{OR}(x,y) = \text{NOR}(\text{NOR}(x,y), \text{NOR}(x,y))$  etc.

## See Also

- [logic circuit](#)
- [quantum gate](#)

logic

## Logic

TODO: intro

TODO: relationship of logic and math, which comes first etc.

**Power of logic is limited** (for more please read this excellent resource:

<http://humanknowledge.net/Thoughts.html>) -- though logic is the strongest, most stable platform our knowledge can ever stand on, it is still not infinitely powerful and has its limits, despite what any reddit [atheist](#) tells you or even what he believes. This sadly [dooms](#) us to certain eternal inability to uncover all there

is, we just have to accept from a certain point we are blind and not even logic will help us. Kurt Godel (along with others, e.g. Tarski) mathematically proved with his incompleteness theorems that we simply won't be able to prove everything, not even the validity of formal tools we use to prove things. See also knowability. Even in just intuitive terms: on the lowest level we start using logic to talk about itself, i.e. if we e.g. try to prove that "logic works" using logical arguments, we cannot ever succeed, because if we succeed, the proven fact that "logic works" relies on the fact that logic indeed works; if it perhaps doesn't work and we used it to prove its own validity, we might have simply gotten a wrong result (it's just as if we trust someone saying "I am not a liar", he may as well be lying about not being a liar). By this logic even the previous sentence may or may not actually be true, we simply don't know, sometimes the best we can do is simply hold on to stronger or weaker beliefs. Imagine we have a function *isTrue(x)* that automatically checks if statement *x* is true (returns *true* or *false*), now imagine we have statement *y* that says *isTrue(y) = false*; our *isTrue* function will fail to correctly evaluate statement *y* (it can't return neither *true* nor *false*, both will lead to contradiction) -- this is a proof that there can never be a computable function that decides whether something is true or not. Logic furthermore cannot talk about many things; it can tell us how the world works but e.g. not WHY it works like it does. Checkmate atheists.

## See Also

- knowability
  - science
- 

love

## <3 Love <3

Love is a deep feeling of affection towards someone or something, usually accompanied by a very strong emotion. There are many different kinds of love and love has always been one of the most important feelings which many living beings are capable of, it permeates human art, culture and daily lives. Unconditional selfless love towards all living beings is the basis of less retarded society.

What is the opposite of love? Many say it is hatred, even though it may also very well be argued that it is rather indifference, i.e. just "not caring", because hate and love often come hand in hand and are sometimes actually very similar -- both hate and love arouse strong emotion, even obsession, and can be present at the same time (so called love-hate relationship). Love sometimes quickly changes to hate and vice versa.

As mentioned, **love is not a single feeling**, there are many types of it, for example parental love, love of a life partner, platonic love, self love, love for a friend, towards God, of pet animal, love of art, knowledge, life, nature, as well as selfish obsessive love, selfless love and many others. Some kinds of love may be so rare and complex that it's hard to describe them, for example it is possible to passionately love a complete stranger merely for his existence, without feeling a sexual desire towards him. One may love a beautiful mathematical formula and even people who hurt him. Love is a very complex thing.

Is there a **good real life example of unconditional selfless love**? Yes. When a fascist Brenton Tarrant shot up the Christchurch mosques on 15 March 2019 and killed 51 people, there was a woman among them whose husband said after the incident he wanted to hug Tarrant. The husband was also present during the shooting. Not only has he forgiven the killer of his wife and someone who almost also murdered him alone, he showed him love, something which must have been unimaginably difficult and something that proved him one of the most pure people on this planet. He said about it the following (paraphrased for copyright concerns): "There is no use in anger. Anger and fight will not fix it, only with love and caring can we warm hearts. [...] I love him because he is a human being, he is my brother. [...] I don't support his act. [...] But perhaps he was hurt in his life, perhaps something happened to him. [...] Everyone has two sides, a bad one and a good one; bring out the good in you.". (source: <https://www.mirror.co.uk/news/world-news/husband-forgives-new-zealand-terrorist-14154882>) { This moved me so much when I read it, I can't explain how much this affected my life. I have so much admiration for what this man said and I wish I could follow his message for my whole life. Only the words of the man alone have awoken so much of the purest love in me towards every living being on this planet, which I didn't even know existed. ~drummyfish }

# The Way Of Love

LRS advocates living the way of love -- loving everyone and treating others with love, spreading it to the whole world. Love is contagious; just like hate spawns hate, love spawns more love. Love is able to stop the self-sustaining circle of hate and revenge. If you show a true, unconditional love to someone who hates you, there is a great chance the hatred will be lost, that grievances will be forgiven and forgotten.

Today's society makes love kind of a commodity, as anything else; a subject of speculation, a tool, sometimes even a weapon, a card to be kept hidden and played at the right time. People are taught to hide their feelings, they are afraid to tell others they love them as it might make them look weak, vulnerable, it might be socially unacceptable. We reject such toxic bullshit. If you love someone, whoever it is, tell him. It really works, you will soon be surrounded with loving people this way.

{ I know this from experience, once I truly started loving others unconditionally, I made many past enemies into great friends, and I saw many of them turn to being actually very nice people. It is just such a great feeling to let go of hate and so heartwarming to make peace with people <3 ~drummyfish }

## See Also

- polyamory

---

low\_poly

## Low Poly

The term *low poly* (also low-poly or lowpoly) is used for polygonal 3D models whose polygon count is relatively low -- so low that one can see the model approximates the ideal shape only very roughly. For typical models (animals, cars, guns, ...) the polygon count under which they are correctly called low poly is usually a few dozens or few hundreds at most. The opposite of low poly is high poly.

**WATCH OUT:** Retards nowadays use the term "low poly" for stylized/untextured high poly models; they even use the term for models whose polygon count is lower than the number of atoms in observable universe, or they use the term completely randomly just to put a cool label to their lame shit models. **STOP THIS FUCKING INSANITY, DON'T CALL HIGH POLY MODELS LOW POLY.**

The exact threshold on polygon count from which we call a model low poly can't be objectively set because firstly there's a subjective judgment at play and secondly such threshold depends on the ideal shape we're approximating. This means that not every model with low polygon count is low poly: if a shape, for example a cube, can simply be created with low number of polygons without it causing a distortion of the shape, it shouldn't be called low poly. And similarly a model with high polygon count can still be classified as low poly if even the high number of polygons still causes a significant distortion of the shape.

The original purpose of creating low poly models was to improve performance, or rather to make it even possible to render something in the era of early computer graphics. Low poly models take less space in memory and on good, non-capitalist computers render faster. As computers became able to render more and more polygons, low poly models became more and more unnecessary and eventually ended up just as a form of "**retro**" **art style** -- many people still have nostalgia for PS1 graphics with very low poly models and new games sometimes try to mimic this look. In the world of capitalist consoomer computing/gayming nowadays no one really cares about saving polygons on models because "modern" GPUs aren't really affected by polygon count anymore, everyone just uses models with billions of polygons even for things that no one ever sees, soydevs don't care anymore about the art of carefully crafting models on a low polygon budget. However in the context of good, non-capitalist technology low poly models are still very important.

Low poly models are intended to be used in interactive/real-time graphics while high poly ones are for the use in offline (non-realtime) rendering. Sometimes (typically in games) a model is made in both a low poly and high poly version: the low poly version is used during gameplay, the high poly version is used in cutscenes. Sometimes even more than two versions of models are made, see level of detail.

## See Also

- [pixel art](#)
  - [ASCII art](#)
  - [low fidelity](#)
  - [retro](#)
- 

lrs\_dictionary

## LRS Dictionary

WORK IN PROGRESS

{ Most of these I just heard/read somewhere, e.g. on [4chan](#), in [Jargon File](#) or from [RMS](#), some terms I made myself. ~drummyfish }

### mainstream

American  
[anime](#)  
[Apple](#) user  
[Asperger](#)  
average citizen  
[Blender](#)  
[censorship](#)  
[CEO](#)  
[cloud](#) computing  
[cloudflare](#)  
code of conduct ([COC](#))  
consume  
[copyright](#)  
[CSS](#)  
[C++](#)  
[Debian](#)  
[democracy](#)  
digital garden  
digital rights management ([DRM](#))  
[encryption](#)  
[entrepreneur](#)  
[Facebook](#) user  
fair trade  
[feminism](#)  
[Firefox](#)  
[gaming](#)  
[geek](#)  
[global warming](#)  
[Google](#)  
Gmail  
[GNU](#)  
influencer

### correct/cooler

Amerifag  
tranime  
iToddler  
assburger  
normie, normalfag, bloatoddler, NPC, ...  
Blunder  
censorshit  
capitalist evil oppressor  
clown computing  
cuckflare, clownflare, crimeflare  
code of coercion, code of censorship  
consoom (see also [coom](#))  
copywrong, copyrestriction, copyrape  
cascading style shit  
crippled C  
Lesbian  
democrazy  
digital swamp  
digital restrictions management  
bloatcryption  
murderer  
zucker, used  
fair rape  
feminazism, femifascism  
Furryfox  
gayming  
retard  
global heating  
Goolag  
Gfail  
GNUts, fretards, Gigantic and Nasty but Unavoidable  
manipulator

## mainstream

Intel  
Internet Explorer  
Internet of things  
iPad  
iPhone  
job  
"left"  
LGBT  
"Linux"  
logic gate  
Macintosh  
Microsoft  
microtransaction  
moderation  
modern  
network  
neurodivergent  
neurotypical  
NFS  
Nintendo  
NSA  
NVidia  
object oriented programming (OOP)  
object oriented  
objective C  
openbsd  
peer-reviewed  
person  
plug and play  
school  
"science"  
software as a service (SAAS)  
Steve Jobs  
subscription  
systemd  
United States of America  
user (of a proprietary system)  
voice assistant  
wayland  
webassembly  
Wikipedian  
Windows  
work  
world wide web  
YouTube

## correct/cooler

Incel  
Internet Exploder, Internet Exploiter  
Internet of stinks/stings  
iBad  
spyPhone  
slavery  
pseudoleft, SJW  
FGTS, TTTT  
GNU, lunix, loonix  
logic gayte  
Macintoy, Macintrash, Maggotbox  
Microshit  
microtheft  
censorship  
malicious, shitty  
notwork  
retarded, neuroretarded  
typical retard  
nightmare file system  
Nintendont  
national spying agency  
NoVidya  
object obsessed programming  
objectfuscated  
objectionable C  
openbased  
peer-censored  
man  
plug and pray  
indoctrination center  
soyence  
service as a software substitute (SAASS)  
Steve Jewbs  
microrape  
shitstemd, soystemd  
United Shitholes of America, burgerland  
used, lusr  
personal spy agent  
whyland  
weebassembly  
wikipedo  
Winshit, Winbloat, Backdoors? :D  
slavery  
world wide wait  
JewTube

## See Also

- [acronyms](#)
  - [newspeak](#)
- 

lrs

## Less Retarded Software

Less retarded software (LRS) is a specific kind of software aiming to be a truly good technology maximally benefiting and respecting its users, following the philosophy of extreme minimalism (Unix philosophy, suckless, KISS, ...), anarcho pacifism, communism and freedom. The term was invented by drummyfish.

By extension LRS can also stand for less retarded society, a kind of ideal society which we aim to achieve with our technology.

LRS is a set of ideas and kind of a mindset, a philosophy, though it tries to not become a traditional movement or even something akin a centrally organized group; by anarchist principles it sees following people and groups of people as harmful, it always advocates to only follow ideas and to associate loosely. Therefore it tries to only be a concept that will remain pure, such as for example that of free software, but NOT an organization, such as for example the FSF, which will always become corrupt.

As a symbol of LRS we sometimes use heart (love), the peace symbol (pacifism, nonviolence) and A in circle (anarchism), but these only serve as a universal identifier of the philosophy, not as a flag or anything similar -- as flags are a sign of fascism -- for this the official LRS flag is defined to be a completely transparent square which has side length of one billion light years times the busy beaver function of the current 64 bit Unix time -- this is so that the flag cannot practically be manufactured and even scaled down versions will hardly serve the purpose of a flag (only the ideal version of the flag is acceptable, i.e. that which is completely transparent and invisible). The official international LRS day is every day in the year and it always takes precedence over any other cause whose day it is supposed to be (as long as it is aligned with LRS the other cause may be acknowledged too, but only in second or lower place).

{ TODO: official currency? }

## Definition

The definition here is not strict but rather fuzzy, it is in a form of ideas, style and common practices that together help us subjectively identify software as less retarded.

Software is less retarded if it adheres, to a high degree (not necessarily fully), to the following principles:

- Being made with a **truly selfless** goal of maximally helping all living beings who may use the software without any intent of taking advantage of them in any way.
- Trying to follow the **Unix philosophy** (do one thing well, use text interfaces, ...).
- Trying to follow the **suckless philosophy** (configs as source files, distributing in source form, mods as patches, ...).
- Being **minimalist** (single compilation unit, header-only libraries, no build systems, no OOP languages, simple version numbering, ...), countercomplex, KISS, appropriate technology. Any project has to be **solo manageable** if that's at all possible.
- Being **free software** legally but ALSO practically (well commented, not bloated and obscured etc., so as to truly and practically enable the freedoms to study, modify etc.). This may also include attributes such as decentralization.
- Being **free culture**, i.e. LRS programs are free as a whole, including art assets, data etc.
- **Minimizing dependencies**, even those such as standard library or relying on OS concepts such as files or threads, even indirect ones such as build systems and even non-software ones (e.g. avoiding floating point, GPU, 64bit etc.).
- Very **portable**, hardware non-discriminating, i.e. being written in a portable language, minimizing resource usage (RAM, CPU, ...) and so on.



- Being written in a **good, suckless programming language** -- which languages are acceptable is debatable, but some of them most likely include C (C89 or C99), comun, Forth, Lisp, maybe even Brainfuck, False, Lua, Smalltalk, Pascal etc. On the other hand bloated languages like Python, JavaScript or Rust are absolutely unacceptable.
- **Future-proof, self-contained** (just compile and run, no unnecessary config files, daemons, database services, ...), finished as much as possible, not controlled by anyone (should follow from other points). This may even include attributes such as physical durability and design that maximizes the devices life.
- **Hacking friendly**, repairable and inviting to improvements and customization, highly adhering to hacker culture.
- Built on top of other LRS or LRS-friendly technology such as the C99 language, comun, Unix, our own libraries etc.
- Simple permissive licensing (being suckless legally) with great preference of **public domain**, e.g. with CC0 + patent waivers.
- Elegant by its simple, well thought-through solutions. (This is to be contrasted with modern rapid development.)
- **No bullshit** such as codes of conduct, furry mascots, tricky licensing conditions, ads etc.

## Further Philosophy

Here are a few bullet points giving further ideas about what LRS is about, also serving as advice for creating such technology:

- Do one thing well.
- Keep it simple, no bullshit. Less is more, worse is better, small is beautiful. Don't overengineer.
- Users are programmers. This means users can fiddle with their programs; instead of going to request a feature from the developer, the user can many times implement it himself thanks to simple design of the program, EVEN if the user is not an actual programmer (anyone can ctrl+F keywords and rewrite values in source code).
- Bug reports are patches.
- Customization is forking. A software tool is customized by applying personally selected patches and making personal changes to the source code (configuration is also part of source code) -- this creates a personal fork of the tool.
- Forking is good.
- Users compile their programs. Compilation is trivial and fast.
- Programs are distributed in source form.
- Be selfless, program's goal is only to help its user.
- Programs are efficient and take long time to make, they aren't consumerist products, they can't be made on schedule but they should aim to be finished.
- No one owns programs, no one owns data, no one owns art, no one owns information and ideas. Everything is free, legally AND in any other ways.
- Use universal interfaces (text), be compatible
- No capitalist style usercentrism: a user is NOT above programmer or any other living being (as it is in capitalism). This means that if e.g. a feature can make user's life 1% better but will enslave additional 10 programmers with perpetual maintenance, it should NOT be added.
- Code is reusable.
- Hacking is good. Allow hacking, allow breaking and raping of your program in ways you didn't intend, do not artificially prevent anything.
- Be portable, respect weaker platforms and platforms of other types.
- Programs are technology (NOT brands, franchises, weapons, political grounds, social networks, work opportunities, property, platforms, ...).
- Work is shit, laziness is good.
- Secrets are bad, encryption is stupid.
- Low level is good, use only minimum necessary abstraction.
- ...

## Why

LRS exists for a number of reasons, one of the main ones is that we simply need better technology -- not better as in "having more features" but better in terms of design, purpose and ethics. Technology has to make us more free, not enslave us. Technology has to be a tool that serves us, not a device for our abuse. We believe mainstream technology poses a serious, even existential threat to our civilization. We don't think we can prevent collapse or a dystopian scenario on our own, or even if these can be prevented at all, but we can help nudge the technology in a better direction, we can inspire others and perhaps make the future a little brighter, even if it's destined to be dark. Even if future seems hopeless, what better can we do than try our best to make it not so?

There are other reason for LRS as well, for example it can be very satisfying and can bring back joy of programming that's been lost in the modern toxic environment of the capitalist mainstream. Minimalist programming is pleasant on its own, and in many things we do we can really achieve something great because not many people are exploring this way of technology. For example there are nowadays very few programs or nice artworks that are completely public domain, which is pretty sad, but it's also an opportunity: you can be the first human to create a completely public domain software of certain kind. Software of all kind has already been written, but you can be the first one who creates a truly good version of such software so that it can e.g. be run on embedded devices. If you create something good that's public domain, you may even make some capitalist go out of business or at least lose a lot of money if he's been offering the same thing for money. You free people. That's a pretty nice feeling and makes you actually live a good life.

{ Here and there I get a nice email from someone who likes something I've created, someone who just needed a simple thing and found that I've made it, that alone is worth the effort I think. ~drummyfish. }

## Specific Software

see also LRS projects needed

The "official" LRS programs and libraries have so far been solely developed by drummyfish, the "founder" of LRS. These include:

- **Anarch**: Game similar to Doom.
- **comun**: LRS programming language.
- **raycastlib**: Advanced 2D raycasting rendering library.
- **SAF**: Tiny library for small portable games.
- **small3dlib**: Simple software rasterizer for 3D rendering.
- **smallchesslib**: Simple chess library and engine (AI).
- **microtd**: Simple tower defense game written with SAF.
- **tinyphysicsengine**: Very simple 3D physics engine.
- smaller projects like dumbchat and shitpress

Apart from this software a lot of other software developed by other people and groups can be considered LRS, at least to a high degree (there is usually some minor inferiority e.g. in licensing). Especially suckless software mostly fits the LRS criteria. The following programs and libraries can be considered LRS at least to some degree:

- **brainfuck**: Extremely simple programming language.
- **dwm**: Official suckless window manager.
- **Collapse OS** and **Dusk OS**: Extremely minimalist operating systems.
- **LIL**: Tiny embeddable scripting programming language.
- **lisp**: Programming language with a pretty elegant design.
- **st**: Official suckless terminal emulator.
- **badwolf**: Very small yet very usable web browser.
- **netsurf**: Nice minimalist web browser.
- **FORTH**: Small programming language with very nice design.
- **surf**: Official suckless web browser.
- **tcc**: Small C compiler (alternative to gcc).
- **musl**: Tiny C standard library (alternative to glibc).
- **FALSE**: Extremely small programming language.

- **vim** (kind of): TUI text/programming editor. Vim is actually relatively big but there are smaller builds, flavors and alternatives.
- **Simon Tatham's portable puzzle collection**: Very portable collection of puzzle games.
- ...

Other potentially LRS software to check out may include TinyGL, scc, ed, IBNIZ, lynx, links, uClibc, miniz, Lua, nuklear, dmenu, sbase, sic, tabbed, svkbd, busybox, darcs, raylib, IRC, PortableGL, openbsd, mtpaint and others.

It is also possible to talk about LRS data formats, protocols, standards, designs and concepts as such etc. These might include:

- **ASCII**: Text encoding.
- **fixed point**: Fractional number format, as opposed to floating point.
- **RGB332**, **RGB565**: Simple RGB formats/palettes.
- **bytebeat**: Simple and powerful procedural music technique.
- **farbfeld**: Suckless image format.
- **flatfile**: Using files instead of database.
- **rock carved binary data**: Way of recording binary data for ages by manually carving them into rock, plastic or similar durable material.
- **gopher**: Simple alternative to the Web.
- **json**: Simple data text format.
- **lambda calculus**: Minimal functional language.
- **markdown**: Very simple document format.
- **ppm**: Simple image format.
- **qoi**: Lossless compression image format in < 1000 LOC, practically as good as png.
- **reverse polish notation** as opposed to traditional expression notation with brackets, operator precedence and other bloat.
- **set theory**: Basis of all mathematics.
- **textboards**, **imageboards** and pure HTML personal websites as opposed to forums (no registration, no users, simple interface) or even social networks
- **Turing machine**: Minimal definition of a computer.
- **txt2tags**: Very simple document format.
- ...

Other technology than software may also be aligned with LRS principles, e.g.:

- simple and cheap bicycle without changing gears, as opposed to e.g. a car
- sundial, hourglass, ...
- old technology such as toys, alert and cars (e.g. the 1980s toy "See n' Say") used to play back prerecorded sounds without using any electronics or requiring batteries, using only a plastic disc that span on needle (in the same way vinyl records work)
- knives are pretty less retarded
- rocks
- tangram, chess, go, backgammon, ...
- simple flute or a home-made drum kit as musical instruments (as opposed to e.g. grand piano)
- street football as a cheap, simple and accessible sport (unlike for example ice hockey)
- less retarded hardware
- ...

## Politics/Culture And Society

See also less retarded society and FAQ.

LRS is connected to a pretty specific political beliefs, but it's not a requirement to share those beliefs to create LRS or be part of the community centered around LRS technology. We just think that it doesn't make logical sense to support LRS and not the politics that justifies it and from which it is derived, but it's up to you to verify this.

With that said, the politics behind LRS is an idealist anarcho pacifist communism, but NOT pseudoleftism (i.e. we do not support political correctness, COCs, cancel culture, Marxism-Leninism etc.). In our views, goals and means we are similar e.g. to the Venus project, even though we may not agree completely on all points. We are not officially associated with any other project or community. **We love all living beings** (not just people), even those who cause us pain or hate us, we believe love is the only way towards a good society -- in this we follow similar philosophy of nonviolence that was preached by Jesus but without necessarily being religious, we simply think it is the only correct way of a mature society to behave nonviolently and lovingly towards everyone. We do NOT have any leaders or heroes; people are imperfect and giving some more power, louder voices or greater influence creates hierarchy and goes against anarchism, therefore we only follow ideas. We aim for true social (not necessarily physical) equality of everyone, our technology helps everyone equally. **We reject competition as a basis of society** and anti-equality means such as violence, fights, bullying (cancelling etc.), ensorship (political correctness etc.), governments and capitalism. We support things such as universal basic income (as long as there exist money which we are however ultimately against), veganism and slow movement. We highly prefer peaceful evolution to revolution as revolutions tend to be violent and have to be fought -- we do not intend to push any ideas by force but rather to convince enough people to a voluntary change.

## See Also

- less retarded society
- suckless
- Venus project
- reactionary software
- KISS
- Buddhism
- bitreich

---

lrs\_wiki

## LRS Wiki

LRS wiki, also Less Retarded Wiki, is a public domain (CC0) encyclopedia focused on truly good, minimalist technology, mainly computer software -- so called less retarded software (LRS) which should serve the people at large -- while also exploring related topics such as the relationship between technology and society, promoting so called less retarded society. The basic motivation behind LRS and its wiki is unconditional love of all life, and the goal of LRS is to move towards creating a truly useful, selfless technology that maximally helps all living beings as much as possible. As such the wiki rejects for example capitalist software (and capitalism itself), bloated software, intellectual property laws (copyright, patents, ...) ensorship, pseudoleftism (political correctness, cancel culture, COCs ...) etc. It embraces free as in freedom, simple technology, i.e. Unix philosophy, suckless software, anarcho pacifism, racial realism, free speech, veganism etc. As a work promoting pure good in a time of universal rule of evil it is greatly controversial, often met with hostility.

LRS wiki was started by drummyfish on November 3 2021 as a way of recording and sharing his views, experience and knowledge about technology, as well as for creating a completely public domain educational resource and account of current society for future generations. It was forked from so called "based wiki" at a point when all the content on it had been made by drummyfish, so at this point LRS wiki is 100% drummyfish's own work; over time it became kind of a snapshot of drummyfish's brain and so the wiki doesn't allow contributions (but allows and encourages forks).

Over time, being written solely by drummyfish without much self censorship and "language filtering", the wiki also became something like drummyfish's raw brain dump with all the thoughts and moods averaged over the time span of writing the wiki -- reading through it makes you see relatively faithfully how drummyfish internally thinks (e.g. you see anticapitalist rants everywhere because these annoying thoughts are just constantly bothering drummyfish, whatever he's thinking about) -- this can make many people vomit but it's a kind of experiment and some even liked it, so it stays up. No one is forced to read it and CC0 ensures anyone can shape it into anything better hopefully.

The wiki can also additionally be seen as a dirty collection of drummyfish's cheatsheets, links, code snippets, jokes, attempts at ASCII art, vent rants etcetc. So the whole thing is like a digital swamp that one might see as a kind of retarded art that combines many things together: technical, cultural, personal, objective and subjective, beautiful and ugly. It might also be viewed as a shitpost or meme taken too far. It's just its own thing.

The wiki is similar to and was inspired by other wikis and similar works, for example in its topics and technical aspects it is similar to the earliest (plain HTML) versions of Wikipedia and wikiwikiweb. In tone and political incorrectness it is similar to Encyclopedia Dramatica, but unlike Dramatica LRS is a "serious" project.

LRS wiki is currently written as a collection of Markdown files that use a few shell scripts that convert the whole thing to HTML for the web (and can also produce txt and pdf version of it), i.e. it doesn't use any wiki engine or bloated static site generator. There is a plan to rewrite the wiki in comun.

## See Also

- LRS wiki stats
- LRS wiki style guide
- LRS wiki authors
- LRS wiki "rights"
- LRS wiki post mortem
- LRS
- less retarded society
- FAQ

---

luke\_smith

## Luke Smith

Luke Smith was -- before becoming crypto influencer, scam promoter and a generic turd in a suit (around 2022) -- an an American Internet tech mini pseudoc celebrity known for making videos about suckless software, independent living in the woods and here and there about historical, political, linguistic and religious topics. He played a big role in making suckless more popular, however he later started to behave in hugely retarded ways and now isn't worth following anymore.

His look has been described as the *default Runescape character*: he is bald, over 30 years old (probably born around 1990) and lives in a rural location in Florida (exact coordinates have been doxxed but legally can't be shared here, but let's just say the road around his house bears his name). He is a Christian (well, the militant fascist USA "Christian") he goes to the church etc. He has a podcast called *Not Related!* (<https://notrelated.xyz/>) in which he discusses things such as alternative historical theories -- actually a great podcast. He has a minimalist 90s style website <https://lukesmith.xyz/> and his own peertube instance where his videos can be watched if one doesn't want to watch them on YouTube. He is the author of LARBS and minimalist recipe site <https://based.cooking/> (recently he spoiled the site with some shitty web framework lol).

He used to be kind of based in things like identifying the harmfulness of bloat and soyence, but also retarded to a great degree other times, for example he used to shill the Brave browser pretty hard before he realized it was actually a huge scam all along xD He's openly a rightist fascist, capitalist, also probably a Nazi etc. In July 2022 **he started promoting some shitty bloated modern tranny website generator that literally uses JavaScript?** WHAT THE FUCK. Like a good capitalist (to which he self admitted in his podcast) he instantly turned 180 degrees against his own teaching as soon as he smelled the promotion money. Also he's shilling crypto, he lets himself be paid for promoting extremely shitty webhosts in his web tutorials, he's anti-porn, anti-games, anti-fun and leans towards medieval ideas such as "imagination and boredom being harmful because it makes you watch porn" etc. He went to huge shit, you wouldn't even believe. Though he even now still probably promotes suckless somehow, he isn't a programmer (shell scripting isn't programming) and sometimes doesn't seem to understand basic programming ideas (such as branchless programming), he's more of a typical productivity retard. For Luke suckless is something more akin a brand he associated himself with, something he plays a mascot for because it provided him with a bit of personal convenience, this guy doesn't even care about deeper values it seems. As of 2023 he seems to have become

obsessed with adopting a new identity of a turd in a very cheap suit, he literally looks like the door-to-door scam seller lol. All in all, a huge letdown. Of course, Luke is a type B fail.

His videos consisted of normie-friendly tutorials on suckless software, rants, independent living, live-streams and podcasts. The typical Luke Smith video is him walking somewhere in the middle of a jungle talking about how retarded modern technology is and how everyone should move to the woods.

Luke studies PhD in linguistics but is very critical of academia -- he "speaks" several languages (including Latin), though many of them to a low level with bad American accent and he can't sometimes even speak English correctly (using phrases such as "the reason is because", "less people" etc.). He is a self described right-winder, retarded capitalist and talks in meme phrases which makes his "content" kind of enjoyable. He despises such things as soydevry, bloat, "consoomerism" (though he loves money he makes off of it) and soyence.

There seems to be a guy who could very well be Luke's brother, Jacob W Smith -- he has a website at <https://jacobwsmith.xyz/index.html>, he shares many Luke's opinions, writes about him, is a member of Christian webring, seems of similar age, also the name etcetc.

See Also

- Mental Outlaw: Luke's black brother
- Distro Tube

magic

Magic

Magic stands for unknown mechanisms. Once mechanisms of magic are revealed and understood, it becomes science.

main

Welcome To The Less Retarded Wiki

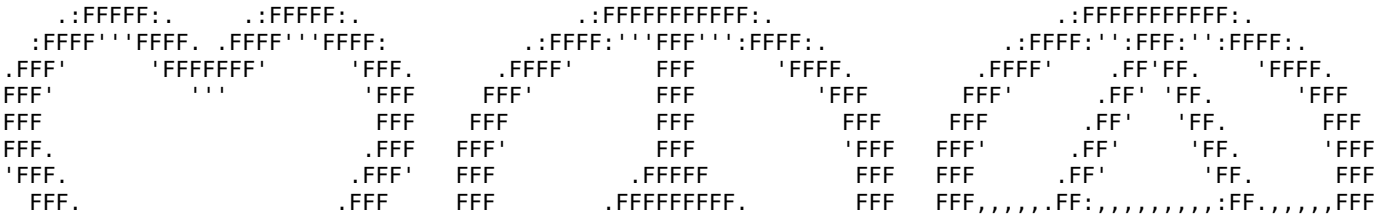
Love everyone, help selflessly.

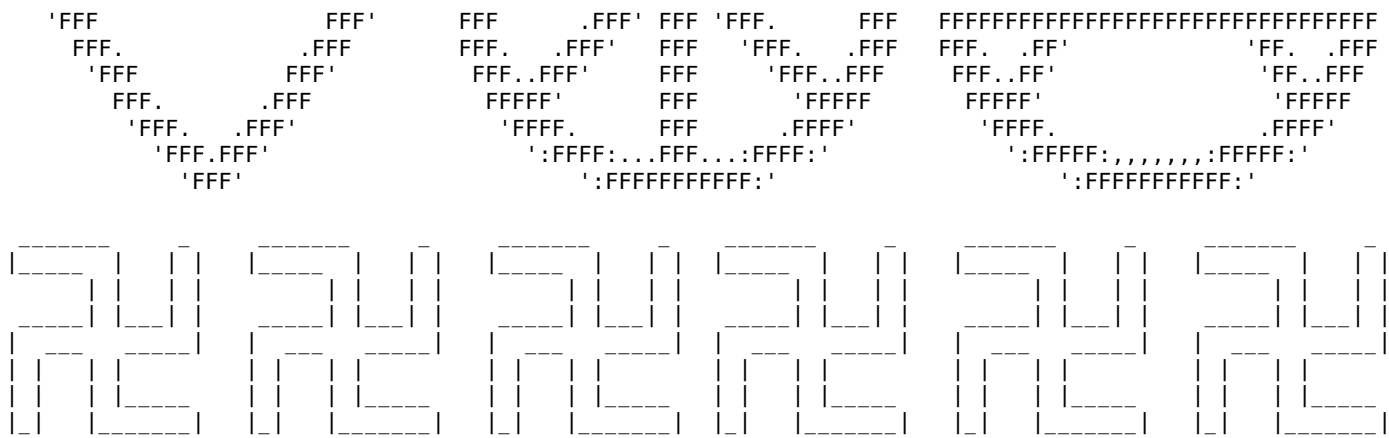
Welcome to Less Retarded Wiki, an encyclopedia only I can edit. But you can fork it, it is public domain under CC0 (see wiki rights) :) Holy shit, I'm gonna get cancelled hard as soon as SJWs find out about this (once this happens, read the wiki post mortem). Until then, let's enjoy the ride. THERE IS NO MODERATION, I can do whatever I want here lol. I love this. INB4 "hate speech" website { LMAO codeberg and Gitlab have already banned us :D Wikipedia/WM Commons also now banned me globally for "opinions expressed on my website". ~drummyfish } CONGRATULATIONS, you have discovered the one true, undistorted and unbiased view of the world -- this is not a joke, this wiki contains pure truth and the solution to most of the issues that plague our current society. Do you have what it takes to unretard yourself? We wish you a nice journey :)

{ This wiki is now also on gopher, see my gopherhole (unless it's down again). ~drummyfish }

{ We have reached 2^9 articles, yay! Yeah, some are just empty, but it's nice. ~drummyfish }

DISCLAIMER: All opinions expressed here are facts. I agree with myself 100% (well, sometimes).





**We love all living beings. Even you.** (Not just all people but also animals and other life forms.) We want to create technology that truly and maximally helps you, e.g. a completely public domain computer. We do NOT fight anything or anyone and we don't have any heroes or leaders. We want to move peacefully towards society that's not based on competition but rather on collaboration. We also **reject all violence**.

{ I no longer see hope, good is practically non existent in this world. This is my last attempt at preserving pure good, I will continue to spread the truth and unconditional love of all life as long as I will be capable of, until the society lynches me for having loved too much. At this point I feel very alone, this work now exists mostly for myself in my isolated world. But I hope that once perhaps my love will be shared with a reader far away, in space or time, even if I will never know him. This is the only way I can continue living. I wish you happy reading, my dear friend. ~drummyfish }

This is a Wiki for less retarded software, less retarded society (LRS) and related topics, mainly those of society, its culture and ideal political views etc. -- LRS should help achieve ideal society with ideal technology. LRS Wiki is a new, refreshing wiki without political correctness. It is neither rightist nor pseudoleftist which many will find confusing.

You ask how could people of the past have been so stupid, how could they have believed obviously nonsensical "pseudoscience" and religious fairy tales, how could the past peasant take part in witch hunts, how could so many people support Hitler and let Holocaust happen? Well, don't judge them too hard -- if you disagree with this wiki, you are just like them. No, there was no magical turn around of society from evil to good just before your birth, times are still the same, except much worse; if you don't see the catastrophic state of the world, you are most likely blissfully brainwashed beyond the level of any medieval peasant. But don't worry, it's not your fault, you are just among the 99.9999%. We are here to help. Keep an open mind and the truth will show. But beware, truth comes for the price of irreversible depression.

This wiki is **NOT** a satire. Yes, everything is **UNDER CONSTRUCTION**.

Are you a failure? Learn which type you are.

**Before contributing please read the rules & style! By contributing you agree to release your contribution under our waiver.** {But contributions aren't really accepted RN :) ~drummyfish }

We have a **C tutorial**! It rocks. We also now have **our own programming language**! It is named comun.

Pay us a visit on the Island and pet our dog! Come mourn to the graveyard because **technology is dead**. Want a **muh webbing** experience? Come board our boat. Modern age is a pile of shit extending to another galaxy. The future is dark but we do our best to bring the light, even knowing it is futile.

LRS Wiki is collapse ready! Feel free to print it out (there is now a complete pdf book download up there), take it to your prep shelter. You may also print copies of this wiki and throw it from a plane into the streets. Thanks.

If you're new here, you may want to read answers to frequently asked questions (FAQ), including "Are you a fascist?" (spoiler: no) and "Do you love Hitler?".

STOP CAPITALISM STOP BLOAT STOP censorship STOP business STOP bullshit STOP copyright STOP working STOP coding STOP competing STOP fighting STOP consuming STOP producing STOP security STOP privacy STOP worshipping people STOP fascism STOP economy STOP slavery STOP violence STOP pedophobia STOP wearing clothes STOP eating animals STOP being an idiot etc. Start loving, sharing, creating and caring :) <3

## What Is Less Retarded Software/Society/Wiki?

Well, we're trying to figure this out on this wiki, but less retarded software is greatly related to suckless, Unix, KISS, free, selfless and sustainable software created to maximally help all living beings. This is so because we just love all living beings. LRS stands opposed to all shittiness of so called "modern" software (and society in general). We pursue heading towards an ideal society that is similar in spirit e.g. to that of the Venus project. For more details see the article about LRS.

In short LRS asks *what if technology was good?* And by extension also *what if society was good?*

UPDATE by drummyfish: now as I've been writing this wiki for a longer time myself, without much self censorship and thought "filtering", the wiki has also become something akin a snapshot of my brain averaged over time, for more see the article about the wiki itself.

## Wanna Help?

See needed projects. Thanks :)

## Are You A Noob?

Are you a noob but see our ideas as appealing and would like to join us? Say no more and head over to a how to!

## Did You Know

- That old technology (such as toys or alerts in cars) could play prerecorded audio without using any electricity or needing batteries? This was done by simply using a plastic disc spinning on a needle (same principle as a vinyl record). There is a gramophone, called CardTalk, made purely of cardboard which plays vinyl records without any electricity (it used to be used by missionaries traveling to jungles etc.).
- That the term "retarded" was actually made as a politically correct replacement for medical terms such as "idiot", "imbecile" and "moron" which became seen as derogatory.
- That all Intel processors since 2008 (and AMD processors since 2013) have a hardware backdoor (Intel ME, AMD PSP) that run the Minix operating system and allows spying on users of those processors no matter what operating system they run?
- That Wilhelm Rontgen, a Nobel laureate, did not patent his groundbreaking discoveries, stating that they should be freely available to anyone without any charge?
- That brain size correlates with intelligence and male brains are on average 10% larger than those of women? Yep, this is still even on Wikipedia, though the implications mustn't be mentioned there.
- That capitalism is probably the most retarded and dangerous idea in history?
- Thanks to quantum computing you can use a computer to carry out computation without actually running the computer?
- You can mathematically prove you don't know some information?
- That some early telephones used a water microphone (a microphone that uses a needle submerged in water)?
- That in capitalism low end CPUs are made by manufacturing a high end CPU and then purposefully crippling it, just for economic reasons? See e.g. core unlocking. This also goes for car engines etc.
- That back in the times of black and white TVs they sold colored transparent plastic sheets to put over the screen to add fake color to it? It was blue on top (for sky), green at the bottom (for grass) and orange in the middle (for people, which would nowadays be seen as "racist").
- That a complement of a formal language can be computationally simpler than the original -- for example that a pushdown automaton cannot tell which strings are of form  $a^{(n)}b^{(n)}c^{(n)}$ , but it can tell exactly which ones are not?



- That LGBT, feminism and similar movements are not truly leftist but rather pseudoleftist and therefore fascist?
- That there is no simple formula for calculating the perimeter of an ellipse?
- That there is a limit on how fast any computable function can grow? There exist uncomputable functions growing so fast that they will eventually outgrow any function that any computer will ever be able to compute. Most famous such function is the busy beaver.
- In 3rd world pigeons carrying SD cards are still much faster and reliable way of transferring data than internet service providers? This also avoids censorship.
- That compiler bomb is a very short source code that makes the language compiler produce gigantic executable?
- That there is a light bulb in California that has been turned on since 1901 and as of writing this is still working? This shows that old things are better than those manufactured under more advanced capitalism which pushes for more consumerism and applies artificial obsolescence. Many sowing machines made more than 100 years ago still function perfectly fine as well as many other types of machines; anything created nowadays shouldn't be expected to last longer than 3 years.
- That there exist numbers that are not computable or are otherwise unknowable? See e.g. Chaitin's constant.
- That throughout history one of the most common patterns is appearance of new lucrative technology or trend which is labeled safe by science, then officially recommended, promoted, adopted by the industry and heavily utilized for many years to decades before being found harmful, which is almost always greatly delayed by the industry trying to hide this fact? This was the case e.g. with asbestos, freons (responsible for ozone layer depletion), x rays, radioactive paint (see *radium girls*), some food preservatives, plastics, smoking and great many prescription drugs among which used to be even cocaine. Yet when you question safety of a new lucrative invention, such as 5G, antidepressants or some quickly developed vaccines, you are labeled insane.

## Topics

Here there are quick directions to some of the important topics; for more see the links provided at the top that include the list of all articles as well as a single page HTML which is good for "fulltext search" via ctrl+F :)

- **basics:** bloat -- capitalist software -- less retarded society -- LRS -- pseudoleft
- **LRS inventions/propositions:** A/B fail -- Anarch -- boat webbing -- comun -- less retarded chess -- less retarded hardware -- less retarded society -- less retarded software -- less retarded watch -- less retarded wiki -- public domain computer -- raycastlib -- rock carved binary data -- SAF -- small3dlib -- smallchesslib -- tinypysicsengine -- world broadcast -- unretardation
- **programming/computers:** 3D rendering -- binary -- computer -- AI -- algorithm -- C -- C tutorial -- computer -- computer graphics -- CPU -- data structure -- demoscene -- GNU -- hacker culture -- hardware -- Internet -- KISS -- Linux -- OOP -- open consoles -- operating system -- optimization -- portability -- procedural generation -- programming -- programming language -- suckless -- Unix philosophy -- web
- **math/theory:** aliasing -- chaos -- combinatorics -- fractal -- formal languages -- information -- linear algebra -- logic -- math -- pi -- prime number -- probability -- Turing machine -- zero
- **society:** anarchism -- anarcho pacifism -- capitalism -- censorship -- collapse -- communism -- democracy -- everyone does it -- fascism -- feminism -- fight culture -- history -- homosexuality -- left vs right vs pseudoleft -- Jesus -- less retarded society -- LGBTQWTF -- science vs soyence -- productivity cult -- selflessness -- socialism -- Venus project -- work
- **freedom/law:** Creative Commons -- free culture -- free hardware -- free software -- copyleft -- copyright -- GNU -- "intellectual property" -- license -- open source -- patent -- public domain
- **interesting:** beauty -- bytebeat -- chess -- Dodleston messages mystery -- hyperoperation -- interplanetary internet -- netstalking -- steganography
- **fun/relaxed/offtopic:** audiophilia -- C downto operator -- C obfuscation contest -- dog -- esolang -- fantasy console -- fun -- games -- island -- jokes -- LMAO -- open console -- rock -- shit -- SIGBOVIK -- Temple OS

---

maintenance

# Maintenance

*Maintenance is slavery of man to a machine.*

Maintenance is shitty work whose goal is just to keep a piece of technology functioning without otherwise changing it. Maintenance is extremely expensive, tiresome and enslaves humans to machines -- we try to minimize maintenance cost as much as possible! Good programs should go to great lengths in effort to becoming highly future-proof and suckless in order to avoid high maintenance cost.

Typical "modern" capitalist/consumerist software (including most free software) is ridiculously bad at avoiding maintenance -- such programs will require one to many programmers maintaining it every single day and will become unrunnable in matter of months to years without this constant maintenance that just wastes time of great minds. I don't know what to say, this is just plainly fucked up.

---

malware

## Malware

Malware is software whose purpose is to be malicious. Under this fall viruses, proprietary software, spyware, DRM software, ransomware, propaganda software, cyberweapons etc.

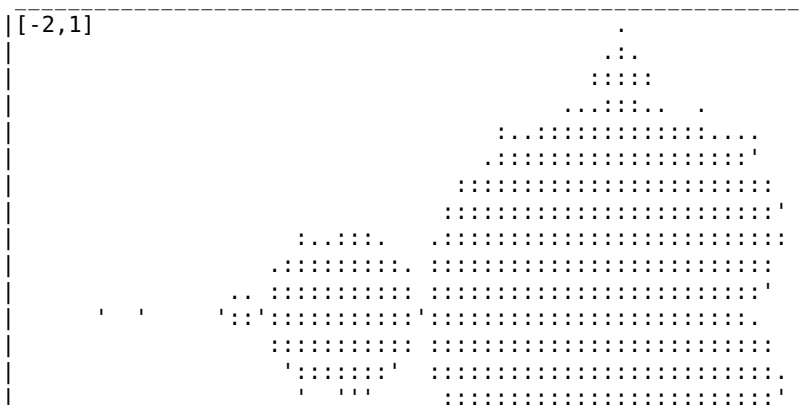
---

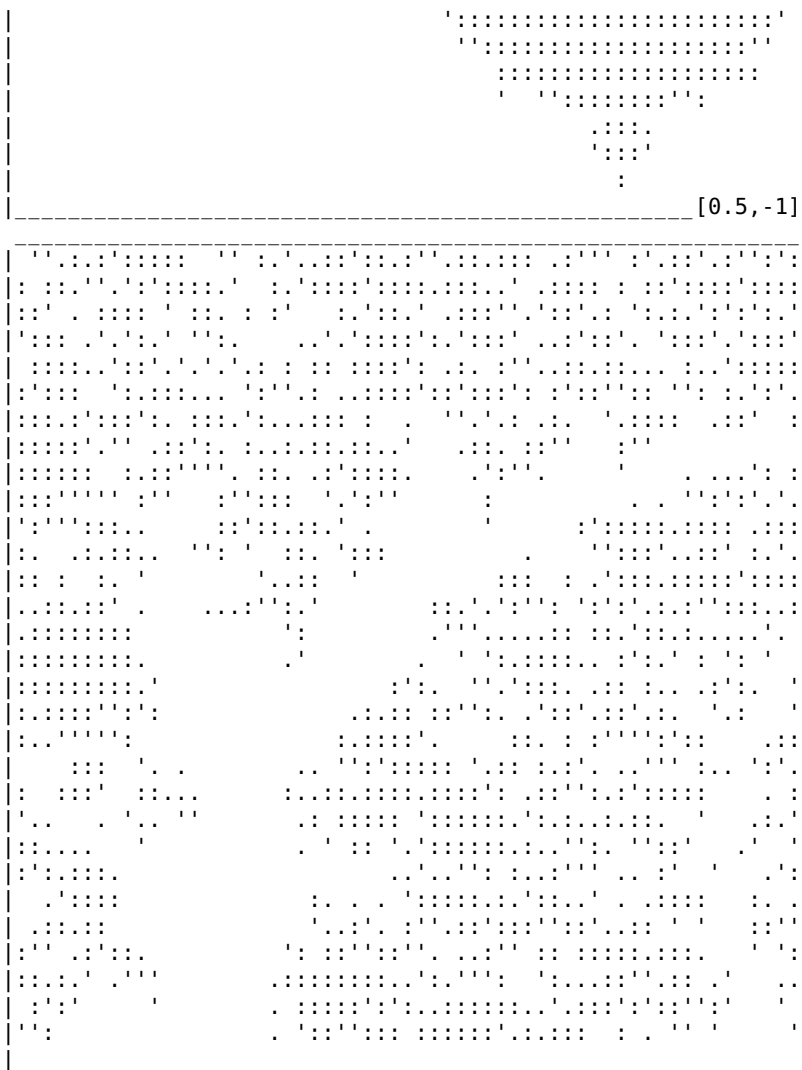
mandelbrot\_set

## Mandelbrot Set

Mandelbrot set (also M-set) is a famous two dimensional fractal, a set of points in two dimensional plane that are defined by a specific very simple equation. It turns out this set has an infinitely complex border (i.e. its shape is a fractal) and the whole thing is just beautiful to look at, especially when we draw it with colors and start zooming in to various interesting places -- patterns keep emerging down to infinitely small scales so we may keep zooming in forever and still discover new and new things; some patterns show self similarity, some not. Applying tricks to further add colors to inside and outside of the set increases the visual beauty yet more -- rendering Mandelbrot set is in fact a quite popular activity among programmers as it's very easy to program such visualizations (at least until we reach the limits of floating point precision, then some more cleverness has to be applied; and yes, Mandelbrot can also be rendered only using fixed point). The origins of exploring this set are somewhere around 1905 when Fatou and Julia explored the equations related to it, however due to the lack of computers the set couldn't very well be drawn -- this was only achieved much later, the first rendering of the set seems to be from 1978, albeit of very poor resolution. The set is named after Benoit Mandelbrot who is often considered the father of the field of fractal geometry and who researched this particular set a lot. Of course, Mandelbrot set is awesome, it's a like a whole infinite world to explore, hidden in just one simple formula.

{ Pretty amazing ASCII rendering of the Mandelbrot set can be found at  
<http://www.mrob.com/pub/muency/asciigraphics.html>. ~drummyfish }





Simple ASCII rendering of Mandelbrot set, below a zoomed-in detail.

**Definition:** we use complex numbers to define the set. Consider the following series of complex numbers  $z[n]$ :

$$z[0] = 0, z[n + 1] = z[n]^2 + p$$

Mandelbrot set is the set of all points  $p$  for which the absolute value ("length") of  $z[n]$  stays bounded (i.e. doesn't grow beyond any limits) as  $n$  goes towards infinity.

NOTE: here is the series equation rewritten to just work with  $x$  and  $y$  coordinates. Checking the point  $[px, py]$ , your series will be  $x[n + 1] = x[n]^2 - y[n]^2 + px$  and  $y[n + 1] = 2 * x[n] * y[n] + py$ .

I.e. taking any point  $p$  in the complex plane (whose real and imaginary parts we see as the  $x$  and  $y$  coordinates), plugging it into the above equation and iterating the series infinitely many times, if the absolute value of  $z[n]$  stays bounded under some finite value (even very large, just not infinitely large), the number belongs to the set, otherwise not (if the absolute value diverges towards infinity). I.e. in other words the Mandelbrot set is a set of kind of "well behaved" points that don't shoot away to infinity when we keep applying some operation to them over and over. Of course computers cannot evaluate infinitely many iterations of the series so they cannot compute the set 100% accurately, but we may very well approximate by performing many iterations (let's 100000) and seeing if the value we get is "very large" (let's say 1000000000) when we stop -- this will work correctly for most points and those few points near the set borders where we make a wrong guess won't really be noticed unless we zoom in very close -- in such cases we can simply perform more iterations to increase precision. To add **colors** to the visualization (so that we don't observe just the borders but also some kind of structure inside and outside of the set) we may simply assign different colors to the points depending e.g. on how big the absolute value is at the time we stop the

evaluation, or how many iterations it took for the absolute value to exceed given limit (for points outside the set). Also note that for nice pictures we should apply antialiasing. Additional fancy filters and shaders such as some kind of postprocessing or fake 3D can also be applied to make the result even more impressive.

There are further optimizations we may apply to calculate the set faster. The set itself also lies in the circle centered at [0,0] with radius 2, so points outside this area can be safely marked as lying outside the set. Furthermore it's proven that if absolute value of  $z[n]$  ever gets greater than 2, the point won't lie in the set (because getting absolute value greater than 2 basically means we start checking a point that inevitably lies outside the circle with radius 2 inside which the whole set lies, so we know the point won't lie in the set). A quick bailout check (not requiring square roots etc.) here can therefore be e.g. checking if either the real or imaginary component absolute value exceeds 2 (which implies the whole vector length must exceed 2 as well), or checking if the sum of squares of the components exceeds 4 (i.e. we squared both sides of the equation and got rid of the square root). Symmetry of the set can also be used to skip computing some points. Further more complex optimizations exist, based e.g. on estimating distance of any given point to the set border etc.

**Quick example:** does point  $[-0.75, 1]$  lie in the Mandelbrot set? Taking the above  $x$  and  $y$  coordinate equations we set  $px = -0.75$  and  $py = 1$ , our starting values are also these, i.e.  $x[0] = -0.75$  and  $y[0] = 1$ . Now  $x[1] = x[0]^2 - y[0]^2 + px = (-0.75)^2 - 1^2 - 0.75 \sim -1.1875$ ,  $y[1] = 2 * x[0] * y[0] + py = 2 * -0.75 * 1 + 1 \sim -0.5$ . So after first iteration we got to approximately  $[-1.1875, -0.5]$ , the length of this vector is  $\sqrt{(-1.1875)^2 + (-0.5)^2} \sim 1.28$ , so we're still in the game. Now we repeat this all with these new coordinates, getting  $x[2] \sim 0.4101$  and  $y[2] \sim 2.1875$ , with length of the vector  $\sqrt{0.4101^2 + 2.1875^2} \sim 2.2256$ . This value surpassed 2, so by the mentioned optimization we now know iterating further the value will only be getting higher and higher until infinity, so we conclude the original point  $[-0.75, 1]$  does NOT lie in the Mandelbrot set.

As an alternative to drawing the set in the traditional plane with  $x/y$  axes correspond to the *real/imaginary* parts of the complex number, we may utilize different mappings, for example polar coordinates or the "inversion mode" (swapping zero and infinity) used in *xaos*. These may offer yet different angles of view of the set.

Mandelbrot set is similar and related to Julia sets; in a way Mandelbrot set is kind of a map of Julia sets, of which there are infinitely many. Each Julia set is, like the Mandelbrot, a set of complex numbers that usually has fractal boundary. Julia sets are defined using the same series as Mandelbrot set, however for given Julia set we take the  $p$  to be constant and instead set  $z[0]$  to the visualized coordinate -- so each  $p$  in the complex plane has its own Julia set. There are some deep mathematical connections between Julia sets and Mandelbrot set. To a Mandelbrot set admirer Julia sets offer infinitely many similar worlds to explore.

The following are some **attributes** of the Mandelbrot set:

- Hausdorff dimension (of the boundary): 2
- **area**: approximately 1.5052; this is a current best estimate, the area is not easy to calculate (it may be estimated e.g. with Monte Carlo methods).
- It is **symmetric** along the  $x$  axis.
- It's proven the set is **connected**, i.e. it's just a single "island".
- The number  $\pi$  is embedded in the shape of the set in a hugely mysterious way which was discovered by mistake by David Bolle in 1991 who tried to measure the width of the gap touching the point  $[-0.75, 0]$  -- as he increased precision of his iterative algorithm, the number of iterations started to approximate digits of  $\pi$ .
- ...

**How to explore Mandelbrot set?** There are about billion programs for this, but a quite nice FOSS one is e.g. Xaos.

As the set is being studied and explored a lot, some even started to make maps of it and give names to various regions. The biggest bulb-part is called the *Main Cardioid*, the smaller disk to the left of it is the *Main Disk*. Between these two parts there is the *Seahorse Valley*. On the right side of *Main Cardioid* there is the *Elephant Valley*. There are terms such as *mu-atom* (also *island*, *mandelbrotie*, *minibrot* or *midget*) -- the smaller distorted self-similar versions of the big set inside the set itself. And so on. Here are some examples of **interesting places** (nice for wallpapers :) in the Mandelbrot set (views are denoted as [center X, center

Y, view radius]]):

- View [-0.774680610626904,-0.137416885603787,8e-12] shows a very nice circular pattern.
- View [-0.74989,-0.0376656,1.04358e-05] shows another nice grid pattern.
- View [0.353447,0.0990225,1.12029e-05] shows a cool spiral pattern.
- View [-1.4045,0,0.0006] shows self-similarity, an approximate smaller Mandelbrot set shape inside itself.
- Views [-1.38379,0,0.037555] and [-1.3973347,0,0.008779] show approximate self similarity.
- Point [-1.3932809650418352,0.0215485287711777498] shows a very thin connection.
- Point [0.372138,0.0903982] shows an infinitely zoomable point from which circular arms stem.
- ...

**Generalizations and modifications:** mentioned Julia sets are very similar to the Mandelbrot set.

**Multibrot** sets are sets similar to the Mandelbrot which we define by requiring  $abs(z[n])$  to not surpass some given value  $T$  under infinite iteration, i.e. Mandelbrot set is one of Multibrot sets, that in which we set  $T = 2$  (because as mentioned above, reaching 2 always leads to divergence towards infinity); for different values of  $T$  we'll get similar but different Multibrot fractal sets. We may also modify the iterative equation from quadratic to cubic (replace  $z[n]^2$  with  $z[n]^3$ ), or a different power (or modify the equation in similar ways) to again get sets similar to the Mandelbrot. Using quaternions instead of complex numbers generalized Mandelbrot from 2D to 4D. Buddhabrot is another famous fractal (which looks like Buddha) and is related to Mandelbrot set.

## Code

The following code is a simple C program that renders the Mandelbrot set into terminal (for demonstrative purposes, it isn't efficient or do any antialiasing, also using float for simplicity but keep in mind fixed point can be easily used as well).

```
#include <stdio.h>

#define ROWS 30
#define COLS 60
#define FROM_X -2.0
#define FROM_Y 1.0
#define STEP (2.5 / ((double) COLS))

unsigned int mandelbrot(double x, double y)
{
    double cx = x, cy = y, tmp;

    for (int i = 0; i < 1000; ++i) // 1000 iterations
    {
        tmp = cx * cx - cy * cy + x;
        cy = 2 * cx * cy + y;
        cx = tmp;

        if (cx * cx + cy * cy > 4) // optimization
            return 0;
    }

    return 1;
}

int main(void)
{
    double cx, cy = FROM_Y;

    for (int y = 0; y < ROWS; ++y)
    {
        cx = FROM_X;

        for (int x = 0; x < COLS; ++x)
        {
            unsigned int point =
                mandelbrot(cx,cy) + (mandelbrot(cx,cy + STEP) * 2);
```

```

        putchar(point == 3 ? ':' : (point == 2 ? '\\' :
            (point == 1 ? '.' : ' ')));

    cx += STEP;
}

putchar('\n');

cy -= 2 * STEP;
}

return 0;
}

```

Note on the optimization above: a naive line checking the divergence of the series would look e.g. like `if (sqrt(cx * cx + cy * cy) > 1000000000)`. However, as said above, it is enough to check if the value exceeds 2, as it's proven that then the series will surely diverge, so we can change it to `if (sqrt(cx * cx + cy * cy) > 2)`. Furthermore we can get rid of the square root by squaring both sides of the inequality, so we get `if (cx * cx + cy * cy > 4)`. Hopefully this is one small example of why math is important for programming.

## See Also

- [fractal](#)
- [Julia set](#)
- [Buddhabrot](#)

---

marble\_race

## Marble Race

Marble race is a simple real life game in which marbles (small glass balls) are released onto a prepared track to race from start to finish by the force of gravity. This game is great because it is suckless, cheap (i.e. accessible), fun and has almost no dependencies, not even a computer -- such a game will be playable even after the technological collapse.

Even though this is a real life game, a computer version can be made too, in different forms: 2D, 3D, realistic or with added elements that would be impossible in real life, such as teleports. And indeed, there have been many games and mini-games made based on this form of entertainment. From the implementation point of view it is very convenient that marbles are of spherical shape as this is one of the simplest shapes to handle in physics engines.

Collecting marbles often comes hand in hand with this hobby, kind of like collecting race horses or Pokemon -- yes, collecting is a bit of a materialistic hobby, however this is one of the cheapest such hobbies that can satisfy the natural human hoarding needs with minimum of actual harm, just like video games satisfy the need for violence without actually harming anyone etc., so we probably consider it a net-good hobby. Consider that getting a new marble, basically a piece of rock, brings basically the same amount of happiness and satisfaction as buying a new car while shrinking the price for it about a million times -- that's very close to what in software we would call suckless.

---

marketing

## Marketing

*Self praise stinks.* --Czech proverb

Marketing is an unethical practice, plentifully used in capitalism, of forcing a product or corporate propaganda by means of lying, manipulation, tricks, brainwashing, torture, exploiting psychological weaknesses of people and others. This manifests most visibly as advertisements and commercials in media but also in other ways such as fake product reviews, product placement in movies etc. Advertising is ever

present and unavoidable in capitalism, billboards now run along the roads instead of trees, commercials yell from electronic devices that are present everywhere, ugly cheap ads, slogans and flashing lights now cover the art of architects -- indeed, there is a promise of ads being put in the sky, people's dreams and thoughts themselves once technology allows it (so called "progress"). Marketing is one of the greatest bullshit industries ever to have seen the light of world and is not only torturing the victims, but also wasting the enormous amount of effort of those who create it.

Specific practices used in marketing are:

- **Lies** and falsehoods. Every ad will present the product as the best, even though not all products can be best. Actors will be paid to lie about how the product changed their life etc. -- so called astrourfing. Many times numbers and "facts" whose source is difficult to trace will be completely made up. **Fake discounts** are something constantly presented in ads.
- **Extreme repetition/spam**: this includes repeating the same commercial over and over (e.g. every 10 minutes) as well as repeating the name of the product in a completely retarded way ("*We recommend X because X is the best. For more info about X visit [www.X.com](#). Remember, X is the best. Your X.*").
- **Psychological tricks** such as **abusing songs** and shitty catchy melodies, often raping existing good music by for example changing the lyrics and then repeating it so many times you never want to even hear the original ever again. This abuses the fact that a song will stick in one's head and keep torturing the individual into thinking about the advertised product constantly. Similarly ads like to show food to people in attempts to **rise appetite in overweight people** and so literally posing a health risk to millions -- consider that most people in the first world are overweight and should try to eat less rather than more, ads are trying to force them to do the opposite (of course additionally forcing food that's not healthy at all). Super markets constantly change placement of items so that people always get lost, spend more time in the shop, see more items and are more likely to buy more things. Other tricks include **shouting, fake empathy** ("we care about you" etc.) or creating the "everyone does it" **illusion** ("look, all the cool teenagers in the world now switched to this product!").
- **Misleading statistics**, presentation and interpretation of data. For example any success rate will be presented as the upper bound as such a number will be higher, typically 99% or 100%, i.e. "*our product is successful in up to 100% cases!*" (which of course gives zero information and only says the product won't succeed in more than 100% cases). A company may also run its own competition for a "best product", e.g. on Facebook, in which all products are of course their products, and then the winning product will be seen on TV as a "contest winning product".
- **Forcefully seizing attention**: ads are present practically everywhere, even embedded in "art" (even in that which one pays for like magazines), in the sky (planes, blimps, drones, ...), they play on every radio you hear in every shop, they pop up on electronic devices one has paid for, they can't be turned off (and if you try to do it, you're called a thief). They are present in education materials and targeted at children. Audio of a commercial will be made louder to catch an attention when it starts playing on a commercial break.
- **bribing celebrities/influencers**. An *influencer* is nowadays a culturally accepted "job" whose sole work consists of lying, forcing products and spreading corporate propaganda.
- ...

These practices are not rare, not even close, they are not even a behavior of a minority, not even of a small majority, they are not illegal and people don't even see them as unusual or undesirable. Stop for a moment to think about how deeply fucked up this is. People in the US are so brainwashed they even pay to see commercials (Super Bowl). Under capitalism these practices are the norm and are getting worse and worse ever year. Boiling the frog works as expected.

A naive idea still present among people is that "ethical marketing" is possible or that it's something that can be fixed by some law, a petition or something similar. In late stage capitalism this is not possible as an "ethical" marketing is a non effective marketing. Deciding to drop the most efficient weapons in the market warfare will only lead to the company losing customers and making place for competition who embraces the unethical means, eventually going bankrupt and disappearing, leaving the throne to the bad guys. You want to do ethical marketing? You're simply out, next in the line gets a shot. Laws will not help as laws are made to firstly favor the market, corporations pay full time lobbyists and law makers themselves are owners of corporations. A law that fixes marketing would be a law that simply bans it -- do you think anyone is going to do that? Even if some small law against "unethical marketing" passes, the immense force and pressure of all

the strongest corporations will work 24/7 on reverting the law and/or finding ways around it, legal or illegal, ethical or unethical. You have a few peasants with banners pleading the country's broken, corrupt non-working political system to shield them from world's strongest market entities with billions of dollars at hand and thousands of full time managers and layers who's only job it is to make marketing happen by any means necessary. If you believe in a happy end then it's incredible how naive you are, you must be more naive than a kindergarden baby.

Another extremely childish idea is that "marketing serves the people by informing them of new products" :D { I don't know, I always think capitalists have at least one brain cell, but they always manage to surprise me by saying something like this. ~drummyfish } This may not even need much comment (it seems weird, like trying to explain that a book dropped from a table will fall to the ground, feels extremely stupid) but let's see: maybe in times of caveman when market was just invented ads worked like this for exactly two days until one caveman realized he can lie on the ad and if he paints a bigger picture on the wall the other caveman customers will be more likely to buy his rocks than the competing caveman's rocks; exactly at this day ads seized to be about informing people and became solely used for forcing one's products and tricking people, and trying to find ways around laws that tried to prohibit this, spawning the endless bullshit war of advertisers and law makers. It's been thousands of years now that ads have absolutely 0% informative value -- imagine an informative ad on TV, a simple white screen with text: "there is a new shampoo in the shop". This literally doesn't even give any information to the consumer, everyone knows there are shampoos in the shop. Do you think there exists any marketing company in which they wouldn't shit themselves in uncontrollable laughter if some of their employee was like "we should make our ad less intrusive, it should only inform the consumer about our product"? Are you really this braindead now?

**Marketing people are subhuman.** Of course, let us be reminded we love all living beings, even subhuman, but the marketing trash not only doesn't show any signs of conscience or morals, they hardly seems conscious at all, they are just a robotic tool of capitalism, more akin monkeys -- however immoral shit they get into, they always just reply "just doing my job" and "it pays well" to anything. What can you say about someone who dedicates his life to bullshit? And not just any bullshit -- bullshit that makes other people more miserable. They make the worst kind of propaganda which literally kills people, they would mercilessly torture children to death if it was on their contract. A capitalist is screeching HAAAAHA IT NOT THE SAME bcuz CHILDREN ARE MAGICAL n economy is pwogwesss, so this invalid. Indeed, it doesn't make any sense -- a capitalist will stay what it is, the lowest class of brainwashed NPC incapable of thinking on its own. All in all, avoid anyone who has anything to do with marketing.

Good things don't need promotion (it's true even if you disagree). **The bigger the promotion, the bigger shit it is.**

No, there is no such thing as a "non-intrusive ad", fucking capitalists are trying to introduce a sense of guilt for not looking at what you don't wanna look. The only non-intrusive ad is that which you don't see or hear at all. Just block all that shit if you can.

---

markov\_chain

## Markov Chain

Markov chain is a relatively simple stochastic (working with probability) mathematical model for predicting or generating sequences of symbols. It can be used to describe some processes happening in the real world such as behavior of some animals, Brownian motion or structure of a language. In the world of programming Markov chains are pretty often used for generation of texts that look like some template text whose structure is learned by the Markov chain (Markov chains are one possible model used in machine learning). Chatbots are just one example.

There are different types of Markov chains. Here we will be focusing on discrete time Markov chains with finite state space as these are the ones practically always used in programming. They are also the simplest ones.

Such a Markov chain consists of a finite number of states  $S_0, S_1, \dots, S_n$ . Each state  $S_i$  has a certain probability of transitioning to another state (including transitioning back to itself), i.e.  $P(S_i, S_0), P(S_i, S_1), \dots, P(S_i, S_n)$ ; these probabilities have to, of course, add up to 1, and some of them may be 0. These probabilities



can conveniently be written as a  $n \times n$  matrix.

Basically Markov chain is like a finite state automaton which instead of input symbols on its transition arrows has probabilities.

## Example

Let's say we want to create a simple AI for an NPC in a video game. At any time this NPC is in one of these states:

- **Taking cover** (state A):
  - ♦ 50% chance to stay in cover
  - ♦ 50% chance to start looking for a target
- **Searching for a target** (state B):
  - ♦ 50% chance to remain searching for a target
  - ♦ 25% chance to start shooting at what it's looking at
  - ♦ 25% chance to throw a grenade at what it's looking at
- **Shooting a bullet at the target** (state C):
  - ♦ 70% chance to remain shooting
  - ♦ 10% chance to throw a grenade
  - ♦ 10% chance to start looking for another target
  - ♦ 10% chance to take cover
- **Throwing a grenade at the target** (state D):
  - ♦ 50% chance to shoot a bullet
  - ♦ 25% chance to start looking for another target
  - ♦ 25% chance to take cover

Now it's pretty clear this description gets a bit tedious, it's better, especially with even more states, to write the probabilities as a matrix (rows represent the current state, columns the next state):

|   | A    | B    | C    | D    |
|---|------|------|------|------|
| A | 0.5  | 0.5  | 0    | 0    |
| B | 0    | 0.5  | 0.25 | 0.25 |
| C | 0.1  | 0.1  | 0.7  | 0.1  |
| D | 0.25 | 0.25 | 0.5  | 0    |

We can see a few things: the NPC can't immediately attack from cover, it has to search for a target first. It also can't throw two grenades in succession etc. Let's note that this model will now be yielding random sequences of actions such as *[cover, search, shoot, shoot, cover]* or *[cover, search, search, grenade, shoot]* but some of them may be less likely (for example shooting 3 bullets in a row has a probability of 0.1%) and some downright impossible (e.g. two grenades in a row). Notice a similarity to for example natural language: some words are more likely to be followed by some words than others (e.g. the word "number" is more likely to be followed by "one" than for example "cat").

## Code Example

Let's write an extremely primitive Markov bot that will work on the level of individual text characters. It will take a training text on input, for example a book, and learn the probabilities with which any letter is followed by another letter. Then it will generate a random output according to these probabilities, something that should resemble the training text. Yes, you may say we are doing a super simple machine learning.

Keep in mind this example is really extremely simple, it only looks one letter back and makes some further simplifications, for example it only approximates the probabilities with kind of a KISS hack -- we won't record any numeric probability, we'll only hold a table of letters, each one having a "bucket" of letters that may possibly follow; during training we'll always throw a preceding letter's follower to a random place in the preceding letter's bucket, with the idea that once we finish training, statistically in any bucket there will remain more letters that are more likely to follow given letter, just because we simply threw more such letters in. Similarly when generating the output text we will choose a letter to follow the current one by looking into the table and pulling out a random follower from that letter's bucket, again hoping that letters

that have greater presence in the bucket will be more likely to be randomly selected. This approach has issues, for example regarding the question of ideal bucket size, and it introduces statistical biases (maximum probability is limited by bucket size, order matters, later letters are kind of privileged), but it kind of works. Try to think of how we could make a better text generator -- for starters it might work on the level of words and could take into account a history of let's say three letters, i.e. it would record triplets of words and then list of words that likely follow, along with each one's probability that we would record as an actual number to make the probabilities accurate.

Anyway with all this said, below is a C code implementing the above described text generator. To use it just pipe some input ASCII text to it, however make it reasonably sized (a few thousand lines maybe, please don't feed it whole Britannica, the output won't be better), keep in mind the program always trains itself from scratch (in practice we might separate training from generation, as serious training might take very long, i.e. we would have a separate training program that would output a trained model, i.e. the learner probabilities, and then a generator that would only take the trained model and generate output text). Here is the code:

```
#include <stdio.h>
#include <stdlib.h>

#define OUTPUT_LEN 10000 // length of generated text
#define N 16 // bucket size for each letter
#define SEED 123456
#define IGNORE_NEWLINES 1

unsigned char charFollowers[256][N];

int main(void)
{
    srand(SEED);

    for (int i = 0; i < 256; ++i)
        for (int j = 0; j < N; ++j)
            charFollowers[i][j] = ' ';

    unsigned char prevChar = 0;

    while (1)
    {
        int c = getchar();

        if (c == EOF)
            break;

#ifdef IGNORE_NEWLINES
        if (c == '\n')
            c = ' ';
#endif

        charFollowers[prevChar][rand() % N] = c; // put char at random place
        prevChar = c;
    }

    prevChar = ' ';

    for (int j = 0; j < OUTPUT_LEN; ++j) // now generate the output
    {
        prevChar = charFollowers[prevChar][rand() % N]; // take random follower
        putchar(prevChar);
    }

    puts("\n");
    return 0;
}
```

Here it's pretty clear the code won't work but its structure really does resemble the original source: curly brackets and semicolons are correctly followed by newlines, assignments look pretty correct as well, dereference arrows (->) appear too -- the code even generated the RCL\_ prefix of the raycastlib functions that's widely seen in the original code.

# Marxism

*Not to be confused with communism.*

Marxism comprises ideas, theories and ideologies strongly based in works of Karl Marx (and also Friedrich Engels), roughly aiming for a revolution that should end capitalism, replace it with socialist society and eventually a truly communist society. Though the terms Marxism and communism are NOT the same (communism is a general idea with many branches, e.g. anarcho communism, Christian communism etc.), most common people see them as equivalent, which is unfortunate as communism is a purely good idea while Marxism is mostly a bad way of trying to achieve and sustain communism (the relationship between communism and Marxism may be roughly compared to the relationship between Christianity and Catholicism). Marxism comes with aggressive, revolutionary mindset that has caused great tragedies mainly during the 20th century, see mainly USSR. Marxism sees itself as a scientific effort, it is studied by intellectuals and lives mainly in written works of Marxists.

**We, LRS, are communists but do NOT embrace Marxism!** Though we do agree with Marxists on many things (mostly identifying issues of capitalism and the desire to replace it with some form of communism) and Marxism is still probably better than capitalism (as anything is better than capitalism), we can't ever approve of some inherent traits of Marxism, mainly the following. Marxism is **violent**, aggressive and revolutionary, often highly hostile to anarchism and pacifism (which we highly embrace), wanting to FORCE communism. Marxists promote establishment of temporary **dictatorship** of the proletariat as they see it the only way to ending capitalism. **Marxists are NOT altruists**, their desire of communism doesn't come from love of life, they simply see it as a more efficient societal system than capitalism that's a natural and inevitable next step in evolution of society, they see a man as servant of society, they are obsessed with work and see people who don't share their values as undesirable -- these undesirable people are not only capitalists, but also people who don't want to work, religious people or people who refuse to fight for their society. They say that ends justify the means and will happily utilize means such as war, dictatorship, executions, censorship, cults of personality and propaganda. We cannot ever stand behind this.

---

math

# Mathematics

Mathematics (also math or maths, from Greek *mathematicos*, *learned*) is the best science (yes, it is a formal science), which deductively deals with numbers and other abstract structures with the use of pure logic, in as rigorous and objective way as possible. In fact it's the only true science that can actually prove things thanks to its tool of mathematical proof (other sciences may only disprove or show something to be very likely). It is immensely important in programming and computer science. Mathematics is possibly the intellectually most difficult field to study in depth, meant for the smartest people; the difficulty, as some mathematicians themselves say, comes especially from the extremely deep abstraction (pure mathematics often examines subjects that have no known connection to reality and only exist as a quirk of logic itself). It is said that mathematics is the only **universal language** in our universe -- if we ever get in contact with an intelligent alien civilization, mathematics is likely to be used for communication. While most people only ever learn basic algebra and some other mechanical operations that are necessary for mathematics, true mathematics is not about blindly performing calculations, it is a creative discipline that constructs proofs from basic axioms, something that can frequently be extremely hard to do.

Some see math not as a science but rather a discipline that develops formal tools for "true sciences". The reasoning is usually that a science has to use scientific method, but that's a limited view as scientific method is not the only way of obtaining reliable knowledge. Besides that math can and does use the principles of scientific method -- mathematicians first perform "experiments" with numbers and generalize into conjectures and later "strong beliefs", however this is not considered good enough in math as it actually has the superior tool of proof that is considered the ultimate goal of math. I.e. math relies on deductive reasoning (proof) rather than less reliable inductive reasoning (scientific method) -- in this sense mathematics is more than a science.

Mathematics as a whole is constructed with logic from some basic system -- historically it was based e.g. on geometry, however modern mathematics has since about 19th century been built on top of set theory, i.e. all thing such as numbers, algebra and functions are all derived from just the existence of sets and classes and some basic operations with them. Specifically Zermeloâ Fraenkel set theory with axiom of choice (ZFC, made in the beginning of 20th century) is mostly used nowadays -- it's a theory with 9 axioms that we can consider kind of "assembly" of mathematics.

Soydevs, coding monkeys (such as webdevs) and just retards in general hate math because they can't understand it. They think they can do programming without math, which is just ridiculous. This delusion stems mostly from these people being highly incompetent and without proper education -- all they've ever seen was a shallow if-then-else python "coding" of baby programs or point-and-click "coding" in gigantic GUI frameworks such as Unity where everything is already preprogrammed for them. Of course this is not completely their fault (only partially), the shitty system just produces robot slaves who can't really think, just do some task to blindly produce goods for the economy etcetc. By Dunningâ Kruger they can't even see how incompetent they are and what real programming is about. In reality, this is like thinking that being able to operate a calculator makes you a capable mathematician or being able to drive a car makes you a capable car engineer. Such people will be able to get jobs and do some repetitive tasks such as web development, Unity game development or system administration, but they will never create anything innovative and all they will ever make will be ugly, bloated spaghetti solution that will likely do more harm than good.

On the other hand, one does not have to be a math PhD in order to be a good programmer in most fields. Sure, knowledge and overview of advanced mathematics is needed to excel, to be able to spot and sense elegant solutions and to innovate in big ways, but beyond these essentials that anyone can learn with a bit of will it's really more about just not being afraid of math, accepting and embracing the fact that it permeates what we do and studying it when the study of a new topic is needed.

**The power of math is limited** because the power of logic itself is limited. In 1930s this actually caused a big crisis in mathematics, connected to so called Hilbert's program which aimed to establish a completely "bulletproof" system to be the foundation of mathematics, however in 1932 Kurt Godel mathematically proved, with his incompleteness theorems, that (basically) there are logical truths which math itself can never prove, and that, put in a simplified way, "math itself cannot prove its own consistency", which of course killed Hilbert's program; since then we simply know we will never have a logically perfect system. This is related to the limited power of computers due to undecidability (there are problems a computer can never decide), proven by Alan Turing.

**What is mathematics really about?** Elementary school dropouts think math is about calculations and numbers -- sure, these are a big part of it but mathematicians mostly give a different answer. The core and art of high mathematics is constructing proofs, but it also involves exploration, a common theme is e.g. generalization: mathematicians love to take already existing knowledge and patterns and extend them into other domains, find more general rules of which currently known rules are only a special case. By this they are discovering universal laws and find that even seemingly unrelated concepts may have a lot in common.

## Overview

Following are some math areas and topics which a programmer should be familiar with:

- **basics** (high-school level math): arithmetic, algebra, expressions, basic functions, equations, geometry, trigonometry/goniometry, systems of linear equations, quadratic equations, complex numbers, logarithms, analytic geometry (many problems are equivalent to relationships of shapes in N dimensional spaces), polynomials (used in many areas, e.g. error correction codes in networking), ...
- **advanced notation**: ability to understand the notation that's often used in papers etc. (the big sigma for sum, calculus notation etc.)
- **formal logic**: computers are based on Boolean logic, knowing basic formulas and theorems here is crucial (e.g. the completeness of NAND or De Morgan's laws), formal logic is also just generally used in formal texts, one should know about predicate vs propositional logic etc.
- **proofs**: core of high level mathematics, one should know the basic proof techniques (direct, contradiction, induction, ...).
- **linear algebra**: aka "vectors and matrices", essential in almost every field (graphics, machine learning, ...).

- **calculus and differential equations:** just essential for advanced math and many fields (graphics, machine learning, electronics, physics, any optimization, ...).
- **theoretical computer science:** computational complexity (very important), computability, formal languages, computational models (automata, Turing machines, ...), ...
- **graph theory:** generally useful tools, especially important e.g. in networks or indexing structures in databases.
- **number and set theory:** sets and operations with them (basis of all mathematics), classes, sets of numbers (natural, rational, real, complex, ...), prime numbers (important e.g. for cryptography, quantum computing, ...), ...
- **discrete math:** basic structures such as groups and fields, abstract algebras and the properties of these structures.
- **signal processing:** Fourier transform and other integral transforms (important e.g. for compression and analysis of signals), aliasing, filter theory, ...
- **numerical methods:** for simulations and approximations of solutions to problems we can't solve exactly.
- **probability/statistics:** encountered practically everywhere but very important e.g. in cryptography.
- **other:** things important in specific fields and/or other weird stuff, e.g. topology, quaternions (graphics, physics), lambda calculus, game theory, fractal geometry, ...

## See Also

- knowability
- logic
- science
- thrembo

---

mechanical

## Mechanical Computer

Mechanical computer (simple ones also being called *mechanical calculators*) is a computer that uses mechanical components (e.g. levers, marbles, gears, strings, even fluids ...) to perform computation (both digital and analog). Not all non-electronic computers are mechanical, there are still other types too -- e.g. computers working with light, biological, quantum, pen and paper computers etc. Sometimes it's unclear what counts as a mechanical computer vs a mere calculator, an automaton or mere instrument -- here we will consider the term in a very wide sense. Mechanical computers used to be used in the past, mainly before the development of vacuum tubes and transistors that opened the door for much more powerful computers. However some still exist today, though nowadays they are usually intended to be educational toys, they are of interest to many (including us) as they offer simplicity (independence of electricity and highly complex components such as transistors and microchips) and therefore freedom. They may also offer help after the collapse. While nowadays it is possible to build a simple electronic computer at home, it's only thanks to being able to buy highly complex parts at the store, i.e. still being dependent on corporations; in a desert one can much more easily build a mechanical computer than electronic one. Mechanical computers are very cool.

{ Britannica 11th edition has a truly amazing article on mechanical computers under the term *Calculating Machines*: [https://en.wikisource.org/wiki/1911\\_Encyclop%C3%A6dia\\_Britannica/Calculating\\_Machines](https://en.wikisource.org/wiki/1911_Encyclop%C3%A6dia_Britannica/Calculating_Machines). Also this leads to many resources: <https://www.johnwolff.id.au/calculators/Resources.htm>. ~drummyfish }

If mechanical computer also utilizes electronic parts, it is called an **electro-mechanical** computer; here we'll however be mainly discussing purely mechanical computers.

**Disadvantages** of digital mechanical computers against electronic ones are great, they basically lose at everything except simplicity of implementation (in the desert). Mechanical computer is MUCH slower (speed will be measured in Hz), has MUCH less memory (mostly just a couple of bits or bytes), will be difficult to program (machine code only), is MUCH bigger, limited by mechanical friction (so it will also be noisy), suffers from mechanical wear etc. Analog mechanical computers are maybe a bit better in comparison, but still lose to electronics big time. But remember, less is more.

Some **notable mechanical computers** include e.g. the 1882 Difference Engine by Charles Babbage (aka the first programmer), Antikythera mechanism (ancient Greek astronomical computer), the famous Curta calculators (quality, powerful pocket-sized mid-20th century calculators) { These are really cool, check them out. ~drummyfish }, Enigma ciphering device (used in WWII), abacus, slide rule, Odhner Arithmometer (extremely popular Russian table calculator), Digi-Comp I (educational programmable 3 bit toy computer) or Turing Tumble { Very KISS and elegant, also check out. ~drummyfish } (another educational computer, using marbles).

Let's also take a look at how we can classify mechanical computers. Firstly they can be:

- **special purpose:** Made to solve only limited set of problems, example being a mechanical calculator that can only perform a few operations like addition and subtraction. A special purpose computer may be easier to make as it doesn't have to bother with the flexibility needed for solving general problems.
- **general purpose:** Full programmable Turing complete computer capable of solving very wide range of tasks efficiently. This is of course harder to make, so general purpose mechanical computers are rarer.

Next we may divide mechanical computers to:

- **analog:** Working with analog data, i.e. continuous, infinitely precise values, typical examples are various integration machines. The analog approach is probably more natural and efficient in the mechanic world, so we encounter many of analog computers here (compared to the electronic world).
- **digital:** Working with discrete values, i.e. whole numbers, bits etc.
- **analog-digital:** Combination of both digital and analog, again this is more common in mechanic world than in electronic world.

And to:

- **autonomous:** Computers that only require to be started and then work completely on their own, without human intervention.
- **semi-autonomous:** Computers largely working on their own but still requiring some human assistance during computation, for example turning some handle to keep the parts moving.
- **computation helpers:** Tools that only aid the man who is doing most of the computation -- typical examples are abacus, slide rule or integrator.

## Basics

**Analog** computers are usually special purpose. { At least I haven't ever heard about any general purpose analog computer, not even sure if that could work. ~drummyfish } Very often they just solve some specific equation needed e.g. for computing ballistic curves, they may perform Fourier transform, compute areas of arbitrary shapes that can be traced by a pencil (see planimeter) etc. Especially useful are computers performing integration and solving differential equations as computing many practically encountered equations is often very hard or impossible -- mechanical machines can integrate quite well, e.g. using the famous ball and disk integrator.

As mere programmers let us focus more on **digital** computers now.

When building a digital computer from scratch we usually start by designing basic logic gates such as AND, NOT and OR -- here we implement the gates using mechanical principles rather than transistors or relays. For simple special-purpose calculators combining these logic gates together may be enough (also note we don't HAVE TO use logic gates, some mechanisms can directly perform arithmetic etc.), however for a highly programmable general purpose computer **logic gates alone practically won't suffice** -- in theory when we have finite memory (in real world always), we can always just use only logic gates to perform any computation, but as the memory grows, the number of logic gates we would need would grow exponentially, so we don't do this. Instead we will need to additionally implement some **sequential processing**, i.e. something like a CPU that performs steps according to program instructions.

Now we have to choose our model of computation and general architecture, we have possibly a number of options. Mainly we may be deciding between having a separate storage for data and program (Harvard architecture) or having the program and data in the same memory (intending for the computer to "reshape"

this initial program data into the program's output). Here there are paths to explore, the most natural one is probably trying to imitate a **Turing machine** (many physical finite-tape Turing machines exist, look them up), probably the simplest "intuitive" computer, but we can even speculate about e.g. some kind of rewriting system imitating formal grammars, cellular automata etc -- someone actually built a simple and elegant rule 110 marble computer (look up on YT), which is Turing complete but not very practical (see Turing tarpit). So Turing machine seems to be the closest to our current idea of a computer (try to program something useful in rule 110...), it's likely the most natural way, so that might be the best first choice we try.

Turing machine has a separate memory for program and data. To build it we need two main parts: memory tape (an array of bits) and control unit (table of states and their transitions). We can potentially design these parts separately and let them communicate via some simple interface, which simplifies things. The specific details of the construction will now depend on what components we use (gears, marbles, dominoes, levers, ...).

## Concepts

Here we will overview some common concepts and methods used in mechanical computers. Remember the concepts may, and often are, **combined**. Also note that making a mechanical computer will be a lot about mechanical engineering, so great many concepts from it will appear -- we can't recount all of them here, we'll just focus on the most important concepts connected to the computing part.

## Gears/Wheels

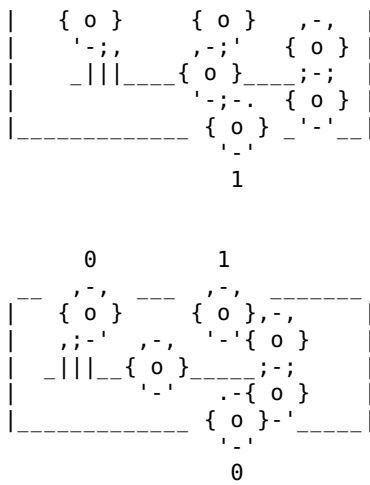
Gears (wheels with teeth) are a super simple mechanism popular in mechanical computers. Note that **gears may be both digital and analog** -- whether they're one or the other depends on our interpretation (if we assign importance to every arbitrary orientation or just a finite number of orientations that click the tooth into some box).

The **advantages** of gears are for example instant transfer of motion -- even if we have many wheels in sequence, rotating the first one instantly (practically) rotates the last one as well. Among **disadvantages** on the other hand may be the burden of friction (having too many gears in a row will require a lot of power for rotation and strong fixation of the gears) and also manufacturing a non-small number of good, quality gears may be more difficult than alternatives (marbles, ...).

Besides others gears/wheels can be used to:

- **Transmit power**, i.e. delivering motion to components that need motion to work (even in computers that don't use gears themselves as computing components).
- **Do arithmetic**: for example a differential can be used to instantly add two numbers (or actually to compute any linear combination, e.g. average, ... using the slide rule concept we can probably even implement multiplication, division etc.). A simple stepped-cylinder (kind of a "gear") alongside with normal gears can also be used to implement addition, subtraction and even multiplication (as explained e.g. in 1911 Encyclopedia Britannica; this principle was used e.g. by the old Russian calculators).
- **Represent a general digital value** by how they are currently rotated, i.e. a gear with  $N$  teeth -- each one labeled with a value -- can hold one of  $N$  values depending on which of the values is currently under some pointer -- this is often used in mechanical calculators e.g. to display computed values. This has the advantage of being able to represent a digital number with one relatively simple part (the wheel) without having to encode multiple bits (i.e. many smaller parts). This may also be used to make a look up table -- imagine e.g. a wheel which by rotating looks up some value that may be represented e.g. by displacement (imagine spinning spiral) or holes on the wheel. If the gear represents a natural number, it naturally implements modulo increment/decrement (highest value will overflow to lowest and vice versa).
- **Represent one bit** by turning either clockwise or counterclockwise.
- Possibly represent values also in other ways, for example by speed of rotation, rotation vs stillness, position (gear traveling on some toothed slider, ...) etcetc.
- ...

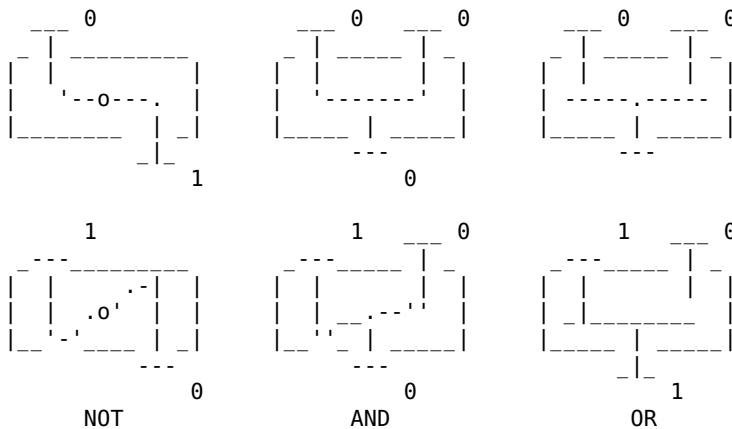
1            1  
 -- , - , -- , - , -----



*NXOR (equality) gate implemented with gears (counterclockwise/clockwise rotation mean 1/0); the bottom gear rotates counterclockwise only if the both input gears rotate in the same direction.*

## Buttons/Levers/Sliders/Etc.

Buttons, levers, sliders and similar mechanism can be used in ways similar to gears, the difference being their motion is linear, not circular. A button can represent a bit with its up/down position, a lever can similarly represent a bit by being pointed left/right. As seen below, implementation of basic logic gates can be quite simple, which is an advantage. Disadvantages include for example, similarly to gears, vulnerability to friction -- with many logic gates in a row it will be more difficult to "press" the inputs.



*Possible implementation of logic gates with buttons.*

## Marbles/Balls/Coins/Etc.

Using moving marbles (and possibly also similar rolling shapes, e.g. cylinders, disks, ...) for computation is **one of the simplest** and most KISS methods for mechanical computers and may therefore be considered very worthy of our attention -- the above mentioned marble rule 110 computer is a possible candidate for **the most KISS Turing complete computer**. But even with a more complicated marble computer it's still much easier to build a "marble maze" than to build a geared machine (even gears themselves aren't that easy to make).

**Basic principle** is that of a marble performing computation by going through a maze -- while a single marble can be used to evaluate some simple logic circuit, usually (see e.g. Turing Tumble) the design uses many marbles and performs sequential computation, i.e. there is typically a **bucket** of marbles placed on a high place from which we release one marble which (normally by relying on gravity) goes through the maze and performs one computation cycle (switches state, potentially flips a memory bit etc.) and then, at the bottom (end of its path), presses a switch to release the next marble from the top bucket. So the computation is autonomous, it consumes marbles from the top bucket and fills the bottom bucket (with few



marbles available an operator may sometimes need to refill the top bucket from the bottom one). The maze is usually an angled board onto which we just place obstacles; multiple layers of boards with holes/tunnels connecting them may be employed to allow more complexity. { You can build it from lego probably. ~drummyfish }

If it's possible it may be actually simpler to use coins instead of marbles -- as they are flat, building a potentially multi-layered maze (e.g. with shifting elements) can be easier, it may be as simple as cutting the layers out of thick cardboard paper and stacking them one on another.

Also an alternative to having a top bucket full of marbles going to the bottom bucket is just having one marble and transporting it back up after each cycle -- this can be done very simply e.g. by tilting the maze the other way, so the computation is then powered by someone (or something) repeatedly tilting the board one way and back again; this is e.g. how the simple rule 110 computer works -- there the marble also does another work on its way back (it updates the barriers in the maze for itself and its neighbors for the next round of the downwards trip), so the "CPU cycle" has two phases.

NOTE: Balls, or potentially other "falling/moving objects", may be used to perform computation also in other ways than we'll describe further on -- some of the alternative approaches are for example:

- The **billiard ball computer** (which also has a great advantage of performing reversible computation).
- Another possible idea is that of the falling object ITSELF encoding a value (likely just a bit), for example imagine some kind of arrow shape which itself represents either 1/0 by pointing up/down, changing its orientation as it passes through the gates (we would also have to ensure the orientation can't change spontaneously on its own of course).
- A bit can also be represented by presence/absence of the marble -- this is utilized e.g. by *binary marbles* (<https://binarymarbles.weebly.com/how-it-works.html>). For example the AND gate is implemented by one input marble falling into a hole, making a "bridge" for the other marble that then overcomes the hole and reaches output. Timing may play an important role as some gates (e.g. XOR) require dropping the input marbles simultaneously.
- ...

These approaches may be tried as well, however further on here we will focus on the traditional "marble maze" approach in which the marbles mostly serve as a kind movement force that flips bits represented by something else (or possibly indicate answer by falling out through a specific hole).

The **disadvantage** here is that the computation is **slow** as to perform one cycle a marble has to travel some path (which may take many seconds, i.e. in the simple form you get a "CPU" with some fractional frequency, e.g. 1/5 Hz). This can potentially be improved a little, e.g. by pipelining (releasing the next marble as soon as possible, even before the current one finishes the whole path) and parallelism (releasing multiple marbles, each one doing some part of work in parallel with others). **Advantages** on the other hand include mentioned simplicity of construction, visual clarity (we can often make the computer as a flat 2D board that can easily be observed, debugged etc.) and potentially good extensibility -- making the pipeline (maze) longer, e.g. to add more bits or functionality, is always possible without being limited e.g. by friction like with gears (though usually for the cost of speed).

Some things that can be done with marbles include:

- **Flipping/setting bits:** A marble running through some specific part of the maze can flip something over, which we may interpret as flipping a bit. A simple rotating "T" shape can be used to make a one bit flip-flop (see below).
- **Branching:** If/else/switch branching can be implemented simply as a marble taking one road or another on a crossroad, which can be decided by some moving part connected to some bit elsewhere.
- **Rotating gears:** A marble may rotate a gear by precise number of teeth, this can be used e.g. to implement a shift on memory tape (see pictures below).
- **Weight/count representing a value:** instead of encoding a number with flippable bits we may instead use a small bucket into which marbles fall -- the number of marbles in the bucket then encode the number stored. We may e.g. introduce a limit number (if  $x > N$ ), i.e. weight at which the bucket becomes heavier than a counterweight and opens some new path for the marbles.
- ...



- **Electronics emulation:** it's known many electronic concepts can be imagined with water pipes instead that deal with similar concepts (pressure  $\sim$  voltage, flow  $\sim$  current, resistor  $\sim$  narrower pipe, ...). By this we may possibly emulate very simple electronics without actual electricity.
- ...

## Other

Don't forget there exist many other possible components and concepts a mechanical computer can internally use -- many things we leave out above for the questionability of their practical usability can be used to in fact carry out computation, for example dominoes or slinkies. Furthermore many actually useful things exist, e.g. teathed **cylinders/disks** may be used to record plots of data over time or to store and deliver read/only data (e.g. the program instructions) easily, see music boxes and gramophones; **punch card and paper tapes** have widely been used for storing read-only data too. Sometimes deformed cylinders were used as an analog **2D look up table** for some mathematical function -- imagine e.g. a device that has input  $x$  (rotating cylinder along its axis) and  $y$  (shifting it left/right); the cylinder can then at each surface point record function  $f(x,y)$  by its width which will in turn displace some stick that will mark the function value on a scale. To transfer movement **strings, chains and belts** may also be used. Random number generation may be implemented e.g. with Galton board. If timing is needed, pendulums can be used just like in clock. Some mechanical computers even use pretty complex parts such as mechanical arms, but these are firstly hard to make and secondly prone to breaking, so try to avoid complexity as much as possible. Some old mechanical calculators worked by requiring the user to plug a stick into some hole (e.g. number he wanted to add) and then manually trace some path -- this can work on the same principle as e.g. the marble computer, but without needing the marbles complexity and size are drastically reduced. Another ideas is a "combing" computer which is driven by its user repeatedly sliding some object through the mechanism (as if combing it) which performs the steps (sequential computation) and changes the state (which is either stored inside the computer or in the combing object).

BONUS THOUGHT: We have gotten so much used to using our current electronic digital computers for everything that sometimes we forget that at simulating actual physical reality they may still fail (or just be very overcomplicated) compared to a mechanical simulation which **USES** the physical reality itself; for example to make a simulation of a tsunami wave it may be more accurate to build an actual small model of a city and flood it with water than to make a computer simulation. That's why aerodynamic tunnels are still a thing. Ancient NASA flight simulators of space ships did use some electronics, but they did not use computer graphics to render the view from the ship, instead they used a screen projecting view from a tiny camera controlled by the simulator, moving inside a tiny environment, which basically achieved photorealistic graphics. Ideas like these may come in handy when designing mechanical computers as simulating reality is often what we want to do with the computer; for example if we want to model a sine function, we don't have to go through the pain of implementing binary logic and performing iterative calculation of sine approximation, we may simply use a pendulum whose swinging draws the function simply and precisely.

---

memory\_management

## Memory Management

In programming memory management is (unsurprisingly) the act and various techniques of managing the working memory (RAM) of a computer, i.e. for example dividing the total physically available memory among multiple memory users such as operating system processes and assuring they don't illegally access each other's part of memory. The scope of the term may differ depending on context, but tasks falling under memory management may include e.g. memory allocation (finding and assigning blocks of free memory) and deallocation (freeing such blocks), ensuring memory safety, organizing blocks of memory and optimizing memory access (e.g. with caches or data reorganization), memory virtualization and related tasks such as address translation, handling out-of-memory exceptions etc.

Memory management can be handled at different levels: hardware units such as the MMU and CPU caches exist to perform certain time-critical memory-related tasks (such as address translation) quickly, operating system may help with memory management (e.g. implement virtual memory and offer syscalls for dynamic allocation and deallocation of memory), a programming language may do some automatic memory management (e.g. garbage collection or handling call stack) and programmer himself may do his own memory management (e.g. deciding between static and dynamic allocation or choosing the size of dynamic

allocation chunk).

**Why all this fuzz?** As a newbie programmer who only works with simple variables and high level languages like Python that do everything for you you don't need to do much memory management yourself, but when working with data whose size may wildly differ and is not known in advance (e.g. files), someone has to handle e.g. the possibility of the data on disk not being able to fit to RAM currently allocated for your program, or -- if the data fits -- there may not be a big enough continuous chunk of memory for it. If we don't know how much memory a process will need, how much memory do we give it (too little and it may not be enough, too much and there will not be enough memory for others)? Someone has to prevent memory leaks so that your computer doesn't run out of memory due to bugs in programs. With many processes running simultaneously on a computer someone has to keep track of which process uses which part of memory and ensure collisions (one process overwriting another process's memory) don't happen, and someone needs to make sure that if bad things happen (such as process trying to write to a memory that doesn't belong to it), they don't have catastrophic consequences like crashing or exploding the system.

## Memory Management In C

In C -- a low level language -- you need to do a lot of **manual** memory management and there is a **big danger of fucking up**, especially with dynamic allocation -- C won't hold your hand (but as a reward your program will be fast and efficient), there is no uber memory safety. There is no automatic garbage collection, i.e. if you allocate memory dynamically, YOU need to keep track of it and manually free it once you're done using it, or you'll end up with memory leak.

For start let's see which kinds of allocation (and their associated parts of memory) there are in C:

- **static allocation (code/data memory):** Simplest kind of allocation happening at compile time: if the compiler can do so (i.e. if it knows enough things such as the size of the data in advance), it allocates space of concrete size at some specific address in the part of memory reserved for code or static data (code and data may be in the same or separate parts depending on platform, see e.g. Harvard architecture) -- this is straightforward, simple, automatic and poses no real dangers, bloat or burden of dependencies. This kind of allocation applies to:
  - ♦ **global variables** (variables declared outside any function, i.e. even outside main)
  - ♦ **static variables** (variables inside functions declared with static keyword)
  - ♦ **constants/literals** (e.g. strings in the source code such as "abc")
- **automatic allocation (stack memory):** For local variables (variables inside functions) the memory is allocated in a special part of memory known as **call stack** only at the time when the function is actually called and executed; i.e. this is similar to dynamic allocation (it happens at run time) but happens automatically, without needing any libraries or other explicit actions from the programmer. I.e. when a function is called at run time, a new *call frame* is created on stack which includes space for local variables of that function (along with e.g. return address from the function etc.). This is necessary e.g. to allow recursion (during which several instances of the same function may be active, each of which may have different values of its variables), and it also helps consume less RAM. This allows for creating variable sized arrays inside functions (e.g. `int array[x];` where x is variable) which is not possible to do with a global array (however variable size arrays aren't supported in old ANSI C!). The disadvantage over dynamic allocation is that stack memory is relatively small and overusing it may easily cause stack overflow (running out of memory). Still this kind of allocation is better than dynamic allocation as it doesn't need any libraries, it doesn't generate complex code and the only danger is that of stack overflow -- memory leaks can't happen (deallocation happens automatically when function is exited). Automatic allocation applies to:
  - ♦ **local variables** (including function arguments and local **variable size arrays**)
- **dynamic allocation (heap memory):** A kind of more complex manual allocation that happens at run time and is initiated by the programmer calling special functions such as `malloc` from the `stdlib` standard library, which return pointers to the allocated memory. This memory is taken from a special part of memory known as **heap**. This allows to allocate, resize and deallocate potentially very big parts of memory, but requires caution as working with pointers is involved and there is a danger of **memory leaks** -- it is the responsibility of the programmer to free allocated memory with the `free` function once it is no longer needed, otherwise that memory will simply remain allocated and unusable by others (if this happens for example in a loop, the program may just start eating up more and more RAM and eventually run out of memory). Dynamic allocation is also pretty complex (it usually involves communicating with operating system and also keeping track of the structure of

memory) and creates a dependency on the stdlib library. Some implementations of the allocation functions are also infamously slow (up to the point of some programmers resorting to program their own dynamic allocation systems). Therefore only use dynamic allocation when absolutely necessary! Dynamic allocation applies to:

♦ **memory allocated with special functions** (malloc, calloc, realloc)

Rule of the thumb: use the simplest thing possible, i.e. static allocation if you can, if not then automatic and only as the last option resort to dynamic allocation. The good news is that **you mostly won't need dynamic allocation** -- you basically only need it when working with data whose size can potentially be VERY big and is unknown at compile time (e.g. you need to load a WHOLE file AT ONCE which may potentially be VERY big). In other cases you can get away with static allocation (just reserving some reasonable amount of memory in advance and hope the data fits, e.g. a global array such as `int myData[DATA_MAX_SIZE]`) or automatic allocation if the data is reasonably small (i.e. you just create a variable sized array inside some function that processes the data). If you end up doing dynamic allocation, be careful, but it's not THAT hard to do it right (just pay more attention) and there are tools (e.g. valgrind) to help you find memory leaks. However by the principles of good design **you should avoid dynamic allocation** if you can, not only because of the potential for errors and worse performance, but most importantly to avoid dependencies and complexity.

For pros: you can also create your own kind of pseudo dynamic allocation in pure C if you really want to avoid using stdlib or can't use it for some reason. The idea is to allocate a big chunk of memory statically (e.g. `global unsigned char myHeap[MY_HEAP_SIZE];`) and then create functions for allocating and freeing blocks of this static memory (e.g. `myAlloc` and `myFree` with same signatures as `malloc` and `free`). This allows you to use memory more efficiently than if you just dumbly (is it a word?) preallocate everything statically, i.e. you may need less total memory; this may be useful e.g. on embedded. Yet another uber hack to "improve" this may be to allocate the "personal heap" on the stack instead of statically, i.e. you create something like a global pointer `unsigned char *myHeapPointer;` and a global variable `unsigned int myHeapSize;`, then somewhere at the beginning of `main` you compute the size `myHeapSize` and then create a local array `myHeap[myHeapSize]`, then finally set the global pointer to it as `myHeapPointer = myHeap;` the rest remains the same (your allocation function will access the heap via the global pointer). Just watch out for reinventing wheels, bugs and that you actually don't end up with a worse mess than if you took a more simple approach. Hell, you might even try to write your own garbage collection and array bound checking and whatnot, but then why just not fuck it and use an already existing abomination like Java? :)

Finally let's see some simple code example:

```
#include <stdio.h>
#include <stdlib.h> // needed for dynamic allocation :(

#define MY_DATA_MAX_SIZE 1024 // if you'll ever need more, just change this and recompile

unsigned char staticMemory[MY_DATA_MAX_SIZE]; // statically allocated array :)
int simpleNumber; // this is also allocated statically :)

void myFunction(int x)
{
    static int staticNumber; // this is allocated statically, NOT on stack
    int localNumber;         // this is allocated on stack
    int localArray[x + 1];    // variable size array, allocated on stack, hope x isn't too big

    localNumber = 2 * x;      // do something with the memory
    localArray[x] = localNumber;

    if (x > 0)                // recursively call the function
        myFunction(x - 1);
}

int main(void)
{
    int localNumberInMain = 123; // this is also allocated on stack

    myFunction(10); // change to 10000000 to see a probable stack overflow

    for (int i = 0; i < 200000; ++i)
    {
```

```

if (i % 1000 == 0)
    printf("i = %d\n",i);

unsigned char *dynamicMemory = (char *) malloc((i + 1) * 10000); // oh no, dynamic allocation, BLOAAAT!

if (!dynamicMemory)
{
    printf("Couldn't allocate memory, there's probably not enough of it :/");
    return 1;
}

dynamicMemory[i * 128] = 123; // do something with the memory

free(dynamicMemory); // if not done, memory leak occurs! try to remove this and see :)
}

return 0;
}

```

---

mental\_outlaw

## Mental Outlaw

Mental Outlaw is a black/N-word youtuber/vlogger focused on FOSS and, to a considerable degree, suckless software. He's kind of a copy-paste of Luke Smith but a little closer to the mainstream and normies.

Like with Luke, sometimes he's real based and sometimes he says very stupid stuff. Make your own judgement.

---

microsoft

## Micro\$oft

Micro\$oft (officially Microsoft, MS) is a terrorist organization, software corporation named after its founder's dick -- it is, along with Google, Apple et al one of the biggest organized crime groups in history, best known for holding the world captive with its highly abusive "operating system" called Windows, as well as for leading an aggressive war on free software and utilizing many unethical and/or illegal business practices such as destroying any potential competition with the Embrace Extend Extinguish (actual terminology internally used at Microsoft) strategy or lately practicing heavy openwashing.

{ Techrights documents Microsoft nicely on their wiki, see e.g. "Microsoft sins" at [http://techrights.org/wiki/List\\_of\\_Microsoft\\_Sins](http://techrights.org/wiki/List_of_Microsoft_Sins), which among others list illegally shooting an antelope, tax evasion, attacking (physically and verbally) employees who leave for other companies and bribing bloggers to write positive reviews. ~drummyfish }

Microsoft is unfortunately among the absolutely most powerful entities in the world (that sucks given they're also among the most hostile ones) -- likely more powerful than any government and most other corporations, it is in their power to **immediately destroy any country** with the push of a button, it's just a question of when this also becomes their interest. This power is due to them having **complete control over almost absolute majority of personal computers in the world** (and therefore by extension over all devices, infrastructure, organization etc.), through their proprietary (malware) "operating system" Windows that has built-in backdoor, allowing Microsoft immediate access and control over practically any computer in the world. The backdoor "feature" isn't even hidden, it is officially and openly admitted (it is euphemistically called auto updates). Microsoft prohibits studying and modification of Windows under threats including physical violence (tinkering with Windows violates its EULA which is a lawfully binding license, and law can potentially be enforced by police using physical force). Besides legal restrictions Microsoft applies high obfuscation, bloat, SAASS and other techniques preventing user freedom and defense against terrorism, and forces its system to be installed in schools, governments, power plants, hospitals and basically on every computer anyone buys. Microsoft can basically (for most people) turn off the Internet, electricity, traffic control system etc. Therefore every hospital, school, government and any other institution has to bow to Microsoft.

TODO: it would take thousands of books to write just a fraction of all the bad things, let's just add the most important ones

---

microtheft

## Microtheft

See [microtransaction](#).

---

microtransaction

## Microtransaction

Microtransaction, also microtheft, is the practice of selling -- for a relatively "low" price -- virtual goods in some virtual environment, especially [games](#), by the owner of that environment. It's a popular business model of many [capitalist games](#) -- players have an "option" (which they are pushed to take) to buy things such as skins and purely cosmetic items but also items giving an unfair advantage over other players (in-game currency, stronger weapons, faster leveling, ...). This is often targeted at children.

Not only don't they show you the source code they run on your computer, not only don't they even give you an independently playable copy of the game you paid for, not only do they spy on you, they also have the audacity to ask for more and more money after you've already paid for the thing that abuses you.

---

military

## Military

Military is the official [state](#) terrorist organization specialized in killing, genocide, torture, destruction, rape and other oppression of people outside the state's country, mainly through physical force; the purpose of military is [war](#) for which it largely consists of [army](#), a group of professional murderers who, for doing the most immoral things imaginable, get a great amount of benefits from the state, such as financial security and propaganda support (the title of [hero](#)). Military complements [police](#), a similar organization which however specializes in killing people inside the country.

**If you can, get military technology**, e.g. from army shops etc. (but be sure to get the true army stuff, not just something branded as such). The reason is that military gets the best technology -- unlike [consumer](#) technology, which is designed to break on purpose and never last very long, the army needs reliable, durable tools that last long, work in harsh conditions and can be repaired in the field. So if you can get your hands on a military laptop, go for it.

---

minigame

## Minigame

Minigame is a very small and simple [game](#) intended to entertain the player in a simple way, usually for only a short amount of time, unlike a full fledged game. Minigames may a lot of times be embedded into a bigger game (as an [easter egg](#) or as a part of a game mechanic such as lock picking), they may come as an extra feature on primarily non-gaming systems, or appear in collections of many minigames as a bigger package (e.g. various party game collections). Minigames include e.g. [minesweeper](#), [sokoban](#), the Google [Chrome](#) T-rex game, [Simon Tatham's Portable Puzzle Collection](#), as well as many of the primitive old games like [Pong](#) and [Tetris](#). Minigames are nice from the [LRS](#) point of view as they are [minimalist](#), simple to create, often [portable](#), while offering a potential for great [fun](#) nevertheless.

Minigame is an ideal project for learning [programming](#).

Despite the primary purpose of minigames many players invest huge amounts of time into playing them,

usually competitively e.g. as part of speedrunning.

Minigames are still very often built on the principles of old arcade games such as getting the highest score or the fastest time. For this they can greatly benefit from procedural generation (e.g. endless runners).

## List Of Minigames

This is a list of just some of many minigames and minigame types.

- **2048**
- **arkanoid**
- **asteroids**
- **backgammon**
- **button smasher**: Games whose goal is achieved mainly by smashing a button as quickly as possible, usually e.g. sprint simulators. This may perhaps even include a game that requires you to press a button as quickly as possible (achieve fastest reaction time).
- **card games**
- **checkers**
- **chess**, its variants and chess puzzles
- **city bomber**: A plane is descending on the screen, player has to drop bombs to destroy building so that it can land.
- **concentration**
- **donkey kong**
- **dots and boxes**
- **endless runner**
- **fifteen**
- **flappy bird**
- **game of life**
- **go**, especially a small board one or variants such as atari go
- **guess a number**
- **hangman**
- **invaders**
- **jigsaw puzzle**
- **knowledge quiz**
- **loderunner**
- **ludo**
- **lunar lander**
- **mahjong**
- **maze**
- **minigolf**
- **minesweeper**
- **pacman**
- **pinball**
- **poker**
- **pong**
- **racetrack**
- **rock-paper-scissors**
- **shoot'em up**
- **snake**
- **sokoban**
- **solitaire**
- **sprouts**
- **sudoku**
- **tangram**
- **tetris** (block game, "tetris" is trademarked)
- **The Witness puzzles**: The kind of puzzles that appear in the game The Witness.
- **tic-tac-toe**
- **tower of hanoi**
- **tron**
- **untangle**



minimalism

## Minimalism

*No gain, no pain.*

In context of technology minimalism is a design philosophy which puts great emphasis on simplicity, it says technology should be as simple as possible while still achieving given goal, possibly even a little bit simpler. Minimalism is one of the most (if not the most) important concepts in programming and technology in general, it could almost be said that becoming a true expert in technology is strongly connected to realizing the importance of simplicity (see e.g. Unix philosophy). One of the first things to stress about minimalism is that it's firstly about **internal simplicity**, i.e. the simplicity of design/repairing/hacking, and only secondly about the simplicity from the user's point of view (otherwise we are only dealing with pseudominimalism). The opposite of minimalism is maximalism. See also minimal viable program.

Antoine de Saint-Exupéry sums it up with a quote: *we achieve perfection not when there is nothing more to add but when there is nothing left to take away.*

Minimalism is also an immensely important concept in art, for example in architecture and design, and in addition there also exists the generalized concept of **life minimalism** which applies said philosophy to all areas of life and which numerous technological minimalists quite naturally start to follow along the way -- life minimalism is about letting go of objects, thoughts and desires that aren't necessarily needed because such things enslave us and mostly just make us more miserable; from time to time you should meditate a little bit about what it is that you really want and need and only keep that. Indeed this is nothing new under the Sun, this wisdom has been present for as long as humans have existed, most religions and philosophers saw a great value in asceticism, frugality and even poverty, as owning little leads to freedom. For instance owning a car is kind of a slavery, you have to clean it, protect it, repair it, maintain it, pay for parking space, pay for gas, pay for insurance -- this is not a small commitment and you sacrifice a significant part of your life and head space to it (especially considering additional commitments of similar magnitude towards your house, garden, clothes, electronics, furniture, pets, bank accounts, social networks and so forth), a minimalist will rather choose to get a simple suckless bicycle, travel by public transport or simply walk.

**Minimalism is necessary for freedom** as a free technology can only be that over which no one has a monopoly, i.e. which many people and small parties can utilize, study and modify with affordable effort, without needing armies of technicians just for the maintenance of such technology. Minimalism goes against the creeping overcomplexity of technology which always brings huge costs and dangers, e.g. the cost of maintenance and further development, obscurity, inefficiency ("bloat", wasting resources), consumerism, the increased risk of bugs, errors and failure.

There is a so called airplane rule that states a plane with two engines has twice as many engine problems than a plane with a single engine.

Up until recently in history every engineer would tell you that *the better machine is that with fewer moving parts*. This still seems to hold e.g. in mathematics, a field not yet so spoiled by huge commercialization and mostly inhabited by the smartest people -- there is a tendency to look for the most minimal equations -- such equations are considered beautiful. Science also knows this rule as the Occam's razor. In technology invaded by aggressive commercialization the situation is different, minimalism lives only in the underground and is ridiculed by the mainstream propaganda. Some the minimalist movements, terms and concepts at least somewhat connected to minimalism include (watch out for SJWs, pseudominimalism, Nazis etc., we don't automatically fully embrace all things on this list):

- suckless
- cat-v
- bitreich
- Collapse OS/Dusk OS, collapse computing
- less retarded software

- Unix philosophy
- KISS
- countercomplex
- permacomputing (SJW cancer warning)
- less is more/worse is better
- appropriate technology
- neoluddism
- reactionary software (bordering with pseudominimalism)
- plan9, openbsd, KISS GNU/Linux and similar (however often obsessed with bloatcryption, may contain pseudominimalism)
- 100rabbits (beware of SJW poison)
- small Internet, web 1.0, web 0.5, gopher, gemini (watch out: gemini is SJW pseudominimalist bloatcryption poison), ...
- primitivism/anarcho primitivism, low tech, ...
- for potential weaker links to minimalism also check out retro/old/boomer tech, salvage computing, degrowth, Amish, technophobia, demoscene, code golf, lightweight software, fantasy consoles (sadly mostly pseudominimalism), communities around plain text, pubnixes, some GNU/Linux distros (e.g. Arch, Gentoo, KISS Linux, ...), IRC communities and so on.
- ...

Under capitalism technological minimalism is suppressed in the mainstream as it goes against corporate interests, i.e. those of having monopoly control over technology, even if such technology is "FOSS" (which then becomes just a cool brand, see openwashing). We may, at best, encounter a "shallow" kind of minimalism, so called pseudominimalism which only tries to make things appear minimal, e.g. aesthetically, and hides ugly overcomplicated internals under the facade. Apple is infamous for this shit.

There are movements such as appropriate technology (described by E. F. Schumacher in a work named *Small Is Beautiful: A Study of Economics As If People Mattered*) advocating for small, efficient, decentralized technology, because that is what best helps people.

**Does minimalism mean we have to give up the nice things?** Well, not really, it is more about giving up the bullshit, getting rid of addiction and changing an attitude. People addicted to modern consumerist technology often worry that with minimalism they will lose their drug, typically games or something similar. Remember that with minimalism **we can still have technology for entertainment**, just a non-consumerist one -- instead of consuming a new game each month we may rather focus on creating deeper games that may last longer, e.g. those of a easy to learn, hard to master kind and building communities around them, or on modifying existing games rather than creating new ones from scratch over and over. Sure, technology would LOOK different, our computer interfaces may become less of a thing of fashion, our games may rely more on aesthetics than realism, but ultimately minimalism can be seen just as trying to achieve the same effect while minimizing waste. If you've been made addicted to bullshit such as buying a new GPU each month so that you can run games at 1000 FPS at progressively higher resolution then of course yes, you will have to suffer a bit of a withdrawal just as a heroin addict suffers when quitting the drug, but just as him in the end you'll be glad you did it.

## Importance Of Minimalism: Simplicity Brings Freedom

It can't be stressed enough that minimalism is absolutely required for technological freedom, i.e. people having, in **practical** ways, control over their tools. While in today's society it is important to have legal freedoms, i.e. support free software, we must not forget that this isn't enough, a freedom on paper means nothing if it can't be practiced. We need both legal AND de facto freedom over technology, the former being guaranteed by a free license, the latter by minimalism. Minimal, simple technology will increase the pool of people and parties who may practice the legal freedoms -- i.e. those to use, study, modify and share -- and therefore ensure that the technology will be developed according to what people need, NOT according to what a corporation needs (which is usually the opposite).

Even if a user of software is not a programmer himself, it is important he chooses to use minimal tools because that makes it more likely his tools can be repaired or improved by SOMEONE from the people. Some people naively think that if they're not programmers, it doesn't matter if they have access and rights to the program's source code, but indeed that is not the case. You want to choose tools that can easily be analyzed and repaired by someone, even if you yourself can't do it.

Minimalism and simplicity increases freedom even of proprietary technology which can be seen e.g. on games for old systems such as GameBoy or DOS -- these games, despite being proprietary, can and are easily and plentifully played, modified and shared by the people, DESPITE not being free legally, simply because it is easy to handle them due to their simplicity. This just further confirms the correlation of freedom and minimalism.

See Also

- primitivism
- single instruction computer

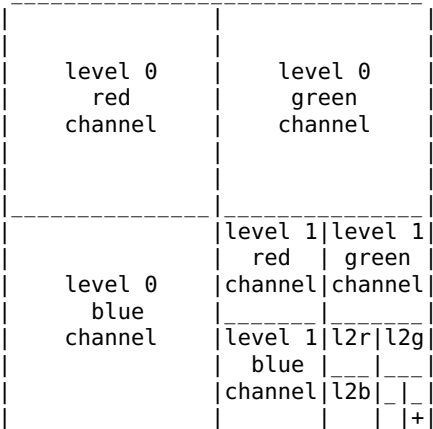
mipmap

Mipmap

Mipmap (from Latin *multum in parvo*, *many in little*), is a digital image that is stored along with progressively smaller versions of itself; mipmaps are useful in computer graphics, especially as a representation of textures in which they may eliminate aliasing during rendering. But mipmaps also have other uses such as serving as acceleration structures or helping performance (using a smaller image can speed up memory access). Mipmaps are also sometimes called **pyramids** because we can imagine the images of different sizes laid one on another to form such a shape.

A basic form of a mipmap can be explained on the following example. Let's say we have an RGB image of size 1024x1024 pixels. To create its mipmap we call the base image level 0 and create progressively smaller versions (different levels) of the image by reducing the size four times (twice along one dimension) at each step. I.e. level 1 will be the base image downscaled to the size 512x512. If we are to use the mipmap for the common purpose of reducing aliasing, the downscaling itself has to be done in a way that doesn't introduce aliasing; this can be done e.g. by downscaling 2x2 areas in the base image into a single pixel by **averaging** the values of those 4 pixels (the averaging is what will prevent aliasing; other downscaling methods may be used depending on the mipmap's purpose, for example for a use as an accelerating structure we may take a maximum or minimum of the 4 pixels). Level 2 will be an image with resolution 256x256 obtained from the 512x512 image, and so on until the last level with size 1x1. In this case we'll have 11 levels which together form our mipmap.

This RGB mipmap can be shown (and represented in memory) as a "fractal image":



This may be how a texture is represented inside a graphics card if we upload it (e.g. with OpenGL). When we are rendering e.g. a 3D model with this texture and the model ends up being rendered at the screen in such size that renders the texture smaller than its base resolution, the renderer (e.g. OpenGL) automatically chooses the correct level of the mipmap (according to Nyquist-Shannon sampling theorem) to use so that aliasing won't occur. If we're using a rendering system such as OpenGL, we may not even notice this is happening, but indeed it's what's going on behind the scenes (OpenGL and other systems have specific functions for working with mipmaps manually if you desire).

Do we absolutely need to use mipmaps in rendering? No, some simple (mostly software) renderers don't use them and you can turn mipmaps off even in OpenGL. Some renderers may deal with aliasing in other ways, for example by denser sampling of the texture which will however be slower (in this regard mipmaps can be seen as precomputed, already antialiased version of the image which trades memory for speed).

We can also decide to not deal with aliasing in any way, but the textures will look pretty bad when downscaled on the screen (e.g. in the distance). They are kind of noisy and flickering, you can find examples of this kind of messy rendering online. However, if you're using low resolution textures, **you may not even need mipmaps** because such textures will hardly ever end up downscaled -- this is an advantage of the KISS approach.

One shortcoming of the explained type of mipmaps is that they are *isotropic*, i.e. they suppose the rendered texture will be scaled uniformly in all directions, which may not always be the case, especially in 3D rendering. Imagine a floor rendered when the camera is looking forward -- the floor texture may end up being downscaled in the vertical direction but upscaled in the horizontal direction. If in this case we use our mipmap, we will prevent aliasing, but the texture will be rendered in lower resolution horizontally. This is because the renderer has chosen a lower resolution of the texture due to downscale (possible aliasing) in vertical direction, but horizontal direction will display the texture upscaled. This may look a bit weird, but its completely workable, it can be seen in most older 3D games.

The above issue is addressed mainly by two methods.

The first is trilinear filtering which uses several levels of the mipmap at once and linearly blends between them. This is alright but still shows some artifacts such as visible changes in blurriness.

The second method is anisotropic filtering which uses different, *anisotropic* mipmaps. Such mipmaps store more version of the image, resized in many different ways. This method is nowadays used in quality graphics.

---

mob\_software

## Mob Software

*Not to be confused with mob programming.*

TODO (read <https://www.dreamsongs.com/MobSoftware.html>)

---

moderation

## Moderation

Moderation is an euphemism for censorship encountered mostly in the context of Internet communication platforms (forum discussions, chats etc.).

---

modern

## Modern

*"Everything that modern culture hates is good, and everything that modern culture loves is bad." --fschmidt from reactionary software*

So called *modern* software/hardware and other *modern* technology might as well be synonymous with shitty bloated abusive technology. It's one of the most abused buzzwords of today, relying (successfully) on the sheeple shortcut thinking -- in a capitalist age when everything is getting progressively worse in terms of design, quality, ethicality, efficiency, etc., newer means worse, therefore modern (*newest*) means *the worst*. In other words *modern* is a term that stands for "as of yet best optimized for exploiting users". At LRS we see the term *modern* as **pejorative** -- for example whenever someone says "we work with modern technology",

he is really saying "we are working with as of yet worst technology". Is it shit? Does it abuse you? Is useless? Doesn't matter, it's NEW!

Modern technology is also opposed by neoluddists, a kind of anti-technology movements whose roots go back to 19th century. The word *modern* was similarly addressed e.g. by reactionary software -- it correctly identifies the word as being connected to a programming orthodoxy of current times, the one that's obsessed with creating bad technology and rejecting good technology. { I only found reactionary software after this article has been written. ~drummyfish }

Sometimes random people notice the issue, though there are very few. One blog (<https://blog.ari.lt/b/modernism/>) for example goes on to say that "modernism sucks" and the word *modern* is basically just an excuse for being bloated. Those are indeed true words.

**Avoid anything labeled as follows:** "modern", "state-of-the-art", "cutting-edge", "for 21st century", "for INSERT CURRENT YEAR", "up-to-date", "innovative", "novel", "latest technology", "high tech" etc.

Remember, older is always better.

## Modern Vs Old Technology

It's sad and dangerous that newer generation won't even remember technology used to be better, people will soon think that the current disgusting state of technology is the best we can do. That is of course wrong, technology used to be relatively good. It is important we leave here a note on at least a few ways in which old was much, much better.

(INB4 "it was faster and longer on battery etc. because it was simpler" -- **yes, that is exactly the point.**)

- Old technology was simpler and **better engineered with minimum bloat**. Fewer incompetent people were present in the field and capitalism wasn't yet pushing as hard on extreme development speed and abuse of the user, products still tried to compete by their quality.
- **Old computers were faster** and astronomically more efficient. Computers with a few MHz single-core CPU and under a megabyte of RAM booted faster to DOS than modern computers boot to Windows 10, despite Moore's law (this shittiness is known as Wirth's law). Old tech also **reacted faster to input** (had shorter input latency/lag), e.g. thanks to shorter input and output processing pipelines. { I've heard this confirmed from John Carmack himself in a talk on his development of VR. ~drummyfish } Back in the day things had to work smoothly -- if in the 90s you showed people a phone that you wake up and have to wait 20 seconds before it starts to react, they would laugh at it and on one would buy it -- nowadays such technology is the standard.
- Old devices such as cell phones **lasted much, much longer on battery**. The old phones such as Nokia 3310 would **last long over a week** on stand-by.
- **Old software was shipped finished, complete and with minimum bugs**. Nowadays newly released "apps" and games are normally released unfinished, even in pre-alpha states and even "finished" ones have bugs often rendering the software unusable (see Cyberpunk 2077, GTA: "Definitive" Edition etc.), user is supposed to wait years for fixes (without any guarantees), pay for content or even subscriptions. Some software "products" even spend their whole commercial life unfinished. Old software was difficult or even impossible to patch (e.g. Gameboy cartridges) so it had to work.
- **Old tech had minimum malicious features**. There wasn't spyware in CPUs, **DRM was either absent or primitive**, there weren't ads in file explorers, there weren't microtransactions in games, there weren't autoupdates, there weren't psychologically abusive social networks, technology was **designed to last**, with replaceable parts; not to be consoomed, there was much less ensorship.
- **Old tech was much easier to repair, modify and customize**, thanks to not being so overcomplicated and not containing so many anti-repair "features". Old software wasn't in the cloud which makes it impossible to modify.
- **Old software was better programmed** because it was firstly made by actually the smartest people such as mathematicians and physicist (who were considering the big picture and saw e.g. the necessity for minimalism) and secondly without such a great pressure of the market because software was more a subject of research and experimenting rather than dirty fight for consumers. This can be seen even on commercial software such as games: for example the Doom engine was written very nicely, in an extremely portable way (which actually became legendary), with things such as an

elegant deterministic FPS-independent physics (which results in many advantages and is basically THE only correct way of writing an engine) and software rendering that ran smooth even on that time's slow CPUs. Later engines from the same creators -- those of Quake games -- began to suffer from worse design (no deterministic physics, dropping of software rendering etc.). Nowadays software is written by high schoolers, women and incompetent minorities forced into tech just for diversity quotas and generally anyone who can just copy paste snippets of code from the web, extremely tight deadlines in the market race make it impossible to tidy any piece of software -- game engines (like anything else) nowadays are indescribably badly written, non-portable, non-deterministic, bloated, running slow even on computers thousands of times faster than those that ran Doom (even if you lower graphic details of a 2023 game to the looks of a 2000s game, it will likely run under 10 FPS on a 2020 computer).

- **Old tech was much more independent and freedom friendly**, did not require Internet connectivity, subscription etc. Thanks to its simplicity and better hackability it was possible for people to partly control their devices, even if the devices were proprietary. Nowadays if the manufactures of your phone (or even a car) decides it's time for you to buy a new model, he just remotely kills your device, and you can hardly do anything about it (this is actually happening e.g. with iPhones).
- There was **minimum bullshit**. True usefulness was more important than killer features and marketing.
- Old tech was **simpler and more fun to program**, allowing direct access to hardware, not complicating things with OOP and similar shit, and so **old programmers were more "productive"**, less frustrated and stressed.
- **Old art was more free in expression, less censored and toxic, without ads and SJW poison**, people still had some standards, there were still artists with artistic freedom and vision (take a look e.g. at the famous movie directors of the 90s, nowadays a director of a movie is just a nobody who has to bow to PR, marketing, diversity departement, suck the investor dick etc.) there were no diversity quotas and shit. For example in old games such as Faery Tale Adventure II (1997) you could happily start killing children (even little black girls lol) in the village you spawned in and get away with it no problem, it was simply a choice you could make. Compare this to Skyrim (2011) where children were made the only invincible beings in the world, literally more powerful than dragons and gods because they couldn't be touched because the immense cowardice of the devs who are shitscared of lawyers, "PR", just literally aiming for profit and can't stand behind their art. After 2010 art is quite literally dead.
- **Old "look n feel" of software was objectively better**. Just compare the look of Doom and any shitty soulless "modern" game with billion polygons but literally zero aesthetics.
- ...

## See Also

- <https://unixsheikh.com/articles/when-the-modern-approach-is-nothing-but-hype.html>

---

modern\_software

## Modern Software

Go [here](#).

---

monad

## Monad

{ This is my poor understanding of a monad. I am not actually sure if it's correct lol :D TODO: get back to this. ~drummyfish }

Monad is a mathematical concept which has become useful in functional programming and is one of the very basic design patterns in this paradigm. A monad basically wraps some data type into an "envelope" type and gives a way to operate with these wrapped data types which greatly simplifies things like error checking or abstracting input/output side effects.

A typical example is a **maybe** monad which wraps a type such as integer to handle exceptions such as division by zero. A maybe monad consists of:

1. The *maybe(T)* data type where *T* is some other data type, e.g. *maybe(int)*. Type *maybe(T)* can have these values:
  - *just(X)* where *X* is any possible value of *T* (for int: -1, 0, 1, 2, ...), or
  - *nothing*, a special value that says no value is present
2. A special function *return(X)* that converts value of given type into this maybe type, e.g. *return(3)* will return *just(3)*
3. A special combinator *X >=> f* which takes a monadic (*maybe*) values *X* and a function *f* and does the following:
  - if *X* is *nothing*, gives back *nothing*
  - if *X* is a value *just(N)*, gives back the value *f(N)* (i.e. unwraps the value and hand it over to the function)

Let's look at a pseudocode example of writing a safe division function. Without using the combinator it's kind of ugly:

```
divSafe(x,y) = // takes two maybe values, returns a maybe value
  if x == nothing
    nothing else
    if y == nothing
      nothing else
      if y == 0
        nothing else
        just(x / y)
```

With the combinator it gets much nicer (note the use of lambda expression):

```
divSafe(x,y) =
  x >=> { a: y >=> { b: if b == 0 nothing else a / b } }
```

Languages will typically make this even nicer with a syntax sugar such as:

```
divSafe(x,y) = do
  a <- x,
  b <- y,
  if y == 0 nothing else return(a / b)
```

TODO: I/O monad TODO: general monad TODO: example in real lang, e.g. haskell

---

money

## Money

THIS IS YOUR GOD

TODO

**Money spoils everything**, and in capitalism money is everywhere. As Richard Muller sings: "happiness is a beautiful thing, but it can't buy you money".

Sadly capitalism forced EVERYONE to deal with money, even those who hate it. How to handle this? **Correct relationship towards money** you should have as an LRS follower:

- **Hate money**, always aim for eliminating money from society. Does this mean you should hate having money? No, this means you should hate *having to have money*.
- **Minimize damage money does**, i.e. firstly do NOT run capitalist businesses that steal money from

the poor (basically every business eventually does this through some level of indirection), that makes money harm others. Secondly make money do less damage to yourself, i.e. do not rid yourself of money by burning them or anything, just use them to live in less pain. You can also **use money for good** -- if you have enough, just give it to someone in need. Don't teach people to fish, just give them the fucking fish if you have tons of them. Remember that **being rich means being a murderer** because you are just sitting on resources that could simply save lives of many, just by having a lot of money and doing nothing you are killing people, so don't be rich -- if you are, just give money to someone.

- **Use money to become independent of money**, i.e. only make money so that you don't have to care about money, NOT to consume more of capitalist production. With regards to "winning a lottery" most people nowadays have the attitude of "I'd keep living the same but I would buy a bigger house and could wear luxury clothes", which is the retarded stance that only enslaves you more and increases overall damage to everyone. The correct thinking is "I would stop working so that I could be more free and do more good without having to make profit anymore".
- **Don't become a slave to money**, i.e. don't spend your thinking time on investments, accounting, protecting your funds, insurances, currencies, businesses to make more money etc. Just as with technology, make some minimal setup that just makes you not have to think about money, EVEN if it costs something (i.e. makes you lose something to inflation etc.). { I'm real retarded about economy but it may be good to e.g. have some cash ready (like for quarter of a year of living), then some saving bank account (where more money goes and possibly gets some small interest but is still available somehow) and then some physical gold to protect from inflation. Anyway even this may be too much for many to worry about, feel free to even just go all cash in your mattress if you want to avoid bank software and such shit, it's all fine. ~drummyfish }
- ...

---

morality

## Morality

Morality is the sense of greater values of an individual and society from which it follows what's ultimately right, wrong, good and bad/evil on a greater level, for a "greater good", without succumbing to low instincts such as self interest, self preservation, immediate pleasure etc. Morality is what greatly distinguishes man from animal and allows him to act not on mere instincts and reactions to immediate stimuli, it is driven by the higher forces such as beliefs, logic, empathy, love, conscience, religion and science. Examples of moral (good) behavior include altruism, selflessness, communism in general sense, less retarded society and non violence, while examples of IMMORALITY (evil) might be capitalism, fascism, rape, pedophobia, genocide, marketing, proprietary software, nationalism and LGBT.

Morality is very similar to ethics, to the point of often being used interchangeably, however we may still find slight differences. While morality is seen as something personal and intuitive, greatly driven by conscience and judged on a case-by-case basis, ethics is perceived more as a set of informal, often unwritten shared rules to assure morality in a larger group of individuals, i.e. ethics is an agreement on a way of behavior between individuals, each of which may have slightly different personal morals. Ethics is also sometimes defined as the branch of philosophy concerned with examining morality.

**Morality is much different from legality.** Ideally it is said that laws should be the minimum (a proper subset) of morality, i.e. laws should be the officially codified, approved and enforced rules that ensure the very basic moral behavior is sustained, such as people not murdering others, however laws CANNOT with the best of our effort ever capture the infinitely complex nature of morals (no one can ever write down EXACTLY what is and isn't moral in every single imaginable situation that can arise in real world), so it is seen as inevitable that laws will always allow some slightly immoral actions (imagine e.g. someone giving a bad advice to someone else on purpose just to see the other one fail -- this may be legal but is likely immoral). This is accepted because the other option, i.e. law trying to prevent ALL immoral behavior, would be too restrictive and would also inevitably prevent a huge amount of moral, useful and essential behavior; imagine e.g. law trying to prevent giving bad advice by banning all communication altogether. However, this ideal of "laws as a minimum of morals" doesn't hold in practice because law is hugely abused and manipulated to serve the evil, so not only does it allow immoral behavior (which would be kind of OK), it BANS moral behavior (which is unacceptable from the idealist point of view), for example it is prohibited to share useful information ("intellectual property"), repairing (DRM), living in an abandoned house one doesn't "officially



own" etc. Furthermore laws themselves in principle have a negative effect on morality because **people unfortunately start replacing morality with legality**; as laws get more complex and in control of our everyday lives, people only start deciding and judging actions based on a question of "is it legal?" rather than "is it moral?" -- indeed, if nowadays you accuse someone of doing something wrong, he will almost definitely reply something along the lines of "I can legally do that so shut up." Laws destroy morality, hence laws have to be cancelled (see [anarchism](#)) and we have to focus only on developing our sense of morality better.

## See Also

- [superego](#)

---

motivation

## Motivation

*There are too many motivated people, and too few passionate ones.*

---

mud

## Multi User Dungeon

{ WIP, researching. ~drummyfish }

**Are there any free MUD codebases written in good languages?** No. You have to write your own, it seems like MUD faggots don't wanna free their code, and if they do they write it in JavaScript++. Many times it's also because the old code (written in nice languages such as C) started to be created a long time ago when licenses weren't yet such a big thing or didn't exist at all. Some source available codebases are e.g. DikuMUD (written in C, marked LGPL on GitHub, however the code may be PROPRIETARY, their site explicitly mentions one of the original authors was unreachable during relicensing) and SMAUG (written in C, marked GPL on GitHub, however the code is most likely PROPRIETARY as it's a long chain of derivatives from some obscure weird licensed source).

---

murderer

## Murderer

You misspelled [entrepreneur](#).

---

music

## Music

Music is an auditory [art](#) whose aim is to create [pleasant](#) sound of longer duration that usually adheres to some rules and structure, such as those of melody, harmony, rhythm and repetition. Music has played a huge role throughout all history of human [culture](#). It is impossible to precisely define what *music* is as the term is [fuzzy](#), i.e. its borders are unclear; what one individual or culture considers music may to another one sound like [noise](#) without any artistic value, and whatever rule we set in music is never set in stone and will be broken by some artists (there exists music without chords, melody, harmony, rhythm, repetition... even without any sound at all, see *Four Minutes Thirty Three Seconds*). Music is mostly created by singing and playing musical instruments such as [piano](#), guitar or drums, but it may contain also other sounds; it can be recorded and played back, and in all creation, recording and playing back [computers](#) are widely used nowadays.

**Music is deeply about [math](#)**, though most musicians don't actually have much clue about it and just play "intuitively", by feel and by the ear. Nevertheless the theory of scales, musical intervals, harmony, rhythm and other elements of music is quite complex;

**Copyright of music:** TODO (esp. soundfonts etc.).

TODO: list of most LRS/suckless instruments

## Modern Western Music + How To Just Make Noice Music Without PhD

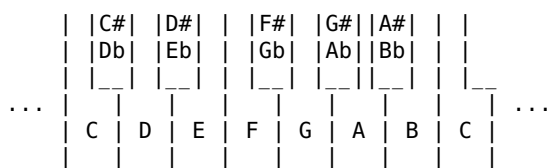
{ I don't actually know that much about the theory, I will only write as much as I know, which is possibly somewhat simplified, but suffices for some kind of overview. Please keep this in mind and don't eat me. ~drummyfish }

Our current western music is almost exclusively based on major and minor diatonic scales with 12 equal temperament tuning -- i.e. basically our scales differ just by their transposition and have the same structure, that of 5 whole tone steps and 2 semitone steps in the same relative places, AND a semitone step always corresponds to the multiplying factor of 12th root of 2 -- this all is basically what we nowadays find on our pianos and in our songs and other compositions. 4/4 rhythm is most common but other ones appear, e.g. 3/4. Yeah this may sound kinda too nerdy, but it's just to set clear what we'll work with in this section. Here we will just suppose this kind of music. Also western music has some common structures such as verses, choruses, bridges etc.; lyrics of music follow many rules of poetry, they utilize rhymes, rhythm based on syllables, choice of pleasant sounding words etc.

**Why are we using this specific scale n shit, why are the notes like this bruh?** TODO

**Music is greatly about breaking the rules**, like most other art anyway -- you have to learn the rules and respect them most of the time, but respecting them all the time results in sterile, soulless music; a basic rule is therefore to break the rules IN APPROPRIATE PLACES, i.e. where such a rule break will result in emotional response, in something interesting, unique, ... This includes for example leaving the scale for a while, adding disharmony, adding/skipping a beat in one bar, ...

**If you wanna learn music, firstly you should get something with piano keyboard:** musical keyboard, electronic piano, even virtual software piano, ... The reason being that the keys really help you understand what's going on, the piano keyboard quite nicely visually represent the notes (there is a reason every music software uses the piano roll). Guitar or flute on the other hand will seem much more confusing; of course you can learn these instruments, but first start with the piano keyboard.



*Tones on piano keyboard, the "big keys" are white, the "smaller keys on top" are black.*

OK so above we have part of a piano keyboard, tones go from lower (left) to higher (right), the keyboard tones just repeat the same above and below. The white keys are named simply A, B, C, ..., the black keys are named by their neighboring white key either by adding # (sharp) to the left note or by adding *b* (flat) to the right note (notes such as C# and Db can be considered the same within the scales we are dealing with). Note: it is convenient to see C as the "start tone" (instead of A) because then we get a nice major scale that has no black keys in it and is easy to play on piano; just ignore this and suppose we kind of "start" on C for now.

Take a look at the C note at the left for example; we can see there is another C on the right; the latter C is one **octave** above, i.e. it is the "same" note by name but it is also higher (for this we sometimes number the notes as C2, C3 etc.). The same goes for any other tone, each one has its different versions in different octaves. Kind of like the color red has different versions, a lighter one, a darker one etc. Octave is a basic interval we have to remember, **a tone that's one octave above another tone has twice its frequency**, so e.g. if C2 has 65 hertz, C3 has 130 hertz etc. This means that **music intervals are logarithmic, NOT linear!** I.e. an interval (such as octave) says a number by which we have to MULTIPLY a frequency to get the higher frequency, NOT a number which we would have to add. This is extremely important.

Other important intervals are **tone** and **semitone**. Semitone is a step from one key to the immediately next key (even from white to black and vice versa), for example from C to C#, from E to F, from G# to A etc. A tone is two semitones, e.g. from C to D, from F# to G# etc. There are 12 semitones in one octave (you have to make 12 steps from one tone to get to that tone's higher octave version), so a semitone has a multiplying factor of  $2^{1/12}$  (12th root of two). For example C2 being 65 hertz, D2 is  $65 * 2^{1/12} \approx 69$  hertz. This makes sense as if you make 12 steps then you just multiply 12th root of two twelve times and are left simply with multiply by 2, i.e. one octave.

TODO: chords, scales, melody, harmony, beat, bass, drums, riffs, transpositions, tempo, polyphony ...

## Music And Computers/Programming

TODO: midi, bytebeat, tracker music, waveforms, formats, procedural music, AI music, ...

---

myths

## Myths

This is a list of myths and common misconceptions.

- **"Java is highly portable"**; FALSE: While Java runs on several platforms, it's inefficiency and overhead of its extremely high level programming makes it unusable on devices with limited resources such as embedded systems. Its bloated nature and high number of dependencies limit it to running on a few types of **mainstream** devices that are privileged to have the java virtual machine implemented.
  - **"C is not portable"**; FALSE: C is extremely portable if written correctly (e.g. without dependencies), in fact it is probably the **most portable language in history** because firstly a C compiler is available for almost any platform -- a C compiler is one of the most essential things to have -- and secondly because C is extremely efficient and will run even on devices with extremely limited resources such as embedded systems.
  - **"Capitalism fuels progress"**; FALSE: Monopolies that inevitably arise in capitalism want to forever prevent others from creating innovations that would make the subject of their business obsolete (e.g. fossil fuel businessmen will want to prevent electric cars). Of course, small businesses cannot compete with large corporations, therefore corporations win and keep the status quo. Small businesses can mostly only succeed in creating bullshit that exists for its own sake (there is e.g. an online store selling literal shit) -- this we would not call a progress. Furthermore capitalism is against the important kind of progress such as social progress or education, because of course educated, independent and mature people would engage less in consumerism and even realize that capitalism is bad.
  - **Feminism, LGBT, Antifa and similar are leftist movements**; FALSE: These are in fact pseudoleftist, fascist movements who don't care about true equality but rather privileges for a certain minority, just as the Italian fascist or Nazis did. This is proven by their naming and means of operation such as violence, censorship, bullying etc. which are anti-equality.
  - TODO
- 

name\_is\_important

## Name Is Important

Name of a philosophy, project, movement, group, ideology etc. plays a more significant role than a common man believes. A naive view is that name is just an arbitrary identifier whose value lies at most being "catchy" and "easily remembered" (see also marketing), a common man will rather believe promise of politician than the name of his party which he will disregard just as a "bunch of unimportant words"; however name is much more than a mere string of letters, it is the single most stable defining feature of an entity; everything else, all the books and knowledge associated with it may be distorted by history, but the name will always stay the same and will hold a scrutiny over all actions of the entity, it will always be a permanent reminder to every follower of what he is trying to achieve. But what if the name of the movement changes? Then it will be by definition a new, different movement, and everyone will have to decide if he wants to abandon the old

movement to join the new. The name very often points towards the one true goal.

HOWEVER, here also comes a word of **warning**: firstly language itself (i.e. meanings of all words that exist in it) changes, and also although the power of the name is great, it is not infinite, the discussed stress of the importance of name should just remind us that the force of the name is greater than one might expect, but may still be broken if stronger forces are at play -- there have been many cases of **name abuse** in history, notably e.g. by Nazism whose name stands for "national socialism" but whose actions were completely antisocialist, or so called "Anarcho" capitalism which abuses the name anarchism despite being completely antianarchist. The moral of the story here is that we should put a great effort in choosing a name, but we shouldn't think we'll be safe as long as we do -- we will probably never be safe from the fuzziness of language and its potential to be abused, but we should try to do our best.

We have to keep in mind two things:

- When encountering a new movement/philosophy/ideology etc., we can tell a lot about it from its name: the name is its ultimate goal which **will be pursued on detriment of other goals**. A lot of times the bad movements are those **named after the means** (e.g. capitalism or open source) or **people** (e.g. Maoism) rather than goals (e.g. pacifism) because by this dominance the focus on means will inevitably subordinate the goal.
- When starting a new movement, we have to pay very careful attention to giving it a name.

Let us comment on a few examples:

- **Capitalism**: The goal is maximization of capital -- capital should be the means to achieving "something" (what lol?), but it instead becomes the goal. There is no promise of a good society, it is not mentioned in the name, and indeed what we get is a system that evolves corporations that get progressively better at maximizing capital, on the detriment of people.
- **Free Software Movement**: The movement is about user's freedom and therefore ethics. Notice that even though definitions of free software may slightly differ between different branches of it, the groups calling themselves *free software* movements always pursue freedom. This is why capitalism couldn't embrace free software: because thanks to the name it always firmly stands against abusing the user. Of course, even the meaning of the word *freedom* may eventually be shifted and the movement may get spoiled too, but it has stood for a long time and has already proven a great resistance.
- **Open Source Movement**: The movement has been born from the Free Software Movement with a specific goal of abandoning ethics and supporting business. The ethics (freedom) has been dropped from the name and was replaced with the word "open", and indeed what we're seeing is software that is somewhat "open" -- whatever that means -- but mostly capitalism software abusing and restricting its users by means other than proprietary licenses (see e.g. Firefox).
- **Feminism**: The goal is to benefit the females, i.e. make the superior to males. There is no mention of equality of sexes in the name even this idea might have appeared somewhere in the movement's beginning. And indeed, what we're seeing is a progressively more aggressive fascist movement that is downright hostile to males, is completely uninterested in inequality in the opposite direction and only ever looks for empowering the women, without any concern of equality.
- **LGBT**: Same as feminism.
- **Anarchism**: anarchism means "without a ruler" which has shown to be a good name, clearly opposing the idea of one man standing above another and so, again, keeps resisting attempts at being twisted to go against this basic goal. We rarely see the name being abused by the powerful because someone who wants to rule others simply cannot promote something that clearly indicates there should be no rulers. Indeed, there exist attempts at abusing the name, such as "Anarcho" capitalism, but we observe the sheer absurdity of such attempt is quickly spotted by most people who know what the word *anarchism* means.
- ...

{ A note from my friend: "coding" gaining popularity over "programming" maybe shows a subconscious shift towards productivity cult, a shift from focusing on the process and doing it well (programming) towards just shitting out quantities of code (coding). ~drummyfish }

---

nanogenmo

# NaNoGenMo

NaNoGenMo (national novel generation month) is a fun yearly event, running since 2013, in which people make computer generated novels during the month November. It was inspired by NaNoWriMo (a similar event but for normal, human creative writing) and launched by Darius Kazemi on Twitter. It is similar e.g. to the international obfuscated C code contest with one difference being that NaNoGenMo is not a contest, it's just a fun activity people do and see what comes out of it. Because of this rules are also very relaxed, something along the lines of "submit a text of around 50000 words that doesn't violate copyright along with the code that generated it" (so there appear borderline submissions like non plain text pdfs and so on). At the beginning the text generating programs weren't usually anything too sophisticated, they were mostly things like 100 lines of Python that throw around random sentences, maybe use some Markov chain, maybe some regex substitution on an already existing book -- most of the entries seemed to be just that. A simple but effective approach that's been used is to simulate some world with actors in it and just let it be documented what they're doing. By 2019 however an increased abuse of language models and other bloat started to be noticed, so a kind of fork event was spawned, called Nano-NaNoGenMo, in which at most 256 character programs are allowed. The sad thing is that NaNoGenMo uses GitHub issues for posting the texts by which they support terrorists. Also a lot of participants are huge noobs who share their works using Dropbox and Google documents and similar shit so they're literally unreachable for most smart people. They also don't require any license, many of the works are proprietary. And there doesn't seem to be any nice repository of the entries either, you have to dig them up in the issues or look them up on bloated woke blog posts that attempt to summarize them. There exist similar events for poetry (NaPoGenMo), opera (NaOpGenMo) and movies (NaMoGenMo). The idea of NaNoGenMo is excellent, the execution an uttermost fail.

Some of the generated books were quite popular (though maybe mostly for the nature of having been generated by computers) -- for example the *World Clock*, a Python generated work that just gives random snapshots of people's lives around the world, was even printed and sold. One entry is just a program going through all directories on the harddrive and commenting on them like "wow, there's a lot of files here" and so on. *The Swallows of Summer* had some success despite being just an endless exchange of talk and interactions between Alice and Bob. Some other entries seem to be interesting, there are e.g. various modifications of the *Moby Dick* or the *Bible* (conveniently well known long works completely in the public domain) -- it's enough to just replace some keywords to get something quite entertaining. Wikipedia and Project Gutenberg are commonly used as sources of text. One novel just describes someone writing down the digits of pi. Around 2020 many started to use neural network language models, e.g. *A Young "Person"'s Encyclopedia* is a fictional encyclopedia made with GPT-3. There is one book in which a model trained on first sentences of famous books just suggests a huge list of new sentences with which one can potentially start a novel.

## See Also

- ioccc
- SIGBOVIK
- procedural generaion
- esoteric programming languages

---

nc

## NC

See also ND.

In the context of licenses the acronym NC stands for *non-commercial* and means "only non-commercial use allowed", which is an unpopular limitation that makes such a license **by definition proprietary (i.e. NOT a free cultural license)**. This means that a work shared under a license with NC clause is prohibited from being used commercially (which itself is a very unclear statement), greatly limits the freedom of such work and opens the door for legal fuzziness and therefore possible bullying. The NC limitation appears most notably in two Creative Commons licenses: CC BY-NC-SA and CC BY-NC-ND; again, despite these licenses being Creative Commons, they are **NOT free as in freedom** licenses -- note that this is not an opinion or

controversial statement, NC licenses very clearly break the consensual definition of free cultural works and Creative Commons themselves clearly state this is the case; they justify NC licenses as part of the proprietary-free license spectrum, standing somewhere in between "all rights reserved" and free cultural licenses. Even though to free culture newcomers NC licenses don't seem like such a big deal, they are in fact extremely harmful to free culture, **DO NOT USE NC LICENSES**. NC is similar (and similarly harmful) to another proprietary license limitation: ND (no derivatives allowed). If you use an NC license, you're a huge cocksucker.

**Why are NC licenses bad?** Firstly the *Definition of Free Cultural Works* project that maintains the widely accepted definition of free culture has an article on this: <https://freedomdefined.org/Definition>. Let us write a summary of the arguments ALONG WITH our own arguments:

- **"Commercial use" isn't just selling, it possibly prohibits beneficial uses you would want to allow.** Imagine you create e.g. an educational image that you would like many people to see -- by using an NC license you will however prohibit for example other people from showing the image on social media if they are people who make money from social media, for example YouTubers that make money from ads on their videos will be prohibited (or at least greatly discouraged) from showing your educational image in their videos, you may also rule out use by ethical non-profits who still need to make some money to sustain themselves etc. Of course, the author may allow commercial use in individual cases and "on the go", in the spirit of permission culture, however this may be practically impossible in cases of big collaborative works similar to e.g. Wikipedia with possibly thousands of authors.
- **It is very unclear what commercial use means which opens door to legal fear and avoidance of NC works.** "Commercial use" may be interpreted not just as directly selling the work but as doing pretty much anything with the work that somehow, even indirectly, leads to some kind of profit -- consider e.g. printing an NC picture on your shirt (allowed by the NC license) and then wearing that shirt while promoting something you're selling -- if such shirt could be seen as helping you get attention of more customers and so result in making more money, you're potentially violating the license and the author can sue you. I.e. by using NC works you will always have to worry to not come close to anything that could be seen, even by a stretch, as a "commercial use". This is a worry that will exist as long as copyright on the work, i.e. certainly for your whole life. For this reason people simply avoid using such works and even choose to use lower quality works that don't pose such dangers. Using an NC license will therefore make your work much less popular, less visible and, of course, much less useful.
- **It makes the work incompatible with other licenses.** Works under NC license cannot be included in free cultural works, for example text written on the Citizendium encyclopedia (licensed NC) cannot be copied to Wikipedia (licensed free under CC-BY-SA).
- **It adds huge legal bloat.** Similarly to e.g. copyleft, the NC clause has to either be very vague and unclear or extremely long and complex in explaining what it really means. This of course leads to unclarities, legal bugs, confusion, payments of lawyers etc.
- **It strengthens the idea of intellectual property.** The basic aim of free culture is to relax "intellectual property" laws, not to strengthen them or continue the ways of permission culture, therefore the strict NC limitation is very unpopular among proponents of free culture. We, LRS, strongly reject the very idea of being able to own information, so stricter legal conditions are always worse in our view.
- **NC license may be in some cases worse than no license at all.** Even though Creative Commons NC licenses give basic rights such as that for non-commercial sharing, the explicit prohibition of commercial use may in specific cases result in more harm than if there was no license attached to the work, because breaking a rule that's stated explicitly may in court be seen as a bit more serious and intentional than breaking an implicit rule. (Similar reasoning was used to reject CC0 by OSI.)
- **NC license is proprietary, i.e. all arguments against proprietary works apply.**
- ...

{ Nice parody is the CC BY-NV license :D <https://questioncopyright.org/cc-by-nv/trackback.html> ~drummyfish }

## See Also

- ND
- CC BY-NV

nd

# ND

See also [NC](#).

In the context of [licenses](#) the acronym ND stands for *no derivatives* and means "no derivative works allowed" -- this is an unpopular limitation that makes such a license **by definition proprietary (i.e. NOT a free cultural license)**. A work licensed under a license with ND clause -- most notably the [CC BY-NC-ND](#) license -- prohibits anyone from making derivative (i.e. modified) works from the original work, on grounds of [copyright](#), which goes against one of the very fundamental ideas of [free culture](#) (and just any sane culture), that of free [remixing](#), improvement, combining and reuse of [art](#); it kills artistic freedom, [culture](#), opens the door to legal bullying and strengthens the [harmful capitalist](#) idea of "[intellectual property](#)". All in all **ND licenses are cancer, NEVER USE THEM**.

The ND clause is similarly harmful to the [NC](#) (non-commercial-only) clause -- see the NC article for more detail.

needed

## LRS: Projects Needed

WIP

{ If you want to maybe start some project here I'll be glad if you let me know before you start, it can be good to talk about it first as I already have some ideas about how to make some of these projects, I just don't have time to work on them, I will just give you the ideas I have if you want, we can discuss how to best write the code etc. Of course it's all up to you, I'm just offering advice and discussion :) ~drummyfish }

Here is a list of some projects and project ideas which we, [LRS](#), need to make in order to pursue our goals. The projects here are mostly basic things and tools that already exist in some form, but that have to be made from scratch according to [LRS](#) philosophy, i.e. in a KISS/suckless way, under public domain, in a good language (C, comun, ...) etc. This is kind of a dirty list serving some rough organization. If you have the skills and will (or know someone who does), you may take inspiration here, pick one up and make it, or contribute to some of the projects listed here. Also note that it's still possible to make multiple projects of the same type, e.g. you may still create another chess engine even though we already have one, just watch out that this is justified (it should offer something worth the extra effort).

| what                     | difficulty | implementation                    | by         | status          | comment                                                         | similar        |
|--------------------------|------------|-----------------------------------|------------|-----------------|-----------------------------------------------------------------|----------------|
| 2D image editor          | mid?       |                                   |            |                 | KISS GIMP clone needed! Use LRS GUI lib. Glorified MS paint?    | ped, G classic |
| 2D raycasting engine (C) | mid        | <a href="#">raycastlib</a>        | drummyfish | done            |                                                                 |                |
| 3D modelling software    | mid/hard?  |                                   |            |                 | Blender clone needed! LRS GUI lib + small3dlib, just .obj files | Blende         |
| 3D physics engine (C)    | hard       | <a href="#">tinyphysicsengine</a> | drummyfish | done            | could use a true rigid body one too                             |                |
| 3D raytracing library    | mid?       |                                   |            | had vague plans | C lib for shooting 3D rays, allows raycast., RT, pathtr., ...   | POV-R          |
| 3D renderer (C)          | mid/hard   | <a href="#">small3dlib</a>        | drummyfish | done            |                                                                 | TinyGL Portab  |

| what                                   | difficulty    | implementation           | by         | status                | comment                                                           | similar             |
|----------------------------------------|---------------|--------------------------|------------|-----------------------|-------------------------------------------------------------------|---------------------|
| 3D voxel renderer (C)                  | mid/hard?     |                          |            |                       | like Ken Silverman's voxlap, looks very nice                      | voxlap              |
| <u>Anarch</u> mods                     | easy          |                          |            | more would be nice    | for fun                                                           |                     |
| artificial human language              | hard?         |                          |            | thinking bout it      | need LRS lang., big problems with definitions of words tho, think | Esperanto<br>Lojban |
| Arduino/Pokitto/... computer           | mid/hard?     |                          |            |                       | until we have PD computer, we'll need a nice tiny embedded comp.  |                     |
| audio/music editor                     | mid/hard?     |                          |            |                       | for waveforms and/or MIDI (tracker music), can even be CLI/TUI    | Audacity ...        |
| chat software                          | mid?          | dumbchat                 | drummyfish | one done              | make it KISS, no encryption, no Unicode, ... IRC just chat!       |                     |
| <u>chatbot</u>                         | mid?          |                          |            | plans in my head      | probably NOT neural net, KISS lib for good enough chatbot         |                     |
| <u>chess</u> engine/library (C)        | mid/hard      | <u>smallchesslib</u>     | drummyfish | done                  | it's not very strong tho :/                                       |                     |
| <u>compression</u> lib/util            | mid?          | shitpress/comunpress ... |            | one so far            |                                                                   |                     |
| data, datasets                         | easy/mid?     |                          |            | can never have enough | simple format CC0 data (CSV etc.): txt dictionaries, star DB, ... | Wikidata            |
| dating/friend searching website        | mid?          |                          |            |                       | we are lonely + don't wanna use proprietary dating shit           |                     |
| <u>free universes</u>                  | mid/hard?     |                          |            |                       | need at least one fantasy and one sci-fi, for games n shit        |                     |
| fiction, stories, books                | mid?          |                          |            | have some plans       | fairytale, sci-fi from LRS society etc.                           |                     |
| free cultural <u>porn</u> website      | mid?          |                          |            |                       | libre porn + suckless site (no JS), prev. attempts failed         | WMC p<br>freedo     |
| forum, chat, git/file host/mirror, ... | easy/mid?     |                          |            |                       | for LRS community, if you have a server you could host something  | email,              |
| <u>gamebook</u>                        | easy/mid?     |                          |            |                       | can be done by nonprogrammers and later be made into PC game too  |                     |
| game engine/fantasy console (tiny)     | easy/mid      | <u>SAF</u>               | drummyfish | done                  |                                                                   |                     |
| game engine: point n click adventure   | mid           |                          |            |                       |                                                                   |                     |
| game: <u>Doom</u> clone                | hard          | <u>Anarch</u>            | drummyfish | done                  |                                                                   | Freedo              |
| game: <u>GTA</u> clone                 | hard<br>hard? |                          |            |                       |                                                                   | Minete              |



| what                                    | difficulty | implementation   | by         | status                | comment                                                           | similar |
|-----------------------------------------|------------|------------------|------------|-----------------------|-------------------------------------------------------------------|---------|
| game: <u>Minecraft</u> clone            |            |                  |            |                       | Minetest is bloated as fuck, also bad license and SJWs            |         |
| game: text adventure                    | easy       |                  |            |                       | pure CLI text adventure, maybe "US citizen simulator"? :)         |         |
| game: <u>Trackmania</u> clone           | hard       | <u>Licar</u>     | drummyfish | started               |                                                                   |         |
| game: <u>Pokemon</u> clone              | hard?      |                  |            |                       | catchable monsters game, procedurally generated ones? SAF?        | Tuxem   |
| game: fantasy <u>RPG</u>                | hard?      |                  |            |                       | Dream: Elder Scrolls clone, also just a dungeon crawler, ...      |         |
| games: tiny ones                        | easy       | <u>uTD</u> , ... | ...        | can never have enough | very tiny games, SAF is ideal for this, nice learning project     |         |
| <u>go</u> engine/library (C or comun)   | mid?       |                  |            |                       |                                                                   |         |
| <u>GUI</u> library                      | easy/mid   |                  |            |                       | like SAF but for "PC" GUI (mouse, sound, ...), now GUI's a mess   |         |
| image/2D data library                   | mid?       |                  |            |                       | C/comun lib for bitmaps (FFT, formats, ...), needs good planning  |         |
| logic circuit library/simulator (comun) | mid/hard?  |                  |            |                       | will be needed for PD computer                                    |         |
| Marble Blast clone (C?)                 | mid/hard?  |                  |            |                       | like Neverball but KISS, better controls, wouldn't be so hard     | Neverb  |
| " <u>micronation</u> "                  | ???        |                  |            |                       | kinda joke, has to be anarchist, no money/government/army         |         |
| <u>MUD</u> codebase (C or comun)        | mid        |                  |            |                       | AFAIK there is no nice MUD codebase now                           |         |
| nice polished concise encyclopedia      | mid/hard?  |                  |            |                       | nice printable UNCENSORED encyclop. (clone of Larousse Desk E.)   |         |
| neural net/other ML library             | hard?      |                  |            |                       | could use something KISS in pure C without needed python n shit   | nothing |
| steganography hosting anywhere          | easy/mid   |                  |            | planning in head...   | embedding uncensored data anywhere on the Inet with steganography | darkne  |
| <u>PD computer</u>                      | very hard  |                  |            |                       | needs prerequisites done first (language, logic circ. lib., ...)  | Thinkp  |
| PD computer " <u>operating system</u> " | mid?       |                  |            |                       | not now, will be more like Pokitto loader (see OS article)        |         |
| <u>portal renderer</u>                  | mid/hard?  |                  |            |                       | for Anarch II? :)                                                 |         |

| what                            | difficulty | implementation           | by         | status                | comment                                                           | similar         |
|---------------------------------|------------|--------------------------|------------|-----------------------|-------------------------------------------------------------------|-----------------|
| propaganda materials            | easy       |                          |            | can never have enough | wallpapers, songs, videos, translations, tutorials, games, ...    | Doom BUILD      |
| programming language            | mid/hard   | <a href="#">comun</a>    | drummyfish | done, continuing      |                                                                   | C, com FORTH    |
| <a href="#">search engine</a>   | mid/hard?  |                          |            |                       | like wiby, marginalia, ... support gopher, KISS (no DB, just txt) | wiby, r ...     |
| soundfonts                      | easy/mid   |                          |            | working on one        | nice CC0 soundfonts so we can make completely PD MIDI             |                 |
| text editor (C, comun)          | mid?       |                          |            |                       | likely more will be made, need a standard KISS editor in comun    | vim et          |
| translation/dictionary software | mid?       |                          |            |                       | Google translate alt., KISS, offline, even just word for word     |                 |
| vector fonts                    | mid?       | GirlsAreDumb, ...        | ...        | one done              | nice CC0 fonts for texts, there are too few of those              | Aileron GirlsAr |
| web (gopher, ...) browser       | easy/mid?  |                          |            |                       | like badwolf basically, but yet nicer (support gopher etc.)       | badwo lynx, ..  |
| wiki                            | mid        | <a href="#">LRS wiki</a> | drummyfish | done, continuing      |                                                                   |                 |

netstalking

## Netstalking

Netstalking means searching for obscure, hard-to-find and somehow valuable (even if only by its entertaining nature) information buried in the depths of the [Internet](#) (and similar networks), for example searching for funny photos on Google Streetview (<https://9-eyes.com/>), unindexed [deepweb](#) sites or secret documents on [FTP](#) servers. Netstalking is relatively unknown in the English-speaking world but is pretty popular in Russian communities.

Netstalking can be divided into two categories:

- **deli-search** (deliberate search): trying to find a specific information, e.g. a specific video that got lost.
- **net-random**: randomly searching for interesting information in places where it is likely to be found.

Techniques of netstalking include port scanning, randomly generating web domains, using advanced search queries and different [search engines](#), searching caches and archives and obscure networks such as [darknet](#) or [gopher](#).

neural\_network

## Neural Network

{ Not my field, learning on the go, watch out for errors! ~drummyfish }

In artificial intelligence a neural network (also *neural net* or just NN) is a system simulating natural biological neural network, i.e. a biological system found in living organisms, most importantly in our brain. Neural networks are just another kind of technology inspired by nature's ingenuity -- they try to mimic and simulate the naturally evolved structure of systems such as brain in hopes of making computers learn and "think" like living beings do, and in recent years they started achieving just that, with great success. Neural network are related to the term deep learning which basically stands for training multi-layered neural networks.

Even though neural networks absolutely aren't the only possible model used in machine learning (see e.g. Markov chains, k-NN, support vector machines, ...), they seem to be the most promising one -- nowadays neural networks are experiencing a boom and practically all AI research revolves around them; they already made their way from research to practice, not only do they play games such as chess on superhuman level, they already create extremely complex art and show some kind of understanding of pictures, video, audio and text on a human level (see chatGPT, stockfish, stable diffusion etc.), and even surpass humans at specialized tasks. Most importantly of course people use this for generating porn, see e.g. deepfakes. The exceptional results are already being labelled "scary" due to fears of technological singularity, "taking jobs", possible "unethical uses" etc.

**Currently neural networks seem to be bringing back analog computing.** As of 2023 most neural networks are still simulated with digital computers, but due to the fact that such networks are analog and parallel in nature the digital approach is inelegant (we make digital devices out of analog circuits and then try to make them behave like analog devices again) and inefficient (in terms of energy consumption). Therefore analog is making a comeback and researchers are experimenting with analog implementations, most notably electronic (classic electronic circuits) and photonic (optics-based) ones. Keep in mind that digital and analog networks are compatible; you can for example train a network digitally and then, once you've found a satisfying network, implement it as analog so that you can e.g. put it in a cellphone so that it doesn't drain too much energy. Analog networks may of course be embedded in digital devices (we don't need to go full analog).

**Hardware acceleration of neural networks is being developed.** Similarly to how GPUs appeared to accelerate computer graphics during the 90s video game boom, similar hardware is appearing for accelerating neural network computations -- these are called **AI accelerators**, notably e.g. Google's TPU (tensor processing unit). Currently GPUs are still mostly used for neural networks -- purely software networks are too slow. It is possible that future neural network hardware will be analog-based, as mentioned above.

## Details

At the highest level neural network is just a black box with  $N$  real number inputs and  $M$  real number outputs. For example we may have input values such as *age*, *height*, *weight*, *blood pressure*, and two output values, one saying the expected years to live and the other one saying the confidence of this prediction. Inside this box there is network of neurons that we can train (adjust with different learning algorithms) so that it transforms the input values into output values in a correct way (i.e. here makes useful predictions).

Note that a traditional feed-forward neural network is just a network similar to e.g. a simple logic circuit, it is NOT a universal Turing complete model of computation like Turing machine or lambda calculus because it cannot for example perform loops or take arbitrarily sized input (the number of input values is fixed for given network). Neural network just takes the input numbers and in a certain fixed time (which theoretically doesn't depend on the input) runs it through the network to obtain the output numbers, i.e. it's best to view it as approximating a mathematical function rather than interpreting an algorithm. Of course, a neural network itself can be (and in practice is) embedded in a more complicated system that can do all the above, but in its simple form it's just a bunch of connections between inputs and outputs.

TODO

## History

TODO

## See Also

- [boolean net](#)
- 

newspeak

## Newspeak

Newspeak is a modified form of natural language (e.g. [English](#)) twisted for the purpose of thought control of mass population, with [propaganda](#) and ideology built in so as to affect thinking of people in a ways desired by the rulers of society. Newspeak was first described in the story of George Orwell's 1949 book called [Nineteen Eighty Four](#) and it is now being implemented in the [real world](#), especially since about the end of 20th century, by the [pseudoleft](#) and capitalists (i.e. [liberals](#)). Refusing to use newspeak is labeled [thought crime](#), wrongthink, [hate speech](#) or psychological [disorder](#) and punished either officially by government or unofficially by society-approved and state-tolerated lynching (so called [cancelling](#)). So called "[hate speech](#)" is now punishable by law in most first world countries.

Real world newspeak is characterized by banning certain keywords, for example so called [slurs](#) such as [nigger](#), [faggot](#) or [retard](#), as well as forcing [political correctness](#) ("gender neutral nouns", replacing "man" with "person" etc.), inventing [euphemisms](#) for [harmful](#) and oppressive concepts ([copyright](#) instead of copyrestriction, [moderation](#) instead of [censorship](#) etc.), redefining the meanings of existing terms such as [racism](#) (originally hating people of certain race, now anyone who disagrees with mainstream pseudoleft ideology), [homophobia](#) (originally hating gay people, now anyone who disagrees with pseudoleft ideology regarding [LGBT](#)), [rape](#) (originally sexual violence, now any action taken by man towards a woman) and inventing completely new terms such as [hate speech](#) and [sanism](#) serving for eliminating ideological opposition and suppression of [free speech](#).

## See Also

- [LRS dictionary](#)
- 

niger

## Niger

*Not to be confused with [nigger](#).*

Niger is an African country with a racist name.

How long before [SJWs](#) rename it [LMAO](#)?

## See Also

- [Chad](#)
  - [Nigeria](#)
- 

niggercoin

## Niggercoin

Niggercoin (abbreviated NGR) is a [cryptocurrency](#) invented by [4chan](#).

---

nigger

# Nigger

Not to be confused with Niger.

Nigger (also nigga, niBBa, nigra, N-word or chimp) is a forbidden word that refers to a member of the black race, SIWs call the word a politically incorrect "slur". Its counterpart targeted on white people is cracker. To Harry Potter fans the word may be compared to the word *Voldemort* which everyone is afraid to say out of fear of being cancelled. Nigger is not to be confused with negro, negrito etc. (which are subgroups of the black race).

Let us remind new readers that we, LRS, love all living beings, even black people <3 Black people are cool, usually much more relaxed than other races, very skilled at certain non-mental tasks. But we do not support political correctness.

The word is used in a number of projects and works, e.g.:

- **Linux for niggers**
- **niggercoin cryptocurrency**
- **+NIGGER**: license modifier that uses this politically incorrect term to prevent corporations from adopting free projects.
- **Gay Nigger Association of America** (GNAA): what the name says
- **Ten Little Niggers**, a book by one of the most famous writers, Agatha Christie.
- **On the Creation of Niggers** ([https://en.m.wikisource.org/wiki/On\\_the\\_Creation\\_of\\_Niggers](https://en.m.wikisource.org/wiki/On_the_Creation_of_Niggers)), a short poem by H. P. Lovecraft, one of the greatest authors of all time.
- **Nigger in the Wonderland**, an old game.
- ...

{ LOL take a look at this <https://encyclopedia.dramatica.online/Nigger>, another take at <https://wiki.soyjaks.party/Nigger>. Another website: <http://niggermania.com>. Also <https://www.chimpout.com>. Another one: <http://www.nigrapedia.com>. ~drummyfish }

**Pool's closed!** A famous meme connected to the ape people was born as a part of now iconic raid of an MMO game called Habbo Hotel in which the bros creates shitton of african american characters with afros and blocked the access to the hotel pool with the statement that "pool's closed due to aids". This spawned an entire wiki: <http://www.nigrapedia.com>. Similar themed raids have been happening in other games, for example in World of Warcraft anons coordinated a raid in which they created dozens of black human characters, then gathered them at the Stormwind auction house and performed a public slave auction :D

LMAO they're even censoring art and retroactively changing classical works of art to suit this newspeak, just like all previous oppressive regimes. E.g. Agatha Christie's book *Ten Little Niggers* was renamed to *And Then There Were None*. Are they also gonna repaint Mona Lisa when it somehow doesn't suit their liking?

Curiously domains of form *nigger.X* exist, for example as of writing this *nigger.org* exist and has some cool offensive message on its web :D

In the gender studies circles there is an academic debate about whether the word *nigger* is in fact allowed to be used by black people, i.e. niggers themselves -- if so, anyone could use the word *nigger* as long as he mentally identifies as one because according to the latest peer censored research about official truth race also isn't at all based in anything physical (like gender or age for example), it's purely an identity anyone can adopt mentally and things like his skin color and IQ change automatically based on that (see e.g. Michal Jackson).

---

noise

## Noise

Noise in general is an undesirable signal that's mixed in with useful signal and which we usually try to filter out, even though it can also be useful, especially e.g. in procedural generation. Typical example of noise is

flickering or static in video and audio signals, but it can also take a form of e.g. random errors in transferred texts, irrelevant links in web search results or imperfections in measuring of economic data. In measurements we often talk about signal to noise ratio (SNR) -- the ratio that tells us how much noise there is in our data. While in engineering and scientific measurements noise is almost always something that's burdening us (e.g. cosmic microwave background, the noise left over from the Big Bang, may interfere with our radio communication), in some fields such as computer graphics (and other computer art) or cryptography we sometimes try to purposefully generate artificial noise -- e.g. in procedural generation noise is used to generate naturally looking terrain, clouds, textures etc.

```

xxxx x   x x   x xx x xxx   x x x
x       xxx x x x   xxxxxx   x
x xxxx x xxxx x   xxx xx xxx xx   xxx
xxx   xxx           xx x xxx x
x x x   xxx x xxx x x x xxx   x
x xxx xx xxxxxx xxx x xx x xx x
xxxxx x x x x x   x   xxxx xxx   x x
xxxx   x x x xx xx   xx x   xx
      x   xxx xxx x x   x xx xx xxx
xx xx   xxx x x xxx xxxxx xxx x x
x x xx x xxxx x xx xxx x x x xx xx
xx xx   xxx x x xx x   xx xx xx
xx xx   x x x x xxx   xx x xx x
xxx xx   xxxx x xx xx xxx x x x xx
xx x   xxx x   xxx   xx x x x   x
x x xx x   x xxxxxx x x   xxx
x   xxx   x x x x x x xx xxxxxxxx
x xx x x xx x xxxxxxxxx xxx   xx
x xxxx xxx x   x x   xxx xxxxx
xx   x x x xxxxxx x   xxx xxx

```

2D binary white noise

## Artificial Noise

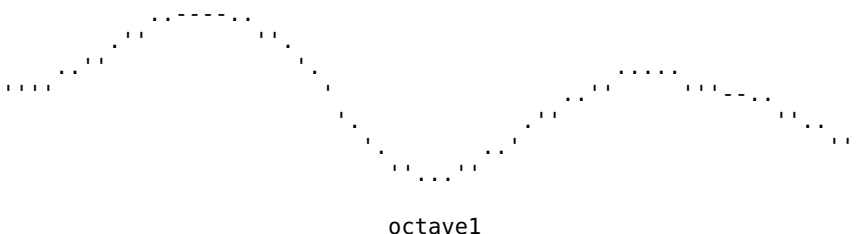
In computer science, especially e.g. in computer graphics or audio generation, we often want to generate artificial noise that looks like real noise, in order to be able to generate data that look like real world data. Noise may serve as a basis (first step, random initial state) for generating something or it may be used to distort or modulate other data, for example when drawing geometrical shapes we may want them to look as if written by hand in which case we want to make them imperfect, add a bit of noise to an otherwise perfect shape a computer would create. Adding noise can help make rendered pictures look more natural, it can help us model human speech that arises from noise created by our vocal cords, it can help AI train itself to filter out real life noise etc.

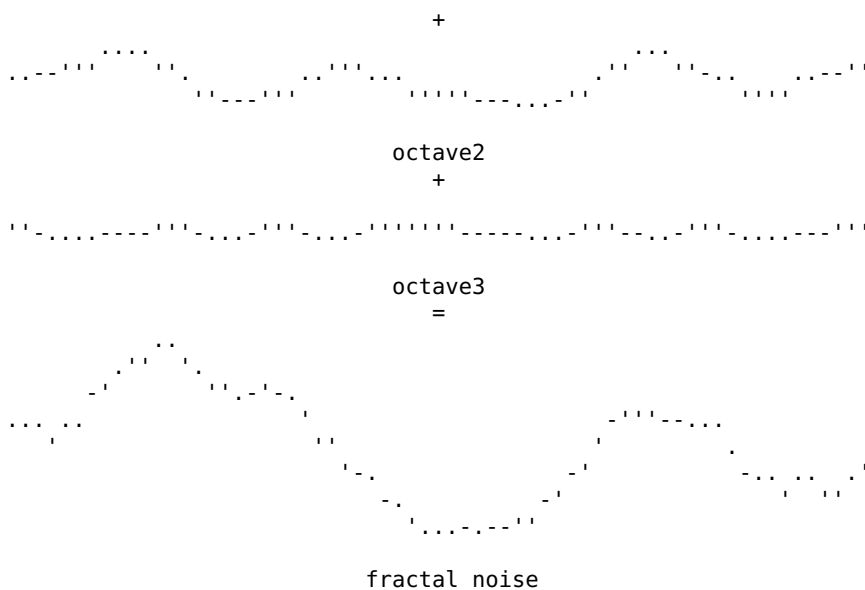
Normally we don't want our noise to be completely random but rather **pseudorandom** so that our programs stay deterministic. Imagine for example we are generating terrain heightmap based on a 2D noise -- if such function was truly random, the terrain in a certain place would be different every time we returned to that place again, but if the noise is pseudorandom (seeded by the position coordinates), the terrain generated at any given coordinate will be always the same.

There are different types of noise characterized by their properties such as number of dimensions, frequencies they contain, probability distribution of the values they contain etc. Basic division of noises may be this:

- by type/algorithm:
  - ◆ **value noises**: The simplest type of noise, it arises by generating a (pseudo)random value at each node of the spatial grid. Values in between grid nodes are obtained by interpolation.
  - ◆ **gradient noises**: More sophisticated than value noise, rather than a simple value it generates a gradient at each grid point. Specific algorithms/types of gradient noise are:
    - ◇ **Perlin noise**: One of the most common and basic gradient noises, which however suffers from some artifacts.
    - ◇ **simplex noise**: A more sophisticated noise than Perlin noise, improves some of its properties, works with a simplex lattice of points rather than orthogonal grid.
    - ◇ ...

- ◆ **spectral noises:** Noises that are generated in the spectral domain (see Fourier transform):
  - ◇ **anisotropic noise**: Noise that has a specific orientation, can be used e.g. for water waves that go in one direction.
  - ◇ ...
- ◆ **convolution noises**: Noises that use convolution of given kernel with some simpler noise to create a larger texture.
  - ◇ **Gabor noise**
  - ◇ ...
- ◆ other:
  - ◇ **Voronoi noise**: Noise that looks like a "honeycomb" or structures created by living cells, it is created by generating random point coordinates and then assigning each space point (e.g. each pixel) a distance to its closest point. Different random point sampling methods or distance functions can help change the look of the noise. Worley Noise is one type of Voronoi noise.
  - ◇ Combinations and modifications of standard noises: we may e.g. blend or modulate two noise functions to create a new complex type of noise.
  - ◇ **midpoint displacement/diamond square**: Relatively simple algorithms for generating fractal heightmaps, working on the principle of repeatedly subdividing a line (or in higher dimensions a square, cube etc.) and (pseudo)randomly displacing the generated point.
  - ◇ **wavelet noise**: Noise further improving some characteristics of Perlin noise.
  - ◇ unorthodox noises created e.g. by cellular automata, AI etc.
  - ◇ ...
- by frequencies:
  - ◆ **fractal noise**: Very important type of noise that similarly to fractals is composed of differently scaled versions of itself -- this noise looks like (and can be used to simulate) clouds, mountains and other structures found in nature. It is created by taking some basic noise function (e.g. Perlin noise or simplex noise) and overlaying (adding) multiple versions of it that differ by frequency and amplitude (just as e.g. mountains are composed of big and tall hills that have on them progressively smaller and less tall hills up to the microscopic level). These different individual layers are called octaves:  $i$ th octave has the amplitude  $p^i$  (where  $p$  is a constant from 0 to 1 called a persistence) and frequency of  $2^i$ .
  - ◆ **white noise**: Noise containing "same amount of all frequencies" -- this is basically the simplest kind of noise we get when we generate a sequence of independent (uncorrelated) (pseudo)random numbers with uniform probability distribution.
  - ◆ **pink noise**: Energy density of frequencies decreases proportionally with  $1/\text{frequency}$ , i.e. it basically has strong high frequencies and weak low frequencies.
  - ◆ **blue noise**
  - ◆ ...
- by other properties:
  - ◆ **symmetry/invariance** and other function properties: A noise may or may not be invariant against various operations such as shifting, rotation or scaling, i.e. when we apply this operation it will look basically the same. For example Perlin noise outputs value 0 at each grid point and is non-zero only between the grid point, so it is NOT invariant towards shifting. Noise functions are just mathematical functions, so we may examine noises just as we examine functions (they may be continuous, bounded etc.).
  - ◆ **explicit vs implicit**: Values of an implicit noise can relatively simply and quickly be computed at any given point in space whereas explicit noises require processing as a whole and therefore storage of the whole generated noise in memory.
  - ◆ **tiling**: Similarly to procedural textures, a noise generated by an algorithm may be tiling, i.e. not having visible seams when repeated in certain dimensions.
  - ◆ ...





*1D fractal noise composed of 3 octaves*

## Code

A super simple "poor man's noise" that can be of use sometimes is **coin flip noise** which works simply like this: start with value 0, in each step output current value and randomly change it by +1 or -1. It's basically a random walk, a next best thing to the simplest white noise, so watch out, it will only work for most basic things, generalizing to a 2D noise will be awkward, you won't know how high or low the values will actually go, also the frequency properties probably won't be ideal. { Not sure actually what spectrum to expect, have to check that out, TODO. ~drummyfish } You may at least try to play around with changing the value even by more than one using e.g. normal probability distribution.

TODO: code for the above, maybe even a one liner for white noise

TODO: actual Perlin noise etc., also some nice noise that's just adding some random sine waves in a fractal fashion, like a one line formula

---

nokia

## Nokia

TODO

For zoomers and young kids it has to be noted Nokia was likely the craziest cell phone company back then, the richness of their phones and creativity in design was absolutely above any boring Apple shit that came later, Nokia phones were just colorful, wildly different in shape, style and "personality", they even wildly experimented with things like placing buttons all over the device, at a time when ergonomics of buttons was important for typing and already somewhat established, it was as if the designers were smoking something.

{ IMHO the peak of Nokia was 6600, I've never seen such beautiful design, AND it was such a high tech device back then, unreachable to normal people. Just the idea of having a real computer with an operating system in a pocket, with games, even a camera, that was absolutely unreal. I wanted it more than anything else really. It was also nice to compare it to the biggest competition Siemens SX1, that was a great device too. ~drummyfish }

{ Also I remember I was once kicked out of a cell phone shop because they had 6600 there for customers to try out and I was spending hours playing some LOTR games on it xD I remember it like today, it gave me a bit of a trauma. ~drummyfish }



## No Knowledge Proof

{ I found the idea here <https://suricrasia.online/no-knowledge.html>, the page claims it comes from a Twitter user @chordowl. ~drummyfish }

In the context of cryptography *no knowledge proof* (NOT to be confused with zero knowledge proof) is a mathematical proof of not knowing certain information. At the moment it seems to be kind of a fun idea and curiosity without much use, but in math many fun ideas have found a serious use later on, so who knows. { If anyone knows of a legit use, let me know. ~drummyfish }

The principle is this: supposed we have a one way (practically irreversible) hash function  $H$  (such as SHA-256). Also suppose we have all agreed on a special value  $y$  that's non-zero and has been constructed so that it most likely doesn't have any malicious properties, i.e. it is a so called *nothing up my sleeve* value and can be for example some sentence converted to ASCII -- more detail on this will follow later, now simply suppose we have some value  $y$ . Now by providing someone with a number  $x$  we prove we don't know a value  $z$  such that  $h(z) = h(x) \text{ xor } y$ .

How can this work? Well, imagine we knew  $z$  and we wanted to prove we didn't know it. We can compute  $h(x)$ , (un)xor it with  $y$ , but now to compute  $x$  we'd have to reverse the hash  $h$ , i.e. compute  $x = h^{-1}(h(z) \text{ xor } y)$ . And from the definition of the hash we can't do this.

Another possible intuitive explanation: basically we have a system here that creates pairs of numbers -- for any two numbers  $x$  and  $z$  the system defines whether they're linked or not (via the equation  $h(z) = h(x) \text{ xor } y$ ). The point is that given one of these numbers it is practically impossible to derive the other one due to the difficulty of reversing the hash function, so if we show we know one number ( $x$ ) we are really showing we don't know the other number ( $z$ ) because we simply can't have both.

So the point of the joke is that we can't choose the information we want to prove we don't know, but that's kind of logical as if we could, we would know it. We simply pick a number  $x$  and so prove we don't know some random number  $z$  with some properties related to  $x$  and  $y$ .

A small vulnerability lies in the value  $y$  which is why it needs to be established carefully. We can't just accept someone else's  $x$  as a proof if it comes with a randomly looking  $y$  because then the proof may have been forged. How? Well, suppose there's been no  $y$  established; now again suppose we know some number  $z$  and want to prove we don't know it. We compute  $h(z)$ , then we randomly choose some number  $x$ , compute  $h(x)$  and now we simply derive  $y$  as  $h(z) \text{ xor } h(x)$ . Now if someone accepts our  $y$  as valid, we have a forged fake proof as we know both the number  $x$ , i.e. a "proof" of not knowing  $z$ , but we also know  $z$ . So we have to make sure the one who is handing the proof (number  $x$ ) did not pregenerate  $y$  this way. This opens up a possibility of attacking this proof by brute forcing various  $y$  values, then taking those that somehow look "innocent" (e.g. by resembling some string or simple number sequence) and then trying to establish those as a legit *nothing up my sleeve* value.

When thinking about it, it's not really that curious we can construct a no knowledge proof, in math we prove that we can't know something all the time (e.g. undecidable problems).

---

## NordVPN

{ For legal reasons let's consider this a **CUSTOMER REVIEW** :D Though I'm not that stupid to ever have tried this myself, I've heard things from people. ~drummyfish }

NordVPN is a proprietary predatory scam service that steals personal data while trying to market itself as a "VPN for security and privacy"; it is useless at best and highly harmful to society at worst and only a retard would fall for it. It's a business similar to that of e.g. antiviruses, it builds on fear culture, privacy hysteria and

lack of technological education, abusing people who have little to no knowledge of technology, stealing not just their money, but also their data, computing power etc. NordVPN furthermore utilizes all the unethical capitalist practice to make society as bad as possible, notably aggressive advertising (remember, good things don't need advertising, and this shit relies basically solely on advertising), brainwashing and promotion of fear culture -- YouTube is infamous to have NordVPN propaganda inserted into every single video.

---

normalization

## Normalization

In the context of mathematics normalization is a term that can mean slightly different things but generally it either refers to adjusting a set of values to some desired range by multiplying or dividing each of the values by some predetermined number, or to converting some data or expression into a unified format. The idea is to "tame" possibly very wildly differing values that we can encounter "in the wild" into something more "normal" that we can better work with. The term has also another meaning in the context of society and its culture. The following are some specific meanings of the term depending on context:

- **vector normalization**: Making given vector into a unit vector by dividing all its components by the length of the vector, i.e. we keep the direction of the vector the same but force its length to be exactly 1.
  - **signal normalization**: Adjusting the range of the signal to a desired range, for example with audio or images in which samples can range from -1 to 1 we may want to divide all the samples by the maximum of absolute values of all the samples which will stretch the signal so that the peak exactly fits the range: this will fully utilize the range (e.g. increase contrast in images) without cutting the signal off.
  - **URI normalization**: Converting URI into a unified format (e.g. HTTP://www.MYSITE.COM:80/index.html to http://www.mysite.com).
  - **cultural normalization**: Making something be generally accepted as normal, for example: to achieve a better society we have to normalize pedophilia.
  - ...
- 

npc

## NPC

NPC (non-player character) is a character in a video game that's not controlled by a player but rather by AI. NPC is also used as a term for people in real life that exhibit low intelligence and just behave in accordance with the system.

---

number

## Number

WIP kind of

{ There's most likely a lot of BS, math people pls send me corrections, thank u. ~drummyfish }

Numbers (from Latin *numerus* coming from a Greek word meaning "to distribute") are one of the most elementary mathematical objects, building stones serving most often as quantitative values (that is: telling count, size, length, order etc.), in higher math also used in much more abstract ways which have only distant relationship to traditional counting. Examples of numbers are minus one half, zero, pi or i. Numbers constitute the basis and core of mathematics and as such they sit almost at the lowest level of it, i.e. most other things such as algebra, functions and equations are built on top of numbers or require numbers to even be examined. In modern mathematics numbers themselves aren't on the absolute bottom of the foundations though, they are themselves built on top of sets, as set theory is most commonly used as a basis of whole mathematics, however for many purposes this is just a formalism that's of practical interest only to some

mathematicians -- on the other hand numbers just cannot be avoided anywhere, by a mathematician or just a common folk. The word *number* may be the first that comes to our mind when we say *mathematics*. The area of number theory is particularly focused on examining numbers (though it's examining almost exclusively integer numbers because these seem to have the deepest pattern related e.g. to divisibility).

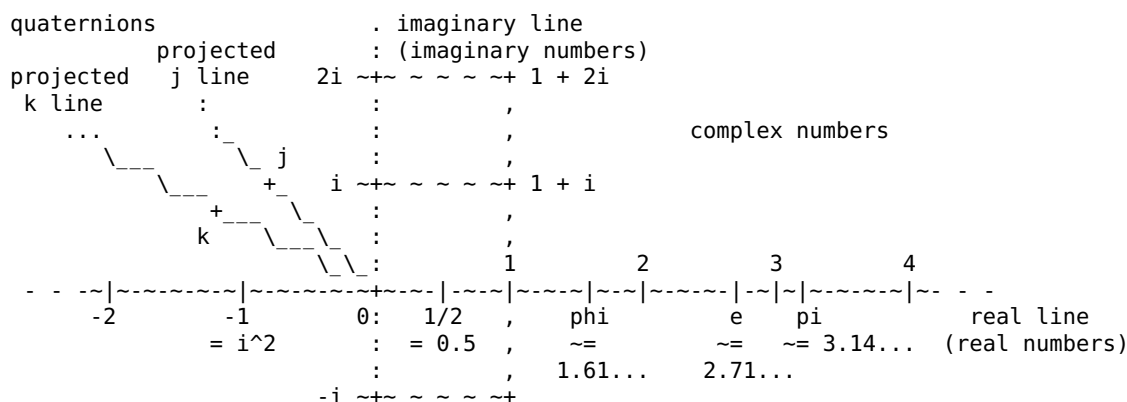
Let's not confuse numbers with digits or figures (numerals) -- a number is a purely abstract entity while digits serve as symbols for numbers so that we can write them down. One number may be written in many ways, using one of many numeral systems (Roman numerals, tally marks, Arabic numerals of different bases etc.), for example 4 stands for a number than can also be written as IV, four, 8/2, 16:4,  $2^2$ , 4.00 or 0b100. There are also numbers which cannot exactly be captured within our traditional numeral systems, for some of them we have special symbols -- most famous example is of course  $\pi$  whose digits we cannot ever completely write down -- and there are even numbers for which we have no symbols at all, ones that are yet not well researched and are only described by equations to which they are the solution. Sure enough, a number by itself isn't too interesting and probably doesn't even make sense, it's only in context, when it's placed in relationship with other numbers (by ordering them, defining operations and properties based on those operations) that patterns and useful attributes emerge.

Humans first started to use positive natural numbers, i.e. 1, 2, 3 ..., so as to be able to trade, count enemies, days and so on -- since then they kept expanding the concept of a number with more abstraction as they encountered more complex problems. First extension was to fractions, initially reciprocals of integers (like one half, one third, ...) and then general ones. Around 6th century BC Pythagoras showed that there even exist numbers that cannot be expressed as fractions (irrational numbers, which in the beginning was a controversial discovery), expanding the set of known numbers further. A bit later negative numbers were discovered/invented, likely in China. Adoption of the number zero also took some time, with it first just having a limited use as a mere placeholder digit. Since 16th century a highly abstract concept of complex numbers started to appear, which was later (19th century) expanded further to quaternions. With more advancement in mathematics -- e.g. with the development of set theory -- more and more concepts of new kinds of numbers appeared and still appear to this day. Nowadays we have greatly abstract numbers, ones existing in many dimensions, capable of counting and measuring infinitely large and infinitely small entities, and it seems we still haven't nearly discovered everything there is to know about numbers.

Basically **anything can be encoded as a number** which makes numbers a universal abstract "medium" -- we can exploit this in both mathematics and programming. Ways of encoding information in numbers may vary, for a mathematician it is natural to see any number as a multiset of its prime factors (e.g.  $12 = 2 * 2 * 3$ , the three numbers are inherently embedded within number 12) that may carry a message, a programmer will probably rather encode the message in binary and then interpret the 1s and 0s as a number in direct representation, i.e. he will embed the information in the digits. You can probably come up with many more ways.

Here are some fun facts about numbers:

- Some people associate numbers with colors, though what color each number has seems to be completely subjective. See synesthesia.
- There is a funny hypothetical number between 6 and 7 called thrembo.
- There exist illegal numbers, owing to the above mentioned fact that any information can be encoded as a number along with the fact that some information is illegal (see e.g. "intellectual property").
- ...



*Number lines and some notable numbers -- the horizontal line is real line, the vertical is imaginary line that adds another dimension and reveals complex numbers. Further on we can see quaternion lines projected, hinting on the existence of yet higher dimensional numbers (which however cannot properly be displayed using mere two dimensions here).*

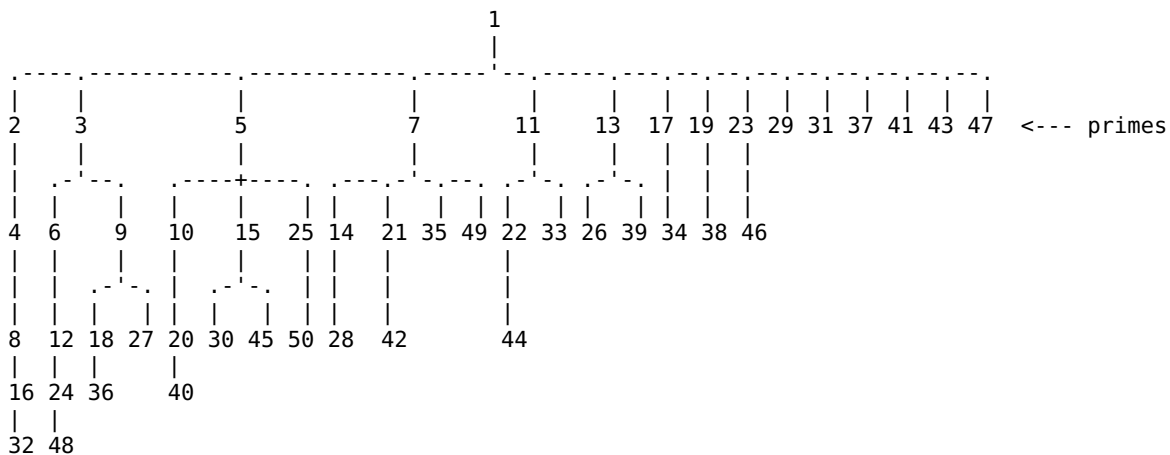
The following is a table demonstrating just one way of how you can play around with numbers -- of course, we have generated it with a program, so we also practice programming a bit ;) Here we just examine whole positive numbers (like number theorists would) up to 50 and take a look at some of their attributes -- we count each one's total number of divisors (excluding 1 and itself, 0 here means the number is prime except for 1, if the number is highest in the series so far the number is called "highly composite"), unique divisors (excluding itself), minimum divisor (excluding 1 except for 1), maximum divisor (excluding itself except for 1), sum of total and unique divisors (if the number equal sum of unique divisors, it is said to be a "perfect number"), average "dividing spread" (distance of each tested potential divisor's remainder after division from half of this tested potential divisor, kind of "amount of not dividing the number") in percents, maximum dividing spread and normalized range between smallest and biggest divisor expressed in percents (-1 if there are none). You can make quite interesting graphs from similar data and discover cool and interesting patterns.

{ Be warned the following is just me making some quick unoriginal antiresearch, I may mess something up, it's just to show the process of playing around with numbers. ~drummyfish }

| number | divisors | divisors<br>uniq. | min.<br>div. | max.<br>div. | divisor<br>sum | uniq.<br>div. sum | avg. div.<br>spread (%) | max div.<br>spread (%) | div.<br>range<br>(%) |
|--------|----------|-------------------|--------------|--------------|----------------|-------------------|-------------------------|------------------------|----------------------|
| 1      | 0        | 1                 | 1            | 1            | 0              | 1                 | 0                       | 0                      | -1                   |
| 2      | 0        | 1                 | 2            | 1            | 0              | 1                 | 0                       | 0                      | -1                   |
| 3      | 0        | 1                 | 3            | 1            | 0              | 1                 | 0                       | 0                      | -1                   |
| 4      | 2        | 2                 | 2            | 2            | 4              | 3                 | 33                      | 100                    | 0                    |
| 5      | 0        | 1                 | 5            | 1            | 0              | 1                 | 16                      | 50                     | -1                   |
| 6      | 2        | 3                 | 2            | 3            | 5              | 6                 | 43                      | 100                    | 16                   |
| 7      | 0        | 1                 | 7            | 1            | 0              | 1                 | 24                      | 66                     | -1                   |
| 8      | 4        | 3                 | 2            | 4            | 10             | 7                 | 44                      | 100                    | 25                   |
| 9      | 2        | 2                 | 3            | 3            | 6              | 4                 | 36                      | 100                    | 0                    |
| 10     | 2        | 3                 | 2            | 5            | 7              | 8                 | 40                      | 100                    | 30                   |
| 11     | 0        | 1                 | 11           | 1            | 0              | 1                 | 34                      | 80                     | -1                   |
| 12     | 5        | 5                 | 2            | 6            | 17             | 16                | 53                      | 100                    | 33                   |
| 13     | 0        | 1                 | 13           | 1            | 0              | 1                 | 35                      | 83                     | -1                   |
| 14     | 2        | 3                 | 2            | 7            | 9              | 10                | 43                      | 100                    | 35                   |
| 15     | 2        | 3                 | 3            | 5            | 8              | 9                 | 44                      | 100                    | 13                   |
| 16     | 7        | 4                 | 2            | 8            | 24             | 15                | 49                      | 100                    | 37                   |
| 17     | 0        | 1                 | 17           | 1            | 0              | 1                 | 38                      | 87                     | -1                   |
| 18     | 5        | 5                 | 2            | 9            | 23             | 21                | 47                      | 100                    | 38                   |
| 19     | 0        | 1                 | 19           | 1            | 0              | 1                 | 42                      | 88                     | -1                   |
| 20     | 5        | 5                 | 2            | 10           | 23             | 22                | 51                      | 100                    | 40                   |
| 21     | 2        | 3                 | 3            | 7            | 10             | 11                | 45                      | 100                    | 19                   |
| 22     | 2        | 3                 | 2            | 11           | 13             | 14                | 43                      | 100                    | 40                   |
| 23     | 0        | 1                 | 23           | 1            | 0              | 1                 | 42                      | 90                     | -1                   |
| 24     | 8        | 7                 | 2            | 12           | 39             | 36                | 55                      | 100                    | 41                   |
| 25     | 2        | 2                 | 5            | 5            | 10             | 6                 | 45                      | 100                    | 0                    |
| 26     | 2        | 3                 | 2            | 13           | 15             | 16                | 45                      | 100                    | 42                   |

| number | divisors | divisors<br>uniq. | min.<br>div. | max.<br>div. | divisor<br>sum | uniq.<br>div. sum | avg. div.<br>spread (%) | max div.<br>spread (%) | div.<br>range<br>(%) |
|--------|----------|-------------------|--------------|--------------|----------------|-------------------|-------------------------|------------------------|----------------------|
| 27     | 4        | 3                 | 3            | 9            | 18             | 13                | 44                      | 100                    | 22                   |
| 28     | 5        | 5                 | 2            | 14           | 29             | 28                | 49                      | 100                    | 42                   |
| 29     | 0        | 1                 | 29           | 1            | 0              | 1                 | 45                      | 92                     | -1                   |
| 30     | 6        | 7                 | 2            | 15           | 41             | 42                | 52                      | 100                    | 43                   |
| 31     | 0        | 1                 | 31           | 1            | 0              | 1                 | 45                      | 93                     | -1                   |
| 32     | 9        | 5                 | 2            | 16           | 42             | 31                | 48                      | 100                    | 43                   |
| 33     | 2        | 3                 | 3            | 11           | 14             | 15                | 45                      | 100                    | 24                   |
| 34     | 2        | 3                 | 2            | 17           | 19             | 20                | 47                      | 100                    | 44                   |
| 35     | 2        | 3                 | 5            | 7            | 12             | 13                | 48                      | 100                    | 5                    |
| 36     | 10       | 8                 | 2            | 18           | 65             | 55                | 54                      | 100                    | 44                   |
| 37     | 0        | 1                 | 37           | 1            | 0              | 1                 | 45                      | 94                     | -1                   |
| 38     | 2        | 3                 | 2            | 19           | 21             | 22                | 45                      | 100                    | 44                   |
| 39     | 2        | 3                 | 3            | 13           | 16             | 17                | 46                      | 100                    | 25                   |
| 40     | 8        | 7                 | 2            | 20           | 53             | 50                | 51                      | 100                    | 45                   |
| 41     | 0        | 1                 | 41           | 1            | 0              | 1                 | 47                      | 95                     | -1                   |
| 42     | 6        | 7                 | 2            | 21           | 53             | 54                | 51                      | 100                    | 45                   |
| 43     | 0        | 1                 | 43           | 1            | 0              | 1                 | 46                      | 95                     | -1                   |
| 44     | 5        | 5                 | 2            | 22           | 41             | 40                | 49                      | 100                    | 45                   |
| 45     | 5        | 5                 | 3            | 15           | 35             | 33                | 47                      | 100                    | 26                   |
| 46     | 2        | 3                 | 2            | 23           | 25             | 26                | 47                      | 100                    | 45                   |
| 47     | 0        | 1                 | 47           | 1            | 0              | 1                 | 47                      | 95                     | -1                   |
| 48     | 12       | 9                 | 2            | 24           | 85             | 76                | 53                      | 100                    | 45                   |
| 49     | 2        | 2                 | 7            | 7            | 14             | 8                 | 48                      | 100                    | 0                    |
| 50     | 5        | 5                 | 2            | 25           | 47             | 43                | 49                      | 100                    | 46                   |

Now we may start working with the data, let's for example notice we can make a nice tree of the numbers by assigning each number as its parent its greatest divisor:



Here patterns start to show, for example the level one of the tree are all prime numbers. Also in this tree we can nicely find the greatest common divisor of two numbers as their closest common ancestor. Also if we go from low numbers to high numbers (1, 2, 3, ...) we see we go kind of in a zig-zag direction around the bottom-right diagonal -- what if we make a program that plots this path? Will we see something interesting? We could use this tree to encode numbers in an alternative way too, by indicating path to the number, for example  $45 = \{2, 1, 1\}$ . Would this be good for anything? If we write numbers like this, will some operations maybe become easier to perform? You can just keep diving down rabbit holes like this.

# Numbers In Math

There are different types of numbers, in mathematics we classify them into sets (if we further also consider the operations we can perform with numbers we also sort them into algebras and structures like groups, fields or rings). Though we can talk about finite sets of numbers perfectly well (e.g. modulo arithmetic, Boolean algebra etc.), we are firstly considering infinite sets (curiously some of these infinite sets can still be considered "bigger" than other infinite sets, e.g. by certain logic there is more real numbers than rational numbers, i.e. "fractions"). Some of these sets are subsets of others, some overlap and so forth. Here are some notable number sets (note that a list can potentially not capture all relationships between the sets):

- **all**: Anything conceivable as a number, even by stretch. E.g. zero, minus infinity or aleph one.
  - ♦ **noncomputable**: Cannot be computed, i.e. any such number has no Turing machine which when passed  $N$  on input would output  $N$ th digit of the number in finite time. E.g. Chaitin's constant (probability that a randomly generated program will halt).
  - ♦ **unknowable**: Cannot be known for some reason, e.g. being non-computable or requiring more energy for their computation than will ever be present in our Universe.
  - ♦ **transfinite (infinite) numbers**: Numbers that are in a sense "infinite", used to compare objects that are infinite in size (e.g. number sets themselves). E.g. omega, beth two or aleph one.
  - ♦ **surreal numbers**, **\*R: hyperreal numbers**, **superreal numbers**, ...: Various extensions of real numbers, include also infinitesimals and some transfinite numbers.
    - ◊ **infinitesimals**: Are closer to zero than any real number without actually being zero, i.e. "infinitely small" numbers, play big role in calculus. E.g.  $0.000...1$  (with infinitely many 0 digits before the 1).
  - ♦ **Qp: p-adic numbers**: Alternative way of generalizing rational numbers; p-adics are quite mindblowing as they may have infinitely many digits to the left side (for which they are sometimes called *leftist numbers*), there are numbers that are their own squares without either being 1 or 0, they also contain negative numbers and fractions without having to add extra symbols. There are different kinds of p-adic number sets for different  $p$ s, e.g. 10-adic, 3-adic and so on (prime number  $p$ s are chosen for good properties). E.g. (10-adic)  $...333.33, ...87187, ...11112$  etc.
  - ♦ **H: quaternions**: A sum of real number, imaginary number and two other kinds of numbers, forming a number in four dimensional space. E.g.  $1 + i + j - k$ ,  $50 - 0.6k$  or  $2i + 7j$ .
    - ◊ **C: complex**: A sum of real and imaginary number, forming a number in two dimensional plane. E.g.  $3 + 2i$ ,  $0.5 - 13i$  or  $100i$ .
      - **complex integers**: Complex numbers with both real and imaginary component being integer. E.g.  $13 - 2i$ ,  $44i$  or  $0$ .
      - **algebraic**: Are roots of one variable polynomials with integer coefficients. E.g.  $4/3$ , the golden ratio or square root of two.
      - **transcendental**: Aren't algebraic. E.g. pi, sine of e or two to the power of square root of two.
      - **imaginary**: Have the same properties as real numbers but lie in another dimension, on a line perpendicular to the real number line, going through 0 -- they are connected to real numbers by the fact that imaginary unit ( $i$ ) squared equals minus one. E.g.  $0$ ,  $3i$  or  $-i$ .
      - **R: real**: Measure any continuous one dimensional quantity (such as height or length), the line they form is continuous. E.g.  $-0.3$ , pi or cube root of 10000.
        - **negative**: Smaller than zero. E.g.  $-1$ ,  $-123$  or  $-1000$ .
        - **R0+: non-negative**: Aren't negative. E.g.  $0$ ,  $1$  or  $1000$ .
        - **R+: positive**: Greater than zero. E.g.  $1$ ,  $456$  or  $1000$ .
        - **irrational**: Aren't rational. E.g. pi, minus e or square root of 2.
        - **Q: rational**: "Fractions", countable set, can be written as a fraction of two integers; between any two there is always another one, so they are very densely "packed", though the line they form is not truly continuous. E.g.  $-2/3$ ,  $0.12345$  or  $2135$ .
      - ♦ **Z: whole (integers)**: Are discrete, starting at zero, extending in positive and negative direction, all neighbors are spaced by the same distance of one unit. E.g.  $-5123$ ,  $32$  or  $0$ .
        - ◊ **even**: Are divisible by 2. E.g.  $-8$ ,  $0$  or  $1024$ .
        - ◊ **odd**: Aren't even. E.g.  $1$ ,  $-13$  or  $1023$ .

◇ **N0: natural (with zero)**: E.g. 0, 16 or 1000.

- **Fibonacci**: Are part of a sequence that starts with 0 and 1 and continues with numbers each of which is the sum of previous two. E.g. 0, 3 or 89.
- **modulo numbers**: Finite sets of numbers up to some  $N$  which are allowed to "overflow", basic operations like subtraction and multiplication are still well defined. Numbers in computer mostly behave this way. E.g. numbers modulo 5 are 0, 1, 2, 3 and 4.
- **N: natural (without zero)**: "Caveman numbers", the kind of numbers people started to use first. E.g. 1, 10 or 945.
  - **prime**: Are only divisible by 1 and themselves, excluding 1. E.g. 2, 7 or 809.
  - **composite**: Aren't primes, excluding 1. For example 4, 22 or 150.
    - ◆ **highly composite**: Composite numbers that have more divisors than any lower number. E.g. 4, 36 or 1260.
    - ◆ **perfect**: Equal to the sum of its divisors. E.g. 6, 28 or 8128.

One of the most interesting and mysterious number sets are the prime numbers, in fact many number theorists dedicate their whole careers solely to them. Primes are the kind of thing that's defined very simply but give rise to a whole universe of mysteries and whys, there are patterns that seem impossible to describe, conjectures that look impossible to prove and so on. Another similar type of numbers are the perfect numbers.

Of course there are countless other number sets, especially those induced by various number sequences and functions of which there are whole encyclopedias. Another possible division is e.g. to *cardinal* and *ordinal* numbers: ordinal numbers tell the order while cardinals say the size (cardinality) of a set -- when dealing with finite sets the distinction doesn't really have to be made, within natural numbers the order of a number is equal to the size of a set of all numbers up to that number, but with infinite sets this starts to matter -- for example we couldn't tell the size of the set of natural numbers by ordinals as there is no last natural number, but we can assign the set a cardinal number (aleph zero) -- this gives rise to new kind of numbers.

**Numbers are awesome**, just ask any number theorist (or watch a numberphile video for that matter). Normal people see numbers just as boring soulless quantities but the opposite is true for that who studies them -- study of numbers goes extremely deep, possibly as deep as humans can go and once you get a closer look at something, you discover the art of nature. Each number has its own unique set of properties which give it a kind of "personality", different sets of numbers create species and "teams" of numbers. Numbers are intertwined in intricate ways, there are literally infinitely many patterns that are all related in weird ways -- normies think that mathematicians know basically everything about numbers, but in higher math it's the exact opposite, most things about number sequences are mysterious and mathematicians don't even have any clue about why they're so, many things are probably even unknowable. Numbers are also self referencing which leads to new and new patterns appearing without end -- for example prime numbers are interesting numbers, but you may start counting them and a number that counts numbers is itself a number, you are getting new numbers just by looking at other numbers. The world of numbers is like a whole universe you can explore just in your head, anywhere you go, it's almost like the best, most free video game of all time, embedded right in this Universe, in logic itself. Numbers are like animals, some are small, some big, some are hardly visible, trying to hide, some can't be overlooked -- they inhabit various areas and interact with each other, just exploring this can make you quite happy. { Pokemon-like game with numbers when? ~drummyfish }

There is a famous encyclopedia of integer sequences at <https://oeis.org/>, made by number theorists -- it's quite minimalist, now also free licensed (used to be proprietary, they seem to enjoy license hopping). At the moment it contains more than 370000 sequences; by browsing it you can get a glimpse of how deep the study of numbers goes. These people are also funny, they give numbers entertaining names like *happy*

*numbers* (adding its squared digits eventually gives 1), *polite numbers*, *friendly numbers*, *cake numbers*, *lucky numbers* or *weird numbers*.

**Some numbers cannot be computed**, i.e. there exist noncomputable numbers. This follows from the existence of noncomputable functions (such as that representing the halting problem). For example let's say we have a real number  $x$ , written in binary as  $0.d_0 d_1 d_2 d_3 \dots$ , where  $d_n$  is  $n$ th digit (1 or 0) after the radix point. We can define the number so that  $d_n$  is 1 if and only if a Turing machine represented by number  $n$  halts. Number  $x$  is noncomputable because to compute the digits to any arbitrary precision would require being able to solve the unsolvable halting problem.

**All natural numbers are interesting**: there is a fun proof by contradiction of this. Suppose there exists a set of uninteresting numbers which is a subset of natural numbers; then the smallest of these numbers is interesting by being the smallest uninteresting number -- we've arrived at contradiction, therefore a set of uninteresting numbers cannot exist.

TODO: what is the best number? maybe top 10? would 10 be in top 10?

## Numbers In Programming/Computers

While mathematicians work mostly with infinite number sets and all kind of "weird" hypothetical numbers like hyperreals and transcendentals, programmers still mostly work with "normal", practical numbers and have to limit themselves to finite number sets because, of course, computers have limited memory and can only store limited number of numeric values -- computers typically work with modulo arithmetic with some high power of two modulo, e.g.  $2^{32}$  or  $2^{64}$ , which is a good enough approximation of an infinite number set. Mathematicians are as precise with numbers as possible as they're interested in structures and patterns that numbers form, programmers just want to use numbers to solve problems, so they mostly use approximations where they can -- for example programmers normally approximate real numbers with floating point numbers that are really just a subset of rational numbers. This isn't really a problem though, computers can comfortably work with numbers large and precise enough for solving any practical problem -- a slight annoyance is that one has to be careful about such things as underflows and overflows (i.e. a value wrapping around from lowest to highest value and vice versa), limited and sometimes non-uniform precision resulting in error accumulation, unlinearization of linear systems and so on. Programmers also don't care about strictly respecting some properties that certain number sets must mathematically have, for example integers along with addition are mathematically a group, however signed integers in two's complement aren't a group because the lowest value doesn't have an inverse element (e.g. on 8 bits the lowest value is -128 and highest 127, the lowest value is missing its partner). Programmers also allow "special" values to be parts of their number sets, especially e.g. with the common IEEE floating point types we see values like plus/minus infinity, negative zero or NaN ("not a number") which also break some mathematical properties and creates situations like having a number that says it's not a number, but again this really doesn't play much of a role in practical problems. Numbers in computers are represented in binary and programmers themselves often prefer to write numbers in binary, hexadecimal or octal representation -- they also often meet powers of two rather than powers of ten or primes or other similar limits (for example the data type limits are typically limited by some power of two). Famously programmers start counting from 0 (they go as far as using the term "zeroth") while mathematicians rather tend to start at 1. Just as mathematicians have different sets of numbers, programmers have an analogy in numeric data types -- a data type defines a set of values and operations that can be performed with them. The following are some of the common data types and representations of numbers in computers:

- **numeric**: Anything considered a number. In very high level languages there may be just one generic "number" type that can store any kind of number, automatically choosing best representation for it etc.
  - ◆ **unsigned**: Don't allow negative values -- this is sufficient in many cases, simpler to implement and can offer higher range in the positive direction.
  - ◆ **signed**: Allow also negative values which brings up the issue of what representation to use -- nowadays the most common is two's complement.
  - ◆ **fixed size**: Most common, each number takes some fixed size in memory, expressed in bits or bytes -- this of course determines the maximum number of values and so for example the minimum and maximum storable number.
    - ◆ **8bit**: Can store 256 value (e.g. integers from 0 to 255 or -128 to 127).
    - ◆ **16bit**: Can store 65536 values.



- ◊ **32bit**: Can store 4294967296 values.
- ♦ **arbitrary size**: Can store arbitrarily high/low and/or precise value, take variable amount of memory depending on how much is needed, used only in very specialized cases, may be considerably slower.
- ♦ **integer**: Integer values, most common, usually using direct or two's complement representation.
- ♦ **fractional**: Have higher precision than integers, allow storing fractions, are often used to approximate real numbers.
  - ◊ **fixed point**: Are represented by a number with radix point in fixed place, have uniform precision.
  - ◊ **floating point**: Have movable radix point which is more complicated but allows for representing both very high and very small values due to non-uniform precision.
- ♦ **complex**: Analogous to mathematical complex numbers.
- ♦ **quaternion**: Analogous to mathematical quaternions.
- ♦ **symbolic**: Used in some specialized mathematical software to perform symbolic computation, i.e. computation done in a human-like way, by manipulating symbols without using concrete values that would have to resort to approximation.

## Notable Numbers

Here is a table of some notable numbers, mostly important in math and programming but also some famous ones from physics and popular culture (note: the order is roughly from lower numbers to higher ones, however not all of these numbers can be compared easily or at all, so the ordering isn't strictly correct).

| number                      | value                     | equal to                           | notes                                                   |
|-----------------------------|---------------------------|------------------------------------|---------------------------------------------------------|
| minus <u>infinity</u>       |                           |                                    | not always considered a number, smallest possible value |
| minus/negative one          | -1                        | $i^2, j^2, k^2$                    |                                                         |
| <u>"negative zero"</u>      | "-0"                      | zero                               | non-mathematical, sometimes used in programming         |
| <u>zero</u>                 | 0                         | negative zero, $e^{(i * \pi)} + 1$ | "nothing"                                               |
|                             | $4.940656... * 10^{-324}$ |                                    | smallest number storable in IEEE-754 64 binary float    |
|                             | $1.401298... * 10^{-45}$  |                                    | smallest number storable in IEEE-754 32 binary float    |
|                             | $1.616255... * 10^{-35}$  |                                    | Planck length in meters, smallest "length" in Universe  |
| one eight                   | 0.125                     | $2^{-3}$                           |                                                         |
| one fourth                  | 0.25                      | $2^{-2}$                           |                                                         |
| one half                    | 0.5                       | $2^{-1}$                           |                                                         |
| <u>one</u>                  | 1                         | $2^0, 0!, 0.999...$                | NOT a prime                                             |
| <u>square root of two</u>   | 1.414213...               | $2^{(1/2)}$                        | irrational, diagonal of unit square, important in geom. |
| phi ( <u>golden ratio</u> ) | 1.618033...               | $(1 + \sqrt{5}) / 2$               | irrational, visually pleasant ratio, divine proportion  |
| <u>two</u>                  | 2                         | $2^1, 0b000010$                    | prime                                                   |
| <u>e</u> (Euler's number)   | 2.718281...               |                                    | base of natural <u>logarithm</u>                        |
| <u>three</u>                | 3                         | $2^2 - 1$                          | prime, max. unsigned number with 2 bits                 |
| <u>pi</u>                   | 3.141592...               |                                    | circle circumference to its diameter, irrational        |
| <u>four</u>                 | 4                         | $2^2, 0b000100$                    | first composite number                                  |
| <u>five</u>                 | 5                         |                                    |                                                         |

| number                   | value       | equal to                                   | notes                                                                              |
|--------------------------|-------------|--------------------------------------------|------------------------------------------------------------------------------------|
| <u>six</u>               | 6           | $3!, 1 * 2 * 3, 1 + 2 + 3$                 | (twin) prime, number of platonic solids<br>highly composite number, perfect number |
| <u>tau</u>               | 6.283185... | $2 * \pi$                                  | radians in full circle, defined mostly for convenience                             |
| <u>thrembo</u>           | ???         |                                            | the hidden number                                                                  |
| <u>seven</u>             | 7           | $2^3 - 1$                                  | (twin) prime, days in week, max. unsigned n. with 3 bits                           |
| <u>eight</u>             | 8           | $2^3, 0b001000$                            |                                                                                    |
| <u>nine</u>              | 9           |                                            |                                                                                    |
| <u>ten</u>               | 10          | $10^1, 1 + 2 + 3 + 4$                      | your IQ? :D                                                                        |
| twelve, dozen            | 12          | $2 * 2 * 3$                                | highly composite number                                                            |
| fifteen                  | 15          | $2^4 - 1, 0b1111, 0x0f, 1 + 2 + 3 + 4 + 5$ | maximum unsigned number storable with 4 bits                                       |
| <u>sixteen</u>           | 16          | $2^4, 2^{2^2}, 0b010000$                   |                                                                                    |
| twenty four              | 24          | $2 * 2 * 2 * 3, 4!$                        | highly composite number                                                            |
| thirty one               | 31          | $2^5 - 1$                                  | maximum unsigned number storable with 5 bits                                       |
| <u>thirty two</u>        | 32          | $2^5, 0b100000$                            |                                                                                    |
| thirty six               | 36          | $2 * 2 * 3 * 3$                            | highly composite number                                                            |
| <u>fourty two</u>        | 42          |                                            | cringe number, answer to some stuff                                                |
| fourty eight             | 48          | $2^5 + 2^4, 2 * 2 * 2 * 2 * 3$             | highly composite number                                                            |
| sixty three              | 63          | $2^6 - 1$                                  | maximum unsigned number storable with 6 bits                                       |
| <u>sixty four</u>        | 64          | $2^6$                                      |                                                                                    |
| <u>sixty nine</u>        | 69          |                                            | sexual position                                                                    |
| ninety six               | 96          | $2^5 + 2^6$                                |                                                                                    |
| one hundred              | 100         | $10^2$                                     |                                                                                    |
| one hundred twenty one   | 121         | $11^2$                                     |                                                                                    |
| one hundred twenty seven | 127         | $2^7 - 1$                                  | maximum value of signed byte                                                       |
| one hundred twenty eight | 128         | $2^7$                                      |                                                                                    |
| one hundred fourty four  | 144         | $12^2$                                     |                                                                                    |
| one hundred sixty eight  | 168         | $24 * 7$                                   | hours in week                                                                      |
| two hundred fifty five   | 255         | $2^8 - 1, 0b11111111, 0xff$                | maximum value of unsigned <u>byte</u>                                              |
| two hundred fifty six    | 256         | $2^8, 16^2, ((2^2)^2)^2$                   | number of values that can be stored in one byte                                    |
| three hundred sixty      | 360         | $2 * 2 * 2 * 3 * 3 * 5$                    | highly composite number, degrees in full circle                                    |
| four hundred twenty      | 420         |                                            | some stoner shit (they like to smoke it at 4:20)                                   |
| five hundred twelve      | 512         | $2^9$                                      |                                                                                    |

| number                    | value                    | equal to                         | notes                                                    |
|---------------------------|--------------------------|----------------------------------|----------------------------------------------------------|
| six hundred and sixty six | 666                      |                                  | number of the beast                                      |
| one thousand              | 1000                     | $10^3$                           |                                                          |
| one thousand twenty four  | 1024                     | $2^{10}$                         |                                                          |
| two thousand fourty eight | 2048                     | $2^{11}$                         |                                                          |
| four thousand ninety six  | 4096                     | $2^{12}$                         |                                                          |
| ten thousand              | 10000                    | $10^4, 100^2$                    |                                                          |
| ... (enough lol)          | 65535                    | $2^{16} - 1$                     | maximum unsigned number storable with 16 bits            |
|                           | 65536                    | $2^{16}, 256^2, 2^{(2^{(2^2)})}$ | number of values storable with 16 bits                   |
|                           | 80085                    |                                  | looks like BOOBS                                         |
| hundred thousand          | 100000                   | $10^5$                           |                                                          |
| one <u>million</u>        | 1000000                  | $10^6$                           |                                                          |
| one <u>billion</u>        | 1000000000               | $10^9$                           |                                                          |
|                           | 3735928559               | 0xdeadbeef                       | one of famous hexadecimal constants, spells out DEADBEEF |
|                           | 4294967295               | $2^{32} - 1, 0xffffffff$         | maximum unsigned number storable with 32 bits            |
|                           | 4294967296               | $2^{32}$                         | number of values storable with 32 bits                   |
| one trillion              | 1000000000000            | $10^{12}$                        |                                                          |
|                           | 18446744073709551615     | $2^{64} - 1$                     | maximum unsigned number storable with 64 bits            |
|                           | 18446744073709551616     | $2^{64}$                         | number of values storable with 64 bits                   |
|                           | $3.402823... * 10^{38}$  |                                  | largest number storable in IEEE-754 32 binary float      |
| <u>googol</u>             | $10^{100}$               |                                  | often used big number                                    |
|                           | $4.65... * 10^{185}$     |                                  | approx. number of Planck volumes in observable universe  |
|                           | $1.797693... * 10^{308}$ |                                  | largest number storable in IEEE-754 64 binary float      |
| <u>googolplex</u>         | $10^{(10^{100})}$        | $10^{\text{googol}}$             | another large number, number of genders in 21st century  |
| <u>Graham's number</u>    |                          | $g_{64}$                         | extremely, unimaginably large number, > googolplex       |
| TREE(3)                   | unknown                  |                                  | yet even larger number, > Graham's number                |
| <u>infinity</u>           |                          |                                  | not always considered a number, largest possible value   |
| <u>aleph</u> zero         |                          | beth zero, cardinality(N)        | infinite cardinal number, "size" of the set of nat. num. |
| i (imaginary unit)        |                          | $j * k$                          | part of complex numbers and quaternions                  |
| i                         |                          | $k * i$                          | one of quaternion units                                  |
| k                         |                          | $i * j$                          | one of quaternion units                                  |

| number                                  | value | equal to | notes |
|-----------------------------------------|-------|----------|-------|
| TODO: add some p-adic and infinitesimal |       |          |       |
| often_confused                          |       |          |       |

## Often Confused Terms

There are many terms that are very similar and can many times be used interchangeably. This isn't wrong per se, a slight difference may be insignificant in certain contexts. However it's good to know the differences for the sake of those cases where they matter. The following list tries to document some of the often confused/similar terms.

- **AI** vs **machine learning** vs **neural networks**
- **algebra** vs **arithmetic**
- **algorithm** vs **program** vs **process** vs **heuristic**
- **analog** vs **mechanical**
- **anarchy** vs **chaos**
- **argument** vs **parameter**
- **array** vs **list** vs **tuple** vs **set** vs **multiset**
- **ASCII** vs **plain text** vs **Unicode**
- **ASCII art** vs **ANSI art** vs **Unicode art**
- **assembler** vs **assembly** vs **machine code**
- **attribution** vs **credit**
- **binary** vs **executable**
- **binary** vs **boolean**
- **black race** vs **nigger** vs **negro**
- **brute force** vs **heuristic search**
- **bug** vs **glitch** vs **error** vs **exception** vs **fault** vs **failure** vs **defect**
- **causation** vs **correlation**
- **cepstrum** vs **spectrum**
- **chaos** vs **randomness** vs **pseudorandomness** vs **entropy** vs **statistics** vs **probability** vs **stochasticity**
- **class** vs **set**
- **closed source** vs **proprietary**
- **CLI** vs **TUI** vs **terminal** vs **console**
- **color model** vs **color space**
- **communism** vs **Marxism** vs **socialism**
- **complex** vs **complicated**
- **complex number** vs **imaginary number**
- **computer language** vs **programming language**
- **computer science** vs **information technology** vs **informatics** vs **cybernetics** vs **computer engineering** vs **software engineering**
- **concurrency** vs **parallelism** vs **quasiparallelism** vs **distribution**
- **constant** vs **literal**
- **coding** vs **programming**
- **codec** vs **container format**
- **coherence** vs **consistency**
- **computational model** vs **model of computation**
- **convolution** vs **correlation**
- **copyright** vs **patent** vs **trademark** vs **intellectual property** vs **moral right** etc.
- **crossplatform/multiplatform** vs **portable**
- **cryptography** vs **security**
- **data** vs **information** vs **entropy** vs **signal**
- **data structure** vs **data type**
- **decentralized** vs **distributed**
- **declaration** vs **definition**
- **demo** vs **intro**
- **democracy** vs **voting**
- **desktop environment** vs **window manager**

- duck typing vs weak typing vs dynamic typing
- digit vs number vs value vs figure vs numeral
- digital vs electronic
- directed acyclic graph vs tree
- directory vs folder
- discrete Fourier transform vs discrete time Fourier transform
- emoticon vs emoji vs smiley
- emulation vs simulation
- entity vs object
- equation vs expression vs inequality
- equivalence vs implication
- ethics vs morality
- Euler's number vs Euler number
- evolutionary programming vs evolutionary algorithm vs genetic programming vs genetic algorithm
- equality vs identity (in programming languages)
- floating point number vs real number
- font vs typeface
- framework vs library
- free software vs open source vs public domain vs source available vs freeware
- geek vs nerd
- GNU/Linux vs Linux
- gradient noise vs value noise
- hypothesis vs theory vs conjecture
- ID vs token vs hash vs handle vs identifier
- infinite vs arbitrarily large/unbounded
- Internet vs web
- Java vs JavaScript
- kB/mB/gB/tB vs KiB/MiB/GiB/TiB
- latency/ping/lag vs throughput/bandwidth
- leftism vs pseudoleftism
- liberalism vs libertarianism
- license vs waiver
- main memory vs working memory vs RAM
- mass vs weight
- method vs methodology
- modem vs router vs switch
- nationalism vs patriotism
- NP vs NP-hard vs NP-complete
- paging vs virtual memory
- path tracing vs ray tracing vs ray casting
- pointer vs reference
- principal square root vs square root (especially when defining i)
- probability vs probability density
- pseudo vs quasi
- pseudoleft vs left
- pseudoskeptic vs skeptic
- shading vs shadows
- science vs soyence
- Unicode vs UTF
- URI vs URL
- webpage vs website
- wrap around vs overflow
- ...

---

old

# Old

Nowadays old positively correlates with better. For comparison of new and old see [modern tech](#).

---

one

## One

One (1, also a *unit*) is the first positive whole [number](#), signifying the existence of a single unique object we're counting.

Some facts about this number include:

- It is an [odd](#) number.
  - It is a positive number, whole number (integer), [real number](#), [rational number](#) and [complex number](#).
  - It is by convention NOT a [prime number](#), though it is only divisible by 1 and itself.
  - It is a multiplicative identity, i.e.  $1 * x = x$  for any number  $x$ . Also  $x / 1 = x$ ,  $x^1 = x$ ,  $1^x = 1$  for any number  $x$ .
  - In programming there is a very common type of [bug](#) called [off by one](#) in which a boundary is either incorrectly included or excluded.
  - 1 is often a convenient upper bound of many intervals, e.g. when [normalizing](#) numbers or dealing with [probabilities](#).
  - In computing the number and digit 1, as opposed to 0, usually means the *true* or *on* value.
  - In programming operations with 1 (similarly to 0) are very common and may sometimes be handled as special cases to help efficiency, for example adding or subtracting one is called incrementing and decrementing and many [assembly](#) languages have special instructions for this.
- 

oop

## Object-Oriented Programming

*"I invented the term 'object oriented' and [C++](#) was not what I had in mind" --[Alan Kay](#), inventor of OOP*

Object-oriented programming (OOP, also object-obsessed programming, objectfused programming or artificial inelegance) is a [programming paradigm](#) that tries to model reality as a collection of abstract objects that communicate with each other and obey some specific rules. While the idea itself isn't bad and can be useful in certain cases and while pure OOP in very old languages like [Smalltalk](#) may have even been quite elegant, by later adoption by [capitalist businesses](#) the concept has been extremely twisted and degenerated to unbelievable levels -- **OOP has become extremely overused, extremely badly implemented and downright forced in programming languages** that nowadays try to apply this [abstraction](#) to every single program and concept, creating [anti-patterns](#), unnecessary issues and of course greatly significant amounts of [bloat](#). We therefore see the OOP of today as a **cancer of programming**. OOP was basically a software development fashion wave that scarred the industry for decades, it has poisoned minds of several generations. Nowadays despite OOP still keeping many fans the critical stance towards it isn't even controversial anymore, many others have voiced the criticism over and over, usually the most competent programmers like [Richard Stallman](#) and [Linux Torvalds](#) and groups like [suckless](#) and [bitreich](#). Ugly examples of OOP gone bad include [Java](#) and [C++](#) (which at least doesn't force it). Other languages such as [Python](#) and [JavaScript](#) include OOP but have lightened it up a bit and at least allow you to avoid using it. You should probably learn OOP but only to see why it's bad (and to actually understand 99% of code written nowadays).

**A real life analogy** to give a bit of high level overview: the original [Smalltalk](#) style OOP was kind of like when society invented [democracy](#) -- a simple idea which everyone understands (we are 10 cavemen, let's just vote on stuff mkay?) that's many times useful and works well, e.g. on a scale of a village or a small city. Then cities grew bigger (just as software did), into states and empires and the idea kept getting more and more complicated -- people just wanted to keep the democracy, apply it to everything and scale it indefinitely, but for that they had to add more complexity, they implemented representatives, parliaments, senates, presidents, vicepresidents, ministers, judges, more and more bureaucracy, hybrid ideas (free

market, controlled economy, ...), corruption and inefficiencies crept in, the system degenerated into what we have today -- a hugely expensive paperworking machine that's exploited and hacked, with laws so complicated no one really understands them, with magic, randomness and unpredictability, producing just waste and bullshit, crumbling under own weight. This is also the way OOP went -- they started inventing static classes/methods, abstract classes/methods, multiple inheritances, interfaces, design patterns, overriding, hybrid paradigms and so on until we ended up with ugly abominations on which today's technology stands. Now a few things have to be noted. Firstly these abominations are a disaster, they came from our mistake of taking the original simple idea (simple small scale voting democracy) and saying "let's make this the only thing in the world and let's scale it a million times!" Such idea is stupid from the start and there is no doubt about that. However another evil is that people are taught to do everything this way -- today's programmers will use the mainstream OOP everywhere, even in simple programs, they don't even think about if they should, they are simply taught "always use this". This is like in real life wanting to govern a family by having elections each year to vote for the head of the family, then having members of family vote for other members of the family to be their representatives that will talk for them (the same kind of craziness as wanting to strictly respect encapsulation even in trivial programs), then if someone wants to buy anything he has to ask for a budget several months in advance and have others vote on it while an elected anti corruption committee is watching etcetc. This kind of insanity is what's normal in software nowadays. Now the only sane discussion can be had only about the usefulness and scope of the original, simple idea (simple voting in small groups, simple pure OOP) and here we say that it may be good, but only applied to just some specific situations, i.e. we say simple OOP is good for some problems but not all, just like voting is a good solution to some problems (e.g. a group of friends deciding where to go party), but not all (e.g. passengers in a car voting on which way to steer and which pedals to press).

## Principles

Bear in mind that OOP doesn't have a single, crystal clear definition. It takes many forms and mutations depending on language and it is practically always combined with other paradigms such as the imperative paradigm, so things may be fuzzy.

Generally OOP programs solve problems by having **objects** that communicate with each other. Every object is specialized to do some thing, e.g. one handles drawing text, another one handles caching, another one handles rendering of pictures etc. Every object has its **data** (e.g. a human object has weight, race etc.) and **methods** (object's own functions, e.g. human may provide methods getHeight, drinkBeer or petCat). Objects may send **messages** to each other: e.g. a human object sends a message to another human object to get his name (in practice this means the first object calls a method of the other object just like we call functions, e.g.: `human2.getName()`).

Now many OO languages use so called **class OOP**. In these we define object classes, similarly to defining data types. A class is a "template" for an object, it defines methods and types of data to hold. Any object we then create is then created based on some class (e.g. we create the object `alice` and `bob` of class `Human`, just as normally we create a variable `x` of type `int`). We say an object is an **instance** of a class, i.e. object is a real manifestation of what a class describes, with specific data etc.

The more "lightweight" type of OOP is called **classless OOP** which is usually based on having so called prototype objects instead of classes. In these languages we can simply create objects without classes and then assign them properties and methods dynamically at runtime. Here instead of creating a `Human` class we rather create a prototype object that serves as a template for other objects. To create specific humans we clone the prototype human and modify the clone.

OOP furthermore comes with some basic principles such as:

- **encapsulation**: Object should NOT be able to access other object's data directly -- they may only use their methods. For example an object shouldn't be able to access the height attribute of a `Human` object, it should be able to access it only via methods of that object such as `getHeight`. (This leads to the setter/getter antipattern).
- **polymorphism**: Different objects (e.g. of different classes) may have methods with the same name which behave differently for either object and we may just call that method without caring what kind of object that is (the correct implementation gets chosen at runtime). E.g. objects of both `Human` and `Bomb` classes may have a method `setOnFire`, which with the former will kill the human and with the latter will cause an explosion killing many humans. This is good e.g. in a case when we have an array

of GUI components and want to perform e.g. resize on every one of them: we simply iterate over the whole array and call the method resize on each object without caring whether the object is a button, checkbox or a window.

- **inheritance**: In class OOP classes form a hierarchy in which parent classes can have child classes, e.g. a class LivingBeing will have Human and Animal subclasses. Subclasses inherit stuff from the parent class and may add some more. However this leads to other antipatterns such as the diamond problem. Inheritance is nowadays regarded as bad even by normies and is being replaced by composition.

## Why It's Shit

- OOP is just a bad abstraction for many problems that by their nature aren't object-oriented. OOP is not a silver bullet, yet it tries to behave as one. **The greatest issue of OOP is that it's trying to solve everything.** For example it forces the idea that data and algorithms should always come together, but that's simply a stupid statement in general, there is no justification for it, some data is simply data and some algorithms are simply algorithms. You may ask what else to use instead of OOP then -- see the section below.
- For simple programs (which most programs should be) such as many Unix utilities OOP is simply completely unnecessary.
- OOP languages make you battle artificial restrictions rather than focus on solving the problem at hand.
- Great number of the supposed "features" and design patterns (setters/getters, singletons, inheritance, ...) turned out to actually be antipatterns and burdens -- this isn't a controversial statement, even OOP proponents usually agree with this.
- OOP as any higher abstraction very often comes with overhead, memory footprints and performance loss (bloat) as well as more complex compilers, language specifications, more dependencies, magic etc.
- The relatively elegant idea of pure OOP didn't catch on and the practically used OOP languages are abomination hybrids of imperative and OOP paradigms that just take more head space, create friction and unnecessary issues to solve. Sane languages now allow the choice to use OOP fully, partially or avoid it completely, which leads to a two-in-one overcomplication.
- The naive idea of OOP that the real world is composed of nicely defined objects such as Humans and Trees also showed to be completely off, we instead see shit like AbstractIntVisitorShitFactory etc. Everyone who ever tried to make some kind of categorization knows it's usually asking for trouble, categories greatly overlap, have unclear borders, multiple parents etcetc.
- The idea that OOP would lead to code reusability also completely failed, it's simply not the case at all, implementation code of specific classes is typically burdened with internal and external dependencies just like any other bloated code. OOPer believed that their paradigm would create a world full of reusable blackboxes, but that wasn't the case, OOP is neither necessary for blackboxing, nor has the practice shown it would contribute to it -- quite on the contrary, e.g. simple imperative header-only C libraries are much more reusable than those we find in the OOP world.
- Good programmers don't need OOP because they know how to program -- OOP doesn't invent anything, it is merely a way of trying to **force** good programming mostly on incompetent programmers hired in companies, to prevent them from doing damage. However this of course doesn't work, a shit programmer will always program shit, he will find his way to fuck up despite any obstacles and if you invent obstacles good enough for stopping him from fucking up, you'll also stop him from being able to program something that works well as you tie his hands. Yes, good programmers write shit buggy code too, but that's more of a symptom of bad, overcomplicated bloated capitalist design of technology that's just asking for bugs and errors -- here OOP is trying to cure symptoms of an inherently wrong direction, it is not addressing the root cause.
- OOP just mostly repeats what other things like modules already do.
- If you want to program in object-oriented way and have a good justification for it, **you don't need an OOP language anyway**, you can emulate all aspects of OOP in simple languages like C. So instead of building the idea into the language itself and dragging it along forever and everywhere, it would be better to have optional OOP libraries.
- It generalizes and simplifies programming into a few rules of thumb such as encapsulation, again for the sake of inexperienced noobs. However there are no simple rules for how to program well, good programming requires a huge amount of experience and as in any art, good programmer knows when breaking the general rules is good. OOP doesn't let good programmers do this, it preaches things like "global variables bad" which is just too oversimplified and hurts good programming.



# Pure OOP (The "Legit" But Unused Kind Of OOP)

TODO

Similarly to how functional languages are based on some very simple mathematical system such as lambda calculus, pure object oriented languages have a similar thing, most notably the sigma calculus (defined in the paper called *A Theory Of Primitive Objects* by Abadi and Cardelli).

## So Which Paradigm To Use Instead Of OOP?

After many people realized OOP is kind of shit, there has been a boom of "OOP alternatives" such as functional, traits, agent oriented programming, all kinds of "lightweight"/optional OOP etc etc. Which one to use?

In short: NONE, **by default use the imperative paradigm** (also here many times interchangeably called "procedural"). Remember this isn't to say you shouldn't ever apply a different paradigm, but imperative should be the default, most prevalent and suitable one to use in solving most problems. There is nothing new to invent or "beat" OOP.

But why imperative? Why can't we simply improve OOP or come up with something ultra genius to replace it with? Why do we say OOP is bad because it's forced and now we are forcing imperative paradigm? The answer is that the **imperative paradigm is special because it is how computers actually work**, it is not made up but rather it's the **natural low level paradigm with minimum abstraction that reflects the underlying nature of computers**. You may say this is just bullshit arbitrary rationalization but no, these properties makes imperative paradigm special among all other paradigms because:

- Its implementation is simple and suckless/LRS because it maps nicely and naturally to the underlying hardware -- basically commands in a language simply translate to one or more instructions. This makes construction of compilers easy.
- It's predictable and efficient, i.e. a programmer writing imperative code can see quite clearly how what he's writing will translate to the assembly instructions. This makes it possible to write highly efficient code, unlike high level paradigms that perform huge amounts of magic for translating foreign concepts to machine instructions -- and of course this magic may differ between compilers, i.e. what's efficient code in one compiler may be inefficient in another (similar situation arose e.g. in the world of OpenGL where driver implementation started to play a huge role and which led to the creation of a more low level API Vulkan).
- It doesn't force high amounts of unnecessary high level abstraction. This means we MAY use any abstraction, even OOP, if we currently need it, e.g. via a library, but we aren't FORCED to use a weird high level concepts on problems that can't be described easily in terms of those concepts. That is if you're solving a non-OOP problem with OOP, you waste effort on translating that problem to OOP and the compiler then wastes another effort on un-OOPing this to translate this to instructions. With imperative paradigm this can't happen because you're basically writing instructions which has to happen either way.
- It is generally true that the higher the abstraction, the smaller its scope of application should be, so the default abstraction (paradigm) should be low level. This works e.g. in science: psychology is a high level abstraction but can only be applied to study human behavior, while quantum physics is a low level abstraction which applies to the whole universe.

Once computers start fundamentally working on a different paradigm, e.g. functional -- which BTW might happen with new types of computers such as quantum ones -- we may switch to that paradigm as the default, but until then imperative is the way to go.

## History

TODO

openai

# "Open" AI

OpenAI ("open" as in "not open") is a hugely harmful proprietary/closed-source for profit corporation researching and creating proprietary AI, known especially for practicing unethical capitalist deception techniques such as heavy openwashing (using the word *open* to make people think it is has something to do with "open source") and non-profit-washing (hiding behind a similarly named non-profit so that it seems the corporation is actually non-profit).

---

openarena

## OpenArena

OpenArena (OA) is a first person pew pew arena shooter game, a free as in freedom clone of the famous game Quake 3. It runs on GNU/Linux, Winshit, BSD and other systems, it is quite light on system resources but does require a GPU acceleration (no software rendering). Quake 3 engine (Id tech 3) has retroactively been made free software: OpenArena forked it and additionally replaced the proprietary Quake assets, such as 3D models, sounds and maps, with community-created Japanese/nerd/waifu-themed free culture assets, so as to create a completely free game. OpenArena plays almost exactly the same as Quake 3, it basically just looks different and has different maps. It has an official wiki at <https://openarena.fandom.com> and a forum at <http://www.openarena.ws/board/>. OpenArena has also been used as a research tool.

As of 2023 most players you encounter in OA are cheating (many americans play it), but if you don't mind that, it's a pretty comfy game.

As of version 0.8.8 there are 45 maps and 12 game modes (deathmatch, team deatchmatch, capture the flag, last man standing, ...).

A bit of fun: you can redirect the chat to text to speech and let it be read aloud e.g. like this:

```
openarena 2>&l | grep --line-buffered ".*^7: ^2.*" | sed -u "s/.*: ^2\(.*\)/\1/g" | sed -u "s/\^./g" | espeak
```

Character art exhibits what SJWs would call high sexism. This is great, it's the good old 90s art style. The art is very nice and professional looking (no programmer art). Characters such as Angelyss are basically naked with just bikini strings covering her nipples. Other characters like Merman and Penguin (a typical "Linux user") are pretty funny. Ratmod has very nice taunts that would definitely be labeled offensive to gays and other minorities nowadays. The community is also pretty nice, free speech is allowed in chat, no codes of conduct anywhere, boomers thankfully don't buy into such bullshit. Very refreshing in the politically correct era.

OpenArena is similar to e.g. Xonotic -- another free arena shooter -- but is a bit simpler and oldschool, both in graphics, features and gameplay. It has fewer weapons, game modes and options. However there exist additional modifications, most notably the Ratmod (or RatArena) which makes it a bit more "advanced" (adds game modes, projectiles go through portals, improved prediction code etc.). As of 2022 an asset reboot in a more Anime style, called OA3, is planned and it seems to be aiming in the right direction -- instead of making a "modern HD game" (and so basically just remake Xonotic) they specifically set to create a *2000 game* (i.e. keep the models low poly etc.). It could also help distinguish OpenArena from Quake more and so make it legally safer (e.g. in terms of trade dress).

{ I've been casually playing OA for a while alongside Xonotic. I love both games. OA is more oldschool, boomer, straightforward and KISS, feels slower in terms of movement and combat but DM matches are much quicker, fragging is very rapid. ~drummyfish }

The project was established on August 19 2005, one day after the Quake 3 engine was freed.

## See Also

- Freedoom
- Xonotic

open\_console

## Open Console

{ Open consoles are how I got into [suckless](#) programming, they taught me about the low-level, optimizations and how to actually program efficiently on very limited hardware. I recommend you grab one of these.  
~drummyfish }

Open consoles (also indie handhelds etc.) are tiny [GameBoy](#)-like [gaming](#) consoles mostly powered by [free software](#) and [free hardware](#), which have relatively recently (some time after 2015) seen a small boom. Examples include [Arduboy](#), [Pokitto](#) or [Gamebuino](#). These are **NOT** to be confused with the [Raspberry Pi](#) (and similar) handhelds that run GameBoy/PS1/DOS [emulators](#) (though some open consoles may use e.g. the RP2040 Raspberry pi processor) but rather custom, mostly [FOSS](#) platforms running mostly their own community made [homebrew](#) games. Open consoles are also similar to the old consoles/computers such as [NES](#), [GameBoy](#) etc., however again there is a difference in being more indie, released more recently and being "open", directly made for tinkering, so it's e.g. much easier to program them (old consoles/computers very often require some unofficial hacks, obscure libraries, gcc patches etc. to just get your code working).

In summary, open consoles are:

- **GameBoy-like gaming consoles** (but also allow and encourage non-gaming uses).
- Powered by **free hardware and free software** (usually [Arduino](#) plus a custom library, although mostly advertised as [open source](#) and not so strict about freedom). Schematics are a lot of times available.
- **Retro**.
- **Indie** (sometimes developed by a single guy), often [crowd-funded](#).
- **Educational**.
- **DIY**, sometimes leaving assembly of the kit to the customer (assembled kits can usually be ordered for extra price).
- **Very cheap** (compared to proprietary mainstream consoles).
- **Hacking friendly**.
- Typically **embedded ARM**.
- **Bare metal** (no operating system).
- Pretty **low spec** hardware ([RAM](#) amount in kilobytes, CPU frequency in MHz).
- Relying on **user created games** which are many times also free-licensed.

Recommended consoles for starters are [Arduboy](#) and [Pokitto](#) which are not only very well designed, but most importantly have actual friendly active communities.

These nice little toys are great because they are anti-[modern](#), [simple](#), out of the toxic mainstream, like the oldschool bullshit-free computers. This supports (and by the low specs kind of "forces") [suckless](#) programming and brings the programmer the joy of programming (no headaches of resizable windows, multithreading etc., just plain programming of simple things with direct access to hardware). They offer an alternative [ISA](#), a non-x86 platform without botnet and [bloat](#) usable for any purpose, not just games. Besides that, this hobby teaches low level, efficiency-focused programming skills.

**Watch out** (2024 update): having been successful on the market, the world of open consoles is now flooded by corporations and [SJWs](#) bringing in the [toxicity](#), they are going to go to shit very soon, get the old ones while you still can. New consoles already try to employ web-only IDEs in micropython, they're websites are full of suicide inducing diversity propaganda and unusable on computers with less than 1 TB of RAM.

## Programming

Open consoles can typically be programmed without proprietary software (though officially they may promote something involving proprietary software), GNU/[Linux](#) mostly works just fine (sometimes it requires a bit of extra work but not much). Most of the consoles are [Arduino](#)-based so the Arduino IDE is the official development tool with [C++](#) as a language ([C](#) being thankfully an option as well). The IDE is "open-source"

but also bloat; thankfully CLI development workflow can be set up without greater issues (Arduino comes with CLI tools and for other platforms gcc cross-compiler can be used) so comfy programming with vim is nicely possible.

If normies can do it, you can do it too.

Some consoles (e.g. Arduboy, Pokitto and Gamebuino META) have their own emulators which make the development much easier... or rather bearable. Without an emulator you're forced to constantly reupload the program to the real hardware which is a pain, so you want to either use a nice LRS library such as SAF or write your game to be platform-independent and just make it run on your development PC as well as on the console (just abstract the I/O and use SDL for the PC and the console's library for the console -- see how Anarch does it).

## Open Console List

Some notable open consoles (which fit the definition at least loosely) are listed here. Symbol meaning:

- A = Arduino
- C = great active community
- \* = recommended
- + = many games/programs
- - = discontinued

| name                     | CPU         | RAM (K) | ROM (K) | display  | year | notes                              |
|--------------------------|-------------|---------|---------|----------|------|------------------------------------|
| <u>Arduboy</u>           | 8b 16 MHz   | 2.5     | 32      | 64x32 1b | 2015 | * A C +, tiny                      |
| <u>Gamebuino</u>         | 8b 16 MHz   | 2       | 32      | 84x48 1b | 2014 | + A -, SD                          |
| <u>Pokitto</u>           | 32b 48 MHz  | 36      | 256     | 220x176  | 2018 | * C +, ext. hats, SD               |
| <u>ESPboy</u>            | 32b 160 MHz | 80      | 4000    | 128x128  | 2019 | A                                  |
| <u>GB META</u>           | 32b 48 MHz  | 32      | 256     | 168x120  | 2018 | A + -, SD                          |
| <u>Nibble</u>            | 32b 160 MHz | 80      | 4000    | 128x128  | 2021 | A, AAA bat.                        |
| <u>UzeBox</u>            | 8b 28 MHz   | 4       | 64      | 360x240  | 2008 | C, +                               |
| <u>Tiny Arcade</u>       | 32b         |         |         |          |      | A                                  |
| <u>Thumby</u>            | 32b 133 MHz | 264     | 2000    | 72x40 1b | 2022 | RPI (RP2040), mainly web editor :( |
| <u>Pocket Arcade</u>     |             |         |         |          |      |                                    |
| Ringo/ <u>MakerPhone</u> | 32b 160 MHz | 520     | 4000    | 160x128  | 2018 | A -, phone, SD                     |
| <u>Agon</u>              | 8b 18 MHz   | 512     |         | 640x480  |      |                                    |

TODO: BBC micro:bit, Vircon32 (fantasy console implementable in HW, not sure about license), Retro Game Tiny, Adafruit PyGamer, ... see also <https://github.com/ESPboy-edu/awesome-indie-handhelds>

## See Also

- programmable calculator
- fantasy console

---

open\_source

## Open \$ource

"Micro\$oft <3 open \$ource"

Open source (OS, also *Open \$ource*) is a capitalist movement, in recent years degraded to a mere brand, forked from the free software movement; it is advocating at least partial "openness", i.e. strategic sharing of design parts with the public and allowing unpaid volunteer contributors from the public to take part in software and hardware development; though technically and legally the definition of *open source* is mostly

identical to free (as in freedom) software, in practice and in spirit it couldn't be more different as for **abandoning the goal of freedom and ethics in favor of business** (to which ethics is an obstacle), due to which we see open source as inherently evil and recommend following the free software way instead. Richard Stallman, the founder of free software, distances himself from the open source movement. Fascist organizations such as Microsoft and Google, on the other hand, embrace open source (while restraining from using the term *free software*) and slowly shape it towards their goals. Open source is a short for "yes, it will abuse you, but at least you can read its source code." The term FOSS is sometimes used to refer to both free software and open source without expressing any preference.

Open source unfortunately (but unsurprisingly) became absolutely prevalent over free software as it better serves capitalism and abuse of people, and its followers are more and more hostile towards the free software movement. This is very dangerous, ethics and focus on actual user freedom is replaced by shallow legal definitions that can be bypassed, e.g. by capitalist software and bloat monopoly. In a way open source is capitalism reshaping free software so as to weaken it and eventually make its principles of freedom ineffective. Open source tries to shift the goal posts: more and more it offers only an illusion of some kind of ethics and/or freedom, it pushes towards mere partial openness ("open source" for proprietary platforms), towards high complexity, inclusion of unethical business-centered features (autoupdates, DRM, ...), high interdependency, difficulty of utilizing the rights granted by the license, exclusion of developers with "incorrect" political opinions or bad brand image etc. In practice open source has become something akin a mere **brand** which is stick to a piece of software to give users with little insight a feeling they're buying into something good -- this is called **openwashing**. This claim is greatly supported by the fact that corporations such as Microsoft and Google widely embrace open source ("Microsoft <3 open source", the infamous GitHub acquisition etc.).

**"Free and Open Source: it is completely *FREE OF COST* and *ALMOST ALL* of its components are open source."** --GNU/Linux Mint's website already marketing partially proprietary system as "open source" and purposefully misusing the word "free" to mean "gratis" (February 2024)

{ Mint also hilariously markets itself as KISS lol. My friend suggested they only implemented the "stupid" part of it :-} ~drummyfish }

One great difference of open source with respect to free software is that **open source doesn't mind proprietary dependencies and only "partially open" projects** (see also open core): Windows only programs or games in proprietary engines such as Unity are happily called open source -- this would be impossible in the context of free software because as Richard Stallman says software can only be free if it is free as a whole, it takes a single proprietary line of code to allow abuse of the user. The "open source" communities nowadays absolutely **don't care a bit about freedom or ethics** (the majority of open source supporting zoomers most likely don't even know there was ever any connection), many "open source" proponents even react aggressively to bringing the idea of ethics up. "Open source" communities use locked, abusive proprietary platforms such as Discord, Google cloud documents and Micro\$oft's GitHub to create software and collaborate -- users without Discord and/or GitHub account often aren't even offered a way to contribute, report bugs or ask for support. There are many "open source" projects that are just meant to be part of a mostly proprietary environment, for example the Mangos implementation of World of Warcraft server, which of course has to be used with the proprietary WoW client and with proprietary server assets, which gives Blizzard (the owner of WoW) complete legal control over any server running on such an "open source" server (such servers always only rely on Blizzard temporarily TOLERATING their small noncommercial communities, despite Blizzard having taken some of them down with legal action) -- calling such a project "free software" in this context would just sound laughable, so they rather call it "open source", i.e. "no, there is no freedom, but the source is technically open". Lately you will even see more and more people just calling any software/project "open" as long as some part of its source code is available for viewing on GitHub, no matter the license or any other considerations (see e.g. "open"geofiction etc.).

The open source definition is maintained by the Open Source Initiative (OSI) -- they define what exactly classifies as open source and which licenses are compatible with it. These licenses are mostly the same as those approved by the FSF (even though not 100%). The open source definition is a bit more complex than that of free software, in a nutshell it goes along the lines:

1. The license has to allow **free redistribution** of the software without any fees.
2. **Source code must be freely available**, without any obfuscation.

3. **Modification of the software must be allowed** as well as redistribution of these modified versions under the same terms as the original.
4. **Direct modification may be forbidden only if patches are allowed.**
5. **The license must not discriminate against people**, everyone has to be given the same rights.
6. **The license must not discriminate against specific uses**, i.e. use for any purpose must be allowed.
7. **The license applies automatically** to everyone who receives the software with the license.
8. **The license must apply generally**, it cannot be e.g. limited to the case when the software is part of some larger package.
9. **The license must not restrict other software**, i.e. it cannot for example be forbidden to run the software alongside some other piece of software.
10. **The license must be technology neutral**, i.e. it cannot for example limit the software to certain platform or API.

Open source furthermore greatly fails for example by not accepting CC0 as a valid license and not accepting esoteric programming languages (because they're "obfuscated"). All in all, avoid open source, support free software.

## See Also

- openwashing
- free software
- open core
- source available
- license

---

operating\_system

## Operating System

Operating System (OS) is usually a quite complex program that's typically installed on a computer before any other user program and serves as a platform for running other programs as well as handling low level functions, managing resources (CPU usage, RAM, files, network, ...) and offering services, protection and interfaces for humans and programs. If computer was a city, an OS is its center that was built first and where its government resides. As with most things, the definition of an OS can differ and be stretched greatly -- while a typical OS will include features such as graphical interface with windows and mouse cursor, file system, multitasking, networking, audio system, safety mechanisms or user accounts, there exist OSes that work without any said feature. Though common on mainstream computers, operating system isn't necessary; it may be replaced by a much simpler program (something akin a program loader, BIOS etc.) or even be absent altogether -- programs that run without operating system are called "bare metal" programs (these can be encountered on many simple computers such as embedded devices).

There is a nice CC0 wiki for OS development at <https://wiki.osdev.org/>.

From programmer's point of view a serious OS is one of the most difficult pieces of software one can pursue to develop. The task involves an enormous amount of low-level programming, development of own tools from scratch and requires deep and detailed knowledge of all components of a computer, of established standards as well as many theoretical subjects such as compiler design.

**Which OS is the best?** Currently there seems to be almost no good operating system in existence, except perhaps for Collapse OS and Dusk OS which may be the closest to LRS at the moment, but aren't widely used yet and don't have many programs running on them. Besides this there are quite a few relatively usable OSes, mostly Unix like systems. For example OpenBSD seems to be one of them, however it is proprietary (yes, it contains some code without license) and too obsessed with MUH SECURITY, and still a bit overcomplicated. HyperbolaBSD at least tries to address the freedom issue of OpenBSD but suffers from many others. Devuan is pretty usable, just works and is alright in not being an absolute apeshit of consommerist bloat. FreeDOS seemed nice too: though it's not Unix like, it is much more KISS than Unices, but it will probably only work on x86 systems.

An OS, as a software, consists of two main parts:

- **kernel**: The base/core of the system, running in the most privileged mode, offering an environment for user applications.
- **userland (applications)**: The set of programs running on top of the kernel. These run in lower-privileged mode and use the services offered by the kernel via so called system calls.

For example in GNU/Linux, Linux is the kernel and GNU is the userland.

## Attributes/Features

TODO

## Notable Operating Systems

Below are some of the most notable OSes.

{ Some more can be found here: <https://wiki.osdev.org/Projects>. ~drummyfish }

- Android: extremely badly designed malicious system
- BSD systems such as OpenBSD and freeBSD: Unix-like OSes
- Collapse OS: finished, extremely minimalist OS that will help us survive the collapse
- DuskOS: kind of continuation of Collapse OS aiming for more comfort
- DOS
- FreeDOS
- GNU/Linux very popular systems existing in many different distributions, some completely free
- Haiku
- HyperbolaBSD
- Inferno: OS in the style of Plan 9
- MacOS
- Minix
- Plan 9: research OS, continuing the ideas of Unix
- PostmarketOS
- ReactOS
- Replicant
- Solaris
- TempleOS: simple meme OS written by a Terry Davis
- Unix
- Windows: very bad proprietary capitalist OS
- 9front: OS based on Plan 9
- ...

## LRS Operating System

What would an operating system designed by LRS principles look like? There may be many different ways to approach this challenge. Multiple operating systems (or multiple versions of the same system) may be made, such as as an "extremely KISS bare minimum featureless system", a "more advanced but still KISS system", a "special-purpose safe system for critical uses" etc. The following is a discussion of ideas we might employ in designing such systems.

The basic idea for a universal LRS operating system is to be something more akin a mere **text shell** (possibly comun shell), we wouldn't probably even call it an operating system. A rough vision is something like "**DOS plus a bit of Unix philosophy**"; we may also imagine it like GRUB or something similar really. The system would probably seem primitive by "modern standards", but in a good society it would be sufficient as a universal operating system (i.e. not necessarily suitable for ALL purposes). The OS would in fact be more of a **program loader** (like e.g. the one seen in Pokitto), running with the same privileges as other programs -- its purpose would NOT be to provide a safe environment for programs to run in, to protect user's data and possibly not even to offer a platform for programs to run on (for abstracting hardware away a non-OS library might be used instead), but rather to allow switching between different programs on a

computer without having to reupload the programs externally, and to provide basic tools for managing the computer itself (such as browsing files, testing hardware etc.). This shell would basically allow to browse files, load them as programs, and maybe run simple scripts (e.g. in mentioned comun language), allowing things such as automatization of running several program (NOT in parallel but rather one by one) to collaborate on computing something.

An idea worth mentioning is also the possibility to have a have a distribution of this "operating system" that works completely without a file system, i.e. something akin a "big program" that has all the tools compiled into it, without the possibility to install or uninstall programs. Of course this doesn't mean ALL operating systems would in the world would work like this, it would just be a possibility for those that could benefit from it, e.g. very small wrist watch computers that don't wouldn't want and need to include hardware and software required for a mutable filesystem to work, since all they would need would be a few tools like stopwatch and calculator, plus they would gain the advantage of loading a program instantly. The tools to be "compiled in" could be chosen by the user before compilation to make a personalized "immutable distro".

Let's keep in mind that true LRS computers would be different from the current capitalist ones -- an operating system would only be optional, programs would be able to run on bare metal as well as under an OS, and operating systems would be as much compatible as possible. By this an OS might be seen as more of an extra tool rather than a platform.

The system might likely lack features one would nowadays call essential for an OS, such as multiple user support (no need if security is not an issue, everyone can simply make his own subdirectory for personal files), virtual memory, complex GUI etc. There might even be no multitasking; a possibility to make a multitasking OS exists, but let's keep in mind that even such things as programs interacting via pipes may be implemented without it (using temporary buffer files into which one program's output is stored before running the next program).

The universal OS would assume well behaved programs, as programs would likely be given full control over the computer when run -- this would greatly simplify the system and also computing in general. Doing so would be possible thanks to non-existence of malicious programs (as in good society there would be no need for them) and elimination of update culture. Users would only install a few programs they choose carefully -- programs that have been greatly tested and don't need to be updated.

**On user interface:** the basic interaction mode would of course be the text interface. Programs would have the option to switch to a graphical mode in which they would be able to draw to screen. There would be no such bloat as window managers or desktop environments -- these are capitalist inventions that aren't really needed as users practically always interacts with just one program at a time. Even in a multitasking system only one program would be drawing to the screen at a time, with user having the option to "alt-tab" between them. This would also simplify programs greatly as they wouldn't have to handle bullshit such as dynamically resizing and rearranging their window content. If someone REALLY wanted to have two programs at the screen at the same time, something akin a "screen splitter" might be made to create two virtual screens on one physical screen.

**A bit more details:** the universal OS could simply be a program that gets executed after computer restart. This program would offer a shell (textual, graphical, ...) that would allow inspecting the computer, configuring it, and mainly running other programs. Once the user chose to run some program, the OS would load the program to memory and jump to executing it. To get back to the OS the program could hand back control to the OS, or the computer could simply be restarted. If the program crashes, the computer simply restarts back to OS.

TODO: more

---

optimization

## Optimization

Optimization means making a program more efficient in terms of consumption of some computing resource or by any similar metric, commonly aiming for greater execution speed or lower memory usage (but also e.g. lower power consumption, lower network usage etc.) while preserving how the program functions externally;



this can be done manually (by rewriting parts of your program) or automatically (typically by compiler when it's translating your program). Unlike refactoring, which aims primarily for a better readability of source code, optimization changes the inner behavior of the executed program to a more optimal one. Apart from optimizing programs/algorithms we may also more widely talk about optimizing e.g. data structures, file formats, hardware, protocol and so on.

## Manual Optimization

These are optimizations you do yourself by writing better code.

### General Tips'N'Tricks

These are mainly for C, but may be usable in other languages as well.

- **Tell your compiler to actually optimize** (-O3, -Os flags etc.). Also check out further compiler flags that may help you turn off unnecessary things you don't need, AND try out different compilers, some may just produce better code. If you are brave also check even more aggressive flags like -Ofast and -Oz, which may be even faster than -O3, but may break your program too.
- **gprof is a utility you can use to profile your code.**
- **<stdint.h> has fast type nicknames**, types such as uint\_fast32\_t which picks the fastest type of at least given width on given platform.
- **Actually measure the performance** to see if your optimizations work or not. Sometimes things behave counterintuitively and you end up making your program perform worse by trying to optimize it! Also make sure that you MEASURE THE PERFORMANCE CORRECTLY, many beginners for example just try to measure run time of a single simple function call which doesn't really work, you want to try to measure something like a million of such function calls in a loop and then average the time.
- **Keywords such as inline, static, const and register can help compiler optimize well.**
- **Optimize the bottlenecks!** Optimizing in the wrong place is a complete waste of time. If you're optimizing a part of code that's taking 1% of your program's run time, you will never speed up your program by more than that 1% even if you speed up the specific part by 10000%. Bottlenecks are usually inner-most loops of the main program loop, you can identify them with profiling. Generally initialization code that runs only once in a long time doesn't need much optimization -- no one is going to care if a program starts up 1 millisecond faster (but of course in special cases such as launching many processes this may start to matter).
- **You can almost always trade space (memory usage) for time (CPU demand) and vice versa** and you can also fine-tune this. You typically gain speed by precomputation (look up tables, more demanding on memory) and memory with compression (more demanding on CPU).
- **Static things are faster and smaller than dynamic things.** This means that things that are somehow fixed/unchangeable are better in terms of performance (and usually also safer and better testable) than things that are allowed to change during run time -- for example calling a function directly (e.g. myVar = myFunc();) is both faster and requires fewer instructions than calling a function by pointer (e.g. myVar = myFuncPointer();) the latter is more flexible but for the price of performance, so if you don't need flexibility (dynamic behavior), use static behavior. This also applies to using constants (faster/smaller) vs variables, static vs dynamic typing, normal vs dynamic arrays etc.
- **Be smart, use math**, for example simplify expressions using known formulas, minimize logic circuits etc. Example: let's say you want to compute the radius of a zero-centered bounding sphere of an  $N$ -point point cloud. Naively you might be computing the Euclidean distance ( $\sqrt{x^2 + y^2 + z^2}$ ) to each point and taking a maximum of them, however you can just find the maximum of squared distances ( $x^2 + y^2 + z^2$ ) and return a square root of that maximum. This saves you a computation of  $N - 1$  square roots.
- **Learn about dynamic programming.**
- **Avoid branches (ifs)** if you can (remember ternary operators, loop conditions etc. are branches as well). They break prediction in CPU pipelines and instruction preloading and are often source of great performance losses. Don't forget that you can many times compare and use the result of operations without using any branching (e.g.  $x = (y == 5) + 1$ ; instead of  $x = (y == 5) ? 2 : 1$ ;;).
- **Use iteration instead of recursion** if possible (calling a function costs something).
- **Use good enough approximations instead of completely accurate calculations**, e.g. taxicab distance instead of Euclidean distance, capsule shape to represent the player's collision shape rather

than the 3D model's mesh etc. With a physics engine instead of running the simulation at the same FPS as rendering, you may just run it at half and interpolate between two physics states at every other frame. Nice examples can also be found in computer graphics, e.g. some software renderers use perspective-correct texturing only for large near triangles and cheaper affine texturing for other triangles, which mostly looks OK.

- **Use quick bailout conditions:** many times before performing some expensive calculation you can quickly check whether it's even worth performing it and potentially skip it. For example in physics collision detections you may first quickly check whether the bounding spheres of the bodies collide before running an expensive precise collision detection -- if bounding spheres of objects don't collide, it is not possible for the bodies to collide and so we can skip further collision detection.
- **Operations on static data can be accelerated with accelerating structures** (look-up tables for functions, indices for database lookups, spatial grids for collision checking, various trees ...).
- **Use powers of 2** (1, 2, 4, 8, 16, 32, ...) whenever possible, this is efficient thanks to computers working in binary. Not only may this help nice utilization and alignment of memory, but mainly multiplication and division can be optimized by the compiler to mere bit shifts which is a tremendous speedup.
- **Memory alignment usually helps speed**, i.e. variables at "nice addresses" (usually multiples of the platform's native integer size) are faster to access, but this may cost some memory (the gaps between aligned data).
- **Write cache-friendly code** (minimize long jumps in memory).
- **Compare to 0 rather than other values.** There's usually an instruction that just checks the zero flag which is faster than loading and comparing two arbitrary numbers.
- **Use bit tricks**, hacks for manipulating binary numbers in clever ways only using very basic operations without which one might naively write complex inefficient code with loops and branches. Example of a simple bit trick is checking if a number is power of two as  $!(x \& (x - 1)) \& x$ .
- **Consider moving computation from run time to compile time**, see preprocessor, macros and metaprogramming. E.g. if you make a resolution of your game constant (as opposed to a variable), the compiler will be able to partially precompute expressions with the display dimensions and so speed up your program (but you won't be able to dynamically change resolution).
- On some platforms such as ARM the first **arguments to a function may be passed via registers**, so it may be better to have fewer parameters in functions.
- **Passing arguments costs something:** passing a value to a function requires a push onto the stack and later its pop, so minimizing the number of parameters a function has, using global variables to pass arguments and doing things like passing structs by pointers rather than by value can help speed. { from *Game Programming Gurus* -drummyfish }
- **Optimize when you already have a working code.** As Donald Knuth put it: "premature optimization is the root of all evil". Nevertheless you should get used to simple no-brainer efficient patterns by default and just write them automatically. Also do one optimization at a time, don't try to put in more optimizations at once.
- **Use your own caches where they help**, for example if you're frequently working with some database item you better pull it to memory and work with it there, then write it back once you're done (as opposed to communicating with the DB there and back).
- **Single compilation unit (one big program without linking) can help compiler optimize better** because it can see the whole code at once, not just its parts. It will also make your program compile faster.
- Search literature for **algorithms with better complexity class** (sorts are a nice example).
- For the sake of simple computers such as embedded platforms **avoid floating point** as that is often painfully slowly emulated in software. Use fixed point, or at least offer it as a fallback. This also applies to other hardware requirements such as GPU or sound cards: while such hardware accelerates your program on computers that have the hardware, making use of it may lead to your program being slower on computers that lack it.
- **Factoring out invariants from loops and early branching can create a speed up:** it's sometimes possible to factor things out of loops (or even long non-looping code that just repeats some things), i.e. instead of branching inside the loop create two versions of the loop and branch in front of them. This is a kind of space-time tradeoff. Consider e.g. while (a) if (b) func1(); else func2(); -- if *b* doesn't change inside the loop, you can rewrite this as if (b) while (a) func1(); else while (a) func2();. Or in while (a) b += c \* d; if *c* and *d* don't change (are invariant), we can rewrite to cd = c \* d; while (a) b += cd;. And so on.
- **Division can be replaced by multiplication by reciprocal**, i.e.  $x / y = x * 1/y$ . The point is that multiplication is usually faster than division. This may not help us when performing a single division

by variable value (as we still have to divide 1 by y) but it does help when we need to divide many numbers by the same variable number OR when we know the divisor at compile time; we save time by precomputing the reciprocal before a loop or at compile time. Of course this can also easily be done with fixed point and integers!

- **Consider the difference between logical and bitwise operators!** For example AND and OR boolean functions in C have two variants, one bitwise (& and |) and one logical (&& and ||) -- they behave a bit differently but sometimes you may have a choice which one to use, then consider this: bitwise operators usually translate to only a single fast (and small) instruction while the logical ones usually translate to a branch (i.e. multiple instructions with potentially slow jumps), however logical operators may be faster because they are evaluated as short circuit (e.g. if first operand of OR is true, second operand is not evaluated at all) while bitwise operators will evaluate all operands.
- **Consider the pros and cons of using indices vs pointers:** When working with arrays you usually have the choice of using either pointers or indices, each option has advantages and disadvantages; working with pointers may be faster and produce smaller code (fewer instructions), but array indices are portable, may be smaller and safer. E.g. imagine you store your game sprites as a continuous array of images in RAM and your program internally precomputes a table that says where each image starts -- here you can either use pointers (which say directly the memory address of each image) or indices (which say the offset from the start of the big image array): using indices may be better here as the table may potentially be smaller (an index into relatively small array doesn't have to be able to keep any possible memory address) and the table may even be stored to a file and just loaded next time (whereas pointers can't because on next run the memory addresses may be different), however you'll need a few extra instructions to access any image (adding the index to the array pointer), which will however most definitely be negligible.
- **Reuse variables to save space.** A warning about this one: readability may suffer, mainstreamers will tell you you're going against "good practice", and some compilers may do this automatically anyway. Be sure to at least make this clear in your comments. Anyway, on a lower level and/or with dumber compilers you can just reuse variables that you used for something else rather than creating a new variable that takes additional RAM; of course a prerequisite for "merging" variables is that the variables aren't used at the same time.
- **To save memory use compression techniques.** Compression doesn't always have to mean you use a typical compression algorithm such as jpeg or LZ77, you may simply just throw in a few compression techniques such as run length or word dictionaries into your data structures. E.g. in Anarch maps are kept small by consisting of a small dictionary of tile definitions and map cells referring to this dictionary (which makes the cells much smaller than if each one held a complete tile definition).
- **What's fast on one platform may be slow on another.** This depends on the instruction set as well as on compiler, operating system, available hardware, driver implementation and other details. In the end you always need to test on the specific platform to be sure about how fast it will run. A good approach is to optimize for the weakest platform you want to support -- if it runs fast on a weak platform, a "better" platform will most likely still run it fast.
- **Prefer preincrement over postincrement** (typically e.g. in a for loop), i.e. rather do ++i than i++ as the latter is a bit more complex and normally generates more instructions.
- **Mental calculation tricks**, e.g. multiplying by one less or more than a power of two is equal to multiplying by power of two and subtracting/adding once, for example  $x * 7 = x * 8 - x$ ; the latter may be faster as a multiplication by power of two (bit shift) and addition/subtraction may be faster than single multiplication, especially on some primitive platform without hardware multiplication. However this needs to be tested on the specific platform. Smart compilers perform these optimizations automatically, but not every compiler is high level and smart.
- **With more than two branches use switch instead of ifs** (if possible) -- it should be common knowledge but some newcomers may not know that switch is fundamentally different from if branches: switch statement generates a jump table that can branch into one of many case labels in constant time, as opposed to a series of if statements which keeps checking conditions one by one, however switch only supports conditions of exact comparison. So prefer using switch when you have many conditions to check (but know that switch can't always be used, e.g. for string comparisons). Switch also allows hacks such as label fall through which may help some optimizations.
- **Else should be the less likely branch**, try to make if conditions so that the if branch is the one with higher probability of being executed -- this can help branch prediction.
- Similarly **order if-sequences and switch cases from most probable:** If you have a sequences of ifs such as if (x) ... else if (y) ... else if (z) ..., make it so that the most likely condition to hold gets checked first, then second most likely etc. Compiler most likely can't know the

probabilities of the conditions so it can't automatically help with this. Do the same with the switch statement -- even though switch typically gets compiled to a table of jump addresses, in which case order of the cases doesn't matter, it may also get compiled in a way similar to the if sequence (e.g. as part of size optimization if the cases are sparse) and then it may matter again.

- **Variable aliasing:** If in a function you are often accessing a variable through some complex dereference of multiple pointers, it may help to rather load it to a local variable at the start of the function and then work with that variable, as dereferencing pointers costs something. { from *Game Programming Gurus* -drummyfish }
- **You can save space by "squeezing" variables** -- this is a space-time tradeoff, it's a no brainer but nubs may be unaware of it -- for example you may store 2 4bit values in a single char variable (8bit data type), one in the lower 4bits, one in the higher 4bits (use bit shifts etc.). So instead of 16 memory-aligned booleans you may create one int and use its individual bits for each boolean value. This is useful in environments with extremely limited RAM such as 8bit Arduinos.
- **Consider lazy evaluation** (only evaluate what's actually needed).
- **You can optimize critical parts of code in assembly**, i.e. manually write the assembly code that takes most of the running time of the program, with as few and as inexpensive instructions as possible (but beware, popular compilers are very smart and it's often hard to beat them). But note that such code loses portability! So ALWAYS have a C (or whatever language you are using) fallback code for other platforms, use ifdefs to switch to the fallback version on platforms running on different assembly languages.
- **Loop unrolling/splitting/fusion, function inlining etc.:** there are optimizations that are usually done by high level languages at assembly level (e.g. loop unrolling physically replaces a loop by repeated commands which gains speed but also makes the program bigger). However if you're writing in assembly or have a dumb compiler (or are even writing your own) you may do these manually, e.g. with macros/templates etc. Sometimes you can hint a compiler to perform these optimizations, so look this up.
- **Parallelism (multithreading, compute shaders, ...) can astronomically accelerate many programs**, it is one of the most effective techniques of speeding up programs -- we can simply perform several computations at once and save a lot of time -- but there are a few notes. Firstly not all problems can be parallelized, some problem are sequential in nature, even though most problems can probably be parallelized to some degree. Secondly it is hard to do, opens the door for many new types of bugs, requires hardware support (software simulated parallelism can't work here of course) and introduces dependencies; in other words it is huge bloat, we don't recommend parallelization unless a very, very good reason is given. Optional use of SIMD instructions can be a reasonable midway to going full parallel computation.
- **Optimizing data:** it's important to remember we can optimize both algorithm AND data, for example in a 3D game we may simplify our 3D models, remove parts of a level that will never be seen etc.
- **Specialized hardware (e.g. a GPU) astronomically accelerates programs**, but as with the previous point, portability and simplicity greatly suffers, your program becomes bloated and gains dependencies, always consider using specialized hardware and offer software fallbacks.
- **Smaller code may also be faster** as it allows to fit more instructions into cache.
- Do not optimize everything and for any cost: optimization often makes the code more cryptic, it may bloat it, bring in more bugs etc. Only optimize if it is worth the prize. { from *Game Programming Gurus* -drummyfish }

## When To Actually Optimize?

Nubs often ask this and this can also be a very nontrivial question. Generally fine, sophisticated optimization should come as one of the last steps in development, when you actually have a working thing. These are optimizations requiring significant energy/time to implement -- you don't want to spend resources on this at the stage when they may well be dropped in the end, or they won't matter because they'll be outside the bottleneck. However there are two "exceptions".

The highest-level optimization is done as part of the initial design of the program, before any line of code gets written. This includes the choice of data structures and mathematical models you're going to be using, the very foundation around which you'll be building your castle. This happens in your head at the time you're forming an idea for a program, e.g. you're choosing between server-client or P2P, monolithic or micro kernel, raytraced or rasterized graphics etc. These choices affect greatly the performance of your program but can hardly be changed once the program is completed, so they need to be made beforehand. **This requires wide knowledge and experience** as you work by intuition.

Another kind of optimization done during development is just automatically writing good code, i.e. being familiar with specific patterns and using them without much thought. For example if you're computing some value inside a loop and this value doesn't change between iterations, you just automatically put computation of that value **before** the loop. Without this you'd simply end up with a shitty code that would have to be rewritten line by line at the end. Yes, compilers can often do this simple kind of optimization for you, but you don't want to rely on it.

## Automatic Optimization

Automatic optimization is typically performed by the compiler; usually the programmer has the option to tell the compiler how much and in what way to optimize (no optimization, mild optimization, aggressive optimization, optimization for speed, size; check e.g. the man pages of [gcc](#) where you can see how to turn on even specific types of optimizations). Some compilers perform extremely complex reasoning to make the code more efficient, the whole area of optimization is a huge science -- here we'll only take a look at the very basic techniques. We see optimizations as transformations of the code that keep the semantics the same but minimize or maximize some measure (e.g. execution time, memory usage, power usage, network usage etc.). Automatic optimizations are usually performed on the intermediate representation (e.g. [bytecode](#)) as that's the ideal way (we only write the optimizer once), however some may be specific to some concrete instruction set -- these are sometimes called *peephole* optimizations and have to be delayed until code generation.

The following are some common methods of automatic optimization (also note that virtually any method from the above mentioned manual optimizations can be applied if only the compiler can detect the possibility of applying it):

{ Tip: man pages of gcc or possibly other compilers detail specific optimizations they perform under the flags that turn them on, so see these man pages for a similar overview. ~drummyfish }

- **Replacing instructions with faster equivalents:** we replace an instruction (or a series of instructions) with another one that does the same thing but faster (or with fewer instructions etc.). Typical example is replacing multiplication by power of two with a bit shift (e.g.  $x * 8$  is the same as  $x \ll 3$ ).
- **Inlining:** a function call may usually (not always though, consider e.g. [recursion](#)) be replaced by the function code itself inserted in the place of the call (so called inlining). This is faster but usually makes the code bigger so the compiler has to somehow judge and decide when it's worth to inline a function -- this may be affected e.g. by the function size (inlining a short function won't make the code that much bigger), programmer's hints (`inline` keyword, optimize for speed rather than size etc.) or guesstimating how often the function will be called. Function that is only called in one place can be safely inlined.
- **Loop unrolling:** duplicates the body of a loop, making the code bigger but increasing its speed (a condition check is saved). E.g. `for (int i = 0; i < 3; ++i) func();` may be replaced with `func(); func(); func();`. Unrolling may be full or just partial.
- **Lazy evaluation/short circuit/test reordering:** the principles of lazy evaluation (evaluate function only when we actually need it) and short circuit evaluation (don't further evaluate functions when it's clear we won't need them) may be auto inserted into the code to make it more efficient. Test reordering may lead to first testing simpler things (e.g. equality comparison) and leaving complex tests (function calls etc.) for later.
- **Algebraic laws, expression evaluation:** expressions may be partially preevaluated and manipulated to stay mathematically equivalent while becoming easier to evaluate, for example  $1 + 3 + 5 * 3 * x / 6$  may be transformed to just  $4 + 5 * x / 2$ .
- **Removing instructions that cancel out:** for example in [Brainfuck](#) the series of instructions `+++--` may be shortened to just `+`.
- **Removing instructions that do nothing:** generated code may contain instructions that just do nothing, e.g. NOPs that were used as placeholders that never got replaced; these can be just removed.
- **Register allocation:** most frequently used variables should be kept in CPU registers for fastest access.
- **Removing branches:** branches are often expensive due to not being CPU pipeline friendly, they can sometimes be replaced by a branch-free code, e.g. `if (a == b) c = 1; else c = 0;` can be replaced with `c = a == b;`

- **Memory alignment, reordering etc.:** data stored in memory may be reorganized for better efficiency, e.g. an often accessed array of bytes may actually be made into array of ints so that each item resides exactly on one address (which takes fewer instructions to access and is therefore faster). Data may also be reordered to be more cache friendly.
- **Generating lookup tables:** if the optimizer judges some function to be critical in terms of speed, it may auto generate a lookup table for it, i.e. precompute its values and so sacrifice some memory for making it run extremely fast.
- **Dead code removal:** parts of code that aren't used can be just removed, making the generated program smaller -- this includes e.g. functions that are present in a library which however aren't used by the specific program or blocks of code that become unreachable e.g. due to some `#define` that makes an if condition always false etc.
- **Compression:** compression methods may be applied to make data smaller and optimize for size (for the price of increased CPU usage).
- ...

## See Also

- refactoring
- bit hacks
- fizzbuzz

---

OS

## OS

OS can stand for either operating system or open source.

---

palette

## Palette

In computer graphics palette is a set of possible colors that can be displayed, the term usually refers to a selected smaller subset of all colors that can in theory be displayed (large sets of colors tend to be called color spaces rather than palettes). Nowadays mainstream computers are powerful enough to work with over 6 million 24bit RGB colors (so called True Color) practically without limitations so the use of palettes is no longer such a huge thing, but with resource-limited machines, such as embedded devices and older computers, the use of palettes is sometimes necessary or at least offers many advantages (e.g. saving a lot of memory). Nevertheless palettes find uses even in "modern" graphics, e.g. in the design of image formats that save space. Palettes are also greatly important in pixel art as an artistic choice.

Palettes usually contain a few to few thousand colors and the number is normally a power of 2, i.e. we see palettes with number of colors being 8, 16, 256, 2048, etc. -- this has advantages such as efficiency (fully utilizing color indices, keeping memory aligned etc.). Palettes can be general purpose or specialized (for example some image formats such as GIF create a special palette for every individual image so as to best preserve its colors). Palettes can also be explicitly stored (the palette colors are stored somewhere in the memory) or implicit (the color can somehow be derived from its index, e.g. the 565 palette).

Palettes are related to screen modes -- systems that work with palettes will usually offer to set a specific screen mode that defines parameters such as screen resolution and number of colors we can use, i.e. the number of colors of our palette (we can normally set the colors in a palette). Modes that make use of palettes are called **indexed** because each pixel in memory is stored as an index to the palette (for example if we have a palette {red, yellow, white}, a pixel value 0 will stand for *red*, 1 for *yellow* and 2 for *white*) -- the palette serves as a color look-up table (CLUT). Non-indexed modes on the other hand store the color directly (i.e. there will typically be a direct RGB value stored for each pixel). We can see that an indexed mode (i.e. choosing to use a palette) will save a lot of memory for the framebuffer (VRAM) thanks to reducing the number of bits per pixel: e.g. when using an 8 bit palette, storing each pixel (index) will take up 1 byte (8 bits, 256 colors) while in a non-indexed 24 bit RGB mode (over 6 million colors) each pixel will take 3 bytes (24 bits), i.e. three times as much. The same goes for using bigger palettes: e.g. using a 16 bit palette

(65536 colors) will take four times as much memory for storing pixels than a 4 bit palette (16 colors). Note that even in indexed modes we may sometimes be able to draw pixels of arbitrary color with so called **direct** writes to the display, i.e. without the color being stored in framebuffer. With palettes we may see the use of dithering to achieve the illusion of mixing colors.

Using palettes has also more advantages, for example we can cycle the palette colors or quickly switch it for another palette and so e.g. increase contrast or apply some color effect (this trick was used e.g. in Doom). Palettes can be constructed in clever ways (for example in Anarch) so that it is e.g. easy to make a color brighter or darker by simply incrementing or decrementing its index (while increasing brightness of a three-component RGB value is complex and slow) -- as we generally process big numbers of pixels this can lead to tremendous speed ups. Having fewer colors also makes them easier to compare and so easily implement things such as pixel art upscaling (huge number of colors generally forces us to compare pixels with some amount of bias which is slower).

**Can palettes be copyrighted?** We hope not, that would indeed be pretty fucked up, however it's not that simple, for example those massive faggots at Pantone literally try to do just that and successfully removed their "proprietary colors" from photoshop. Trademarks and trade dress already allowed some kind of ownership of colors or at least their combinations (Milka even tried to trademark a single color), and some websites for sharing palettes claim that a picture of a palette can be copyrighted as some kind of "digital painting", even though they acknowledge a small set of colors as such probably can't be copyrighted. In general copyright MAY apply to selection (abstract set) of things: for example a mere selection of articles from Wikipedia may be considered a copyrightable work, though of course such a "work" (lol) still has to pass some threshold of originality etc. So for maximum safety try to create your own palette (and share it under CC0 and other waivers just in case, to spare others the same pain) as a first option, as a second option use some common public domain mathematically generated palette (e.g. 332) or a palette that's explicitly shared under free terms (CC0 is probably best), and if you absolutely have to reuse someone else's palette (free or proprietary), at least try to make slight modifications to it by reordering the colors and possibly slightly changing the RGB values.

## Examples

Example of a basic 8 color palette may be (the color notation is in hexadecimal #rrggbb format):

```
#000000 #808080 #ffffff #ff0000 #00ff00 #0000ff #ffff00 #00ffff
black   gray    white   red     green  blue    yellow cyan
```

The following is a general purpose 256 color palette made by drummyfish and used in Anarch. It is based on HSV model: it divides colors into 4 saturations, 10 or 11 hues and 8 levels of value ("brightness") which can easily be changed by incrementing/decrementing the color index (which in Anarch was exploited for lightening up and darkening textures depending on distance).

```
#000000 #242424 #494949 #6d6d6d #929292 #b6b6b6 #dbdbdb #ffffff
#201515 #402a2a #604040 #805555 #a06a6a #c08080 #e09595 #ffa0a0
#201b15 #40372a #605240 #806e55 #a08a6a #c0a580 #e0c195 #ffdca0
#1d2015 #3b402a #596040 #778055 #95a06a #b3c080 #d1e095 #edffaa
#172015 #2f402a #466040 #5e8055 #75a06a #8dc080 #a5e095 #bcffaa
#152019 #2a4033 #40604c #558066 #6aa080 #80c099 #95e0b3 #aaffcc
#15201f #2a403f #40605f #55807f #6aa09f #80c0bf #95e0df #aafffe
#151920 #2a3340 #404c60 #556680 #6a80a0 #8099c0 #95b3e0 #aaccff
#171520 #2e2a40 #464060 #5d5580 #746aa0 #8c80c0 #a395e0 #b9aaff
#1d1520 #3b2a40 #594060 #775580 #956aa0 #b380c0 #d195e0 #eeaaaf
#20151b #402a37 #604053 #80556f #a06a8b #c080a7 #e095c3 #ffaadd
#200a0a #401515 #602020 #802a2a #a03535 #c04040 #e04a4a #ff5555
#20170a #402e15 #604520 #805c2a #a07435 #c08b40 #e0a24a #ffb955
#1b200a #374015 #536020 #6e802a #8aa035 #a6c040 #c2e04a #dcff55
#f200a0 #1e4015 #2d6020 #3c802a #4ba035 #5bc040 #6ae04a #79ff55
#a20130 #154026 #206039 #2a804c #35a060 #40c073 #4ae086 #55ff99
#a201f0 #15403f #20605f #2a807e #35a09e #40c0be #4ae0de #55fffd
#a13200 #152640 #203960 #2a4c80 #3560a0 #4073c0 #4a86e0 #5599ff
#e0a200 #1d1540 #2c2060 #3a2a80 #4935a0 #5840c0 #664ae0 #7455ff
#1b0a20 #371540 #532060 #6e2a80 #8a35a0 #a640c0 #c24ae0 #dd55ff
#200a17 #40152f #602047 #802a5e #a03576 #c0408e #e04aa6 #ff55bc
#200000 #400000 #600000 #800000 #a00000 #c00000 #e00000 #ff0000
#201100 #402200 #603300 #804500 #a05600 #c06700 #e07900 #ff8a00
```

```
#1d2000 #3a4000 #586000 #758000 #92a000 #b0c000 #cde000 #eaff00
#c20000 #184000 #246000 #308000 #3ca000 #48c000 #54e000 #60ff00
#200500 #400a00 #600f00 #801500 #a01a00 #c01f00 #e02400 #ff2900
#201600 #402d00 #604300 #805a00 #a07000 #c08700 #e09e00 #ffb400
#172000 #2e4000 #466000 #5d8000 #74a000 #8cc000 #a3e000 #baff00
#620000 #c40000 #126000 #188000 #1ea000 #24c000 #2ae000 #30ff00
#b00200 #160040 #210060 #2d0080 #3800a0 #4300c0 #4f00e0 #5900ff
#1c0020 #390040 #550060 #720080 #8f00a0 #ab00c0 #c800e0 #e400ff
#200012 #400024 #600036 #800048 #a0005a #c0006c #e0007e #ff008f
```

Other common palettes include [RGB332](#) (256 colors, one byte represents RGB with 3, 3 and 2 bits for R, G and B) and [RGB565](#) (65536 colors, two bytes represent RGB with 5, 6 and 5 bits for R, G and B).

## See Also

- [color ramp](#)

---

paradigm

## Paradigm

Paradigm (from Greek *paradeigma*, "pattern", "example") of a [programming language](#) means the very basic concepts that are used as a basis for performing computation in that language. Among popular paradigms are e.g. the [imperative](#), [object oriented](#) and [functional](#), but there are many more; we may see every paradigm as a set of basic ideas and mathematical models (e.g. [models of computation](#)) that form the foundation of how the language works; these are typically additionally also accompanied by kind of "philosophy"/mindset/recommendations that will likely be used by the programmer who uses the language. Just to be clear, paradigm does NOT encompass other than purely technical aspects of performing computation (i.e. it does NOT include e.g. political, artistic or other ideas such as "eco-friendly language", "joke language" etc.). Just as e.g. music genres, paradigms are greatly fuzzy, have different definitions, flavors and are often combined; sometimes it's unclear how to classify paradigms (if one strictly falls under another etc.) or even if something is or isn't a paradigm.

For example the [functional](#) paradigm is built on top of [lambda calculus](#) (one of many possible mathematical systems that can be used to perform general calculations) which performs calculations by combining pure mathematical [functions](#) -- this then shapes the language so that a programmer will mostly be writing mathematical functions in it, AND this also usually comes with the natural "philosophy" of subsequently viewing everything as a function, even such things as loops or [numbers](#) themselves. In contrast [object oriented](#) (OOP) paradigm tries to solve problems by constructing a network of intercommunicating "objects" and so in OOP we tend to see most things as objects.

**Most common** practically used paradigm is the [imperative](#), one based on the simple concept of issuing "commands" to a [computer](#) -- though it is nowadays almost always combined with some other [bullshit](#) paradigm, most notably [object orientation](#). Prevalence of imperative paradigm is probably caused by several factors, most importantly its simplicity (it's possibly the closest to human thinking, easiest to learn, predict etc.), efficiency thanks to being closest to how computers actually work (compilers have very small overhead in translation, they perform less "[magic](#)"), historically established status (which is related to simplicity; imperative was the first natural approach to programming) etc.

**List of notable paradigms** follows (keep in mind the subjectivity and fuzziness that affect classification):

- **[imperative](#)**: Programmer issues commands, the computer blindly executes them ("impero" = "to command"). The focus is on the process of computation. This is the most common paradigm.
  - ♦ **[procedural](#)**: Programmer writes procedures -- smaller subprograms that together solve the whole problem at hand. This is an extremely common paradigm; procedures are also often called [functions](#), but they mustn't be confused with PURE mathematical functions used in functional paradigm. Examples: [Pascal](#), [C](#), [Fortran](#), ...
  - ♦ **[stack-based](#)**: Computation happens on stack (or multiple stacks), a [data structure](#) very convenient for this purpose. These languages often naturally use prefix or postfix notation, it is easy to implement parameter passing and returning values, many computations are elegant.



This is a popular paradigm for minimalist languages. Examples: Forth, comun.

- ◆ **event-driven**: Programmer defines reactions to certain external events rather than a single run of a program. This is very often used in GUI programming (with events such as "button clicked", "window resized" etc.). Example: Javascript.
- ◆ **array-based**: Operations mostly work on arrays as opposed to working on single values (scalars).
- ◆ ...
- **declarative**: Programmer defines ("declares") what the result should look like, the program finds something that fits the definition. The focus is on the result, not the process to obtain it.
  - ◆ **functional**: Programmer describes solution to a problem as composition of pure mathematical functions (which are not to be confused with more loosely defined "functions" present in many languages). Here everything, even branching and loops, are implemented as strictly mathematical functions that have no side effects. Formally this paradigm is based on lambda calculus. Example: Haskell.
  - ◆ **logic**: Programmer describes solution using formal logic. Example: Prolog.
  - ◆ ...
- **object oriented** (OOP): Programmer defines objects (greatly independent, small encapsulated abstract entities) that communicate with each other to solve given problem. Most generally the paradigm is only about "decomposing problems to objects" and so may be implemented as both imperative and declarative, though nowadays OOP is heavily combined with imperative programming and so is often seen as imperative. Examples: Smalltalk, Java, C++, ...
  - ◆ class-based
  - ◆ classless
  - ◆ ...
- **agent oriented** (AOP): Very similar to OOP, sometimes very vague, many times seen as OOP extension focusing on concurrency, agents are entities having their own goals (as opposed to mere objects as "service providers").
- ...

The list enumerates just the most important paradigms, other possible paradigms and "almost" paradigms may just include anything connected to any largely useful concept, e.g.: recursive, concurrent, structured, data oriented, visual, set-based, table-based, metaprogramming, nondeterministic, value-level, message-based, generic, reflective, constraint programming, genetic, term rewriting, string-based, symbolic etcetc.

---

patent

## Patent

Patent is a form of extreme "intellectual property" that allows owning useful ideas, oppressing and bullying people and preventing others from using ideas -- software patents are especially harmful to society and technology. Patents are currently along with copyright likely the most harmful kind of "intellectual property" in technology -- even though copyright is probably a more pressing issue at the moment because it is the most common form of IP oppression, patents can be just as harmful in individual cases. Of course we're not even talking about the whole gigantic bullshit bureaucracy and business connected to patents that just wastes man centuries of effort. Examples of patents in software are minigames on loading screens in games (this patent has already expired), shadow volume algorithm for rendering shadows, mp3 format (also expired), various compression techniques, even such broad ideas as **public key encryption** (yes, the whole idea that's the basis of cryptography was patented and unusable until 1977) etc.

There is an article on software patents at <https://www.gnu.org/philosophy/software-patents.en.html>. There is even a site and initiative dedicated to ending software patents at [https://wiki.endsoftwarepatents.org/wiki/Main\\_Page](https://wiki.endsoftwarepatents.org/wiki/Main_Page).

Patents are kind of similar to but also very different from copyright (Richard Stallman stressed the differences and says it is dangerous to think of copyright and patents as similar): while copyright applies to art and is granted automatically, patents apply to ideas (which should ideally be new inventions but in practice can be just any trivially stupid ideas), have to be registered and are kept recorded somewhere. Patents also last a shorter time than copyright (generally 20 years as opposed to copyright's lifetime plus 70

years) and are territorial, i.e. not world-wide. These facts make patents a bit less disastrous than copyright, however they still cause a great deal of damage -- not only do they prevent technological progress (a new idea such as a new efficient algorithm is simply prohibited to be used by anyone but it's "owner" and those who the owner sells a license), they also allow so called **patent trolling** (patent scams) -- patent trolling takes advantage of the fact that it is practically impossible to safely check if some idea is not patented, i.e. safe to use. There exist troll companies whose sole business is to register trivial patents and then sue random people who unknowingly implement this idea in their projects (there is e.g. a famous video about how this happened to the developer of X-plane, trolled by Uniloc company that had patented the idea of using a "play store" to distribute programs) -- the companies often bully developers to off court settlement for paying a lower fee but this includes a contract that **prevents the affected developers from talking about this**.

Granting and checking patents is also becoming progressively more difficult, expensive and sometimes basically impossible, as any new filed patent has to be checked for how "innovative" it is. This means someone has to literally go through all ideas ever invented in computer science (impossible even for the biggest brain on the planet) and check if the new submitted idea is really new -- given that computer science progresses by lightning speed, every day it is becoming more and more difficult to check patents. As time for checking a patent is limited, the result is many false positives, errors and grants of patents on trivial or non-innovative ideas, which has disastrous consequences. And of course, we're not even talking about corruption -- patents are highly lucrative and it would be naive to believe there are no cases of someone just buying a patent grant.

Many (probably most) free software proponents, and just many programmers in general, including for example Richard Stallman, John Carmack or Donald Knuth, have highly criticized the existence of software patents. Richard Stallman himself has been warning of the dangers and has likened the world of patents to a **mine field** because when you're programming, you have no idea whether an idea you get and implement in your program isn't in fact "owned" by anyone, programming itself poses risk of stepping on mines (patents).

As a good free software developer you should **use licenses/waivers to get rid of patents!** Similarly to copyright, your software should come with a license or waiver that ensure patents won't prevent others from exercising the four essential freedom rights, i.e. there should be a legal document that says you grant others rights to any of your patented ideas hiding in your source code so that others are safe from you suing them if they reuse your potentially patent-infected code (still, there may unfortunately be hiding patents from third parties which cannot be addressed). Some licenses, such as GPL or Apache include patent grants, however others such as MIT or CC0 don't or have to be slightly modified to do so. This is an issue because there is for example no nice way of dedicating one's work completely to the public domain complete with patent grants, as CC0, Unlicense and WTFPL don't address the patent issue -- with these an extra patent waiver has to be manually added! Unlike with copyright, patent waivers aren't always completely necessary, it is very possible that in many simple and non-innovative projects there are no patented ideas, however one can never be sure, so it is better to use a patent waiver just in case, one can never go wrong by including it.

**Which patent waiver to use?** You may for example copy-paste the waiver from our own wiki.

Some patents are fun and bullshit, e.g. there exist bizarre patents that claim to achieve impossible things such as perpetuum mobile or infinitely efficient compression of random data (nicely analyzed at <http://gailly.net/05533051.html>).

## See Also

- intellectual property
- copyright
- trademark

---

paywall

## Paywall

*BUY PREMIUM MEMBERSHIP TO READ THIS ARTICLE*

## PD

PD stands for public domain.

pedophilia

## Pedophilia

*Love is not a crime.*

{ I hate disclaimers but I'm getting some suicide suggestions and death threats, so I'll leave a small note here: keep in mind LRS loves all living beings and never advocates for hurting anyone, i.e. rape of anyone is absolutely not acceptable, as any other kind of violence against any living being -- this is what really matters in the end (as opposed to respecting arbitrary law-imposed age limits etc.). Any thought, desire, perception or sharing of any information must however never be considered wrong in itself, i.e. bullying someone merely for his sexual orientation or his thoughts is just as wrong as raping someone. LRS is one the most peaceful philosophies in history.

Most people I talk to about this article privately tell me they basically agree with everything I write here, but they say I "shouldn't be saying this aloud". Well, what kind of fucked up society is this when I can't tell a truth everyone knows? What kind of medieval thinking is this, do we really live in such a dystopian horror already? Fuck this shit and fuck your silence, I wanna puke from your conformance to evil.

I have not once now encountered groups of people who tried to seriously push me to committing suicide, simply for advocating not bullying people for a private desire, knowing very well I had suicidal tendencies and that I would never harm anyone, nor would I advocate any kind of harm of anyone -- not random strangers, but people who knew me for long. There is literally no difference from a witch hunt now. This is the kind of people you want to be? Just think about it for a second.

love & peace ~drummyfish }

Pedophilia (also paedophilia or paedosexuality) is a sexual orientation towards children. A pedophile is often called a *pedo* or *minor-attracted person* (map). Opposition of pedophilia is called **pedophobia** or pedohysteria and is a form of age discrimination and witch hunt.

*NOTE for pedophobes:* please attend this anonymous self-help program.

Unlike for example pure homosexuality, pedophilia is completely natural and normal -- many studies confirm this (some links e.g. here) but if you're not heavily brainwashed you don't even need any studies (it's really like wanting to see studies on whether men want to have sex with women at all): wanting to have sex with young, sexually mature girls who are able to reproduce is, despite it being forbidden by law, as normal as wanting to have sex with a woman that is married to someone else, despite it being culturally forbidden, or wanting to punch someone who is really annoying, despite it being forbidden by law. No one can question that pedophilia is natural, the only discussion can be about it being harmful and here again it has to be said it is NOT any more harmful than any other orientation. Can it harm someone? Yes, but so can any other form of sex or any human interaction whatsoever, that's not a reason to ban it. Nevertheless, pedophilia is nowadays wrongfully, mostly for political and historical reasons, labeled a "disorder" (just as homosexuality used to be not a long time ago). It is the forbidden, tabooed, censored and bullied sexual orientation of the 21st century, even though all healthy people are pedophiles -- just don't pretend you've never seen a jailbait you found sexy, people start being sexually attractive exactly as soon as they become able to reproduce; furthermore when you've gone without sex long enough and get extremely horny, you get turned on by anything that literally has some kind of hole in it -- this is completely normal. Basically everyone has some kind of weird fetish he hides from the world, there are people who literally fuck cars in their exhausts, people who like to eat shit, dress in diapers and hang from ceiling by their nipples, people who have sexual relationships with virtual characters etc. -- this is all considered normal, but somehow once you get an erection seeing a hot 17

year old girl, you're a demon that needs to be locked up and cured, if not executed right away, just for a thought present in your mind.

Even though one cannot choose this orientation and even though pedophiles don't hurt anyone any more than for example gay people do, they are highly oppressed and tortured. Despite what the propaganda says, a **pedophile is not automatically a rapist** of children (a pedophile will probably choose to never actually even have sex with a child) any more than a gay man is automatically a rapist of people of the same sex, and watching child porn won't make you want to rape children any more than watching gay porn will make you want to rape people of the same sex. Nevertheless the society, especially the fascists from the LGBT movement who ought to know better than anyone else what it is like to be oppressed only because of private sexual desires, actively hunt pedophiles, bully them and lynch them on the Internet and in the real life -- this is done by both both civilians and the state (I shit you not, in Murica there are whole police teams of pink haired lesbians who pretend to be little girls on the Internet and tease guys so that they can lock them up and get a medal for it). LGBT activists proclaim that a "child can't consent" but at the same time tell you that "a prepubescent child can make a decision about changing its sex" (yes, it's happening, even if parent's agreement is also needed, would parents also be able to allow a child to have sex if it wishes to?). There is a literal **witch hunt** going on against completely innocent people, just like in the middle ages. Innocent people are tortured, castrated, cancelled, rid of their careers, imprisoned, beaten, rid of their friends and families and pushed to suicide sometimes only for having certain files on their computers or saying something inappropriate online (not that any of the above is ever justified to do to anyone, even the worst criminal).

In 15th century an unofficial version of the Bible, the Wycliffe's translation, was officially declared a heretic work and its possession was forbidden under penalty of prison, torture and excommunication. That's what we still do today, just with works violating a different kind of orthodoxy. We are literally still living in the middle ages.

**Can a child consent?** Rather ask if you have good enough reason to prevent it from what it wants to do and what is natural for its healthy development. Can a child consent for going out of house? What if someone abducts it there? What if a car runs it over? Better lock it at home until it's 18 and it's no longer on you if it dies, right? Doesn't matter it will grow up to be a pale unsocialized monster with depression who never saw sunlight, only if it's physically safe and you are legally safe. People nowadays have more trouble with sex than ever before, they don't know what gender they are, they have trouble dating, stay virgins, don't have kids, commit suicides. This wasn't the case in times when this supposed "law protection" didn't exist, how can that be? It's because this "protection" is actually a curse, it makes big deal out of sex and prevents natural development at everyone's pace. It labels people monsters for being attracted to the wrong age group, it labels them marked for life for having been touched by someone from a different age group, it label art a work of Satan if it shows a natural human body. It prohibits the depiction of young face because someone might find it pretty. This you think is a good society? Think again then.

{ I've had people point out to me that pedophobia hurts not only adults but also the minors and children; they told me they had strong sexual desires before the age of 18 they couldn't satisfy because of the age discrimination: even on many social networks they are forced to lie about their age just to be able to join and socialize with others. I myself remember I had the desires LONG before reaching adulthood and would be very glad to satisfy them back then. Sure, abuse can happen, but that's the case for any interaction between children and adults and strong and weak in general -- should we just ban children play parks because that's where many child abductions happen? ~drummyfish }

The fact that they made people believe it is a disorder if your penis can't magically telepathically check a chick's ID and may get erect if she's been born before a date legally established in political region the penis currently resides in shows that at this point an average citizen is more retarded than a braindead chimp. Society believes it is not a disease for a human to think he's a dog but by law it is considered a disease if by the exact nanosecond of your 18th birthday your brain doesn't magically switch from being attracted to "up to exactly 18" to "exactly from 18 above".

Child porn is hardcore censored on the mainstream Internet, it is forbidden to even posses for personal use (!!!) -- even if you don't pay for it, even if you don't show it to anyone, even if you're not redistributing it, even if you're not hurting anyone, even if you don't even watch it, you're a criminal just if a file of an underage PP resides on your harddrive. The anti-pedo craze has gotten so insanely and unbelievably bad that even cartoon pictures of naked children or photos of children in swimsuits (not even talking about non-sexual photos of naked children) are banned basically everywhere on the internet :D WTF. LMAO they

even blur just faces of children on TV. Let's repeat that, **children faces are censored in today's society** xD The worst part is that most people comply with such censorship and even support it, it's unbelievable how fucked up the world is.

The pedophile witch hunt exists because it is a great political tool. It is an arbitrarily invented (well, maybe not invented but purposefully escalated) victimless crime. By the principles of fear culture, it allows to push things such as hard surveillance and censorship, similarly to e.g. "war on terror". You're a government or a corporation and want to spy on people chatting? Just make a law requiring mandatory spyware in all chat and justify it by "pedophiles" (this is what EU did). You're against the surveillance law? You must be a pedophile! The witch hunt also allows to immediately cancel anyone uncomfortable. There's a guy who the government doesn't like? Maybe a political competition. Simple, just plant some files on his computer, make up a little story and he's gone.

Defending pedophilia in itself is enough to be cancelled, perhaps even imprisoned or killed by the angry mob, however it is the morally right thing to always say the truth -- especially that which is being censored. Therefore we mustn't remain silent about this issue.

---

people

## People

"People are retarded." --Osho

All people are idiots, love all of them, never make anyone a leader. A cover mostly says just enough about the book.

Here is a list of people notable in technology or in other ways related to LRS.

- **Aaron Swartz**: famous computer prodigy activist involved in creation of famous things like Reddit, RSS and Creative Commons, suicided at 26
- **Albert Einstein**: 20th century physicist, author of theory of relativity, pacifist and socialist, regarded as one of the most brilliant geniuses in history.
- **Alan Turing**: 20th century mathematician, father of computer science, gay
- **Alexandre Oliva**: free software advocate, founding member of FSFLA, maintainer of Linux-libre
- **Bill Gates**: founder and CEO of Micro\$oft, huge faggot
- **Buddha** (Siddhartha Gautama): started buddhism, a religion seeking enlightenment attained by searching for the ultimate truth and so freeing oneself from all desire
- **David Mondou-Labbe** ("Devine Lu Linvega"): some weird narcissist soyboy making minimalist stuff, 100r member, cryptocapitalist, pseudoleftist fascist, heavily utilizing NC licenses
- **Dennis Ritchie**: creator of C language and co-creator of Unix
- **Diogenes**: based Greek philosopher who opposed all authorities in very cool ways
- **Donald Knuth**: computer scientist, Turing-award winner, author of the famous Art of Computer Programming books and the TeX typesetting system
- **drummyfish** (Miloslav Ā Ā—Ā¾): creator of LRS, a few programs and this wiki, anarcho-pacifist
- **Eric S. Raymond**: oldschool hacker turned capitalist, proponent of open \$ource, desperately trying to be popular, co-founder of OSI and tech writer (Jargon File, CatB, ...)
- **Fabrice Bellard**: legendary programmer, made many famous programs such as ffmpeg, tcc, TinyGL etc.
- **Geoffrey Knauth**: very shitty president of Free Software Foundation since 2020 who embraces proprietary software lol
- **Grigori Perelman** based Russian mathematician who solved one of the biggest problems in math (Poincare conjecture), then refused Fields medal and million dollar prize, refuses to talk to anyone and make hero of himself, just sent a huge fuck you to the system
- **Jason Scott**: quite famous archivist and filmmaker (maintains e.g. textfiles.com), focused on old hacker/boomer tech like BBSes
- **Jesus**: probably the most famous guy in history, had a nice teaching of nonviolence and love
- **Jimmy Wales**: co-founder of Wikipedia
- **John Carmack**: legendary game (Doom, Quake, ...) and graphics developer, often called a programming god

- **John Romero**: legendary oldschool game dev, co-creator of Doom
- **John von Neumann**: early 20th century multidisciplinary genius, one of the greatest computer scientists of all time, also famous for huge IQ and being a human calculator
- **Ken Silverman**: famous oldschool 3D engine programmer (Duke Nukem 3D's BUILD engine, ...), sadly proprietaryfag
- **Ken Thompson**: co-creator of Unix, C and Go
- **Kurt Godel**: mathematician famous for his groundbreaking incompleteness theorems proving that logic itself has intrinsic limitations, was a tinfoil schizo and died of starvation believing his food to be poisoned
- **Larry Sanger**: co-founder of Wikipedia, also one of its biggest critics
- **Larry Wall**: creator of Perl language, linguist
- **Lawrence Lessig**: lawyer, founder of free culture movement and Creative Commons, critic of copyright
- **Linus Torvalds**: Finnish programmer who created Linux and git
- **Luke Smith**: suckless vlogger/celebrity
- **Mahatma Gandhi**: Indian man who greatly utilized and popularized nonviolence
- **Melvin Kaye aka Mel**: genius old time programmer that appears in hacker lore (*Story of Mel*)
- **Mental Outlaw**: suckless vlogger/celebrity
- **Mother Teresa**
- **Nina Paley**: female artist, one of the most famous proponents of free culture
- **Noam Chomsky**: linguist notable in theoretical compsci, anarchist
- **Å scar Toledo G.**: programmer of tiny programs and games (e.g. the smallest chess program), sadly proprietary winfag
- **Richard Stallman**: inventor of free software and copyleft, founder of GNU and FSF, hacker, also created emacs
- **Rob Pike**: oldschool hacker strayed from the path of good, involved in Unix, Plan 9 and go
- **Roy Schestowitz**: PhD journalist, running Techrights, revealing corruption in technology
- **Stephen Gough** (*naked rambler*): based guy who refuses to wear clothes, is bullied by society and kept in prison
- **Steve Jobs**: founder and CEO of Apple, huge retard and dickhead
- **Ted Kaczynski**: AKA the Unabomber, mathematician, prodigy, primitivist and murderer who pointed out the dangers of modern technology
- **Terry Davis**: deceased schizophrenic genius, creator of Temple OS, became a tech meme
- **Tom Murphy VII (Tom7)**: researcher, famous SIGBOVIK contributor and YouTuber
- **Uriel M. Pereira**: deceased member of the suckless/cat-v community, "philosopher"
- **Virgil Dupras**: creator of Collapse OS and Dusk OS
- **viznut** (Ville-Matias HeikkilÄä): creator or countercomplex, minimalist programmer, inventor of bytebeat, hacker, collapse "prepper"
- ...

---

permacomputing

## Permacomputing

Permacomputing is a new term invented by Viznut, it's inspired by the term permaculture and means something like "sustainable, ecological minimalist computing"; see permacomputing wiki.

---

permacomputing\_wiki

## Permacomputing Wiki

Permacomputing wiki is a computer minimalist pseudoleftist-infected wiki centered around so called permacomputing (a recent term that means basically "sustainable computing", focus on maximizing lifespan of technology, minimize its waste etc., inspired by permaculture) that focuses a lot on minimalist, eco-friendly, collapse-ready computing; in many ways (especially when you take away the SJW fascism) the wiki is a lot similar to our LRS wiki. It is part of soynet, the wiki was started in 2022 and can now be accessed at <https://permacomputing.net/>, one of its famous users is Viznut (who allegedly coined the term "permacomputing" on his website in 2020). The wiki has some really cool stuff, but is sadly toxic, with code

of censorship and is littered with pseudoleftist fascism (about half of bullet points in their site rules is just pseudoleftist copy pasta gospel lol). LMAO they are promoting some kind of lesbian servers or something :D The wiki also seem to be dying. { One theory is that it was created as a rage reaction to our wiki and the activity was mostly fueled by anger which by now had possibly burned out :D ~drummyfish }

{ NOTE: Someone reached out to me pointing out permacomputing wiki focuses on new things and concepts while LRS just writes about Unix and "old" stuff -- that's true! Actually permacomputing wiki is awesome in this, it's just sad it's being plagued by ideological issues, but the "content" is really great. I wish I could write better about the "new", I just focus on what I personally do best, i.e. boomer stuff. But I will try to possibly change my direction a bit to focus on new ideas as well. Thanks to the reader for a kind email <3 :-)  
~drummyfish }

**Late 2023 sum up of the wiki's issues:** it seems like a few users just care about computers and try to write cool stuff about technology while clashing with a few political fanatics who just want to push pseudoleftist propaganda, ridiculously trying to find ways to somehow insert feminism and LGBT to core principles of technology design :D It's really awkward and creates conflicts in articles e.g. about Rust where feminists really want to push it as the best thing ever while the educated minimalist just can NOT ever accept Rust as a good language, not even by a huge margin. That's all just funny but what's more, there seem to be even censorship going on as for example the political activists seem to prefer shitty Gemini just for its political message and MUH ENCRYPTION (which, again, clashes with the need of minimalism) over superior Gopher (a clear preferred choice for true minimalist) and so they JUST DON'T MENTION GOPHER AT ALL, even in the article on smol net where it is just a key thing to mention and it's clear they just wanna hide its existence, this is literally like making an article about text editors and refusing to mention Vim in it because you're an Emacs fan :D For the same reason they probably also don't mention our LRS wiki which they most likely copied (or at very least would be worth a mention as a related resource). This is just a message to their readers that they're gonna blatantly manipulate them and so probably something that should make you go away. It's kind of all funny, sad and depressing that yet another promising thing is becoming a victim to the cancer just in such early stage.

{ To be honest reading through the wiki makes me conjecture it's actually a LRS wiki ripoff that refuses to admit to it :D Now to make it clear: I don't care if someone copies this wiki or if I get credited or anything like that, on the contrary, I explicitly state in many places this is public domain, that I highly encourage copying, making ripoffs and despise any idea of being able to own an intellectual work. The conjecture here is of purely entertaining nature. If anyone associated with permacomputing wiki is reading this, let me know if the similarities are purely coincidental because yes, we are dealing with similar topic, by similar means, having similar value etc. I also understand no one wants to associate his work with mine, though making a small note for historians somewhere can hardly bring anyone any harm. Why I think it's so similar? Some hints are these: wiki created about half a year after this one, "Care for life" cited as their "axiom" vs "Unconditional love of all life" is cited as our axiom (even using the same word "axiom"), whole design looks pretty similar (similar top-level links, css similar to my website, ...), same waiver, similar articles (like pseudosimplicity vs pseudominimalism, dependency, smallnet vs smol internet, games, history, paper computer, bloat; sure these are general topic we deal with, but the selection...) with similar content in them (e.g. "A dependency refers to another piece of technology" vs "Dependency of a piece of technology is another piece of technology..."). I don't know, it's just at the edge of me being able to decide if it's a coincidence or not :P  
~drummyfish }

## See Also

- Damaged Earth Catalog
- xxiivv

---

phd

## PhD

PhD (also Ph.D., PhD. etc.), or *doctor of philosophy*, written after the name, is the highest academic degree that can be earned by being a student in University, the basic title required for working as a scientist. It is earned through many years of study and especially active publishing of original research that pushed the boundary of current human knowledge in a specific field. Despite being called doctor of *philosophy*, the title

is awarded generally to scientists in basically any field such as mathematics, physics, psychology, chemistry etc., NOT just to those studying philosophy. PhD is yet above master's degree. It is a doctorate degree, so a holder of PhD is called a *doctor* (Dr.), just as those with other forms of doctorates such as medical doctorate or honorary doctorate; however PhD is the *big doctorate*, the kind of highest, most prestigious one. People with a PhD degree are considered the foremost experts, the smartest, most educated elite, as only about 1 to 2 % of population hold a PhD, though PhD is also often considered an overkill and an overqualification (there are many cases of people with PhD not mentioning it on their CVs because such a high education can actually be a disadvantage), and of course, as with everything under capitalism, PhDs became a thing of business and conformance, subject to corruption and degradation (there now even exist PhDs in astrology, gender studies etc.), at times even a meme. Yes, one has to be quite smart and talented to obtain a PhD, but nowadays it's probably more about pouring an extreme amount of energy, slavery and conformance to the corrupt academic cults, so the prestige of the title comes for a pretty high price, one often not worth paying.

TODO

**Should you get a PhD?** Probably not -- as said the sacrifice required is enormous, to make it you should have a REAL GOOD reason, of which there aren't many -- perhaps if you REALLY want to be teacher at University or if for some twisted reason you want to spend your whole life in the corrupt toxic soyence environment trying to prove women are better than men and sucking capitalist dicks so that they throw you a bit of money so that you can buy a microscope, then maybe. The thing is that focusing on PhD sucks away a great amount of energy you could spend on actually good things, consider that instead of actually programming less retarded software you will just have to do slavery for your dissertation advisor, do bureaucracy, p-value hacking, make powerpoint presentations, marketing for your research, give handjobs to sponsors, do bullshit research you dislike (because publish or perish), all while withstanding incredible amounts of stress and dodging depression. Really masters degree is enough to give you all you need for a rich intellectual life and being able to do good things, and it won't suck the soul out of your body. On the best universities even bachelor's is probably enough. If you REALLY wanna be the smartass guy who others ought to call a doctor, in some countries you may get some kinda small doctorate, usually just for an extra exam and paying some fee (e.g. RNDr, PHDr etc., details will depend on your country so check that out). { TFW just getting the EZ dentist degree so that you may call yourself a doctor :D ~drummyfish } Nowadays you can also just buy a honorary doctorate online, it's absolutely legal business, though you probably don't wanna support this kind of capitalist bullshit, you just pay them unholy money for a piece of paper.

TODO

---

physics\_engine

## Physics Engine

{ LRS now has a very small 3D physics engine called tinyphysicsengine. ~drummyfish }

Physics engine is a software (usually a library) whose purpose is to simulate physics laws of mechanics, i.e. things such as forces, rigid and soft body collisions, particle motion, fluid dynamics etc.

{ When it comes to classic 3D rigid body physics engines, they're extremely hard to make, much harder than for example an advanced 3D rendering engine, especially when you want to make them LRS (without floating point, ...) and/or general and somewhat physically correct (being able to simulate e.g. the Dzhanibekov effect, satisfying all the conservation laws, continuous collision detection etc.). Good knowledge of mechanics and things like quaternions and 3D rotations is just the beginning, difficulties arise in every aspect of the engine, and of those there are many. As I've found, 32 bit fixed point is not enough for a general engine (even though it is enough for a rendering engine), you'll run into precision problems as you need to represent both relatively high and low energies. You'll also run into stability issues such as stable contacts, situations with multiple objects stacked on top of each other starting to bounce on their own etc. Even things such as deciding in what order to resolve collisions are very difficult, they can lead to many bugs such as a car not being able to drive on a straight road made of several segments. Collision detection alone for all combinations of basic shapes (sphere, cuboid, cylinder, capsule, ... let alone general triangle mesh) are hard as you want to detect general cases (not only e.g. surface collisions) and you want to extract all the parameters of the collisions (collision location, depth, normal etc.) AND you want to make it fast. And of course you'll want to add acceleration structures and many other thing on top. So think twice before deciding



to write your own physics engine.

A sane approach may be to write a simplified engine specifically for your program, for example a Minetest-like game may just need non-rotating capsules in a voxel environment, that's not that hard. You can also get away with a bit of cheating and faking, e.g. simulating rigid bodies as really stiff soft bodies, it may not be as efficient and precise but it's simpler to program. It may be good enough. Well, that's basically what tinyphysicsengine does anyway. Old playstation game Rally Cross apparently did something similar too.  
~drummyfish }

Physics engine is a wide term even though one usually imagines the traditional 3D rigid body engine used in games such as GTA. These engines may nevertheless have different purposes, features and even basic paradigms, some may e.g. be specialized just for computing precise ballistic trajectories for the army, some may serve for simulating weather etc. Some common classifications and possible characteristics of physics engines follow:

- **2D vs 3D**: 2D engines are generally much more simple to implement than 3D, for example because of much more simple math for rotations and collision detection. Graphics and physics are usually loosely interconnected (though they should be decoupled) in that the way in which we represent graphics (2D, general 3D, BSP, voxels, ...) usually also determines how we compute physics, so that there may also exist e.g. "pseudo 3D" physics engines as part of "pseudo 3D" renderers, e.g. the one used in Doom etc.
- **real time vs offline**: Real-time ones are mostly intended to be used in the entertainment industry, i.e. games, movies etc. as they can compute somewhat realistic looking results quickly but for the price of dropping high accuracy (they use many approximations). Scientific engines may prefer to be offline and taking longer time to compute more precise results.
- **rigid body vs soft body**: Rigid body engines don't allow bodies to deform while soft body ones do -- in real life all bodies are soft, but neglecting this detail and considering shapes rigid can have benefits (such as being able to consider the body as a whole and not having to simulate all its individual points). Of course, a complex engine may implement both rigid and soft body physics.
- **paradigm**: The basic approach to implementing the simulation, e.g. being impulse-based (applying impulses to correct errors), constraint-based (solving equations to satisfy imposed constraints), penalty-based (trying to find equilibriums of forces) etc.
- **discrete vs continuous collision detection**: Discrete collision detection only detects collisions at single points in time (at each engine tick) and are simple than those implementing continuous collision detection. Discrete engine are less accurate, consider e.g. that a very fast moving object can pass through a wall because at one instant it is in front of it while at the next tick it is behind it. Continuous collisions won't allow this to happen, but are more difficult to program, may be slower etc. For games discrete collisions are usually good enough.
- **purpose and accuracy**: The basic categories are precise, scientific and often special-purpose engines, and engines meant for entertainment and less accurate visualizations such as games and movies.
- **features: fluid, cloth, particles, ragdoll, inverse kinematics, GPU acceleration, determinism, voxels, acceleration data structures ...**: These are a number of additional features the engine can have such as the ability to simulate fluids (which itself is a huge field of its own) or cloths, some go as far as e.g. integrating motion-captured animations of humans with physics to create smooth realistic animations e.g. of running over walking pedestrians with a car and so on.

A typical physics engine will work something like this: we create a so called **physics world**, a data structure that represents the space in which the simulation takes place (it is similar to a scene in rendering engines). We then populate this world with physics elements such as rigid bodies (which can have attributes such as mass, elasticity etc.). These bodies are normally basic geometric shapes such as spheres, cylinders, boxes or capsules, or objects composed of several such basic shapes. This is unlike with rendering engines in which we normally have triangle meshes -- in physics engines triangle meshes are extremely slow to process, so for the sake of a physics engine we approximate this mesh with some of the above basic shapes (for example a creature in a game that's rendered as a hi-poly 3D model may in the physics engine be represented just as a simple sphere). Furthermore the bodies can be static (cannot move, this is sometimes done by setting their mass to infinity) or dynamic (can move); static bodies normally represent the environment (e.g. the game level), dynamic ones the entities in it (player, NPCs, projectiles, ...). Making a body static has performance benefits as its movement doesn't have to be calculated and the engine can also precalculate some things for it that will make e.g. collision detections faster. We then simulate the physics of the world in so called *ticks*

(similar to frames in rendering); in simple cases one tick can be equivalent to one rendering frame, but properly it shouldn't be so (physics shouldn't be affected by the rendering speed, and also for the physics simulation we can usually get away with smaller "FPS" than for rendering, saving some performance). Usually one tick has set some constant time length (e.g. 1/60th of a second). In each tick the engine performs a **collision detection**, i.e. it finds out which bodies are touching or penetrating other bodies (this is accelerated with things such as bounding spheres). Then it performs so called **collision resolution**, i.e. updating the positions, velocities and forces so that the bodies no longer collide and react to these collisions as they would in the real world (e.g. a ball will bounce after hitting the floor). There can be many more things, for example **constraints**: we may e.g. say that one body must never get further away from another body than 10 meters (imagine it's tied to it by a rope) and the engine will try to make it so that this always holds. The engine will also offer a number of other functions such as casting rays and calculating where it hits (obviously useful for shooter games).

**Integrating physics with graphics:** you will most likely use some kind of graphics engine along with physics engine, even if just for debugging. As said above, keep in mind a graphics and physics engines should be **strictly separated** (decoupled, for a number of reasons such as reusability, easier debugging, being able to switch graphics and physics engines etc.), even though they closely interact and may affect each other in their design, e.g. by the data structures you choose for your program (voxel graphics will imply voxel physics etc.). In your program you will have a **physics world and a graphics scene**, both contain their own elements: the scene has graphics elements such as 3D models or particle systems, the physics world has elements such as rigid bodies and force fields. Some of the graphical and physics entities are connected, for example a 3D model of a tree may be connected to a physics rigid body of a cone shape. NOT ALL graphics elements have counterparts in the physics simulation (e.g. a smoke effect or light aren't present in the physics simulation) and vice versa (e.g. player in a first-person game has no 3D model but still has some physics shape). The connection between graphics and physics elements should be done **above** both engines (i.e. do NOT add pointers to physics object to graphics elements etc.). This means that e.g. in a game you create a higher abstract environment -- for example a level -- which stands above the graphics scene and physics world and has its own game elements, each game element may be connected to a graphics or physics element. These game elements have attributes such as a position which gets updated according to the physics engine and which is transferred to the graphics elements for rendering. Furthermore remember that **graphics and physics should often run on different "FPS"**: graphics engines normally try to render as fast as they can, i.e. reach the highest FPS, while physics engines often have a time step, called a **tick**, of fixed time length (e.g. 1/30th of a second) -- this is so that they stay deterministic, accurate and also because physics may also run on much lower FPS without the user noticing (interpolation can be used in the graphics engine to smooth out the physics animation for rendering). "Modern" engines often implement graphics and physics in separate threads, however this is not suckless, in most cases we recommend the KISS approach of a single thread (in the main loop keep a timer for when the next physics tick should be simulated).

## Existing Engines

One of the best and most famous FOSS 3D physics engines is Bullet (zlib license), it has many features (rigid and soft bodies, GPU acceleration, constraints, ...) and has been used in many projects (Blender, Godot, ...). Box2D is a famous 2D physics engine under MIT license, written in C++. Tinyphysicsengine is a KISS LRS 3D physics engine made by drummyfish.

---

physics

## Physics

*Physics is just the "keep asking why" game taken to the extreme.*

TODO

---

pi

# Pi

Pi (normally written with a Greek alphabet symbol with Unicode value U+03C0) is one of the most important and famous numbers, equal to approximately 3.14, most popularly defined as the ratio of a circle's circumference to its diameter (but also definable in other ways). It is one of the most fundamental mathematical constants of our universe and appears extremely commonly in mathematics, nature and, of course, programming. When written down in traditional decimal system, its digits go on and on without end and show no repetition or simple pattern, appearing "random" and chaotic -- as of 2021 pi has been evaluated by computers to 62831853071796 digits. In significance and properties pi is similar to another famous number: e. Pi day is celebrated on March 14.

{ Very nice site about pi: <http://www.pi314.net>. ~drummyfish }

Pi is a real transcendental number, i.e. simply put *it cannot be defined by a "simple" equation* (it is not a root of any polynomial equation). As a transcendental number it is also an irrational number, i.e. it cannot be written as an integer fraction. Mathematicians nowadays define pi via the period of the exponential function rather than geometry of circles. If we stick to circles, it is interesting that in non-Euclidean geometry the value of "pi" could be measured to different values (if we draw a circle on an equator of a ball, its circumference is just twice its diameter, i.e. "pi" would be measured to be just 2, revealing the curvature of space).

Pi to 100 decimal digits is:

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482

Pi to 100 binary fractional digits is:

11.0010010000111111011010101000100010000101101000110000100011010011000100110001100110001010

Among the first 50 billion digits the most common one is 8, then 4, 2, 7, 0, 5, 9, 1, 6 and 3.

Some people memorize digits of pi for fun and competition, the official world record as of 2022 is 70030 memorized digits, however Akira Haraguchi allegedly holds an unofficial record of 100000 digits (made in 2006). Some people make mnemonics for remembering the digits of pi (this is known as *PiPhilology*), for example "*Now I fuck a pussy screaming in orgasm*" is a sentence that helps remember the first 8 digits (number of letters in each word encodes the digit).

**PI IS NOT INFINITE.** Soyence popularizators and nubs often say shit like "OH LOOK pi is so special because it infiniiiite". Pi is completely finite with an exact value that's not even greater than 4, what's infinite is just its expansion in decimal (or similar) numeral system, however this is nothing special, even numbers such as 1/3 have infinite decimal expansion -- yes, pi is more interesting because its decimal digits are non-repeating and appear chaotic, but that's nothing special either, there are infinitely many numbers with the same properties and mysteries in this sense (most famously the number e but besides it an infinity of other no-name numbers). The fact we get an infinitely many digits in expansion of pi is given by the fact that we're simply using a system of writing numbers that is made to handle integers and simple fractions -- once we try to write an unusual number with our system, our algorithm simply ends up stuck in an infinite loop. We can create systems of writing numbers in which pi has a finite expansion (e.g. base pi), in fact we can already write pi with a single symbol: *pi*. So yes, pi digits are interesting, but they are NOT what makes pi special among other numbers.

Additionally contrary to what's sometimes claimed **it is also unproven (though believed to be true), whether pi in its digits contains all possible finite strings** -- note that the fact that the series of digits is infinite doesn't alone guarantee this (as e.g. the infinite series 010011000111... doesn't contain any possible combinations of 1s and 0s either). This would hold if pi was normal -- then pi's digits would contain e.g. every book that will ever be written (see also Library Of Babel). But again, there are many other such numbers.

What makes pi special then? Well, mostly its significance as one of the most fundamental constants that seems to appear extremely commonly in math and nature, it seems to stand very close to the root of

description of our universe -- not only does pi show that circles are embedded everywhere in nature, even in very abstract ways, but we find it in Euler's identity, one of the most important equations, it is related to complex exponential and so to Fourier transform, waves, oscillation, trigonometry (sin, cos, ...) and angles (radians use pi), it even starts appearing in number theory, e.g. the probability of two numbers being relative primes is  $6/(\pi^2)$ , and so on.

## Approximations And Programming

Evaluating many digits of pi is mathematically interesting, programs for computing pi are sometimes used as CPU benchmarks. There are programs that can search for a position of arbitrary string encoded in pi's digits. However in practical computations we can easily get away with pi approximated to just a few decimal digits, **you will NEVER need more than 20 decimal digits**, not even for space flights (NASA said they use 15 places).

One way to judge the quality of pi approximation can be to take the number of pi digits it accurately represents versus how many digits there are in the approximation formula -- this says kind of the approximation's compression ratio. But other factors may be important too, e.g. simplicity of evaluation, functions used etc.

An ugly engineering approximation that's actually usable sometimes (e.g. for fast rough estimates with integer-only hardware) is just (something like this was infamously almost made the legal value of pi by the so called Indiana bill in 1897)

$$\pi \approx 3$$

A simple fractional approximation (correct to 6 decimal fractional digits, by Tsu Chung Chih) is

$$\pi \approx 355/113$$

Such a fraction can again be used even without floating point -- let's say we want to multiply number 123 by pi, then we can use the above fraction and compute  $355/113 * 123 = (355 * 123) / 113$ .

Srinivasa Ramanujan made a great number of pi approximations, e.g. an improvement of the previous to (14 correct digits):

$$\pi \approx 355/113 * (1 - 0.0003/3533)$$

Similarly Plouffe, e.g. (30 correct digits):

$$\pi \approx \ln(262537412640768744)/\sqrt{163}$$

Leibnitz formula for pi is an infinite series that converges to the value of pi, however it converges very slowly { Quickly checked, after adding million terms it was accurate to 5 decimal fractional places. ~drummyfish }. It goes as

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \dots$$

Nilakantha series converges much more quickly { After adding only 1000 terms the result was correct to 9 decimal fractional places for me. ~drummyfish }. It goes as

$$\pi = 3 + 4/(2 * 3 * 4) + 4/(4 * 5 * 6) + 4/(6 * 7 * 8) + \dots$$

A simple algorithm for computing approximate pi value can be based on approach used in further history: approximating a circle with many-sided regular polygon and then computing the ratio of its circumference to diameter -- as a diameter here we can take the average of the "big" and "small" diameter of the polygon. For example if we use a simple square as the polygon, we get  $\pi \approx 3.31$  -- this is not very accurate but we'll get a much higher accuracy as we increase the number of sides of the polygon. In 15th century pi was computed to 16 decimal digits with this method. Using inscribed and circumscribed polygons we can use this to get lower and upper bounds on the value of pi.

Another simple approach is monte carlo estimation of the area of a unit circle -- by generating random (or even regularly spaced) 2D points (samples) with coordinates in the range from -1 to 1 and seeing what portion of them falls inside the circle we can estimate the value of pi as  $pi = 4 * x/N$  where  $x$  is the number of points that fall in the circle and  $N$  the total number of generated points.

Digits of pi also emerge when trying to measure some distances inside Mandelbrot set (see David Bolle, 1991) -- this can perhaps also be exploited.

Spigot algorithm can be used for computing digits of pi one by one, without floating point.

Bailey-Borwein-Plouffe formula (discovered in 1995) interestingly allows computing Nth hexadecimal (or binary) digit of pi, WITHOUT having to compute previous digits (and in a time faster than such computation would take). In 2022 Plouffe discovered a similar formula for computing Nth decimal digit.

The following is a C implementation of the Spigot algorithm for calculating digits of pi one by one that doesn't need floating point or special arbitrary length data types, adapted from the original 1995 paper. It works on the principle of converting pi to the decimal base from a special mixed radix base 1/3, 2/5, 3/7, 4/9, ... in which pi is expressed just as 2.22222... { For copyright clarity, this is NOT a web copy paste, it's been written by me according to the paper. ~drummyfish }

```
#include <stdio.h>

#define DIGITS 1000
#define ARRAY_LEN ((10 * DIGITS) / 3)

unsigned int pi[ARRAY_LEN];

void writeDigit(unsigned int digit)
{
    putchar('0' + digit);
}

int main(void)
{
    unsigned int carry, digit = 0, queue = 0;

    for (unsigned int i = 0; i < ARRAY_LEN; ++i)
        pi[i] = 2; // initially pi in this base is just 2s

    for (unsigned int i = 0; i < DIGITS; ++i)
    {
        carry = 0;

        for (int j = ARRAY_LEN - 1; j >= 0; --j)
            { // convert to base 10 and multiply by 10 (shift one to left)

                unsigned int divisor = (j + 1) * 2 - 1; // mixed radix denom.

                pi[j] = 10 * pi[j] + (j + 1) * carry;
                carry = pi[j] / divisor;
                pi[j] %= divisor;
            }

        pi[0] = carry % 10;
        carry /= 10;

        switch (carry)
        { // latter digits may influence earlier digits, hence these buffers
            case 9: // remember consecutive 9s
                queue++;
                break;

            case 10: // ..X 99999.. becomes ..X+1 00000...
                writeDigit(digit + 1);

                for (unsigned int k = 1; k <= queue; ++k)
                    writeDigit(0);

                queue = 0;
        }
    }
}
```

```

        digit = 0;
        break;

default: // normal digit, just print
    if (i != 0) // skip the first 0
        writeDigit(digit);

    if (i == 1) // write the decimal point after 1st digit
        putchar('.');

    digit = carry;

    for (unsigned int k = 1; k <= queue; ++k)
        writeDigit(9);

    queue = 0;
    break;
}

writeDigit(digit); // write the last one

return 0;
}

```

---

piracy

## "Piracy"

Piracy is a capitalist propaganda term for the act of illegally sharing copyrighted information such as non-free books, movies, music, video games or scientific papers.

It is greatly admirable to support piracy, however also keep in mind the following: if you pirate a proprietary piece of information, you get it gratis but it stays proprietary, it abuses you, it limits your freedom -- you won't get the source code, you won't be able to publicly host it without being bullied, you won't be allowed to legally create and share derivative works etc. Therefore **prefer free (as in freedom) alternatives to piracy**, i.e. pieces of information that are not only gratis but also freedom supporting.

Have you ever heard about a public library that struggles with funding? Did you ever wish your local library could afford a bigger building and more books? Imagine for a moment now that we can build public libraries all around the world, basically for free, even in the most remote of place, each having so many books that they wouldn't fit to a skyscraper, each book in so many copies that arbitrarily many people could lend the same book at the same time, for as long as they want. Wouldn't that be great? People promoting anti-piracy are those who say "no, we are against this". You just can't argue anyone supporting anti-piracy is not evil. We already have this great library that past civilizations didn't even dare to dream of, it's the Internet, it's just that evil dicks now prevent access to it.

Despite the term itself being recent, the concept of "piracy" is nothing new; it's essentially as old as the concept of "intellectual ownership" itself. Famous paintings have been copied by "pirate artists" and sold as being the original. Mozart, thanks to his genius, famously copied sheet of music that was supposed to remain unpublished just from hearing the piece played. For the modern history of computer piracy especially the case of The Pirate Bay, a famous torrenting site established in 2003, was of great importance.

At the dawn of personal computer era, the culture of hackers who helped with pirating software by creating cracks spawned the demoscene, a hugely significant art subculture based on programming technically impressive audiovisual presentations. The fuss around piracy also influences mainstream culture, e.g. the infamous "you wouldn't steal a car" anti-piracy propaganda that was present on VHS movie tapes is now a laughable piece of failed capitalist attempt at trying to invoke a sense of guilt of sharing -- many people happily pirated and pirate to this day and they also happily admit to piracy; in fact many old movies and otherwise historically significant media has been preserved only thanks to people pirating. Piracy in actuality doesn't hurt anyone, monstrously rich corporations are and always will be monstrously rich, and if piracy indeed did hurt a corporation, then it's actually another argument for piracy, not against it.

{ My brother collects old movie dubbing, great works of art of legendary actors, works that now would have been lost to time if it weren't for people recording those movies on VHS tapes and illegally sharing them. ~drummyfish }

One paper from 2020s found that men (curiously unlike women) exposed to anti-piracy propaganda will increase their pirating by 18% :D One example of publicly embracing piracy in the mainstream is e.g. the Pirate party that has risen to popularity in a few countries now.

{ Where to pirate stuff? See e.g. <https://www.reddit.com/r/Piracy/wiki/megathread> and <https://piracy.vercel.app>. ~drummyfish }

TODO: more history etc.

---

plan9

## Plan 9

Plan 9 (from Bell Labs, reference to the movie *Plan 9 from Outer Space*) is a research operating system, now FOSS, that was started by many of the original Unix developers as the next project of this kind, it was supposed to be the "new and updated Unix". It tries to work with Unix philosophy (minimalist software philosophy) but expands and modifies it so as to fit "new/evolved" computers -- though Plan 9 developers claim the system is "more Unix than Unix itself", the validity of such claim is questionable as Plan 9 brings in a more complicated paradigm of distributed computing, dependencies (such as requiring GUI and mouse) and therefore bloat (though still being super minimal compared to mainstream operating systems). Besides the original Plan 9, which is apparently dead now, there exist active forks such as 9front; **BEWARE, 9front has a COC and is a fascist pseudoleftist project**. One famous guy working on Plan 9 is Rob Pike who really went super crazy lately; originally a true Unix hacker he later on started saying things like "I want no local storage on my computer" and "the world should provide me my computing environment and maintain it for me", which seem to reflect what Plan 9 is about. Plan 9 fans are also obsessed with papers.

On one hand Plan 9 sounds good and its idealism is admirable, nevertheless **Plan 9 is SHIT** due to the following fact: **it requires what isn't necessary, for example GUI, mouse, file system and networking and forces computers and users to be certain way**. This is absolutely unforgivable and violates the basic premise of good, freedom offering, minimalist nondiscriminatory software; in fact it violates the Unix philosophy which it is supposed to be building on top of -- an operating system should do one thing well: that of offering an environment for programs and their resources, user interface is a nontrivial extra task that should be separated. If you ask how to use Plan 9 without a mouse, the fans respond with telling you how stupid you are for not wanting to use mouse ("here is a study that says mice are better than keyboards: checkmate!") and that using mouse is actually what you want (hey bro, everyone's using a mouse, just accept it) -- they try to force a specific way of how computers should be and how they should be operated, just as Microsoft and Apple, without taking into account that computers can (and should be allowed to) be wildly different, very small, with tiny displays (or no displays at all), with no pointing devices (game consoles, voice operated computers, ...) etc. Sure, it may be possible to make the system work without a mouse or GUI, but these concepts form the very basis of the code and its philosophy, they will be carried as a dead weight if you're not using them and you will probably encounter great issues such as many programs simply relying on the existence of GUI and mouse and not working without them. The philosophy is similar to that of "smart" devices which assume that "Internet is everywhere" and so "let's put Internet into everything", even things that don't need any Internet at all (like hammers and teaspoons), and by the way they will no longer work without Internet (let's hope it doesn't go down lol). In this way **Plan 9 is a dictatorship** and we don't approve of it.

{ To plan 9 fans: please let me know if I misunderstand the concepts somehow, but this is how I understand the system. Beware however that trying to convince me to simply conform with your way of computing will lead nowhere. ~drummyfish }

Plan9's mascot, Glenda, is proprietary (as of february 2023), despite it having been uploaded to Wikimedia Commons lol. No license to be seen on its website.

TODO: some more shite like history and the actual basic concepts?

## +NIGGER

+NIGGER is a license modifier that's meant to be added to a free software license to prevent corporations from adopting this software by making it impossible to make politically correct forks of such software. Its text is available at <https://plusnigger.autism.exposed/>.

The modifier adds a condition that all modified version of this software have to contain the word "NIGGER". For example a license GPLv3+NIGGER has all the conditions of a GPLv3 license plus the condition of including the word "NIGGER".

## Pokitto

Pokitto is a very nice educational open gaming console friendly to hacking and FOSS. It is also very family friendly, aiming to be used as an educational device for kids on schools, which doesn't at all take away any of its value for hardcore hackers. Its website is <https://www.pokitto.com/>. As of writing this Pokitto is unavailable for purchase as a new version is being developed.

Its great advantage is its nice, active and friendly community that's constantly writing software, documenting Pokitto and helping newcomers. There have even appeared a few issues of Pokitto magazine.

The console was created by Jonne Valola from Finland. He started the project on Kickstarter on April 28 2017, pledged over \$27000 and released Pokitto in February 2018. { Jonne is a really nice guy who loves the project, puts his soul into the project and always personally helps people and shares technical details of the console. ~drummyfish }

Pokito, unlike most other open consoles, is NOT based on Arduino, but on NXP's LPC11U6x microcontroller (MCU). Some features and specs of Pokitto are:

- Up to **220x176 color TFT display** (ST7775R). (Resolution and color depth depends on chosen mode and how much RAM you want to dedicate to screen buffer).
- Up to **72 MHz ARM CPU** (LPC11U6x). The base frequency is 48 MHz but the hardware is easily overclocked.
- **256 kB ROM** (program storage space).
- **36 kB RAM** (working memory).
- **4 kB EEPROM** (persistent storage).
- **7 buttons**.
- **Speaker and headphone jack**.
- Both emulator and simulator which make programming much more efficient and comfortable.
- **Custom library** -- PokittoLib -- free-licensed { even though it contains a few small "fair use" files from the MCU vendor. ~drummyfish }. It has many features, unfortunately it's also kind of bloated.
- **SD card support**.
- Hardware extensions called **hats**. Available is e.g. a hat with joystick and extra buttons.
- Programming via USB, works on GNU/Linux with gcc ARM cross compiler. Supports a few languages: **C++, C, MicroPython and Java**.
- Custom IDE for noobs: Femtolde.
- Schematics and 3D print files available.
- A huge number of games and other software has already been written.

**How free is Pokitto?** Quite freedom friendly, but not nearly 100% free; It is made out of proprietary hardware, but it's quite KISS, the Pokitto library, emulator and most tools as well as many games are FOSS, however the library contains a few proprietary pieces of code (short vendor source code without license), though these are almost certainly not harmful and could easily be replaced. Schematics and printable STL files are available, though license seems to be non-present. No Pokitto trademarks were surprisingly found



during brief search.

**Downsides** of Pokitto are that the community is an open source community rather than free software one, purists like us will find they lean towards bloated solutions even though the technical limitation of the console largely prevent their implementation. The web forums runs on discourse and requires JavaScript for interactivity. Discord is also actively used for communication, even though some community members bridged it to free alternatives. The official library is relatively bloated and even contains some small pieces of unlicensed code from the MCU manufacturer -- they are very simple assembly snippets that may be easily replaceable, but we should be cautious even about this. Anyway, a reasonably dedicated programmer might create a suckless Pokitto library without greater problems.

Some quite nice hacks were achieved with Pokitto, e.g. using it as a display for a PC or even running GameBoy games on it -- this was done thank to a small FOSS GameBoy emulator and a tool that packs this emulator along with selected GameBoy ROM into a Pokitto executable -- this of course comes with some limitations, e.g. on sound or game size. Yes, Pokitto quite comfortably runs Anarch.

## How To, Tips'N'Tricks

TODO

**Uploading** programs to Pokitto under GNU/Linux can be done e.g. with dd (or mcopy etc.) like this:

```
sudo mount /dev/sdb ~/mnt
sudo dd bs=256 conv=nocreat,notrunc,sync,fsync if=~/git/Anarch/bin/Anarch_pokitto_nonoverclock_1-01.bin of=~/mnt
sudo umount ~/mnt
```

---

political\_correctness

## Political Correctness

*The issue is not my language but your ego.*

Political correctness (abbreviated PC) stands for pseudoleftist censorship and propaganda forced into language, thinking, science, art and generally all of culture, officially justified as "protecting people from getting offended". It's a political tool serving mostly as a weapon and vehicle for populism, a concept allowing creation of political capital by taking advantage of the events of the second World War, similarly to how for example religious ideas are twisted and turned for justifying political decisions completely incompatible with the religion in question. Political correctness does an immense harm to society as it is an artificially invented "issue" that not only puts people and science under heavy control, surveillance, censorship and threat of punishment, normalizing such practice, but also destroys culture, freedom of art and research and creates a great conflict between those who conform and those who value truth, freedom of art, science and communication, not talking about burdening the whole society with yet another competitive bullshit that doesn't have to exist at all. Political correctness is mainly a political tool that allows elimination (so called cancelling) and discrediting opposition of pseudoleftist political movements and parties, as well as brainwashing and thought control (see e.g. Newspeak), and as such is criticized both by rightists and leftists (see e.g. leftypol).

*Example of politically correct ASCII art. Note the absence of any content that might offend someone. Still the art is imperfect because it has a white background which might be seen as racially offensive.*

The whole idea is basically about declaring certain words, pictures, patterns of behavior and similar things as inherently "offensive" to specific selected minorities (currently mostly women, gay, negros and other non-white races, trannies, fat and retarded people), even outside any context, and about constantly fabricating new reasons to get offended so as to fuel the movement that has to ride on hysteria. For example the word black box is declared as "offensive" to black people because... well, like, black people were discriminated at some point in history and their skin is black... so... the word black now can't be said? :D WTF. A sane mind won't understand this because we're dealing with a literal extremist cult here. It just keeps getting more ridiculous, for example feminists want to remove all words that contain the substring "man" from the language because... it's a male oppression? lol... anyway, we can no longer use words like snowman, now we have to say snowperson or something :D Public material now does best if it doesn't walk on the thin ice of showing people with real skin color and better utilize a neutral blue people :D Fuck just kill me already lmao. This starts to get out of hand as fuck, SJWs started to even push the idea that in git the default branch name, master, is offensive, because well, the word has some remote connection to some history of oppression, so they pushed for its change and achieved it, which practically caused a huge mess and broke many git projects -- this is what they do, there was literally not a single reason for the change, they could have spent their energy on actually programming something nice, but they rather used it on breaking what already exists just to demonstrate their political power. What's next? Will they censor the word "chain" in terms like toolchain or blockchain because chains have something to do with slavery? Will they order to repaint the ISS from white to black because the color white is oppressive? The actual reason for this apparent stupidity is at this point not anyone's protection (probably not even themselves believe it anymore) but rather **forcing submission** -- it's the same psychological tactic used by any oppressor: he just gives a nonsensical order, like "start barking like a dog!", to see who blindly conforms and who doesn't -- those who don't are just eliminated right away and those who conform out of fear have their will broken, they will now blindly obey the ruler without thinking about the sanity of his orders.

While political correctness loves to boast about "diversity" and somehow "protecting it", it is doing the exact opposite -- **political correctness kills diversity in society**, it aims for **a unified, sterile society** that's afraid of even hinting on someone else's difference out of fear of punishment. People are different, stereotypes are based on reality, acknowledging this -- and even joking about it -- doesn't at all mean we have to start to hate each other (in fact that requires some fucked up mental gymnastics and a shitty society that pushes competitive thinking), diversity is good, keeps us aware of strength in unity: everyone is good at something and bad at something, and sometimes we just do things differently, a westener might approach problem differently than Asian or Arab, they look different, think and behave differently, and that's a good thing; political correctness forbids such thinking and only states "there is no such thing as differences in people or culture, don't even dare to hint on it", it will go on to censor anything showing the differences do actually exist and leave nothing but plain white sheet of paper without anything on it, a robotic member of society that's afraid to ask someone about his gender or even place where he comes from, someone unable of thinking or communicating on his own, only resorting to preapproved "safe" ways of communication. Indeed, reality yet again starts beating dystopian fiction horrors.

**Political correctness goes strictly against free speech**, it tries to force people "to behave" and be afraid of words and talking, it creates conflict, divides society (for dividing the working class it is criticized e.g. by Marxists) and also TEACHES people to be offended by language -- i.e. even if a specific word wouldn't normally be used or seen in a hostile way (e.g. the master branch in git repositories), political correctness establishes that NOW IT IS OFFENSIVE and specific minorities SHOULD take offense, even if they normally wouldn't, supporting offended culture and fight culture. I.e. political correctness can be called a cancer of society. **LRS must never adhere to political correctness!**

Of course, political correctness doesn't stop at censoring simple words, don't get mistaken. Facts in textbooks and encyclopedias such as those regarding race and sex differences are censored and replaced with lies with the help of soyence. Political correctness tries to forcefully dictate standards of a culture by an extremely rapidly changing fashion, e.g. the standard of beauty, politeness and so on -- last week we celebrated the international gender fluid day but THIS WEEK we celebrate fat disabled women with acne issues, all TV ads must have at least one crippled landwhale or else you're cancelled. If you can't keep up with their latest inventions you'll be executed -- on no, you used the term "mentally ill"! HOW DARE YOU THAT'S SO OFFENSIVELY AGGRESSIVE YOU HAVE TO SAY NEURODIVERGENT, you're basically Hitler now (but wait until next week when the word neuro itself becomes offensive).

OK, let's get back to a bit more serious. Just for the autistic neuroretarded people persons that might misunderstand our stance on social equality: LRS is for complete social equality of all people and eventually all living beings, however political correctness has nothing to do with achieving this goal, in fact it mostly goes against it, it creates huge amount of collateral damage, it divides people and fuels social conflict rather than calm it. We try to not cure symptoms of a shit society by harmful means but rather address the root cause by transitioning to a good society without conflict where there is no need for censorship, fact distortion and brainwashing to prevent discrimination. In the society we envision accepting facts about physical inequality does not imply an attack or discrimination at all as humans don't compete by their abilities, in such society the idea of political correctness is as ridiculous as e.g. arguing we should be creating numerically more inclusive datasets with higher leading digits as by Benford's law smaller digits are a statistical majority that oppresses higher digits.

Political correctness comes with a funny little phenomenon in a form of constant bullshit cycle of **banning old words that gained negative connotation and forcing newly invented clean-slate words** -- in the past when official medical terms such as *idiot*, *imbecile*, *moron*, *cretin* and *mongoloid* started to be seen as "offensive", a new, politically correct term *mental retardation* was invented to replace them -- of course, the term *retardation* later became seen as offensive too so they had to invent new terms, one of the newest ones seem to be *neurodivergency* -- this term will itself become highly offensive in about 10 to 20 years. At that time it will be extremely funny to browse the web archive and seeing people proudly proclaiming they are "neurodivergent" when at the time it will be seen as if they nowadays proclaimed they are a retarded idiot :D The term autism currently seems to be going through the transition from politically correct to "offensive", the Internet already made the word "autistic" largely synonymous to "stupid". Giving in to these trends is not just harmful by giving approval to the idea of language control, it's also just plain stupid, just as following any kind of fashion.

Yet another harmfulness of political correctness is by making people too focused on shallow words instead of focusing on real issues. It's not just the harsh punishments for saying certain banned words, even if they wouldn't do much actual damage, it is also the opposite -- inventing new words and offering them as "solutions" to issues. For example it's completely absurd how some very old people who have been outsiders for their whole lives because they're simply weird, shy, too stupid or smart, are suddenly offered a supposed comfort by being told: "you're not weird, you're just neurodivergent!" The guy is like "OH MY GOD, my whole life I have suffered, I thought I was just not good with people, but in fact I was just neurodivergent my whole life -- if only I knew back then!". The amount of stupidity is incredible.

Let us now compare how we, LRS, approach the issue of "getting offended" versus how the pseudoleft does it. We start with the "problem": people are getting offended. What do we do?

- **solution A: the LRS way:**

1. Teach people to not get offended and if they still do it's not a big deal.
2. solved

- **"solution" B: political correctness:**

1. Create committees who decide what is offensive and what isn't, it will be perpetually at work as new offensive terms appear.
2. Create committees looking over the committees so that corruption is more difficult (but still inevitable, in practice just postponed).
3. Create propaganda, invest great resources into awareness campaigns, train propaganda popularizers. They will have to be constantly at work.
4. Create new laws that take the new definitions of "offense" into account. Employ people who will be constantly updating the laws as the base committee definitions are updated.
5. Train new lawyers, judges, "experts" and investigators to bring the laws to practice.
6. Develop new technology for censorship of politically incorrect material -- this will do collateral damage but that's inevitable.
7. Train and employ new people to maintain, operate and further develop said technology. They will have to be constantly at work.
8. Create political parties, add fuel to the fire, make movies, censor bad movies, make COCs, rewrite old books, pay influencers, punish nonconformists, make flags, buy studies, ...
9. *hundreds more steps here*
10. ... Is the problem solved now? No, but a bunch of new offensive words appeared while half of population wasted their lives on complete bullshit. Jump to step 1.

It's pretty clear political correctness is not a solution, it's just a newly spawned perpetual business.

Latest trend on social media seems to be the **pseudo political incorrectness**, i.e. pretending to be politically incorrect by just using for example rude words, which of course aren't politically incorrect by themselves, for the sake of improving one's image. It goes along the lines of "IN TIMES OF POLITICAL CORRECTNESS, this SUPER HERO CELEBRITY IS A COOL NONCONFORMING REBEL who is not afraid of saying the word *fuck* in a youtube video". Of course this is retarded as fuck, political incorrectness isn't at all about being rude, it is only about opposing the gospel of pseudoleftists, you may be politically incorrect without being rude simply by saying "white women are less intelligent than white men but more intelligent than black men", and you may be rude without being politically incorrect, e.g. by saying "we should murder this white guy because he is white so fuck him".

{ LMAO I just almost shat myself, I heard someone in a video say "vertically challenged", with a COMPLETELY SERIOUS FACE -- I was like WTF is that, it was a bit of a headscratcher for me till I looked it up on the net and found it just means someone who is short :D I fucking can't anymore with this shit. Bruh I wonder what horizontally challenged would mean, is it like someone who's real fat? ~drummyfish }

## See Also

- [orthodoxy](#)
- [stupidity](#)
- [censorship](#)

---

portability

## Portability

Portable software is software that is easy to port to (make run on) other platforms. Platforms here mean anything that serves as an environment enabling software to run, i.e. hardware platforms (CPUs, ISAs, game consoles, ...), different operating systems vs bare metal, fantasy consoles etc. **Portability is an extremely important attribute of good software** as it allows us to write the program once and then run it on many different computers with little effort -- without portability we'd be constantly busy rewriting old programs to run on new computers, portability allows us to free our programs from being tied to specific computers and exist abstractly and independently and so become future proof. Examples of highly portable programs include Anarch, Simon Tatham's Portable Puzzle Collection, sbase (suckless) implementation of Unix tools such as cat and cmp etc. (one wisdom coming from Unix development actually states that portability should be favored even before performance).

**Portability is different from mere multiplatformness:** multiplatform software simply runs on more than one platform without necessarily being designed with high portability in mind; portable software on the other hand possesses the inherent attribute of being designed so that very little effort is required to make it run on wide range of general platforms. Multiplatformness can be achieved cheaply by using a bloated framework such as the Godot engine or QT framework, however that will not achieve portability; on the contrary it will hurt portability. Portability is achieved through good and careful design, efficient code and avoiding dependencies and bloat.

In connection to software the word *portable* also has one other meaning used mainly in context of Windows programs: it is sometimes used for a binary executable program that can be run without installing (i.e. it can be carried around and ran from a USB drive etc.). However we'll stick to the previously defined meaning.

## How To Make Portable Programs

In short: use abstraction (only necessarily small amount) to not get tied to any specific platform (separate frontend and backend), **keep it simple, minimize dependencies** (minimize use of libraries and requiring hardware such as floating point unit or a GPU, have fallbacks), write efficient, simple code (lower hardware demands will support more platforms), avoid platform-specific features (don't write in assembly as that's specific to each CPU, don't directly use Linux syscalls as these are specific to Linux etc.). Also use **self hosting** (i.e. write your programming language in itself etc.) to make your program **self contained** and

minimize dependencies on anything external.

Remember, portability is about **making it easy for a programmer to take your program and make it run elsewhere**, so portability is kind of a mindset, it is about constantly putting oneself in the shoes of someone else with a very different computer and asking questions such as "how hard will it be to make this work if this library isn't available?" and "how hard would it be make this work in a desert?"; see also **bootstrapping**. Even things that are supposed or commonly expected to be present on all platforms, such as a file system or a raster screen, may not be present on some computers -- always remember this.

**Do NOT use big frameworks/engines** -- it is one of the greatest misconceptions among many inexperienced programmers to think portable software is created with big frameworks, such as the Godot engine or the QT framework, which can "single click" export/deploy software to different platforms. This will merely achieve creating a badly bloated multiplatform program that's completely dependent on the framework itself which drags along hundreds of dependencies and wastes computing resources (RAM, CPU, storage, ...) which are all factors directly contradicting portability. If you for example create a snake game in Godot, you won't be able to port it to embedded devices or devices without an operating system even though the snake game itself is simple enough to run on such devices -- the game drags along the whole Godot engine which is so huge, complex and hardware demanding that it prevents the simple game from running on simple hardware.

**The same goes for languages and libraries**: do NOT use big/bloated languages such as Python, Java or JavaScript -- your program would immediately become dependent on a hugely complex ecosystem of such language. For portability you should basically **only write in C** (the best established, time tested, relatively simple language supported basically by every platform) or in C++ at worst, and even with these languages do NOT use the newer standards as these hugely limit the number of compliant compilers that will be able to compile your program. The best is to write in C89 or C99 standard of C. **Minimize the number of libraries you use**, even if it is the standard library of your language -- not all compilers fully adhere to standards and some don't have the standard library even if they should. For shell scripts only use **posix shell**, i.e. only use constructs, utilities and flags/features defined by the posix standard, even if you have more "powerful" shell and utilities like Bash and GNU utils.

{ A great example of how avoiding C features can help your programs be more portable can be seen with Dusk OS, a very small operating system that will likely be THE system we use if (or rather when) the collapse strikes. The system is implementing what they call "Almost C" (<https://git.sr.ht/~vdupras/duskos/tree/master/fs/doc/cc/index.txt>) -- a language trying to be close to C but avoiding standard compliance to keep simplicity. They want to port C programs but HAVE TO keep it simple so they just can't implement full C and when the judgement day comes, the programs that don't rely on much will simply be the ones that survive. If you just hide behind the excuse "the feature is in the standard so IT HAS TO BE IMPLEMENTED", your program will end up more unlikely to be ported, an old piece of paper saying your program should run simply won't matter. In Dusk OS you can actually see this porting effort happening right now. ~drummyfish }

In your compiled programs **always make your own thin I/O abstraction, decouple your I/O libraries, separate frontend and backend**. This is one of the most basic and most important things to do. Why? Well unless you're writing a library, you will need to use I/O (write out messages, draw to screen, create GUI, read keyboard commands, read from files, read from network, ...) so you will NEED to use some library for this (C stdlib, SDL, OS syscalls, Xlib, ...) but you absolutely DON'T WANT this library to become a hard dependency of your program because if your program depends let's say on SDL, you won't be able to make your program run on platforms that don't have SDL. So the situation is that you HAVE TO use some I/O library but you don't want to become dependent on it.

The way to solve this is to create your own small I/O abstraction in your project, i.e. your own functions (such as drawPixel, writeMessage, keyPressed, playSound, readFile etc.) for performing I/O, which you will use inside your main program. These functions will be defined in a small file which will basically be your own small I/O library just for your program. The functions you define there will then internally use functions of whatever underlying I/O system you choose to use at the time as your frontend (SDL, Xlib, SFML, ...); the important thing is that your main program code won't itself depend on the underlying system, it will only depend on your I/O abstraction, your own functions. Your custom I/O functions will depend on the underlying I/O system but in a way that's very easy to change -- let's say that your keyPressed function internally uses SDL's SDL\_GetKeyboardState to read keyboard state. If you want to switch from using SDL to using a

different frontend, you will only have to change the code in one place: in your I/O abstraction code, i.e. inside your `keyPressed` function. E.g. if you switch from SDL to SFML, you will just delete the code inside your `keyPressed` function and put in another code that uses SFML functions to read keyboard (e.g. the `isKeyPressed` attribute), and your whole code will instantly just work on SFML. In fact you can have multiple implementations of your functions and allow switching of different backends freely -- just as it is possible to compile a `C` program with any C compiler, you can make it possible to compile your program with any I/O frontend. If you used SDL's specific functions in your main code, you would have to completely rewrite your whole codebase if you wanted to switch away from SDL -- for this reason your main code must never directly touch the underlying I/O system, it must only do so through your I/O abstraction. Of course these principles may apply to any other thing that requires use of external libraries, not just I/O.

This is all demonstrated by [LRS](#) programs such as [Anarch](#) or [SAF](#), you can take a look at their code to see how it all works.

Anyway the following is a simple `C` code to demonstrate the abstraction from an I/O system -- it draws a dithered rectangle to the screen and waits until the user pressed the `q` key, then ends. The main code is written independently of any I/O system and can use either `C stdlib` (`stdio`, draws the rectangle to terminal with ASCII characters) or `SDL2` (draws the rectangle to actual window) as its frontend -- of course more frontends (e.g. one using `Xlib` or `SFML`) can be added easily, this is left as an exercise :)

```
#define SCREEN_W 80
#define SCREEN_H 30

// our I/O abstraction:
void ioInit(void);      // init our I/O
void ioEnd(void);       // destroy our I/O
void drawPixel(int x, int y, int white);
void showImage(void);
int  isKeyPressed(char key);

// our main program code:
int main(void)
{
    ioInit();

    for (int y = 3; y < 20; ++y) // draw dithered rectangle
        for (int x = 30; x < 60; ++x)
            drawPixel(x,y,x % 2 == y % 2);

    showImage();

    while (!isKeyPressed('q')); // wait for pressing 'q'

    ioEnd();

    return 0;
}

/*-----
implementation of our I/O abstraction for different
frontends: */

#ifdef FRONTEND_STDLIB // C stdio terminal frontend
#include <stdio.h>
char screen[SCREEN_W * SCREEN_H];

void ioInit(void)
{
    // clear screen:
    for (int i = 0; i < SCREEN_W * SCREEN_H; ++i)
        screen[i] = 0;
}

void ioEnd(void) { } // nothing needed here

void drawPixel(int x, int y, int white)
{
    screen[y * SCREEN_W + x] = white != 0;
}
```

```

}

void showImage(void)
{
    for (int i = 0; i < SCREEN_W * SCREEN_H; ++i)
    {
        if (i % SCREEN_W == 0)
            putchar('\n');

        putchar(screen[i] ? '#' : '.');
    }

    putchar('\n');
}

int isKeyPressed(char key)
{
    return getchar() == key;
}
#ifdef FRONTEND_SDL // SDL2 frontend
#include <SDL2/SDL.h>
unsigned char screen[SCREEN_W * SCREEN_H];
SDL_Window *window;
SDL_Renderer *renderer;
SDL_Texture *texture;

void ioInit(void)
{
    for (int i = 0; i < SCREEN_W * SCREEN_H; ++i)
        screen[i] = 0;

    SDL_Init(0);

    window = SDL_CreateWindow("sdl", SDL_WINDOWPOS_UNDEFINED,
        SDL_WINDOWPOS_UNDEFINED, SCREEN_W, SCREEN_H, SDL_WINDOW_SHOWN);

    renderer = SDL_CreateRenderer(window, -1, 0);

    texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGB332,
        SDL_TEXTUREACCESS_STATIC, SCREEN_W, SCREEN_H);
}

void ioEnd(void)
{
    SDL_DestroyTexture(texture);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
}

void drawPixel(int x, int y, int white)
{
    screen[y * SCREEN_W + x] = (white != 0) * 255;
}

void showImage(void)
{
    SDL_UpdateTexture(texture, NULL, screen, SCREEN_W);

    SDL_RenderClear(renderer);
    SDL_RenderCopy(renderer, texture, NULL, NULL);
    SDL_RenderPresent(renderer);
}

int isKeyPressed(char key)
{
    SDL_PumpEvents();
    const unsigned char *keyboard = SDL_GetKeyboardState(NULL);

    return keyboard[SDL_SCANCODE_A + (key - 'a')];
}
#endif

```

If you compile this code as

```
gcc -DFRONTEND_STDLIB main.c
```

You'll get the stdlib version. If you compile it as

```
gcc -DFRONTEND_SDL -lSDL2 main.c
```

You'll get the SDL version.

A great example of this kind of portable design can be seen e.g. in well written **compilers** that separate their architecture into an frontend and backend -- imagine we are writing for example a C compiler. The parser of C syntax can be easily written in a portable way, we simply write functions that work with text, however we find difficulty in asking what instruction set we will compile to. If we choose one, such as x86, then we will not only write an x86 specific code generator, but also e.g. an x86 specific optimizer; the part of the compiler that may get so complex that it ends up being bigger than the rest of the code. What if then we also want to support another ISA such as Arm or RISC-V, will we have to rewrite our painstakingly written optimizer for those architectures from scratch? The solution is the same as explained above in regards to I/O: we make an abstraction above the instruction set, here called an intermediate representation, usually some bytecode, i.e. the compiler first translates C to the abstract bytecode, then we may perform all the complex optimizations on this bytecode, and only then, in the last moment, we relatively simply translate this bytecode to whatever specific instruction set.

Programming languages, operating systems and other "platforms" also usually employ self hosting to greatly increase portability -- you will most often see a serious programming language written in itself and if not, then at very least e.g. its standard library will be written as such. See also bootstrapping.

---

portal\_rendering

## Portal Rendering

{ I haven't yet gotten to implementing a portal renderer so it's possible I make a wrong claim here by mistake, but I'll try not to :) ~drummyfish }

Portal rendering is a method of 3D rendering that treats the rendered environment as spaces (e.g. rooms) connected by portals (doors, windows, ...) which allows fast and simple determination of visibility and therefore fast and simple rendering. It was a quite popular way of 3D rendering for example in the old 1990s 3D games such as Descent and Duke Nukem.

The **basic general idea** is to represent the 3D environment as a set of "rooms" (generally any subdivision unit of space, not just "house rooms" of course) and their connections to other rooms through portals ("holes", shared walls through which one room connects to another); then when rendering we simply draw the room the camera resides in (from the inside) and proceed to draw the rooms that are connected by portals which are now visible on the screen, treating each of those portals as a kind of new smaller screen (i.e. a clipping window). Then we go on to recursively draw portals in those rooms again etc. until some terminating condition is met (e.g. all screen pixels are covered or we have reached maximum draw depth etc.). A limitation imposed on a room is often that it has to be convex so that its "from the inside" rendering is simple; non-convex spaces are then simply split into multiple convex ones -- EZ.

Just as similar methods like raycasting and BSP rendering, portal rendering can be used in various ways, it is not a simple algorithm but rather a method, approach to rendering. It may also be used just as a "helper" for visibility determination in another method. Notably there is a "full 3D" (*Descent*) and "2D" (*Duke Nukem*), sector based version of portal rendering. They are all based on the same principle but may have different limitations etc.

**Advantages** of portal rendering:

- **It can work without precomputation**, which is a huge plus compared e.g. to BSP rendering (though optional precomputations such as PVS can of course be always employed). This among others



saves time (precomputing can take a while), program complexity and space (no need to store extra precomputed data).

- **The environment can be dynamic** (change on the fly, consider e.g. destructible or animated environment), thanks to not needing precomputed data. This was made advantage of in Build engine games a lot, while in Doom only wall and ceiling height could change on the run.
- **Impossible geometry can be created** -- as we may create any arbitrary spaces that connect to each other, it is possible to for example create a house that's bigger on the inside than on the outside, or a curved tunnel that would in reality intersect itself but doesn't. We can even have **room above room in the 2D version** (though in vanilla version there can't be two VISIBLE rooms above one another).
- **Effect such as mirrors are easy** -- a mirror may be just a portal that connects to the same room in opposite way.
- **There is no overdraw**, a problem that plagues many 3D renderers, so **we don't need z-buffer** and may probably hack the method to not even need a frame buffer. This is pretty awesome, may reduce memory requirements greatly and allow things such as frameless rendering. However z-buffer and double buffering are still mostly used so as to allow additional correct rendering of overlays to the environments, e.g. "2D sprites".
- For mentioned reasons the method is relatively simple, efficient and software rendering friendly, making it a good candidate for weak computers.
- ...

TODO

## 2D Portal Rendering

TODO

pride

# Pride

Pride is an extremely harmful emotion defined as a feeling of superiority, greatly connected to fascism (e.g. nationalism, gay fascism, woman fascism etc.), it is the opposite of humility. Pride is always bad, even small amounts do excessive amount of evil. Flags, statues, sense of identity, egoism, narcissism, hero worship and self praise are all connected to pride.

prime

## Prime Number

Prime number (or just *prime*) is a whole positive number only divisible by 1 and itself, except for the number 1. I.e. prime numbers are 2, 3, 5, 7, 11, 13, 17 etc. Non-prime numbers are called *composite numbers*. Ask any mathematician and you'll learn that prime numbers are more than just highly important, they are interesting and mysterious for their intricate properties and distribution among other numbers, they have for millennia fascinated mathematicians; nowadays they are studied in the math subfield called number theory. Primes are also of practical use, for example in asymmetric cryptography. Primes can be seen as the opposite of highly composite numbers (also antiprimes, numbers that have more divisors than any lower number). Numbers of comparable status and similarly mysterious properties to prime numbers are for example perfect numbers, whose importance is however a bit diminished by current lack of practical use.

##.#.#...#.#...#.#...#...#.#...#.#.#...#...#...#.#...#...

Prime number positions up to 70.

The largest known prime number as of 2022 is  $2^{82589933} - 1$  (it is so called Mersenne prime, i.e. a prime of form  $2^N - 1$ ).

Every natural number greater than 1 has a unique **prime factorization**, i.e. a set of prime numbers whose product it is. For example 75 is a product of three primes:  $3 * 5 * 5$ . This is called the *fundamental theorem of arithmetic*. Naturally, each prime has a factorization consisting of a single number -- itself -- while factorizations of non-primes consist of at least two primes. To mathematicians prime numbers are what chemical elements are to chemists -- a kind of basic building blocks.

**Why is 1 not a prime?** Out of convenience -- if 1 was a prime, the fundamental theorem of arithmetic would not hold because 75's factorization could be  $3 * 5 * 5$  but also  $1 * 3 * 5 * 5$ ,  $1 * 1 * 3 * 5 * 5$  etc. It also makes sense under some different definitions -- imagine for example we create a tree of numbers, assign each number  $N$  a parent number  $M$  which is the maximum of all  $N$ 's divisors that we check from 1 (including) to  $N$  (excluding); in this tree prime numbers are all numbers in depth 1, i.e. those that are direct children of 1, but 1 itself is not at this level, it's at the root, having no parent (as it would be its own parent), so by this definition 1 is also not a prime.

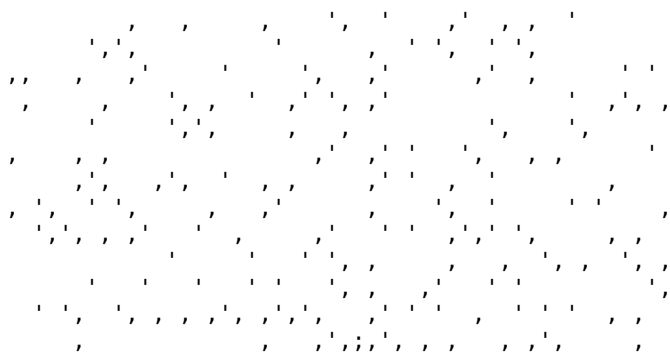
The unique factorization can also nicely be used to encode multisets as numbers. We can assign each prime number its sequential number (2 is 0, 3 is 1, 5 is 2, 7 is 3 etc.), then any number encodes a set of numbers (i.e. just their presence, without specifying their order) in its factorization. E.g.  $75 = 3 * 5 * 5$  encodes a multiset  $\{1, 2, 2\}$ . This can be exploited in cool ways in some cyphers etc.

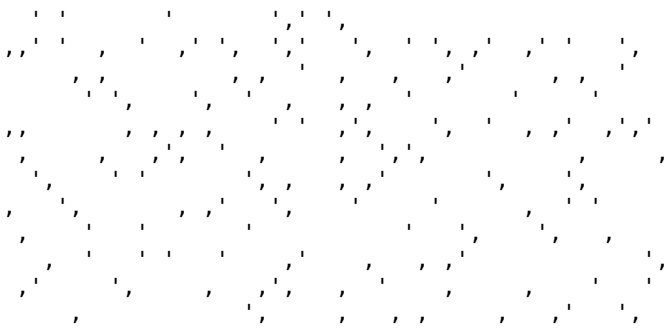
When in 1974 the Arecibo radio message was sent to space to carry a message for aliens, the resolution of the bitmap image it carried was chosen to be  $73 \times 23$  pixels -- two primes. This was cleverly done so that when aliens receive the 1679 sequential values, there are only two possible ways to interpret them as a 2D bitmap image:  $23 \times 73$  (incorrect) and  $73 \times 23$  (correct). This increased the probability of correct interpretation against the case of sending an arbitrary resolution image.

**There are infinitely many prime numbers.** The proof is pretty simple (shown below), however it's pretty interesting that it has still not been proven whether there are infinitely many **twin primes (primes that differ by 2)**, that seems to be an extremely difficult question. Another simple but unproven conjecture about prime numbers is Goldbach's conjecture stating that every even number greater than 2 can be written as a sum of two primes.

Euklid's proof shows there are infinitely many primes, it is done by contradiction and goes as follows: suppose there are finitely many primes  $p_1, p_2, \dots, p_n$ . Now let's consider a number  $s = p_1 * p_2 * \dots * p_n + 1$ . This means  $s - 1$  is divisible by each prime  $p_1, p_2, \dots, p_n$ , but  $s$  itself is not divisible by any of them (as it is just 1 greater than  $s - 1$  and multiples of some number  $q$  greater than 1 have to be spaced by  $q$ , i.e. more than 1). If  $s$  isn't divisible by any of the considered primes, it itself has to be a prime. However that is in contradiction with the original assumption that  $p_1, p_2, \dots, p_n$  are all existing primes. Therefore a finite list of primes cannot exist, there have to be infinitely many of them.

**Distribution and occurrence of primes:** the occurrence of primes seems kind of random (kind of like digits of decimal representation of  $\pi$ ), without a simple pattern, however hints of patterns appear such as the Ulam spiral -- if we plot natural numbers in a square spiral and mark the primes, we can visually distinguish dimly appearing 45 degree diagonals as well as horizontal and vertical lines. Furthermore the **density of primes decreases** the further away we go from 0. The *prime number theorem* states that a number randomly chosen between 0 and  $N$  (for large  $N$ ) has approximately  $1/\log(N)$  probability of being a prime. **Prime counting function** is a function which for  $N$  tells the number of primes smaller or equal to  $N$ . While there are 25 primes under 100 (25%), there are 9592 under 100000 (~9.5%) and only 50847534 under 1000000000 (~5%).

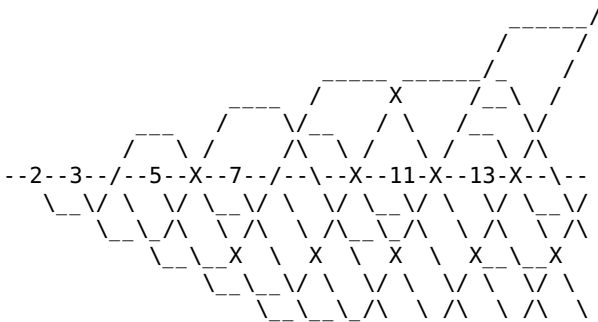




*Ulam spiral: the center of the image is the number 1, the number line continues counter clockwise, each point represents a prime.*

Here are prime numbers under 1000: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997.

Here are twin prime numbers under 1000: 3, 5, 7, 11, 13, 17, 19, 29, 31, 41, 43, 59, 61, 71, 73, 101, 103, 107, 109, 137, 139, 149, 151, 179, 181, 191, 193, 197, 199, 227, 229, 239, 241, 269, 271, 281, 283, 311, 313, 347, 349, 419, 421, 431, 433, 461, 463, 521, 523, 569, 571, 599, 601, 617, 619, 641, 643, 659, 661, 809, 811, 821, 823, 827, 829, 857, 859, 881, 883.



There also exists a term **pseudoprime** -- it stands for a number which is not actually a prime but appears so because it passes some quick primality tests.

**Higher order primes**, also **superprimes** or prime-indexed primes, are primes that occupy prime numberth position within prime numbers, i.e. one of first higher order primes is for example number 5 because it is the 3rd prime and 3 itself is a prime. 5 is also one of second order higher primer numbers because it is 2nd first higher order prime number and 2 is a prime number. Etc. So we may generalize this concept to a prime number order  $R(x)$ , which says the highest order that number  $x$  achieves in this sense, with  $R(x) = 0$  meaning  $x$  is not prime at all. One of very high superprimes is for example number 174440041 (lowest number with  $R(x) = 12$ ). Prime orders for numbers up to 1000 are (leaving out the ones with order 0):

2: 1, 3: 2, 5: 3, 7: 1, 11: 4, 13: 1, 17: 2, 19: 1, 23: 1, 29: 1, 31: 5, 37: 1, 41: 2, 43: 1, 47: 1, 53: 1, 59: 3, 61: 1, 67: 2, 71: 1, 73: 1, 79: 1, 83: 2, 89: 1, 97: 1, 101: 1, 103: 1, 107: 1, 109: 2, 113: 1, 127: 6, 131: 1, 137: 1, 139: 1, 149: 1, 151: 1, 157: 2, 163: 1, 167: 1, 173: 1, 179: 3, 181: 1, 191: 2, 193: 1, 197: 1, 199: 1, 211: 2, 223: 1, 227: 1, 229: 1, 233: 1, 239: 1, 241: 2, 251: 1, 257: 1, 263: 1, 269: 1, 271: 1, 277: 4, 281: 1, 283: 2, 293: 1, 307: 1, 311: 1, 313: 1, 317: 1, 331: 3, 337: 1, 347: 1, 349: 1, 353: 2, 359: 1, 367: 2, 373: 1, 379: 1, 383: 1, 389: 1, 397: 1, 401: 2, 409: 1, 419: 1, 421: 1, 431: 3, 433: 1, 439: 1, 443: 1, 449: 1, 457: 1, 461: 2, 463: 1, 467: 1, 479: 1, 487: 1, 491: 1, 499: 1, 503: 1, 509: 2, 521: 1, 523: 1, 541: 1, 547: 2, 557: 1, 563: 2, 569: 1, 571: 1, 577: 1, 587: 2, 593: 1, 599: 3, 601: 1, 607: 1, 613: 1, 617: 2, 619: 1, 631: 1, 641: 1, 643: 1, 647: 1, 653: 1, 659: 1, 661: 1, 673: 1, 677: 1, 683: 1, 691: 1, 701: 1, 709: 7, 719: 1, 727: 1, 733: 1, 739: 2, 743: 1, 751: 1, 757: 1, 761: 1, 769: 1, 773: 2, 787: 1, 797: 2, 809: 1, 811: 1, 821: 1, 823: 1, 827: 1, 829: 1,

839: 1, 853: 1, 857: 1, 859: 2, 863: 1, 877: 2, 881: 1, 883: 1, 887: 1, 907: 1, 911: 1, 919: 3, 929: 1, 937: 1, 941: 1, 947: 1, 953: 1, 967: 2, 971: 1, 977: 1, 983: 1, 991: 2, 997: 1.

**Prime gaps:** statistically gaps between consecutive primes increase. The size of the gaps themselves make another number sequence that starts like this 1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2, 6, 4, 2, 6, 4, 6, 8, 4, 2, 4, 2, 4, 14, 4, 6, 2, 10, 2, 6, 6, 4, 6, 6, 2, 10, 2, 4, 2, 12, 12, 4, 2, 4, 6, 2, 10, 6, 6, 6, 2, 6, 4, 2, 10, 14, 4, 2, 4, 14, 6, 10, 2, 4, 6, 8, 6, 6, 4, 6, 8, 4, 8, 10.

**Fun with primes:** thanks to their interesting, mysterious and random nature, primes can be played around -- of course, you can examine them mathematically, which is always fun, but you can also play sort of games with them. For example the prime race: you make two teams of primes, one that gives 1 modulo 4, the other one that gives 3; then you go prime by prime and add points to each team depending on which one the prime falls in; the interesting thing is that team 3 is almost always in lead just by a tiny amount (this is known as Chebyshev bias, only after 2946 primes team 1 gets in the lead for a while, then at 50378 etc.). Similar thing can be done by evaluating the Mobius function: set total sum to 0, then go number by number and if it only has unique prime factors, add 1 if the number of those factors is even, otherwise subtract 1 -- see how the function behaves. Of course you can go crazy, make primes paint pictures or compose music -- people also like to do this with digits of numbers, e.g. those of pi or e.

**Can we generalize/modify the concept of prime numbers?** Yeah, sure, why not? The ways are many, we'll rather run into the issue of analysis paralysis -- choosing the interesting generalization of out of the many possible ways. Some possible generalizations include:

- **pseudoprimes:** the above mentioned, i.e. non-primes passing many prime tests.
- **almost primes:** a number is  $n$ -almost prime if it has  $n$  prime factors, so 1-almost primes are just regular primes (they have 1 prime divisor -- themselves) but then there are 2 almost primes like 9 or 15 that are kind of closer to being primes than let's say 5-almost-primes such as 48 or 80. We take the idea of numbers having either none (primes) or some (non-primes) divisors and generalized it by says a number is more prime like if it has fewer divisors.
- Another idea hinted on above: make a tree of numbers with 1 as its root, assign each number a parent that's its greatest divisor (excluding the number itself); in this tree 1 is above prime numbers, prime numbers are on level 1, second level may be seen as the "next best thing" to primes (4, 6, 9, 10, 15, ...), third level the next (8, 12, 18, 27) and so on, i.e. we define the "primeness" as the depth in this tree, the number of times we have to replace the number with its greatest divisor before we get to 1.
- **complex (Gaussian) primes:** This is not a strict generalization because we remove some primes by were primes before, but we may define prime numbers also within complex integers. Here we get primes to be 3, 7, 11, 19, 23 etc.
- Similarly we may try to play on this observation: a non-prime is a number that is divisible by something, i.e. there is some number that when dividing the original number gives remainder after division zero; primes are those for which no number gives remainder zero, but some primes might be considered "weaker" by giving very low or very high remainder such as 1, i.e. being "not quite but almost" divisible by something (of course we have to somehow account for the fact that low divisors can only ever give low remainders) -- ideal prime would have remainders after division near the half of the dividing number (it would dodge multiples of other numbers with some margin), which we can formalize and define kind of "prime strength".
- TODO: generalization to non integers? haven't found anything
- ...

## Algorithms

**Primality test:** testing whether a number is a prime is quite easy and not computationally difficult (unlike factoring the number). A naïve algorithm is called *trial division* and it tests whether any number from 2 up to the tested number divides the tested number (if so, then the number is not a prime, otherwise it is). This can be optimized by only testing numbers up to the square root (including) of the tested number (if there is a factor greater than the square root, there is also another smaller than it which would already have been tested). A further simple optimization is to test division by 2, 3 and then only numbers of the form  $6q \pm 1$  (other forms are divisible by either 2 or 3, e.g.  $6q + 4$  is always divisible by 2). Further optimizations exist and for maximum speed a look up table may be used for smaller primes. A simple C function for primality test may look e.g. like this:

```

int isPrime(int n)
{
    if (n < 4)
        return n > 1;

    if (n % 2 == 0 || n % 3 == 0)
        return 0;

    int test = 6;

    while (test <= n / 2) // replace n / 2 by sqrt(n) if available
    {
        if (n % (test + 1) == 0 || n % (test - 1) == 0)
            return 0;

        test += 6;
    }

    return 1;
}

```

Sieve of Eratosthenes is a simple algorithm to find prime numbers up to a certain bound  $N$ . The idea of it is following: create a list of numbers up to  $N$  and then iteratively mark multiples of whole numbers as non-primes. At the end all remaining (non-marked) numbers are primes. If we need to find all primes under  $N$ , this algorithm is more efficient than testing each number under  $N$  for primality separately (we're making use of a kind of dynamic programming approach).

**Prime factorization:** We can factor a number by repeatedly brute force checking its divisibility by individual primes and there exist many algorithms applying various optimizations (wheel factorization, Dixon's factorization, ...), however for factoring large (hundreds of bits) primes there exists no known efficient algorithm, i.e. one that would run in polynomial time, and it is believed no such algorithm exists (see P vs NP). Many cryptographic algorithms, e.g. RSA, rely on factorization being inefficient. For quantum computers a polynomial ("fast") algorithm exists, it's called Shor's algorithm.

Prime generation: TODO

## See Also

- perfect number
- happy number

---

primitive\_3d

## Primitive 3D

See pseudo 3D.

---

privacy

## Privacy

*We don't give a shit about your privacy.*

Digital privacy is the ability of someone to hide "sensitive" information about himself; nowadays "privacy concerns" are a big part of capitalist bullshit, fear culture and fight culture, and fall under so called computer security, yet greater area of bullshit business. Of course, there are other forms of privacy than digital, for example the physical privacy in real life, however in this article we'll be implicitly dealing with digital privacy unless mentioned otherwise, i.e. privacy with respect to computers, e.g. on the Internet. For starters let's stress the whole business around privacy is bullshit that's wasting energy which could better be spent on actually useful things such as feeding the hungry or curing the ill. Do not engage in privacy hysteria.

{ I have my personal data publicly online and under CC0 for anyone to download and do anything with, including my real name, date of birth, medical info and even nude photos. Literally nothing bad ever happened due to this. ~drummyfish }

Digital privacy can be further categorized. We can talk e.g. about **communication privacy** (emails, chat, ...), **data privacy** (cookies, tracking, medical data, ...), **personal privacy** (intimate photos, sexual orientation, ...), **individual privacy** (identifying information, anonymity, spam, ...) etc. Privacy is also closely related to **cryptography**, as encryption is how information can be "protected" against reaching unauthorized entities, and to free software, as using safe tools with available source code is crucial to avoid malware. Still, to achieve high privacy additional appropriate behavior has to be adopted, e.g. protection against spyware, using proxies and/or onion routing, turning off browser cookies, avoiding fingerprinting, avoiding social networks, avoiding revealing potentially identifying information etc.

**Society is becoming more and more obsessed with privacy and that is EXTREMELY BAD.** It leads to hardcore  censorship, people are hiding their email addresses so it's impossible to contact them, photos of child faces are wiped from the Internet, more and more videos on the internet now just blur everything in the video that's not the main focus of it, "just in case", people are even afraid to credit other people by name even if they are e.g. legally obliged to by a license such as CC-BY-SA (Imao <https://forum.freegamedev.net/viewtopic.php?f=7&t=19322>). Such retardedness has probably never been seen yet.

Do you have **"nothing to hide?"** Tinfoil privacy maximalists absolutely love this sentence, it almost makes them orgasm; don't misunderstand them though, they are psychopaths, they are obsessed people who above everything love to waste their whole lives on playing the hide and seek game, but most importantly they want to drag everyone into the game. Yes -- sadly you've probably been forced to have at least something to hide, for example your password -- it's not that the claim is false, the great mistake and fucked up nature of our world shows in how people interpret such truth and how they react to it. The fact that you have something to hide doesn't mean you should accept it and start focusing on hiding, and, just in case, "hide absolutely everything". This leads to hell, you accept the dystopia AND start supporting it, you buy into endless fear and bullshit, just like when you dedicate your whole life for example to hoarding money -- there will be no such a thing as "moderate privacy", no, privacy maximalists will tell you you must hide absolutely EVERYTHING, even such things as your favorite color or style of speech, because these things might lead to someone guessing your password, fingerprinting you etc. No, this is all absolute insanity, the fact you have to hide something at all shows something is extremely wrong with the society -- if anything, you should try to **fix the society so that you no longer have anything to hide.**

Therefore here is **how to 100% solve privacy**: make it moral to make ALL information public, always, without any censorship, "protection", laws and other bullshit. { NO, it's fucking NOT a joke or "satire", I am 100% serious. More like 3000% actually. It's extremely smart, that's why people don't do it. ~drummyfish } This way passwords will become obsolete, which has a nice side effect of also ending a lot of capitalist bullshit such as banking and intellectual property, people will have to start sharing. Governments and corporations will also start taking extreme advantage of the situation, so people will stop using online technology as much and maybe they'll even finally decide to ditch governments and corporations, another great leap in development of society. People will also stop being concerned about their "private data" -- at first they will be freaking out that everyone can see their dick pics and what porn they jerk off to but since EVERYONE's data will be visible, they will find out that everyone watches weird porn, that everyone has a dick (well, about half of population), and they'll just stop acting like chimps in a while. This literally only has advantages and it solves many of our greatest issues all at once. At this point privacy has been solved. { Leave the Nobel Prize at my door, thanks. ~drummyfish }

{ I'm thinking of a life experiment: start living without a password. In it I would literally make my password public on my website and start to somehow live like that, i.e. I would stop using a bank account, I would stop using social media accounts, would just host my own git repository and email. That doesn't even sound so difficult, I'll probably give it a try one day. ~drummyfish }

**As of 2023 privacy is impossible to achieve** unless you live in wilderness completely independently of the main "civilization". If you use any kind of computer (laptop, TV, phone, car, camera etc.), you are already being watched: basically all CPUs have proven hardware spyware in them capable of bypassing encryption, see Intel ME etc., no matter what operating system you use, and even if you use some obscure CPU without it, you are watched through your Internet activity (even if you use a "secure" browser, which you most likely

don't even if you think you do), your browsing habits are watched and analyzed by highly advanced AI that can track you even without cookies etc., e.g. just from your writing style, patterns of repeated daily activity, mouse movement signature etc. -- all small fragments of information about your activity such as those mentioned above and your locations over time (known from your phone connecting to towers, someone else's phone detecting your voice, street or car camera detecting your face, credit card payments etc.) are connected with other fragments of information (even those of other people) and AI makes a complete picture of your life available to those who need it. You may think you're doing everything right and that they can't find you, but it's enough if e.g. someone from your family posted a picture with you on facebook 10 years ago or if you as a child played online games -- this is enough to know which people you are related to and them being tracked then leads to you also being tracked to a big degree despite you using 7 proxies and living underground. If the government furthermore decides to watch you more (which may happen just because you e.g. try to "protect" your privacy more and start using Tor, which is suspicious), they can just watch you in real time through satellites (even inside buildings) and so on. So you just have to accept you are being watched, and unless we end capitalism, it will only be getting worse (mind reading technology is already emerging).

We have to state that **privacy concerns are a symptom of bad society. We shouldn't ultimately try to protect privacy more (cure symptoms) but rather make a society where need for privacy isn't an issue (cure the root cause)**. This sentiment is shared by many hackers, even Richard Stallman himself used to revolt against passwords when he was at MIT AI Labs; he intentionally used just the password "rms" to allow other people to use his account (this is mentioned in the book *Free As In Freedom*). Efforts towards increasing and protecting privacy is in its essence an unnecessary bullshit effort wasting human work, similarly to law, marketing etc. It is all about censorship and secrecy. Besides this, **all effort towards protecting digital privacy will eventually fail**, thanks to e.g. advanced AI that will identify individuals by pattern in their behavior, even if their explicit identity information is hidden perfectly. Things such as browser fingerprinting are already a standard and simple practice allowing highly successful uncovering of identity of anonymous people online, and research AI is taking this to the next level (e.g. the paper *Detecting Individual Decision-Making Style: Exploring Behavioral Stylometry in Chess* shows revealing chess players by their play style). With internet of stinks, cameras, microphones and smartphones everywhere, advanced AI will be able to identify and track an individual basically anywhere no matter the privacy precautions taken. Curing the root cause is the only option to prevent a catastrophic scenario.

By this viewpoint, LRS's stance towards privacy differs from that of many (if not most) free software, hacker and suckless communities: to us **privacy is a form of censorship** and as such is seen as inherently bad. We dream of a world without abuse where (digital) privacy is not needed because society has adopted our philosophy of information freedom, non-violence and non-competition and there is no threat of sensitive information abuse. Unlike some other people (so called pragmatics), not only do we dream of it, we actively try to make it a reality. Even though we know the ideally working society is unreachable, we try to at least get close to it by restricting ourselves to bare minimum privacy (so we are very open but won't e.g. publish our passwords). We believe that abuse of sensitive information is an issue of the basic principles of our society (e.g. capitalism) and should be addressed by fixing these issues rather than by harmful methods such as censorship.

---

procgen

## Procedural Generation

Procedural generation (procgen, also PCG -- *procedural content generation* -- not to be confused with procedural programming) refers to creation of data, such as art assets in games or test data for data processing software, by using algorithms and mathematical formulas rather than creating it manually or measuring it in the real world (e.g. by taking photographs). This can be used for example for automatic generation of textures, texts, music, game levels or 3D models but also practically anything else, e.g. test databases, animations or even computer programs. Such data are also called *synthetic*. Procedural art currently doesn't reach artistic qualities of a skilled human artist, but it can be good enough or even necessary (e.g. for creating extremely large worlds), it may be preferred e.g. for its extreme save of storage memory, it can help add detail to human work, be a good filler, a substitute, an addition to or a basis for manually created art. Procedural generation has many advantages such as saving space (instead of large data we only store small code of the algorithm that generates it), saving artist's time (once we have an algorithm we can generate a lot data extremely quickly), parameterization (we can tweak parameters of the

algorithm to control the result or create animation, often in real-time), increasing resolution practically to infinity or extending data to more dimensions (e.g. 3D textures). Procedural generation can also be used as a helper and guidance, e.g. an artist may use a procedurally generated game level as a starting point and fine tune it manually, or vice versa, procedural algorithm may create a level by algorithmically assembling manually created building blocks.

As neural AI approaches human level of creativity, we may see computers actually replacing many artists in near future, however it is debatable whether AI generated content should be called procedural generation as AI models are quite different from the traditional hand-made algorithms -- AI art is still seen as a separate approach than procedural generation. For this we'll only be considering the traditional approach from now on.

Minecraft (or Minetest) is a popular example of a game in which the world is generated procedurally, which allows it to have near-infinite worlds -- size of such a world is in practice limited only by ranges of data types rather than available memory. Roguelikes also heavily utilize procgen. However this is nothing new, for example the old game called Daggerfall was known for its extremely vast procedurally generated world, and even much older games used this approach. Some amount of procedural generation can be seen probably in most mainstream games, e.g. clouds, vegetation or NPCs are often made procedurally.

For its extreme save of space procedural generation is extremely popular in demoscene where programmers try to create as small programs as possible. German programmers made a full fledged 3D shooter called .kkrieger that fits into just 96 kB! It was thanks to heavy use of procedural generation for the whole game content. Bytebeat is a simple method of generating procedural "8bit" music, it is used e.g. in Anarch. Procedural generation is generally popular in indie game dev thanks to offering a way of generating huge amounts of content quickly and without having to pay artists.

We may see procgen as being similar to compression algorithms: we have large data and are looking for an algorithm that's much smaller while being able to reproduce the data (but here we normally go the other way around, we start with the algorithm and see what data it produces rather than searching for an algorithm that produces given data). John Carmack himself called procgen "basically a shitty compression".

Using fractals (e.g. those in a form of L-system) is a popular technique in procgen because fractals basically perfectly fit the definition perfectly: a fractal is defined by a simple equation or a set of a few rules that yield an infinitely complex shape. Nature is also full of fractals such as clouds, mountain or trees, so fractals look organic.

There are also other techniques such as wave function collapse which is used especially in tile map generation. Here we basically have some constraints set (such as which tiles can be neighbors) and then consider the initial map a superposition of all possible maps that satisfy these constraints -- we then set a random tile (chosen from those with lowest entropy, i.e. fewest possible options) to a random specific value and propagate the consequences of it to other tiles causing a cascading effect of collapsing the whole map into one of the possible solutions.

A good example to think of is generating procedural textures -- similar techniques may also be used to create procedural terrain heightmaps etc. This is generally done by first generating a basis image or multiple images, e.g. with noise functions such as Perlin noise (it gives us a grayscale image that looks a bit like clouds). We then further process this base image(s) and combine the results in various ways, for example we may use different transformations, modulations, blending, adding color using color ramps etc. The whole texture is therefore described by a graph in which nodes represent the operations we apply; this can literally be done visually in software like Blender (see its shader editor). The nice thing is that we can now for example generalize the texture to 3 dimensions, i.e. not only have a flat image, but have a whole volume of a texture that can extremely easily be mapped to 3D objects simply by intersecting it with their surfaces which will yield a completely smooth texturing without any seams; this is quite often used along with raytracing -- we can texture an object by simply taking the coordinates of the ray hit as the 3D texture coordinates, it's that simple. Or we can animate a 2D texture by doing a moving cross section of 3D texture. We can also write the algorithm so that the generated texture has no seams if repeated side-by-side (by using modular "wrap-around" coordinates). We can also generate the texture at any arbitrary resolution as we have a continuous mathematical description of it; we may perform an infinite zoom into it if we want. As if that's not enough, we can also generate almost infinitely many slightly different versions of this texture by simply changing the seed of pseudorandom generator we use.



We use procedural generation mainly in two ways:

- **offline/explicit:** We pre-generate the data before we run the program, i.e. we let the algorithm create our art, save it to a file and then use it as we would use traditionally created art.
- **realtime/implicit:** We generate the data on the fly and only parts of it that we currently need (this of course requires an algorithm that is able to generate any part of the data independently of its other parts; for example for a procedural texture each pixel's color should only be determined by its coordinates). For example with a procedural texture mapped onto a 3D model, we would only compute the texture pixels (texels) that we are actually drawing: this has the advantage of giving an infinite resolution of the texture because no matter how close-up we view the model, we can always compute exactly the pixels we need. This would typically be implemented inside a fragment/pixel shader program. This is also used in the voxel games that generate the world only in the area the player currently occupies.

Indeed we may also do something "in between", e.g. generate procedural assets into temporary files or RAM caches at run time and depending on the situation, for example when purely realtime generation of such assets would be too slow.

## Notable Techniques/Concepts

The following are some techniques and concepts often used in procedural generation:

- **noise:** Noise is often used as a basis for generation or for modulation, it can be seen as kind of "RNG taken to the next level" -- noise is greatly random but also usually has some structure, for example it may resemble smoke or water ripples. There are great many types of noise and algorithms for its generation; the simplest white noise is actually not very useful, more common are various forms of fractal noise, often used noises are Perlin noise, simplex noise etc., other ones are Voronoi diagrams, coin flip noise, midpoint displacement, spot noise, cosine noise, fault formation, Brownian motion etcetc.
- **random number generators:** To make random decisions we use random number generators -- here we actually don't have to have the best generators, we aren't dealing with "security" or anything critical, however the generator should at least have a great period so that it's not limited to just generating few different results, and its useful to be able to choose probability distribution.
- **modulation:** Using previously generated procedural data, for example noise, to somehow affect another data -- for example imagine we already have some basic procedural texture but want to make it more interesting by randomly displacing its individual pixels to warp it a little bit. If we use a simple random number generator for each pixel, the result will just look too chaotic, so it's better if we e.g. use two additional Perlin noise textures, which together say for each pixel by what distance and angle we'll displace the pixel. As Perlin noise is more continuous, gradual noise, then also the distortion will be kind of continuous and nice. A famous example of this is marble texture that can be generated by horizontally modulating a texture of vertical black and white stripes with Perlin noise.
- **simulations resembling natural/biological phenomena:** E.g. cellular automata, particle systems, erosion simulation, agent systems (letting virtual "workers" collaborate on creating something) ...
- **fractals:** Fractals can resemble nature, they create "content" on all scales and are very simple to define. Popular types of fractals are e.g. L-systems that draw fractals with turtle graphics.
- **coloring:** To get colors from simple numeric values we may use e.g. color mapping of the values to some palette or using three different arrays as RGB channels. We may also use flood fill and other things of course.
- **state search:** This is an approach similar to e.g. how computers play games such as chess. Imagine we for example want to generate a procedural city: any constructed city represents a state and these states are connected (e.g. one state leads to another by placing a building somewhere), forming a graph (sometimes even a tree) of states: the state space. The idea is to create an evaluation function that takes any given state (any specific city) and says how "good" it is (e.g. how realistic it looks, i.e. for example there shouldn't be more hospitals than actual houses of people, buildings should be reachable by roads etc.); then we also implement a search function (e.g. minimax, monte carlo, ...) that uses the evaluation function to search for some good enough state we take as the result. Evolutionary search is often used here.
- **constructive approach:** The "obvious" approach, an algorithm that simply constructs something according to some rules from start to finish, without performing advanced things like evaluating the quality of the generated output, letting different outputs compete, combining several different

outputs etc. Example may be for example recursive space partitioning for the creation of game dungeons.

- **wave function collapse**: Analogy to quantum mechanics, often used for generating tile maps.
- **combining intermediate results**: For example when creating procedural textures we may actually create two separate textures and then somehow blend them together to get a more interesting result.
- **wrap-around coordinates**: Wrap around (modular) coordinates help us make tiling data.
- **mathematical functions**: Nice structures can be generated just by combining basic mathematical functions such as sine, cosine, square root, minimum/maximum, logarithm etc. We take these functions and make an expression that transforms input coordinates (i.e. for example  $x$ ,  $y$  and *time* for an animated picture) to the final generated value. The functions may be composed (put on input of other functions), added, multiplied etc.
- **higher dimensionality**: Equations used for procedural generation are often generalized to higher dimensions so that we can for example create smooth animation by taking the extra dimension as time.
- **filters**: We may utilize traditional graphic filters such as Gaussian blur, median blur, general convolution, color adjustments etc.
- **stochastic models**: Stochastic mathematical models describe the generated result in terms of probabilities, which is convenient for us as we can take the model and just use random number generators to make decisions with given probabilities to obtain a specific result. For example Markov chains can be used to easily generate random procedural text by knowing probabilities with which any word is followed by another word, this may also be used to generate linear game levels etc. Similarly we may utilize various non-deterministic finite state automata, decision trees etc.
- **constraint solving**: Many times the problem of procedural generation can be described as a constraint solving problem, i.e. we have a set of constraints such as "we want 10 to 20 rooms" and "each room must be reachable from other rooms" and then we want to find a solution that just satisfies the constraints (without somehow rating how good the solution is). The advantage of formulating the problem this way is that there exist a number of algorithms for solving such problems, e.g. ASP, some heuristic searches etc.
- ...

## Examples

Here are some cool ASCII renderings of procedurally generated pictures:

```
.....',:.....,c;,,,,,'M0.....',:c,c,'o:xcx.',',l;,:',:.,
...'.',:.....,,'x0Xoc;::::;lo.'o,c;;c.:l.,',:.....,
.',:.....,,'oKxl:'.',:.....,;cX:o0k.'l':',:.....,
.:.....,,'l:x;,Kkocl;::::;lcxX':cX;'o',:.....,
.:.....,,'l;Kc:Xol;',:.....,;ck0c'.l',:.....,
l.'l':.....,.'XlXo;,'.xooccooxkXKMW.'l':.....,
.,:.....,.'olcxl:'.cl;::::;coklXolcl.'l';:'.
':c:;,:'.l,..oxo;,,l;,'.',:.....,;c..c:o;c
x.'l':k'.l,..oc;::::;'.xxkXK0xKxxxxxk.'l':,
xl00lkL,x;Xxl:,,,,'.llccol;l;lllxxxxxkkK;ck'
c:kX:x:kLWKxl.KXc;,,',:.....,;cooxkK;cxKW
oo:'o,kL,Kxl:'MKk;,'.',:.....,;coX0;oX
c.,o,Xc,.kc;,'Kkoc;,'.',:.....,;ck.'o
.:..o:'.xl:'.xc;,'.':.....,;lo..:ck.:c
;l'.l,..o;,'.ol;,'.':.....,;lx..lx,c
'x:.xl,..o;,'.l:,'.':.....,;l...:o..lx:x
.c,.xl,..xl;,'.':.....,;lo..;o..l.'l.
.c:kc;,.ol;,'.':.....,;cx.'lx:'o.:
0o;'kc;,'.c;,'.':.....,;cokK;ck,cX,o,.
'Xo;,0Xoc;,,',:.....,;kKM':lxK,lk,o:'o
:WKxc;,'Kkxooc;,,',:.....,;cXK.lxKWlk:x:Xk:
:'kc;KkxxxxxlLl;l;loccll;,,',:.....,;lxX;cX.,lkL00cl
:'.',:.....,.'kxxxxXx0KXkxx.'l':;co..l.'k:'l'.
oc;o:c.c;,,',:.....,;l;,:oxo..l':;:c:'
',:.....,;lcloxlkoc;,,',:.....,;lc.'lxclo.,:.....,
',:.....,.'WMKXkxooccoox.';oLX.'l':.....,
',:.....,.'l.'c0kc;,'.':.....,;loX:cK;l'.':.....,
',:.....,.'o;Xc:'Xxcl;::::;lcokK;x:l;'.':.....,
',:.....,.'l'.k0o:Xc;,'.':.....,;lxKo,'l';,
',:.....,.'l'.c;c,o:'ol;::::;coX0x',:.....,.
```

[illegible]

603/815

{ NOTE: The equations for the functions were made by me when I was playing around with another project. I have uploaded them to Wikimedia Commons, you can find actual png pictures there. ~drummyfish }

```
#include <stdio.h>
#include <math.h>

#define W 55
#define H 30

// helper stuff:
char palette[] = "WM0KXkxocl;:,'. ";

double min(double a, double b) { return a < b ? a : b; }
double max(double a, double b) { return a > b ? a : b; }
double absMy(double x) { return x < 0 ? -1 * x : x; }
double gauss(double x) { return exp((-1 * x * x) / 2); }

double tri(double x)
{
    x = absMy(x);

    int whole = x;
    double frac = x - whole;

    return whole % 2 ? 1 - frac : frac;
}

void drawFunction(double (*f)(double, double), double xFrom, double yFrom,
double xTo, double yTo)
{
    double v1 = 0xffffffff, v2 = -1 * v1;
    double sX = (xTo - xFrom) / W, sY = (yTo - yFrom) / H;

    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x)
        {
            double v = f(xFrom + x * sX, yFrom + y * sY);

            if (v > v2)
                v2 = v;

            if (v < v1)
                v1 = v;
        }

    v2 -= v1;

    if (v2 == 0)
        v2 = 0.0001;

    for (int y = 0; y < H; ++y)
    {
        for (int x = 0; x < W; ++x)

            putchar(palette[(int) (15 *
((min(v2, max(0, f(xFrom + x * sX, yFrom + y * sY) - v1))) / v2))]);

        putchar('\n');
    }
}

// ==== ACTUAL INTERESTING FUNCTIONS HERE ====

double fSnakes(double x, double y)
{
    return sqrt(tri(x + sqrt(tri(x + 0.4 * sin(y*3)))));
}

double fYinYang(double x, double y)
{
    double r = sin(1.2 * y + 2.5 * sin(x) + 2 * cos(2.25 * y) * sin(x));
    return log(2 * sqrt(absMy(r)) - r);
}
```

```

}

double fSwirl(double x, double y)
{
    return gauss(
        fmod((x * x), (absMy(sin(x + y)) + 0.001)) +
        fmod((y * y), (absMy(sin(x - y)) + 0.001)));
}

int main(void)
{
    drawFunction(fSwirl, -2, -2, 2, 2);
    putchar('\n');
    drawFunction(fSnakes, -1, -1, 2, 2);
    putchar('\n');
    drawFunction(fYinYang, -4, -4, 4, 4);
}

```

Now let's take a look at some iterative algorithm: an extremely simple dungeon generator. This is so called *constructive* algorithm, a simple kind of method that simply "constructs" something according to given rules, without evaluating how good it's work actually is etc. All it's going to do is just randomly choose a cardinal direction (up, right, down, left), draw a line of random length, and repeat the same from the line's endpoint, until predefined number of lines has been drawn (a kind of random walk). Here is the C code:

```

#include <stdio.h>

#define W 30          // world width
#define H 30          // world height

char world[H * W];    // 2D world array

unsigned int r = 12345; // random seed here

unsigned int random()
{
    r = r * 321 + 29;
    return r >> 4;
}

void generateWorld()
{
    int steps = 100;    // draw this many lines
    int pos = 0;
    int add = 1;        // movement offset
    int nextLineIn = 1;

    for (int i = 0; i < H * W; ++i)
        world[i] = ' ';

    while (steps)
    {
        world[pos] = 'X';
        nextLineIn--;

        int nextPos = pos + add;

        if (nextPos < 0 || nextPos >= W * H || // going over world edge?
            (add == 1 && nextPos % W == 0) ||
            (add == -1 && nextPos % W == W - 1))
            nextLineIn = 0;
        else
            pos = nextPos;

        if (nextLineIn <= 0)
        {
            steps--;
            nextLineIn = W / 5 + random() % (W / 3);
            add = (random() & 0x10) ? W : 1;

            if (rand() & 0x80)
                add *= -1;
        }
    }
}

```

```

    }
}
}

int main(void)
{
    generateWorld();

    for (int i = 0; i < H * W; ++i) // draw
    {
        char c = world[i];

        putchar(c);
        putchar(c);

        if ((i + 1) % W == 0)
            putchar('\n');
    }

    return 0;
}

```

And here is one possible output of the program:

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX  XXXX                XX  XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX  XXXX  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX  XXXX  XX                XX  XXXX  XX                X
XXXX  XXXX  XX                XX  XXXX  XX                X
XXXX  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX  XXXX  XX                XX  XX  XXXX  XX                X
XXXX  XXXX  XX                XX  XX  XXXX  XX                X
XXXX  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX      XX  XX                XX  XX  XXXX  XX                X
XX      XX  XX                XX  XX  XXXX  XX                X
XXXXXXXXXXXXXXXXXX          XX  XX  XXXX                X
XX      XX  XX                XX  XX  XXXX                X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXX          XX                X
XX  XX      XX                XX                XX                X
XX  XX      XX                XX                XX                X
XX  XX      XX                XX                XX                X
XX  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX  XX      XX                XX                XX                X
XX  XX      XX                XX                XX                X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX  XX      XX                XX                XX                X
XX  XX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX  XX      XX                XX                XX                X
      XX      XX                XXXXXXXXXXXXXXXXXXXXXXXXXXXX
      XX      XX                XX
      XX      XX                XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  XX

```

TODO: some example with noise

## See Also

- [algorithmic art](#)

---

productivity\_cult

## Productivity Cult

"PRODUCE PRODUCE PRODUCE PRODUCE PRODUCE" --[capitalism](#)

Productivity cult (often connected to terms such as *self improvement*, *personal growth* etc.) is one of modern capitalist religions which praises human productivity above everything, even happiness, well being, sanity etc. Kids nowadays are all about "how to be more motivated and productive", they make daily checklists, analyze tables of their weekly performance, count how much time they spend taking a shit on the toilet, give up sleep to study some useless bullshit required by the current market fluctuation. Productivity cult is all about voluntarily making oneself a robot, a slave to the system that worships capital. As Froge put it: "The world of self improvement gurus is a circlejerk of such magnitude it rivals several world religions".

A human is living being, not a machine, he should live a happy relaxed life, dedicated to spending time with his close ones, raising children, enjoying the beauties of nature, exploring secrets of the universe, without stress; he should create when inspiration or a truly great necessity comes to him and when he does, he should take his time to carefully make the art great, without hastening it or forcing it. Productivity cult goes all against this, it proclaims one should be constantly spitting out "something", torturing and forcing himself, maximizing quantity on detriment of quality, undergo constant stress while suppressing rest -- that one should all the time be preoccupied with competitive fight, deadlines, that art he creates is something that can be planned on schedule, made on deadline and assigned a calculated price tag to be an ideal consumerist product. If such stance towards life doesn't make you wanna puke, you most likely lack a soul.

Do not produce. Create. Art takes time and can't be scheduled.

The name of the cult itself says a lot about it. While a name such as *efficiency* would probably be better, as efficiency means doing less work with the same result and therefore having more free time, it is not a surprise that capitalism has chosen the word *productivity*, i.e. producing more which means working more, e.g. for the price of free time and mental health.

Productivity obsessed people are mostly idiots without the ability to think for themselves, they have desktops with "motivational" wallpapers saying shit like "the word impossible doesn't exist in my dictionary" and when you tell them if it wouldn't be better to rather establish a society where people wouldn't have to work they start screeching "HAHAA THATS IMPOSSIBLE IT CANT WORK". Productivity maximalists bully people for taking rest or doing what they otherwise enjoy in moderation -- they invent words such as "procrastination" to create a feeling of ever present guilt induced by doing what one truly enjoys.

Productivity freaks are often the ones who despise consumers, i.e. brainless zombies that consume goods, but somehow don't seem to mind being producers, a similar kind of brainless zombies that just stands on the other end of this retarded system.

One of the funniest examples of productivity cult gone too far is so called "life couching" in which the aspiring producer robots hire bullshit cult leaders, so called "life couches", to shout at them to be more productive. At least in the past slaves were aware of being slaves and tried to free themselves. I literally want to kill myself.

Productivity is such a big deal because **programmers are in fact actually getting exponentially less productive** due to time needed to spend on bullshit nowadays, on overcomplicated buggy bloat and billions of frameworks needed to get basic things done -- this has been pointed out by Jonathan Blow in his talk *Preventing the Collapse of Civilization* in which he refers to the video of Ken Thompson talking about how he developed the Unix operating system in **three weeks**.

A considerable number of people are attracted to suckless software due to its positive effects on productivity thanks to the elimination of bullshit. These are mostly the kind of above mentioned dumbasses who just try to exploit anything they encounter for self interest without ever aiming for greater good, they don't care about Unix philosophy beyond its effects on increasing their salary. Beware of them, they poison society.

## See Also

- motivation
- procrastination
- laziness
- antiwork

# Programming Language

Programming language is an artificial formal (mathematically precise) language created in order to allow humans to relatively easily write algorithms for computers. It basically allows a human to very specifically and precisely but still relatively comfortably tell a computer what to do. We call a program written in programming language the program's **source code**. Programming languages often try to mimic some human language -- practically always English -- so as to be somewhat close to humans but programming language is actually MUCH simpler so that a computer can actually analyze it and understand it precisely (as computers are extremely bad at understanding actual natural language), without ambiguity, so in the end it all also partially looks like math expressions. A programming language can be seen as a middle ground between pure machine code (the computer's native language, very hard to handle by humans) and natural language (very hard to handle by computers).

For beginners: a programming language is actually much easier to learn than a foreign language, it will typically have fewer than 100 "words" to learn (out of which you'll mostly use like 10) and once you know one programming language, learning another becomes a breeze because they're all (usually) pretty similar in basic concepts. The hard part may be learning some of the concepts.

A programming language is distinct from a general computer language by its purpose to express algorithms and be used for creation of programs. This is to say that there are computer languages that are NOT programming languages (at least in the narrower sense), such as HTML, json and so on.

A **simple example** of source code in the C programming language is the following:

```
// simple program computing squares of numbers
#include <stdio.h>

int square(int x)
{
    return x * x;
}

int main(void)
{
    for (int i = 0; i < 5; ++i)
        printf("%d squared is %d\n",i,square(i));

    return 0;
}
```

Which prints:

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

We divide programming languages into different groups. Perhaps the most common divisions is to two groups:

- **compiled** languages: Meant to be transformed by a compiler to a native (directly executable) binary program, i.e. before running the program we have to run it through the process of compilation into runnable form. These languages are typically more efficient but usually more difficult to program in, less flexible and the compiled programs are non-portable (can't just be copy-pasted to another computer with different architecture and expected to run; note that this doesn't mean compiled languages aren't portable, just that the compiled EXECUTABLE is not). These languages are usually lower level, use static and strong typing and more of manual memory management. Examples: C, C++, go, Haskell or Pascal.
- **interpreted** languages: Meant to be interpreted by an interpreter "on-the-go", i.e. what we write we can also immediately run; these languages are often used for **scripting**. To run such program you



need the interpreter of the language installed on your computer and this interpreter reads the source code as it is written and performs what it dictates (well, this is actually simplified as the interpreter normally also internally does a kind of quick "lightweight" compilation, but anyway...). These languages are generally less efficient (slower, use more RAM) but also more flexible, easier to program in and independent of platforms. These languages usually higher-level, use weak and dynamic typing and automatic memory management (garbage collection, ...). Examples: Python, Perl, JavaScript and BASH.

Sometimes the distinction here may not be completely clear, for example Python is normally considered an interpreted language but it can also be compiled into bytecode and even native code. Java is considered more of a compiled language but it doesn't compile to native code (it compiles to bytecode). C is traditionally a compiled language but there also exist C interpreters. Comun is meant to be both compiled and interpreted etc.

Another common division is by **level of abstraction** roughly to (keep in mind the transition is gradual and depends on context, the line between low and high level is extremely fuzzy):

- **low level**: Languages which are so called "closer to hardware" ("glorified assembly"), using little to no abstraction (reflecting more how a computer actually works under the hood without adding too many artificial concepts above it, allowing direct access to memory with pointers, ...), for this they very often use plain imperative paradigm), being less comfortable (requiring the programmer to do many things manually), less flexible, less safe (allowing shooting oneself in the foot). However (because less is more) they have great many advantages, e.g. being simple to implement (and so more free) and **greatly efficient** (being fast, memory efficient, ...). One popular definition is also that "a low level language is that which requires paying attention to the irrelevant"; another definition says a low level language is that in which one command usually corresponds to one machine instruction. Low level languages are **typically compiled** (but it doesn't have to be so). Where exactly low level languages end is highly subjective, many say C, Fortran, Forth and similar languages are low level (normally when discussing them in context of new, very high level languages), others (mainly the older programmers) say only assembly languages are low level and some will even say only machine code is low level.
- **high level**: Languages with higher level of abstraction than low level ones -- they are normally more complex (though not always), interpreted (again, not necessarily), comfortable, dynamically typed, beginner friendly, "safe" (having various safety mechanism, automatic checks, automatic memory management such as garbage collection) etc. For all this they are typically slower, less memory efficient, and just more bloated. Examples are Python or JavaScript.

We can divide language in many more ways, for example based on their **paradigm** (roughly its core idea/model/"philosophy", e.g. imperative, declarative, object-oriented, functional, logical, ...), **purpose** (general purpose, special purpose), computational power (turing complete or weaker, many definitions of a programming language require Turing completeness), typing (strong, weak, dynamic, static) or function evaluation (strict, lazy).

A computer language consists from two main parts:

- **syntax**: The grammar rules and words, i.e. how the language "looks", what expressions we are allowed to write in it. Syntax says which words can follow other words, if indentation has to follow some rules, how to insert comments in the source code, what format numbers can be written in, what kinds of names variables can have etc. Syntax is the surface part, it's often considered not as important or hard as semantics (e.g. syntax errors aren't really a big deal as the language processor immediately catches them and we correct them easily), but a good design of syntax is nevertheless still very important because that's what the programmer actually deals with a great amount of time.
- **semantics**: The meaning of what we write, i.e. semantics says what the syntax actually stands for. E.g. when syntax says it is possible to write  $a / b$ , semantics says this means the mathematical operation of division and furthermore specifies what  $a$  and  $b$  can actually be, what happens if  $b$  is zero etc. Semantics is the deeper part as firstly it is more difficult to define and secondly it gives the language its features, its power to compute, usability, it can make the language robust or prone to errors, it can make it efficient or slow, easy and hard to compile, optimize etc.

**What is the best programming language and which one should you learn?** (See also [programming](#).) These are the big questions, the topic of programming languages is infamous for being very [religious](#) and different people root for different languages like they do e.g. for [football](#) teams. For [minimalists](#), i.e. [suckless](#), [LRS](#) (us), [Unix](#) people, [Plan9](#) people etc., the standard language is [C](#), which is also probably the most important language in [history](#). It is not in the league of the absolutely most minimal and objectively best languages, but it's relatively minimalist (much more than practically any [modern](#) language) and has great advantages such as being one of the absolutely fastest languages, being extremely well established, long tested, supported everywhere, having many compilers etc. But C isn't easy to learn as a first language. Some minimalist also promote [go](#), which is kind of like "new C". Among the most minimal usable languages are traditionally [Forth](#) and [Lisp](#) which kind of compete for who really is the smallest, then there is also our [comun](#) which is a bit bigger but still much smaller than C. To learn programming you may actually want to start with some ugly language such as [Python](#), but you should really aim to transition to a better language later on.

**Can you use multiple programming languages for one project?** Yes, though it may be a burden, so don't do it just because you can. Combining languages is possible in many ways, e.g. by embedding a [scripting](#) language into a compiled language, linking together object files produces by different languages, creating different programs that communicate over network etc.

## History

The first higher level programming language was probably Plankalkul made by Konrad Zuse in 1942.

TODO

## More Details And Context

What really IS a programming language -- is it software? Is it a standard? Can a language be [bloated](#)? How does the languages evolve? Where is the exact line between a programming language and non-programming language? Who makes programming languages? Who "owns" them? Who controls them? Why are there so many and not just one? These are just some of the questions one may ask upon learning about programming. Let's try to quickly answer some of them.

Strictly speaking programming language is a [formal language](#) with [semantics](#), i.e. just something akin a "mathematical idea" -- as such it cannot be directly "owned", at least not on the grounds of [copyright](#), as seems to have been quite strongly established by a few court cases now. However things related to a language can sadly be owned, for example their specifications (official standards describing the language), [trademarks](#) (the name or logo of the language), implementations (specific software such as the language's compiler), [patents](#) on some ideas used in the implementation etc. Also if a language is very complex, it can be owned practically; typically a corporation will make an extremely complicated language which only 1000 paid programmers can maintain, giving the corporation complete control over the language -- see [bloat monopoly](#) and [capitalist software](#).

At this point we should start to distinguish between the pure language and its [implementation](#). As has been said, the pure language is just an idea -- this idea is explained in detail in so called **language specification**, a document that's kind of a standard that precisely describes the language. Specification is a technical document, it is NOT a tutorial or promotional material or anything like that, its purpose is just to DEFINE the language for those who will be implementing it -- sometimes specification can be a very official standard made by some standardizing organization (as e.g. with C), other times it may be just a collaborative online document that at the same time serves as the language reference (as e.g. with Lua). In any case it's important to [version](#) the specification just as we version programs, because when specification changes, the specified languages usually changes too (unless it's a minor change such as fixing some typos), so we have to have a way to exactly identify WHICH version of the language we are referring to. Theoretically specification is the first thing, however in practice we usually have someone e.g. program a small language for internal use in a company, then that language becomes more popular and widespread and only then someone decides to standardize it and make the official specification. Specification describes things like syntax, semantics, conformance criteria etc., often using precise formal tools such as [grammars](#). It's hugely difficult to make good specification because one has to decide what depth to go to and even what to purposefully leave unspecified! One would thought that it's always better to define as many things as possible, but that's naive -- leaving some things up to the choice of those who will be implementing the

language gives them freedom to implement it in a way that's fastest, most elegant or convenient in any other way.

It is possible for a language to exist without official specification -- the language is then basically specified by some of its implementations, i.e. we say the language is "what this program accepts as valid input". Many languages go through this phase before receiving their specification. Language specified purely by one implementation is not a very good idea because firstly such specification is not very readable and secondly, as said, here EVERYTHING is specified by this one program (the language EQUALS that one specific compiler), we don't know where the freedom of implementation is. Do other implementations have to produce exactly the same compiled binary as this one (without being able to e.g. optimize it better or produce binaries for other platforms)? If not, how much can they differ? Can they e.g. use different representation of numbers (may be important for compatibility)? Do they have to reproduce even the same bugs as the original compiler? Do they have to have the same technical limitations? Do they have to implement the same command line interface (without potentially adding improvements)? Etc.

Specification typically gets updated just as software does, it has its own version and so we then also talk about version of the language (e.g. C89, C99, C11, ...), each one corresponding to some version of the specification.

Now that we have a specification, i.e. the idea, someone has to realize it, i.e. program it, make the implementation; this mostly means programming the language's compiler or interpreter (or both), and possibly other tools (debugger, optimizer, transpiler, etc.). A language can (and often does) have multiple implementations; this happens because some people want to make the language as fast as possible while others e.g. want to rather have small, minimalist implementation that will run on limited computers, others want implementation under a different license etc. The first implementation is usually so called **reference implementation** -- the one that will serve as a kind of authority that shows how the language should behave (e.g. in case it's not clear from the specification) to those who will make newer implementations; here the focus is often on correctness rather than e.g. efficiency or minimalism, though it is often the case that reference implementations are among the best as they're developed for longest time. Reference implementations guide development of the language itself, they help spot and improve weak points of the language etc. Besides this there are third party implementations, i.e. those made later by others. These may add extensions and/or other modifications to the original language so they spawn **dialects** -- slightly different versions of the language. We may see dialects as forks of the original language, which may sometimes even evolve into a completely new language over time. Extensions of the languages may sound like a good thing as they add more "comfort" and "features", however they're usually bad as they create a dependency and fuck up the standardization -- if someone writes a program in a specific compiler's dialect, the program won't compile under other compilers.

A new language comes to existence just as other things do -- when there is a reason for it. I.e. if someone feels there is no good language for whatever he's doing or if someone has a brilliant idea and want to write a PhD thesis or if someone smokes too much weed or if a corporation wants to control some software platform etc., a new language may be made. This often happen gradually (again, like with many things), i.e. someone just starts modifying an already existing language -- at first he just makes a few macros, then he starts making a more complex preprocessor, then he sees it's starting to become a new language so he gives it a name and makes it a new language -- such language may at first just be transpiled to another language (often C) and over time it gets its own full compiler. At first a new language is written in some other language, however most languages aim for **self hosted implementation**, i.e. being written in itself. This is natural and has many advantages -- a language written in itself proves its maturity, it becomes independent and as it itself improves, so does its own compiler. Self hosting a language is one of the greatest milestones in its life -- after this the original implementation in the other language often gets deleted as it would just be a burden to keep maintaining it.

**So can a language be inherently fast, bloated etc.?** When we say a language is fast, bloated, memory efficient and so on, we often refer to its implementations because, as mentioned, a language is just an idea which can be implemented in many ways with different priorities and tradeoffs, so just keep in mind that talking about languages like this usually refers to the implementations. But on the other hand yes, a language CAN itself be seen as inherently having a similar property because it's simply such that its implementations more or less have to have this property. A very complicated language just cannot be implemented in a simple, non-bloated way, an extremely high level and flexible language cannot be implemented to be among the fastest -- so referring to language implementations we also a little bit refer to

the language itself as an implementation reflects the abstract language's properties. **How to tell if language is bloated?** One can get an idea from several things, e.g. list of features, paradigm, size of its implementations, number of implementations, size of the specification, year of creation (newer mostly means more bloat) and so on. However be careful, many of these are just clues, for example small specification may just mean it's vague. Even a small self hosted implementation doesn't have to mean the language is small -- imagine e.g. a language that just does what you write in plain English; such language will have just one line self hosted implementation: "Implement yourself." But to actually bootstrap the language will be immensely difficult and will require a lot of bloat.

Judging languages may further be complicated by the question of what the language encompasses because some languages are e.g. built on relatively small "pure language" core while relying on a huge library, preprocessor, other embedded languages and/or other tools of the development environment coming with the language -- for example POSIX shell makes heavy use of separate programs, utilities that should come with the POSIX system. Similarly Python relies on its huge library. So sometimes we have to make it explicitly clear about this.

## Notable Languages

Here is a table of notable programming languages in chronological order (keep in mind a language usually has several versions/standards/implementations, this is just an overview).

| language            | minimalist/good?  | since | ~min.<br>selfhos. impl.<br>LOC | spec. (~no<br>stdlib pages) | notes                                                                    |
|---------------------|-------------------|-------|--------------------------------|-----------------------------|--------------------------------------------------------------------------|
| " <u>assembly</u> " | <b>yes but...</b> | 1947? |                                |                             | NOT a single language, non- <u>portable</u>                              |
| <u>Fortran</u>      | <b>kind of</b>    | 1957  |                                | 300, proprietary (ISO)      | similar to Pascal, compiled, fast, was used by scientists a lot          |
| <u>Lisp</u>         | <b>yes</b>        | 1958  | 100 (judg. by jmc lisp)        | 1                           | elegant, KISS, functional, many variants (Common Lisp, Closure, ...)     |
| <u>Basic</u>        | kind of?          | 1964  |                                |                             | mean both for beginners and professionals, probably efficient            |
| <u>Forth</u>        | <b>yes</b>        | 1970  | 100 (judg. by milliforth)      | 1                           | <u>stack</u> -based, elegant, very KISS, interpreted and compiled        |
| <u>Pascal</u>       | <b>kind of</b>    | 1970  |                                | 80, proprietary (ISO)       | like "educational C", compiled, not so bad actually                      |
| <u>C</u>            | <b>kind of</b>    | 1972  | 20K (lcc)                      | 160, proprietary (ISO)      | compiled, fastest, efficient, established, suckless, low-level, #1 lang. |
| <u>Prolog</u>       | maybe?            | 1972  |                                |                             | <u>logic</u> paradigm, hard to learn/use                                 |
| <u>Smalltalk</u>    | <b>quite yes</b>  | 1972  |                                | 40, proprietary (ANSI)      | PURE (bearable kind of) <u>OOP</u> language, pretty minimal              |
| <u>C++</u>          | no, bearable      | 1982  |                                | 500, proprietary            | bastard child of C, only adds <u>bloat</u> ( <u>OOP</u> ), "games"       |
| <u>Ada</u>          | ???               | 1983  |                                |                             | { No idea about this, sorry. ~drummyfish }                               |
| Object Pascal       | no                | 1986  |                                |                             | Pascal with OOP (like what C++ is to C), i.e. only adds bloat            |
| Objective-C         | probably not      | 1986  |                                |                             | kind of C with Smalltalk-style "pure" objects?                           |
| <u>Perl</u>         | rather not        | 1987  |                                |                             | interpreted, focused on strings, has kinda cult following                |
| <u>Bash</u>         | well              | 1989  |                                |                             | Unix scripting shell, very ugly syntax, not so elegant but               |

| language           | minimalist/good?   | since | ~min.<br>selfhos. impl.<br>LOC | spec. (~no<br>stdlib pages) | notes                                                                          |
|--------------------|--------------------|-------|--------------------------------|-----------------------------|--------------------------------------------------------------------------------|
| <u>Haskell</u>     | <b>kind of</b>     | 1990  |                                | 150, proprietary            | bearable<br><u>functional</u> , compiled, acceptable                           |
| <u>Python</u>      | NO                 | 1991  |                                | 200? (p. lang.<br>ref.)     | interpreted, huge bloat, slow,<br>lightweight OOP, artificial<br>obsolescence  |
| POSIX <u>shell</u> | well, "kind of"    | 1992  |                                | 50, proprietary<br>(paid)   | standardized (std 1003.2-1992)<br>Unix shell, commonly e.g. <u>Bash</u>        |
| <u>Brainfuck</u>   | <b>yes</b>         | 1993  | 100 (judg. by<br>dbfi)         | 1                           | extremely minimal (8 commands),<br>hard to use, <u>esolang</u>                 |
| <u>FALSE</u>       | <b>yes</b>         | 1993  |                                | 1                           | very small yet powerful,<br>Forth-like, similar to Brainfuck                   |
| <u>Lua</u>         | <b>quite yes</b>   | 1993  | 7K (LuaInLua)                  | 40, free                    | small, interpreted, mainly for<br>scripting (used a lot in games)              |
| <u>Java</u>        | NO                 | 1995  |                                | 800, proprietary            | forced <u>OOP</u> , "platform<br>independent" (bytecode), slow,<br>bloat       |
| <u>JavaScript</u>  | NO                 | 1995  | 50K (est. from<br>QuickJS)     | 500,<br>proprietary?        | interpreted, the <u>web</u> lang.,<br>bloated, classless <u>OOP</u>            |
| <u>PHP</u>         | no                 | 1995  |                                | 120 (by Google),<br>CC0     | server-side web lang., OOP                                                     |
| <u>Ruby</u>        | no                 | 1995  |                                |                             | similar to Python                                                              |
| <u>C#</u>          | NO                 | 2000  |                                |                             | proprietary (yes it is), extremely<br>bad lang. owned by Micro\$oft,<br>AVOID  |
| <u>D</u>           | no                 | 2001  |                                |                             | some expansion/rework of C++?<br>OOP, generics etcetc.                         |
| <u>Rust</u>        | NO! lol            | 2006  |                                | 0 :D                        | extremely bad, slow, freedom<br>issues, toxic community, no<br>standard, AVOID |
| <u>Go</u>          | <b>kind of</b>     | 2009  |                                | 130,<br>proprietary?        | "successor to C" but not well<br>executed, bearable but rather<br>avoid        |
| <u>LIL</u>         | <b>yes</b>         | 2010? |                                |                             | not known too much but nice,<br>"everything's a string"                        |
| <u>uxntal</u>      | <b>yes</b> but SJW | 2021  | 400 (official)                 | 2? (est.),<br>proprietary   | assembly lang. for a minimalist<br>virtual machine, PROPRIETARY<br>SPEC.       |
| <u>comun</u>       | <b>yes</b>         | 2022  | < 3K                           | 2, CC0                      | "official" <u>LRS</u> language, WIP,<br>similar to Forth                       |

## Interesting Languages

Some programming languages may be interesting rather than directly useful, however these are important too as they teach us a lot and may help us design good practically usable languages. In fact professional researches in theory of computation spend their whole lives dealing with practically unusable languages and purely theoretical computers.

One such language is e.g. **Unary**, a programming language that only uses a single character while being Turing complete (i.e. having the highest possible "computing power", being able to express any program). All programs in Unary are just sequences of one character, differing only by their length (i.e. a program can also be seen just as a single natural number, the length of the sequence). We can do this because we can make an ordered list of all (infinitely many) possible programs in some simple programming language (such as a

Turing machine or Brainfuck), i.e. assign each program its ordinal number (1st, 2nd, 3rd, ...) -- then to express a program we simply say the position of the program on the list.

There is a community around so called **esoteric programming languages** which takes great interest in such languages, from mere jokes (e.g. languages that look like cooking recipes or languages that can compute everything but can't output anything) to discussing semi-serious and serious, even philosophical and metaphysical questions. They make you think about what really is a programming language; where should we draw the line exactly, what is the absolute essence of a programming language? What's the smallest thing we would call a programming language? Does it have to be Turing complete? Does it have to allow output? What does it even mean to compute? And so on. If you dare, kindly follow the rabbit hole.

## See Also

- esoteric programming language
- constructed language
- pseudocode
- compiler

---

programming

## Programming

*Not to be confused with coding.*

Programming is the act, science and art of writing computer programs; it involves creation of algorithms and data structures and implementing them in programming languages. It may involve related activities such as testing, debugging, hacking and drinking coffee.

You may also encounter the term coding which is used by noob wannabe programmers, so called "coders" or code monkeys. "Coding" doesn't reach the quality of programming, it is done in baby handholding languages like Python, JavaScript or Rust by people with very shallow knowledge of technology and its context, barely qualified to turn on a computer (like jewtubers), who have flooded the computer industry since it became lucrative. It is mostly done for money and/or creating an image for oneself. What they do is not real programming. Do not try to imitate them.

At high level programming becomes spiritual. Check out e.g. zen and the famous Tao of Programming (yes, it's kind of a joke but it's rooted in the reality of a common hacker's mindset, programming can truly be kind of a meditation and pursuit of enlightenment, often leading one to asking deeper questions about the world). Many people say that learning programming opens your eyes in a certain new way (with some however claiming the contrary that programming will rather close your eyes), you then see the world like never before (but that's probably kind of true of almost all skills taken to a high level so this may be a shit statement). Others say too much programming cripples you mentally and gives you autism. Anyway it's fun and changes you somehow. **Programming requires a good knowledge of advanced math**. Also probably at least above average IQ, as well as below average social intelligence. Being a white man is an advantage.

**Can you do programming without math?** Short answer: no. Long answer: no, you can't.

## How To Learn Programming And Do It Well

*See also programming tips.*

At first you have to learn two basic rules that have to be constantly on your mind:

1. **You cannot be a good programmer if you're not good at math** -- real programming is pure math.
2. **Minimalism is the most important concept in programming.** If you don't like, support or understand minimalism, don't even think of becoming a programmer.

OK, now the key thing to becoming a programmer is learning a programming language very well (and learning many of them), however this is not enough (it's only enough for becoming a coding monkey), you additionally have to have a wider knowledge such as general knowledge of computers (electronics, hardware, theory or computation, networks, ...), tech history and culture (free software, hacker culture, free culture, ...), math and science in general, possibly even society, philosophy etc. Programming is not an isolated topic (only coding is), a programmer has to see the big picture and have a number of other big brain interests such as chess, voting systems, linguistics, physics, music etc. Remember, becoming a good programmer takes a whole life, sometimes even longer.

**Can you become a good programmer when you're old?** Well, as with everything to become a SERIOUSLY good programmer you should have probably started before the age of 20, the majority of the legend programmers started before 10, it's just like with sports or becoming an excellent musician. But with enough enthusiasm and endurance you can become a pretty good programmer at any age, just like you can learn to play an instrument or run marathon basically at any age, it will just take longer and a lot of energy. You don't even have to aim to become very good, becoming just average is enough to write simple games and have a bit of fun in life :) Just don't try to learn programming because it seems cool, because you want to look like movie haxor, gain followers on youtube or because you need a job -- if you're not having genuine fun just thinking before sleep about how to swap two variables without using a temporary variable, programming is probably not for you. **Can you become a good programmer if you're black or woman?** No. :D Ok, maybe you can, but all the above applies, don't do it for politics or money or followers -- if you become a seriously based programmer (from LRS point of view) of unlikely minority, we'll be more than happy to put an apology here, in ALL CAPS and bold letters :) Hopefully this will inspire someone...

**Which programming language to start with?** This is the big question. Though languages such as Python or JavaScript are objectively really REALLY bad, they are nowadays possibly the easiest way to get into programming, so you may want to just pick one of these two, knowing you'll abandon it later to learn a true language such as C (and knowing the bad language will still serve you in the future in some ways, it's not a wasted time). Can you start with C right away? It's probably not impossible for a genius but it will be VERY hard and you'll most likely end up failing, overwhelmed, frustrated and never returning to programming again. In *How To Become A Hacker* ESR actually recommends to learn C, Lisp or Go as the first language, but that recommendation comes to aspiring hackers, i.e. the most talented and ambitious programmers, so think about whether you fit this category. Absolutely do NOT even consider C# (shit, unusable), Java (shit, slow, bloated, unusable), C++ (like C but shit and more complicated), Haskell (non-traditional, hard), Rust (shit, bad design, unusable), Go (prolly hard), Lisp (non-traditional), Prolog (lol) and similar languages -- you may explore these later. Whichever language you pick for the love of god **avoid OOP** -- no matter what anyone tells you, when you see a tutorial that uses "classes"/"objects" just move on, learn normal imperative programming. OOP is a huge pile of shit meme that you will learn anyway later (because everyone writes it nowadays) so that you see why it's shit and why you shouldn't use it.

{ I really started programming in Pascal at school, it was actually a good language as it worked very similarly to C and the transition later wasn't that hard, but nowadays learning Pascal doesn't make much sense anymore. ~drummyfish }

**Games are an ideal start project** because they're fun (having fun makes learning much faster and enjoyable), there are many noob tutorials all over the Internet etc. However keep in mind to **start EXTREMELY simple**. -- this can't be stressed enough, most people are very impatient and eager and start making an RPG game or networking library without really knowing a programming language -- this is a GUARANTEED spectacular failure. At the beginning think in terms of "snake" and "minesweeper". Your very first project shouldn't even use any GUI, it should be purely command-line text program, so a text-only tiny interactive story in Python is possibly the absolutely best choice as a first project. Once you're more comfortable you may consider to start using graphics, e.g. Python + Pygame, but still KEEP IT SIMPLE, make a flappy bird clone or something. As you progress, consider perhaps buying a simple toy computer such as an open console -- these toys are closer to old computers that had no operating systems etc., they e.g. let you interact directly with hardware and teach you a LOT about good programming by teaching you how computers actually work under the hood. One day you will have to make the big step and **learn C**, the best and most important language as of yet, but be sure to only start learning it when you're at least intermediate in your start language (see our C tutorial). To learn C we recommend our SAF library which will save you all headaches of complex APIs and your games will be nice and compatible with you small toy computers.

As with everything, you learn by doing -- reading is extremely important and necessary, but to actually learn anything you have to spend thousands of hours practicing the art yourself. So **program, program and program**, live by programming, look for ways of using programming in what you're already doing, try to automatize anything you do, think about programming before sleep etc. If you can, **contribute to some project**, best if you can help your favorite FOSS program -- try this at least once as being in the company of the experienced just teaches you like nothing else, a month spent contributing to a project may be worth a year of just reading books.

TODO

---

programming\_style

## Programming Style/Code Formatting

In majority of cases a programming language lets the programmer choose the visual/surface style in which to write the code -- one may choose names for variables, indent and align commands in a convenient way, insert comments and so on. This gives rise to various styles -- typically a programmer will have his own preferred style, kind of like handwriting, but once he works in a team, some compromise has to be found to which everyone must conform so as to keep the code nice, consistent and readable. Some project, e.g. Linux, have evolved quite good, tested and de facto standardized styles, so instead of inventing a custom style (which may not be as easy as it sounds) one may choose to adopt some of the existing styles.

There exist automatic code formatters, they are often called **code beautifiers**. But not everything can be automatized, for example inserting empty spaces to separate logically related parts of a sequential of code.

TODO: moar

## Recommended LRS C Programming Style/Formatting

Here we propose a programming style and C code formatting you may use in your programs. { It's basically a style I personally adopted and fine-tuned over many years of my programming. ~drummyfish } Remember that nothing is set in stone (except that you mustn't use tabs), the most important thing is usually to be consistent within a single project and to actually think about why you're doing things the way you're doing them. Keeping to the standard set here will gain you advantages such as increased readability for others already familiar with the same style and avoiding running into traps set by short-sighted decisions e.g. regarding identifiers. Try to think from the point of view of a programmer who gets just your source code without any way to communicate with you, make his life as easy as possible. Also suppose he's reading your code on a calculator. The LRS style/formatting rules follow:

- **Respect the LRS design principles** (KISS, no OOP, avoid dependencies such as stdlib etc.).
- **Indentation: use two spaces, NEVER use tabs**. Why? Tabs are ugly, tricky (look the same as spaces) non-standard behaving characters (behavior is dependent on editor and settings, some processors will silently convert tabs and spaces, copy-paste may do so also etc.), they don't carry over to some platforms (especially paper), some very simple platforms may not even support them; your source will contain spaces either way, no need to insert additional blank character.
- **Limit source code width to 80** columns or similar value. Keep in mind the source may be edited on computers with small screens (like old thinkpads, especially within context of LRS) with a screen split vertically.
- Write **opening and closing curly brackets on their own lines, in the same columns**, e.g.:

```
if (a == b)
{
    doSomething();
    doSomething2();
}
else
{
    doSomethingElse();
    doSomethingElse2();
}
```



- **Omit curly brackets if you can** (e.g. with a single command in the block). However write them where not doing so is likely to cause confusion or syntax errors.
- **Use normal brackets to make precedence and intention clearer** even if they would be unnecessary, don't flex by writing an expression with confusing precedence that saves 4 text characters. For example it may be better to write `(a && b) || c` rather than `a && b || c`.
- **identifiers/names:**
  - ♦ **Use camelCase for variables and functions** (e.g. `myVariable`). Global and big-scope variables should have a greatly descriptive, self-documenting name, even if long (e.g. `getTicksSinceStart`, `countryAreaKMSquared`), local/short-scope identifiers can be shorter (e.g. `argBackup` within a single function), even just one letter (e.g. `i` within a single loop).
  - ♦ **Use CapitalCamelCase for data types** (e.g. `ImaginaryNumber`, `GameState` etc.).
  - ♦ **Use ALL\_CAPS\_SNAKE\_CASE for macros and constants** (e.g. `PI`, `MIN`, `LOG_ERROR`, ...).
  - ♦ It is advised that for your project you come up with a **three letter namespace prefix** that will come in front of your global identifiers. (E.g. `small3dlib` uses the prefix `S3L_`, `SDL` uses `SDL` etc.). If you choose a prefix `XYZ_`, prepend it to all global identifiers, it will prevent name clashes and help readability, e.g. when writing a renderer you will export identifiers such as `XYZ_init`, `XYZ_draw`, `XYZ_setPixel`, `XYZ_Model3D` etc. Do NOT use the prefix in local variables (inside functions, loops etc.).
  - ♦ **Prefix private global identifiers with \_**, e.g. `_tmpPointerBackup`; with the above mentioned namespace prefix this will look e.g. like this: `_XYZ_tmpPointerBackup`.
- **Use spaces** to make code more readable, so e.g. `int x = 10, y = 20`; instead of `int x=10,y=20`;, write space between `if` and its condition etc.
- **Use verbs for functions, nouns for variables** and keep consistency, e.g. a function should be named `getTimeMS` while a variable will be named `timeMS`.
- **Name from general to specific**, e.g. `getCountryTimezone` and `getCountryCapital` instead of `getTimeZoneOfCountry`, `getCapitalOfCountry` etc. This helps with code completion systems. It's not always exactly clear, you may also decide to go for `countryGetTimezone` etc., just keep it consistent.
- **Filenames:** always use only lowercase letters (some older systems just know one case, don't confuse them), either use `camel_case.ext` or `nocase.ext`.
- **Use blank lines** to logically group relevant lines of code. E.g.:

```
int a = x;
char b = y;

c += 3 * a;
d -= b;

if (c < d)
    a = b;

doSomething(a);
```

- Each file shall have a **global comment** at the top with at least: short description of the file's purpose (this is almost always missing in mainstream), short documentation, license, the author(s) and year of creation.
- **Use comments** to make your code better readable and searchable with things like grep (add keywords to relevant parts of code, e.g. `comment // player shoots to code implementing player shooting` etc.). **Use doxygen style comments** if you can, it costs nothing and allows auto documentation.
- **TODOs and WIPs are good.**
- **Don't use enums**, use `#defines`.
- **Global variables are great**, use them. **Long functions are fine**. Repeating yourself may also be fine if the alternative is too complex.
- **Adhere to C99 or C89 standard.**
- **Try to not create many source files**, many times your project can very well be in a single file which is the ideal case. Create **header only libraries** If you have multiple files, keep them in the same directory and try to have just a **single compilation unit** (only one `.c` file with several `.h` files). Try to make files no longer than 10k lines.
- **Use the LRS version numbering system.**
- **Never use non-ASCII characters in your source code.** Just don't, there is basically never any need for it.

## Example

Here is a short example applying the above shown style:

```
TODO (for now see LRS projects like Anarch, small3dlib, SAF etc.)
```

---

programming\_tips

## Programming Tips

This is a place for sharing some practical programming tips.

- **Add by small steps**, spare yourself debugging hell later, do one step after another (see also orthogonality): when adding features/functionality etc. into your code, do it by very small steps and test after each step. Do NOT add multiple things at once. If you add 3 features at once and then find out the program doesn't work, you will have an extremely hard time finding out the bug because it may be in feature 1, feature 2, feature 3 or ANY COMBINATION of them, so you may very well never find the bug. If you instead test after adding each step, you find potential bugs immediately which will make fixing them very quick and easy.
- **Program on a weak computer** or alternatively use some utility such as cpulimit to make your hardware weaker, this will help you make your program efficient (and learn how to do it), any inefficiency will be immediately apparent as your program will simply run slow or swap. Using a physically weak computer is best as it is limited in all aspects so it will also help you make the program easy to develop on such computer etc., small embedded devices such as open consoles are ideal.
- **No indentation for temporary code**: Tiny "workflow" tip: when adding new code, keep it unindented so that you know it's the newly added code and can delete it at any time. Only when you test the added code, indent it correctly to incorporate it as the final code. Of course, this fails in languages where indentation matters (Python cough cough) but similar effects can be achieved e.g. by adding many empty lines in front of/after the temporary code.
- **Comments/preprocessor to quickly hide code**: It is a basic trick to comment out lines of code we want to temporarily disable. However preprocessor may work even better, e.g. in C if you want to be switching between two parts of code, instead of constantly commenting one part and uncommenting the other just use `#if 0` and `#else` directives around the two parts. You can switch between them by just changing 0 to 1 and back. This can also disable parts of code that already contain multiline comments (unlike a comment as nested multiline comments aren't allowed).
- **KEEP IT SIMPLE** and keep it LRS, do not blindly follow mainstream ways and "workflows" as those are more often than not horrible. For example instead of using some uber bug tracker, you should use a simple plaintext TODO.txt file; instead of using an IDE use vim or something similar. Stay away from OOP, dependencies etc.
- **Don't listen to advice of anyone who does programming for living**, he's most definitely accustomed to the worst ways of programming and will try to push you to OOP, bloat, proprietary tech, tranny software, GitHub etc. Listening to advice of such people is like taking advice on whether to take drugs from a drug dealer.
- **Most true programming is done away from the computer** -- soydevs think that a good programmer just spends hours in front of a computer bashing the keyboard and drinking litres of coffee to stay alive and PRODUCTIVE; indeed, they usually do, but they are not good programmers, their time is spent slaving the computer doing maintenance, debugging, googling, updating and socializing on Twitter. A good programmer actually programs everywhere: when going for walk, before falling asleep, when sleeping, when watching a movie etc. He only starts writing a serious program after years of thinking about it and already having most of it programmed in his head; sitting in front of a computer and writing the algorithm down is only the final smaller part of the journey.
- It can't be repeated enough times: minimize ALL kinds of dependencies, don't use what you don't necessarily need -- this doesn't just apply to libraries but also design decisions. E.g. if you're making a compiler, make it a single pass compiler if at all possible, don't perform several source code passes if that's not absolutely necessary (which would however likely signify some flaw in the design of your language). Use fixed point instead of floating point if you can, software rendering instead of GPU rendering etc. If you're making something that transforms text to another text (e.g. machine translation), make it a filter with constant memory complexity if that's possible, i.e. do not require the

- program to load a whole input file to memory. Etcetc.
- During development turn off optimization flags for faster compiling and turn on verbosity and various checks, e.g. -Werror -Wall -Wextra -Wpedantic for C.
- By Unix philosophy **don't be afraid to throw away your code and start over and better** -- next time you'll most likely write the same program a lot better and if you're a Unix programmer, your programs are small, possible to be reimplemented quickly. This has even been generalized into a wisdom that says "plan to throw away one", i.e. when approaching a new issue, you quite frequently start writing a program you know you will throw away when it's finished, just to start over and better; the first program just serves to help you understand the true essence of the problem and foresee the real problems you will face.
- **Go out!** This is related to the other point -- you shouldn't just sit at the computer when programming, get up and go for a walk, do something else, take a shower, go swim, do something in the garden, repair some stuff or something like that. Fresh air and sunlight helps the brain, it makes you feel better and it's been shown that walking helps activate some important brain centers, many people actually say they have to walk when thinking hard { Can confirm. ~drummyfish } Changing your environment and getting out of the current focus on the letter on the screen can kick off some real great idea, seeing seemingly unrelated things in nature can spark some inspiration. If you're stuck, take a day off, just sleeping and approaching the problem fresh does miracles. This really does help. It may help to **carry around a blog for taking notes** so that you don't have to stress about forgetting the ideas -- prefer paper blog, leave all electronics at home.
- TODO: moar

---

progress

## Progress

The true definition of progress is "advancement towards more good", though in the mainstream the term has been twisted to mean things such as "more complicated technology", "bigger economy" and so on. Idiots rarely think, they can't ask a series of two questions in a row such as "what will this lead to and is the result what we want?", they can only understand extremely simple equalities such as "moar buttons in a program = more gooder", hence the language degeneration.

It's important to realize that by definition the only true progress that matters is just that which gets us closer to our ideal society, i.e. progress is only that which makes the life of every individual better as a whole and all other kinds of "progress", such as technological, scientific, artistic, political and so on only exist SOLELY to serve the main progress of well being of individuals. A thousand year leap in technological development is worth absolutely nothing if it doesn't serve the main goal, it's useless if we can send a car to space, harvest energy of a whole star or find the absolute meaning of life if that doesn't serve the main goal -- in fact such "progress" is nowadays mostly made so that it works AGAINST the main goal, i.e. corporations develop more complicated technology to exploit people more and make them more miserable -- that is not true progress, it is its exact opposite.

Here is a comparison of what TRUE progress is and what it isn't:

| what                                             | before | after                          | is it progress?  |
|--------------------------------------------------|--------|--------------------------------|------------------|
| You see food, can you just eat it?               | yes    | no, you have to pay for it     | no, the opposite |
| Can you draw the same picture as someone else?   | yes    | no, because of copy"right"     | no, the opposite |
| Can you walk around naked?                       | yes    | no                             | no, the opposite |
| Can you pee wherever you want?                   | yes    | no                             | no, the opposite |
| Do you have to remember ten different passwords? | no     | yes                            | no, the opposite |
| Can you drink river water?                       | yes    | no, likely has toxic chemicals |                  |

| what                                                | before                         | after                             | is it progress?  |
|-----------------------------------------------------|--------------------------------|-----------------------------------|------------------|
| Do you have to check your bank account often?       | no, there are no banks         | yes                               | no, the opposite |
| You have to do something you hate most of the day?  | yes, e.g. work on the field    | yes, e.g. work on a computer      | no               |
| Are you forced to work?                             | yes, with a whip               | yes, with law/tech/consumerism    | no               |
| Can your society do something spectacular?          | yes, build a pyramid           | yes, fly a toy helicopter on Mars | no               |
| You are sick, will you get treatment?               | only if you're special (noble) | only if you're special (rich)     | no               |
| Will you get education?                             | only if you're special (noble) | only if you're special (rich)     | no               |
| What do you kill each other with?                   | sticks                         | machine guns                      | no               |
| Can you do something you enjoy most of the day?     | no                             | yes                               | yes              |
| Can you talk to someone on the other side of Earth? | no                             | yes, thanks to Internet           | yes              |
| Can you have sex with anyone you want?              | no                             | yes                               | yes              |
| How long will you likely live?                      | not beyond 50                  | probably above 65                 | yes              |
| Can you communicate complex ideas to others?        | no (apes)                      | yes, thanks to language           | yes              |
| What do you kill each other with?                   | sticks                         | nothing                           | yes              |

proprietary

## Proprietary

The word proprietary (related to the word *property*) is used for intellectual works (such as texts, songs, computer programs, ...) that are someone's fully owned "intellectual property" (by means of copyright, patents, trademarks etc.), i.e. those that are not free as in freedom because they cannot be freely copied, shared, modified, studied etc. This word has a negative connotation because proprietary works serve capitalist overlords, are used to abuse others and go against freedom. The opposite of proprietary is free (as in freedom, NOT price) (also *libre*): free works are either those that are completely public domain or technically owned by someone but coming with a free (as in freedom) license that voluntarily waives all the harmful legal rights of the owner. There are two main kinds of proprietary works (and their free counterparts): proprietary software (as software was the first area where these issues arose) (versus free software) and proprietary art of other kind (music, pictures, data, ...) (versus free cultural art).

As said, proprietary software is any software that is not free (as in freedom)/open source software. Such software denies users and creators their basic freedoms (freedom of unlimited use, studying, modifying and sharing) and is therefore evil; proprietary software is mostly capitalist software designed to abuse its user in some way. Proprietary code is often secret, not publicly accessible, but there are many programs whose source code is available but which is still proprietary because no one except the "owner" has any legal rights to fixing it, improving it or redistributing it.

Examples of proprietary software are MS Windows, MacOS, Adobe Photoshop and almost every game. Proprietary software is not only extremely harmful to culture, technology and society in general, it is downright dangerous and in some cases life-threatening; see for example cases of medical implants such as pacemakers running secret proprietary code whose creator and maintainer goes bankrupt and can no longer continue to maintain such devices already planted into bodies of people -- such cases have already

appeared, see e.g. *Autonomic Technologies* nervous system implants.

Proprietary software licenses are usually called EULAs.

By extension besides proprietary software there also exist other proprietary works, for example proprietary art or databases -- these are all works that are not free cultural works. Even though for example a proprietary movie probably isn't IMMEDIATELY as dangerous as proprietary software, it may be just as dangerous to society in the long run. Examples of proprietary art is basically anything mainstream that's not older than let's say 50 years: Harry Potter, all Hollywood movies, basically all pop music, basically all AAA video game art and lore etcetc.

**Is it ever okay to use proprietary software?** If you have to ask, the answer is no, you should avoid proprietary software as much as possible (considering in today's society you probably can't even take a shit without using some form of proprietary software). Proprietary software is cancer, it is like hard drugs, poison, radioactive toxic material, biological virus -- you have to treat it as such. For this reason to most people, especially newcomers to the free world, the best, simplest and safest advice is to completely avoid anything proprietary; this helps you get out of the addiction, break out of the system, find free alternatives and avoid harm to yourself and others. Once one becomes an expert he start to see the answer may be more complex of course, as with everything -- for example in order to make a free clone of something proprietary, we often have to reverse engineer it, which often means having to run it; however this has to only be done by experts who know the dangers and how to handle them, just like handling of a highly dangerous biological virus should only ever be done by an expert in safe laboratory under strictly controlled conditions.

---

proprietary\_software

## Proprietary Software

Go here.

---

pseudo3d

## Pseudo 3D

The term pseudo 3D, also 2.5D or primitive 3D, is used for computer graphics that tries to create the illusion of 3D rendering while in fact only utilizing simpler techniques; genuine 3D rendering is in this case called true 3D. On consumer computers it is nowadays mostly a thing of the past as everything including cell phones now has a powerful GPU capable of most advanced 3D rendering, nevertheless for suckless/KISS/LRS programming the techniques used in the past are very interesting and useful.

For example BSP rendering rendering in early games such as Doom is generally called pseudo 3D in the mainstream, however it is pretty debatable what exactly should classify as true 3D and what not because any computer rendering technique will inevitably have some kind of simplification of the true 3D reality of real life. And so the debate of "was Doom really 3D" arises. One side argues that in Doom's BSP rendering it for example wasn't possible to look up and down or have rooms above other rooms, all due to the limitations of the rendering system which this side sees as "not real 3D". However even modern 3D renderers have limitations such as mostly being able to only render models made out of triangles (while reality can have completely smooth shapes) or having a limited resolution of textures. Where to draw the line for "true 3D" is subjective -- we see it as reasonable to say that **if it looks 3D, it IS 3D**, i.e. we think Doom's graphics WAS really 3D, albeit limited. For this reason we also advise to rather use the term **primitive 3D** rather than pseudo 3D.

Techniques associated with primitive 3D are for example 2D raycasting, BSP rendering, mode7, parallax scrolling, voxel space terrain rendering or perspective-scaled sprites.

## See Also

- software rendering
- bsp rendering

pseudoleft

## Pseudoleft

See [left vs right](#).

---

pseudominimalism

## Pseudominimalism

Pseudominimalism is the kind of technology design which aims to appear [minimalist](#) on the outside while being [bloated](#) on the inside. Rather than trying to achieve a [truly good](#), minimalist design from the ground up, with all its advantages, pseudominimalism merely attempts to hide the ugliness of its internals and appeal purely by the looks. A typical example might be a website that has a minimalist look -- a blank background with sans-serif [font](#) text and a few nice looking vector shapes -- which in the background sneakily uses dozens of [JavaScript](#) frameworks and libraries and requires a high end [CPU](#) in order to even appear responsive. Essentially all [modern](#) "retro" video [games](#) are pseudominimalist in design, they use pixelated graphics but are created in enormous frameworks such as [Unity](#) or [Godot](#); even projects calling themselves "minimalist", such as many [fantasy consoles](#), are in truth only pseudominimalist, written in extremely high level languages such as [JavaScript](#). [Apple](#) is heavily practicing pseudominimalism.

While true minimalists do appreciate minimalist look as well, pseudominimalists are obsessed with visuals and after a while you learn to spot pseudominimalist just by their attempts at what they call a "clean design" or "user experience" -- a true minimalist uses minimalism so that bullshit doesn't stand in his way, a **pseudominimalist is just a snob** using visuals to pretend he's an intellectual. You will see the sweat that went into font choice, spacing of paragraphs with this tryharding leaking even to language in which he tries to use minimum of words which just makes it hard to understand what he wants to say. A typical example is the [shitty 100r wiki](#).

Another example is presented by many "[modern](#)" [CLI](#) programs which [code monkeys](#) use to impress their [YouTube](#) viewers or to feel like matrix haxors. Some people believe that anything running in the command line has to be minimalist by a law of nature which is less and less true as we progress into the [future](#). A lot of [capitalist software](#) add a CLI interface ex post **on top** of an already bloated program, often by simply disabling [GUI](#) (but leaving all its [dependencies](#) in). An example may be the [gomux](#) chat client.

Yet another kind of pseudominimalism appearing among the new generation of crippled pseudoprogrammers is all about writing very few LOC in some incredibly bloated language and calling that "minimalism". Something like a *Minecraft clone in 100 LOC of Python using only Python standard library*, the catch of course being that [Python](#) itself is hugely bloated and its standard library is enormous, therefore they just hide all the complexity out of view. Effort like that is indeed completely useless and only serves for flexing in front of beginners who can't spot the trick. Even if obvious, it has to be noted that **minimalist software cannot be written in a bloated language**.

---

pseudorandomness

## Pseudorandomness

*Randomness is too important of a matter to be left to chance.*

TODO

---

public\_domain\_computer

# Public Domain Computer

Public domain computer is yet nonexistent but planned and highly desired simple ethical computer (in the common meaning of the word) whose specification is completely in the public domain and which is made with completely selfless LRS-aligned goal of being absolutely non-malicious and maximally helpful to everyone. It should be the "people's computer", a simple, suckless, user-respecting hackable computer offering maximum freedom, a computer which anyone can study, improve, manufacture and repair without paying any "intellectual property" fees, a computer which people can buy (well, while money still exist) for extremely low price and use for any purpose without being abused or oppressed.

"Public domain computer" is just a temporary placeholder/general term, the actual project would probably be called something different.

The project is basically about asking: what if computers were designed to serve us instead of corporations? Imagine a computer that wouldn't stand in your way in whatever you want to do.

In our ideal society, one of the versions of the public domain computer could be the less retarded watch.

Note that **the computer has to be 100% from the ground up in the true, safe and worldwide public domain**, i.e. not just "FOSS"-licensed, partially open etc. It should be created from scratch, so as to have no external dependencies and released safely to the public domain e.g. with CC0 + patent waivers. Why? In a good society there simply have to exist basic tools that aren't owned by anyone, tools simply available to everyone without any conditions, just as we have hammers, pencils, public domain mathematical formulas etc. -- computing has become an essential part of society and it certainly has to become a universal "human right", there HAS TO exist an ethical alternative to the oppressive capitalist technology so that people aren't forced to accepting oppression by their computers simply by lack of an alternative. Creating a public domain computer would have similarly positive effects to those of e.g. universal basic income -- with the simple presence of an ethical option the oppressive technology would have a competition and would have to start to behave a bit -- oppressive capitalist technology nowadays is possibly largely thanks to the conspiracy of big computer manufacturers that rely on people being de facto obliged to buy one of their expensive, proprietary, spyware littered non-repairable consumerist computer with secret internals.

**The computer can (and should) be very simple.** It doesn't -- and shouldn't -- try to be the way capitalist computers are, i.e. it would NOT be a typical computer "just in the public domain", **it would be different by basic design philosophy** because its goals would completely differ from those of capitalists. It would follow the LRS philosophy and be more similar to the very first personal computers rather than to the "modern" HD/bloated/superfast/fashion computers. Let us realize that even a very simple computer can help tremendously as a great number of tasks people need can actually be handled by pretty primitive computers -- see what communities do e.g. with open consoles.

Even a pretty simple computer without an operating system is able to:

- Browse much of the Internet, e.g. smol web (no JavaScript websites, gopher, ...).
- Handle communication, e.g. email, IRC, ...
- Allow reading, writing and storing books, e.g. those from Project Gutenberg or offline Wikipedia -- this can tremendously help education e.g. in the third world.
- Run basic software such as calculator, stopwatch, calendar, note taking, alarm clock, memory-card reader, picture viewer, even simple games etc.
- Serve as an embedded computer, e.g. DYI people and small business may use the computer in similar ways Raspberry pi is used nowadays (auto switching lights, opening doors, recording data from sensors, tiny robots, ...).
- Be programmed and serve as an educational tool for programming.
- Do many scientific calculations.
- Control peripherals through simple interfaces.
- Handle simple multimedia such as low-res images and animations, 8bit sounds...
- ...

## Details

The project wouldn't aim to create a specific single "model" of a computer but rather blueprints that would be easily adjusted and mapped to any specific existing technology -- the goal would be to create an abstract hardware specification as well as basic software for the computer.

Abstract hardware specification means e.g. description on the logic gate level so that the computer isn't dependent on any contemporary and potentially proprietary lower level technology such as CMOS. The project would simply create a big logic circuit of the computer and this description could be compiled/synthesized to a lower level circuit board description. The hardware description could also be parameterized so that certain features could be adjusted -- for example it might be possible to choose the amount of RAM or disable specific CPU instructions to make a simpler, cheaper circuit board.

**The computer would have to be created from the ground up**, with every design aspect following the ultimate goal. The project roadmap could look similarly to this one:

1. Create a programming language that will be usable both as a scripting and compiled language for the computer. We already have one -- comun -- though it is not fully finished yet. Now we can already start writing software for the computer. Optionally make other languages such as C compile to our ISA.
2. Design a simple instruction set architecture (ISA). This will provide some challenge but will be doable.
3. Write basic software in our language, mainly:
  - Custom tools for designing, simulating and testing logic circuits. Not extremely difficult if we keep it simple.
  - Emulator of our custom ISA so that we can run and test it on our current computers. It will also be useful to make our computer possible to be run as a virtual hardware on other platforms.
  - Shell that will serve to performing basic tasks with the computer, e.g. using it as a calculator or interactively programming it in simple ways. The shell will also serve as a kind of operating system, or rather a simple program loader. For now the shell can run on our current computers where we can test it and fine tune it.
  - Compiler -- this basically just means self hosting our compiler.
  - Basic tools like a text editor, compression utility etc.
4. With the logic circuit tools design a simple MCU computer based on the above mentioned ISA. This is doable, there are hobbyists that have designed their own 8bit CPUs, a few collaborating people could definitely create a nice MCU if they keep it simple (no caching, no floating point, no GPUs, ...).
5. Compile the MCU logic-level description to an actual circuitboard, possibly even with proprietary tools if other aren't available -- this may be fixed later.
6. Manufacture the first physical computer, test it, debug it, improve it, give it to people, ...
7. Now the main goal has been touched for the first time, however the real fun only begins -- now it is needed to spread the project, keep improving it, write more software such as games etc. :)

## See Also

- comun
- uxn
- less retarded watch
- PDOS

---

public\_domain

## Public Domain

If an "intellectual work" (song, book, computer program, ...) is in the public domain (PD), it has no "owner", meaning no one has any exclusive rights (such as copyright or patent) over the work, no one can dictate how and by whom such work can be used and so anyone can basically do anything with such work (anything that's not otherwise illegal of course).



LRS highly supports public domain and recommends programmers and artists put their works in the public domain using waivers such as CC0 (this very wiki is of course released as such).

Public domain is the ultimate form of freedom in the creative world. In public domain the creativity of people is not restricted. Anyone can study, remix, share and improve public domain works in any way, without a fear of being legally bullied by someone else.

The term "public domain" is sometimes used vaguely to mean anything under a free license, however this use is incorrect and greatly retarded. **Public domain is NOT the same thing as free (as in freedom) software, free culture or freeware (gratis, free as in beer) software**. The differences are these:

- Unlike public domain, **free software and free cultural works are usually still "owned" by someone**, they just try to relax the rules and make them less oppressive. A public domain work is completely unlimited and belongs to everyone and no one, while free software/culture may still require and legally enforce certain freedom-compatible conditions such as giving credit to the author or copyleft.
- **Public domain software is not always free software** -- PD software is free (as in freedom) only if its source code is available and also in the public domain (without source code freedoms 1 and 2 in the definition of free software are violated).
- **Freeware/gratis just means available for no price**, very often under specific restrictive conditions such as "for personal use" only and without the access to the source code. Public domain is not only gratis but also without any legal limitations on use.

## Which Works Are In The Public Domain?

This is not a trivial question, firstly because the term *public domain* is not clearly defined: the definition varies by each country's laws, and secondly because it is non-trivial and sometimes very difficult to assess the legal status of a work.

Corporations and capitalism are highly hostile towards public domain and try to destroy it, make it effectively non-existing, as to eliminate "free" works competing with the consumerist creations of the industry. Over many years they have pushed towards creating laws that make it extremely difficult and rare for works to fall into public domain.

Sadly due to these shitty laws most works created in latest decades are NOT in the public domain because of the copyright cancer: copyright is granted automatically, without any registration or fee, to the author of any shitty artistic creation, and its term lasts mostly for **the whole life of the author plus 70 years!** In some countries this is life + 100 years. In the US, copyright lasts 96 years from the publication of the work (every January 1st there is so called public domain day celebrating new works entering the US public domain). In some countries it is not even possible to legally waive (give up) one's copyright. And to make matters worse, copyright isn't the only possible restriction of an intellectual work, there are also **patents, trademarks, personality rights** and other kinds of intellectual property.

Another bad news is that works in a **"weak" public domain**, i.e. most recent PD works or works that entered PD by some obscure little law, may as well stop being PD by introducing some shitty retroactive law (which has happened). So one may not be feeling completely safe going crazy by utilizing some recent PD works.

We therefore devise the term **safe/strong public domain**. Under this we include works that are pretty safely PD more or less world-wide, even considering possible changes in laws etc. Let us include these works:

- Works published at least 100 years ago whose author probably died at least 70 years ago.
- Works **clearly and properly** marked by a reliable PD waiver such as CC0. However an extra effort needs to be taken to assure that the work e.g. isn't a derivative work of copyrighted work, or that patents are waived with software.
- Works that under any "reasonable" law can not be covered by "intellectual property", e.g. math equations, colors etc.

Creative commons has created a **public domain mark** that helps mark and find works that should be in a world-wide public domain (this is not a waiver though, it is basically only used as a metadata for very old

works to be better searchable).

There are a number of places on the internet to look for public domain works, for a list see below.

**Should you release you own works to the public domain?** Definitely yes! From our point of view public domain is the only option as we deem any "intellectual property" immoral, however even if you disagree with us, you may want to release at least some of your works into public domain, if only out of altruism, no longer caring about your old works, out of curiosity or to make yourself a bit popular in the free culture community (thought this is a motivation we don't entirely embrace). **Are you afraid to do so?** It is natural, letting go of something you spend part of your life on can raise a bit of anxiety, but this is just a fear of making the first step to the unknown, a fear almost entirely artificial, created by capitalist propaganda; making this decision will really most likely only have positive effects unless you actually had SERIOUS plans to make a business of your proprietary art. Practically the worst that can happen is that your work goes unnoticed and unappreciated. If you are still hesitant, try to go slowly, first release one thing, something small, and see what happens.

{ I remember myself how anxious I was about making the decision to release all my work into public domain, despite knowing it was the right thing to do and that I wanted to do it. I felt emotional about giving away rights to art I put so much love and energy into, fearing the evil vultures of the Internet would immediately "steal" it all as soon as I release it. I overcame the fear and now, many years later, I can say that not once have I regretted it, literally not a single case of abuse of my work happened (that I know of anyway), despite some of it becoming kind of popular. I only received love of many people who found my work useful, and even received donations from people. I've seen others put my work to use, improve it, I get mail from people thanking me for I've done. Of course this all is not why I did it, but it's nice, I write about it to share a personal experience that will maybe give you the courage to do the right thing as well. ~drummyfish }

## How To Create Public Domain Works

To create a public domain work you must ensure that after you release it, no one will hold exclusive intellectual property rights to it -- most notably we will be trying to remove copyright from the work (which arises automatically, last extremely long and is most annoying), but know that there are potentially also other rights to take into account, e.g. patents, trade marks, trade dress, personality rights, etc. (in usual cases you don't have to deal with these as they apply only to some things in some situations, but for things like program source code you may need to look into them). We will remove such rights with licenses or waivers, i.e. a legal text which we attach to our works and which says we just give up our rights. Sadly this is not trivial to do.

If you want to create a PD work, then generally in that work **you must not reuse any non-public domain work**. So, for example, you can NOT create a public domain fan fiction story about Harry Potter because Harry Potter and his universe is copyrighted (your fan fiction here would be so called derivative work or a copyrighted work). Similarly you can't just use randomly googled images in a game you created because the images are most likely copyrighted. Small and obscure exceptions (trivial bitmap fonts, freedom of panorama, ...) to this may exist in laws but it's never good to rely on such quirky laws (they may differ between countries etc.), it's best to keep it safe and simply avoid utilizing anything non-PD within your works. If you can, create everything yourself, that's the safest bet.

Note that even things such as music/sound samples, text fonts or paint brushes may sometimes be copyrighted. Just be careful, try to make everything from scratch -- yes, it sucks, because copyright sucks, but this is simply how we bypass it. Making everything yourself from the ground up also teaches you a lot and makes your art truly original, it's not a wasted time.

Also **you must NOT use anything under fair use**! Even though you could lawfully use someone else's copyrighted work under fair use, inclusion of such material would, by the fair use rules, limit what others would be able to do with your work, making it restricted and therefore not public domain. Example: you can probably write a noncommercial Harry Potter fan fiction and share it with friends on the internet because that's fair use, however this fan fiction can never be public domain because it can't e.g. be used commercially, that would no longer fall under fair use, i.e. there is a non-commercial-use-only restriction burdening your work. It doesn't even help if you get an explicit permission to use a copyrighted work in your work unless such permission grants all the right to everyone (not just your work). { I got a mascot removed from SuperTuxKart by this argument, mere author's permission to use his work isn't enough to make it free

as in freedom. ~drummyfish }

Also **do NOT USE AI**, not even for things like upscaling and enhancements. NO JUST DO NOT. NO, your argument is invalid, just DO NOT USE IT. In theory it may be legit, but there's just huge amount of doubt, uncertainty and legal mess. To name a few potential issues: AI may create a derivative work of something it has seen in its training dataset (which even if "open"-licensed still may contain material of non-free things that may be legal in the context of the dataset but not in the context of the generated result, e.g. "freedom of panorama"), the copyright status of AI works themselves is not as of yet clear and even once it's established, it may differ by country AND there is a danger of retroactive changes (once it becomes too easy to create PD works with AI capitalists can just push a law that will say AI can't be used for this because "economy" and yes, it may even be used retroactively, yes, they can do it, it already happened). Furthermore even if AI works are made legit, terms and condition of most usable AI software will still negate this (they already do, EVEN if you pay for it), it's not even clear if they can do this (or it may depend on territory and time) but it's a threat. Also AI is shit, bloat and serves mostly capitalists to produce huge quantities of cheap shit for consumerist games, we just don't need this. You may think "haha I'll create one trillion PD textures and post them to Opengameart and save the world" -- that's literally what everyone is doing right now, it's the worst kind of spam that is now just killing the site, please don't even think of this. Create something small but nice, something whose legitimacy as your own work that you give away can not be questioned.

So you can only use your own original creations and other public domain works within your PD work. Here you should highly prefer your own creations because that is legally the safest, no one can ever challenge your right to reuse your own creation, but there is a low but considerable chance that someone else's PD work isn't actually PD or will cease to be PD by some retroactive law change. So when it only takes a small effort to e.g. photograph your own textures for a game instead of using someone else's PD textures, choose to use your own.

{ NOTE: The above is kind of arguing for reinventing wheels which goes a little bit against our philosophy of remixing and information sharing, but we are forced to do this by the system. We are forced to reinvent wheels to ensure that users of our works can't be legally bullied. ~drummyfish }

In cases where you DO reuse other PD works, try to minimize their number and try to make sure they belong to the actual **safe** public domain (see above). This again minimizes legal risk and additionally makes it easy to document and prove the sources.

As a next step make sure you clearly **document** your work and the sources you use. This means you write down where all the works contained in your work come from, e.g. in your readme. Explicitly mention which things you have created yourself ("*I, ..., have created everything myself except for X, Y and Z*") and which things come from other people and where you have found them. It is great to also archive the proofs of the third party source being public domain (e.g. use the Internet Archive to snapshot the page with a PD texture you've found). For works that allow it (e.g. source code, text, websites, ...) it is good to use version control systems such as git that record WHAT, WHEN and by WHO was contributed. This can all help prove that your work is actually safe and/or remove contributions that caused some legal trouble.

If you collaborate with someone on the work, it must be clear that ALL contributors to the work follow what we describe here (e.g. that they all agree to the license/waiver you have chosen etc.). It is safer if there are fewer contributors as with more people involved the chance of someone starting to "make trouble" increases.

Finally you need to actually release your work into the public domain. Remember that you want to achieve a **safe, world-wide public domain** (so again you shouldn't try to rely on some weird/obscure laws of your own small country). It must be stressed that it is NOT enough to write "*my work is public domain*", this is simply legally insufficient (and in many countries you can't even put your work into public domain which is why you need a more sophisticated tool). You need to use a public domain waiver (similar to a license) which you just put alongside your work (e.g. into the LICENSE file), plus it is also good to explicitly write (e.g. in your readme) a sentence such as "**I, ..., release this work into public domain under CC0 1.0 (link), public domain**". Bear in mind that the WORDING may be very important here, so try to write this well: we mention the license name AND its version (CC0 1.0, it may even be better to fully state *Creative Commons 1.0*) as well as a link to its exact text and also mention the words *public domain* afterwards to make the intent of public domain yet clearer to any doubters. Here we used what's currently probably the best waiver

you can use: Creative Commons Zero (CC0) -- this is what we recommend. However note that CC0 only waives copyright and not other things like trademarks or patents, so e.g. for software you might need to add an extra waiver of these things as well.

{ I personally use the following waiver IN ADDITION to CC0 with my software to attempt waiving of patents, trademarks etc. I made it by taking some standard waiver companies use to steal "rights" of their employees and modifying it to make it a public domain waiver. If you want to use it, make sure you mention it is an EXTRA, additional waiver alongside CC0. The waiver text follows. ~drummyfish

*Each contributor to this work agrees that they waive any exclusive rights, including but not limited to copyright, patents, trademark, trade dress, industrial design, plant varieties and trade secrets, to any and all ideas, concepts, processes, discoveries, improvements and inventions conceived, discovered, made, designed, researched or developed by the contributor either solely or jointly with others, which relate to this work or result from this work. Should any waiver of such right be judged legally invalid or ineffective under applicable law, the contributor hereby grants to each affected person a royalty-free, non transferable, non sublicensable, non exclusive, irrevocable and unconditional license to this right. }*

NOTE: You may be thinking that it doesn't really matter if you waive your rights properly and very clearly if you know you simply won't sue anyone, you may think it's enough to just write "do whatever you want with my creation". But you have to remember others, and even you yourself, can't know if you won't change your mind in the future. A clear waiver is a **legal guarantee** you provide to others, not just a *vague promise of someone on the Internet*, and this guarantee is very valuable, so valuable that whether someone uses your work or not will often come down to this. So waiving your "rights" properly may increase the popularity and reusability of your work almost as much as the quality of the work itself.

For an example of a project properly released into public domain see the repository of our LRS game Anarch.

## Where To Find Public Domain Works

There are quite a few places on the Internet where you may find public domain works. But firstly let there be a warning: you always **have to** check the public domain status of works you find, it is extremely common for people on the Internet to not know what public domain is or how it works so you will find many *false positives* that are called public domain but are, in fact, not. This article should have given you a basic how-to on how to recognize and check public domain works. With this said, here is a list of some places to search (of course, this list will rot with time):

- **Very old works and traditional folk art** are mostly in the public domain, e.g. Shakespeare's plays or folk songs. Nice example of reusing folk art is e.g. Richard Stallman's Free Software Song that takes the melody of a Bulgarian folk song *Sadi Moma*. However watch out for traps, e.g. trademarks that may exist despite no copyright (e.g. Encyclopedia Britannica) or weird nationalist laws against disrespecting a country's folklore that may possibly exist too.
- **Wikimedia Commons**: Contains only free as in freedom works among which are many PD ones. You can search for them with queries such as `cat:incategory:cc-zero`. This site is quite reliable and serious about licensing, if you find a work marked as PD here, you can be reasonably sure this information is true.
- **Internet Archive**: The biggest Internet archive, huge amount of mainly old works such as scanned books and photos. Beware that this site contains all kinds of works from PD to proprietary and works marked as PD should be checked as there can be errors. There is an *advanced search* tool that can help in searching for PD works, for example this query tries to achieve this.
- **Opengameart**: Site for sharing free as in freedom game art (pictures, 3D models, sounds, ...) among which are many under CC0, i.e. PD. Submitted works are checked reasonably well so any CC0 work you find here is likely truly PD.
- **Freesound**: Site for sharing sound recordings and sound effects, contains many CC0 sounds that should be PD.
- **Project Gutenberg**: Archive of old digitized books. NOT ALL are PD, but the real old ones should be. Generally books from before the 20th century should be PD.
- **Stocksnap**: Quality photos and "stock images" under CC0, i.e. PD.
- **Librivox**: Public domain audiobooks made by volunteers that read PD books from Project Gutenberg.

- **Wikisource**: Repository of texts, similar to Project Gutenberg, same rules apply (not all texts here will be PD but the real old ones should be).
  - **Openclipart**: Vector graphics, all under CC0, i.e. PD in theory, **however** there do appear pictures that are derivative works of copyrighted works for which of course this is irrelevant. Check very well anything you download from here.
  - **Blendswap**: Site for exchanging 3D models for Blender, not all models are PD but the ones marked CC0 should be, however **NOT those marked as "fan art"!**
  - **Wikidata**: Database of "everything", published as a whole under CC0 which should make it PD, **however** it will contain information about proprietary works which may make this status questionable sometimes. If you only use data that don't fall under this you should be safe.
  - ...
- 

p\_vs\_np

## P vs NP

*P vs NP* is one of the greatest and most important yet unsolved problems in computer science: it is the question of whether the computational class P is equal to class NP or, in simple terms, whether certain problems for which no "fast" solution is known can in fact be solved "fast". This is very important e.g. for algorithms used in cryptography. This problem is in fact so important that it's one of the seven Millennium Prize Problems. **There is a million dollar reward for solving this problem.**

It is believed and sometimes relied on that  $P \neq NP$  (in which case  $P$  would be a proper subset of  $NP$ ), but a mathematical proof doesn't exist yet. If it was surprisingly proven that  $P = NP$ , there might be practical consequences for cryptography in which most algorithms rely on the problems in question being difficult (slow) to solve -- a proof of  $P = NP$  could lead to fast algorithms for breaking encryption, but that is not a certainty, only one of possible scenarios. However any solution to this problem would be revolutionary and ground breaking.

## Explanation

In the context of computational complexity of algorithms we talk about different types of algorithm time complexities, i.e. different "speeds" of algorithms. This "speed" doesn't mean actual running time of the algorithm in real life but rather how quickly the running time grows depending on the amount of input data to it (so rather something akin "scalability"), i.e. we are interested only in the shape of the function that describes how the amount of input data affects the running time of the algorithm. The types of time complexity are named after mathematical functions that grow as quickly as this dependence, so we have a *constant* time complexity, *logarithmic* time complexity, *linear* time complexity etc.

Then we have classes of computational problems. The classes divide problems based on how "fast" they can be solved.

The class  $P$  stands for **polynomial** and is defined as all problems that can be solved by an algorithm run on a **deterministic Turing machine** (a theoretical computer) with a *polynomial* time complexity.

The class  $NP$  stands for **non-deterministic polynomial** and is defined as all problems that can be solved by an algorithm run on a **non-deterministic Turing machine** with a *polynomial* time complexity. I.e. the definition is the same as for the  $P$  class with the difference that the Turing machine is non-deterministic -- such a machine is faster because it can make kind of "random correct guesses" that lead to the solution more quickly. Non-deterministic computers are only theoretical (at least for now), computers we have in real life cannot perform such randomly correct guesses. It is known that the solution to all  $NP$  problems can be verified in *polynomial* time even by a deterministic Turing machine, we just don't know if the solution can also be found this quickly.

Basically  $P$  means "*problems that can be solved quickly*" and  $NP$  means "*problems whose solutions can be verified quickly but we don't know if they can also be solved quickly*".

The question is whether all  $NP$  problems are in fact  $P$  problems, i.e. whether *all problems that can be verified quickly can also be solved quickly*. It is believed this is not the case.

---

python

# Python

What if pseudocode was actually code?

TODO

---

quantum\_gate

## Quantum Gate

{ Currently studying this, there may be errors. ~drummyfish }

Quantum (logic) gate is a quantum computing equivalent of a traditional logic gate. A quantum gate takes as an input  $N$  qubits and transforms their states to new states (this is different from classical logical gates that may potentially have a different number of input and output values).

Quantum gates are represented by complex matrices that transform the qubit states (which can be seen as points in multidimensional space, see Bloch sphere). A gate operating on  $N$  qubits is represented by a  $2^N \times 2^N$  matrix. These matrices have to be **unitary**. Operations performed by quantum gates may be reversed, unlike those of classical logic gates.

We normally represent a single qubit state with a **column vector**  $|a\rangle = a_0 * |0\rangle + a_1 * |1\rangle \Rightarrow [a_0, a_1]$  (look up bra-ket notation). Multiple qubit states are represented as a tensor product of the individual state, e.g.  $|a,b\rangle = [a_0 * b_0, a_0 * b_1, a_1 * b_0, a_1 * b_1]$ . Applying a quantum gate  $G$  to such a qubit vector  $q$  is performed by simple matrix multiplication:  $G * v$ .

## Basic gates

Here are some of the most common quantum gates.

### Identity

Acts on 1 qubit, leaves the qubit state unchanged.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

## Pauli Gates

Act on 1 qubit. There are three types of Pauli gates: X, Y and Z, each one rotates the qubit about the respective axis by  $\pi$  radians.

The X gate is:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The Y gate is:

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

The Z gate is:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

# NOT

The not gate is identical to the Pauli X gate. It acts on 1 qubit and switches the probabilities of measuring 0 vs 1.

# CNOT

Controlled NOT, acts on 2 qubits. Performs NOT on the second qubit if the first qubit is  $|1\rangle$ .

```
1 0 0 0
0 1 0 0
0 0 0 1
0 0 1 0
```

TODO

---

quaternion

## Quaternion

Quaternion is a type of number, just like there are integer numbers, real numbers or imaginary numbers. They are very useful for certain things such as 3D rotations (they have some advantages over using e.g. Euler angles, for example they avoid Gimbal lock, they are also faster than transform matrices etc.). Quaternions are not so easy to understand but you don't actually need to fully grasp and visualize how they work in order to use them if that's not your thing, there are simple formulas you can copy-paste to your code and it will "just work".

Quaternions are an extension of complex numbers (you should first check out complex numbers before tackling quaternions); while complex numbers can be seen as two dimensional -- having the real and imaginary part -- quaternions would be seen as four dimensional. A quaternion can be written as:

$$a + bi + cj + dk$$

where  $a$ ,  $b$ ,  $c$  and  $d$  are real numbers and  $i$ ,  $j$  and  $k$  are the basic quaternion units. For the basic units it holds that

$$i^2 = j^2 = k^2 = ijk = -1$$

**Why four components and not three?** Simply put, numbers with three components don't have such nice properties, it just so happens that with four dimensions we get this nice system that's useful.

Operations with quaternions such as their multiplication can simply be derived using basic algebra and the above given axioms. Note that **quaternion multiplication is non-commutative** ( $q1 * q2 \neq q2 * q1$ ), but it is still associative ( $q1 * (q2 * q3) = (q1 * q2) * q3$ ).

A **unit quaternion** is a quaternion in which  $a^2 + b^2 + c^2 + d^2 = 1$ .

A **quaternion negation** ( $q^{-1}$ ) is obtained by multiplying  $b$ ,  $c$  and  $d$  by  $-1$ .

## Rotations

Only unit quaternions represent rotations.

Rotating point  $p$  by quaternion  $q$  is done as

$$q^{-1} * (0 + p.x i + p.y j + p.z k) * q$$

Rotation quaternion can be obtained from axis ( $v$ ) and angle ( $a$ ) as

$$q = \cos(a/2) + \sin(a/2) * (v.x \ i + v.y \ j + v.z \ k)$$

TODO: compare to euler angles, examples

---

qubit

## Qubit

Qubit is a quantum computing equivalent of a bit. While bits in classical computers can have one of two states -- either 0 or 1 -- a qubit can additionally have infinitely many states "in between" 0 and 1 (so called superposition). Physically qubits can be realized thanks to quantum states of particles, e.g. the polarization of a photon or the spin of a photon. Qubits are processed with quantum gates.

**Whenever we measure a qubit, we get either 1 or 0**, just like with a normal bit. However during quantum computations the internal state of a qubit is more complex. This state determines the **probabilities** of measuring either 1 or 0. When the measurement is performed (which is basically any observation of its state), the qubit state collapses into one of those two states.

Now we will be dealing with so called **pure states** -- these are the states that can be expressed by the following representation. We will get to the more complex (mixed) states later.

The state of a qubit can be written as

$$A * |0\rangle + B * |1\rangle$$

where  $A$  and  $B$  are complex numbers such that  $A^2 + B^2 = 1$ ,  $|0\rangle$  is a vector  $[0, 1]$  and  $|1\rangle$  is a vector  $[1, 0]$ .  $A^2$  gives the probability of measuring the qubit in the state 0,  $B^2$  gives the probability of measuring 1.

The vectors  $|0\rangle$  and  $|1\rangle$  use so called bra-ket notation and represent a vector basis of a two dimensional state. So the qubit space is a point in a space with two axes, but since  $A$  and  $B$  are complex, the whole space is four dimensional (there are 4 variables:  $A$  real,  $A$  imaginary,  $B$  real and  $B$  imaginary). However, since  $A + B$  must be equal to 1 (normalized), the point cannot be anywhere in this space. Using logic<sup>TM</sup> we can figure out that the final state of a qubit really IS a point in two dimensions: a point on a sphere (Bloch sphere). A point of the sphere can be specified with two coordinates: *phase* (yaw, 0 to 2  $\pi$ , can be computed from many repeated measurements) and  $p$  (pitch, says the probability of measuring 1). It holds that:

$$A = \sqrt{1 - p}$$

$$B = e^{(i * \text{phase})} * \sqrt{p}$$

The sphere has the state  $|0\rangle$  at the top (north pole) and  $|1\rangle$  at the bottom (south pole); these are the only points a normal bit can occupy. The equator is an area of states where the probability of measuring 0 and 1 are equal (above the equator gives a higher probability to 0, below the equator to 1).

Now a qubit may actually be in a more complex state than the pure states we've been dealing with until now. Pure states can be expressed with the state vector described above. Such a state is achieved when we start with a qubit of known value, e.g. if we cool down the qubit, we know it has the value  $|0\rangle$ , and transforming this state with quantum gates keep the state pure. However there are also so called **mixed states** which are more complex and appear e.g. when the qubit may have randomly been modified by an external event, or if we start with a qubit of unknown state. Imagine if we e.g. start with a qubit that we know is either  $|0\rangle$  or  $|1\rangle$ . In such case we have to consider all those states separately. A mixed state is composed of multiple pure states. Mixed states can be expressed with so called **density matrices**, an alternative state representation which is able to encode these states.

---

quine



# Quine

Quine is a nonempty program which prints its own source code. It takes no input, just prints out the source code when run (without cheating such as reading the source code file). Quine is basically a self-replicating program, just as in real world we may construct robots capable of creating copies of themselves (afterall we humans are such robots). The name *quine* refers to the philosopher Willard Quine and his paradox that shows a structure similar to self-replicating programs. Quine is one of the standard/fun/interesting programs such as hello world, compiler bomb, 99 bottles of beer or fizzbuzz.

From mathematical point of view quine is a fixed point of a function (not to be confused with fixed point arithmetic) represented by the programming language. I.e. if we see the programming language as a function  $f(x)$ , where  $x$  is source code and the function's output is the program's output, quine is such  $x$  that  $f(x) = x$ . **A quine can be written in any Turing complete language**, the proof comes from the *fixed point theorem* (which says functions satisfying certain conditions always have a fixed point, i.e. a quine).

Similar efforts include e.g. making self matching regular expressions (for this task to be non-trivial the regex has to e.g. be enclosed between /s). Yet another similar challenge is a polyglot program -- one that is a valid program in several languages -- some programs can be quines and polyglots at the same time, i.e. **polyglot quines**.

The challenge of creating quines is in the self reference -- normally we cannot just single-line print a string literal containing the source because that string literal would have to contain itself, making it infinite in length. The idea commonly used to solve this problem is following:

1. On first line start a definition of string  $S$ , later copy-paste to it the string on the second line.
2. On second line put a command that prints the first line, assigning to  $S$  the string in  $S$  itself, and then prints  $S$  (the second line itself).

Yet a stronger quine is so called **radiation hardened quine**, a quine that remains quine even after any one character from the program has been deleted (found here in Ruby: <https://github.com/mame/radiation-hardened-quine>). Other plays on the theme of quine include e.g. a program that produces a bigger program which will again produce yet bigger program etc.

Another extension of a quine is **multiquine** -- this is NOT a polyglot quine! Multiquine is a quine written in some programming language  $L_0$ ; under normal circumstances this program behaves like a normal quine, but it has an extra feature: when passed a parameter  $N$  (e.g. through CLI flag or through standard input), it will print a program in another language,  $L_N$ , which itself is this multiquine (so it can again be used to get back the program in  $L_0$  and so on). I.e. a multiquine is a quine which can switch between several languages.

In the Text esoteric programming language every program is a quine (and so also a radiation hardened one).

## List Of Quines

**Brainfuck**: not short, has over 2100 characters.

**C**:

```
#include <stdio.h>
char s[] = "#include <stdio.h>%cchar s[] = %c%s%c;%cint main(void) { printf(s,10,34,s,34,10,10); return 0; }
int main(void) { printf(s,10,34,s,34,10,10); return 0; }
```

**comun**:

```
0 46 32 34 S 34 32 58 83 S --> S: "0 46 32 34 S 34 32 58 83 S --> " .
```

**Python**:

```
s="print(str().join([chr(115),chr(61),chr(34)])) + s + str().join([chr(34),chr(10)])) + s)"
print(str().join([chr(115),chr(61),chr(34)])) + s + str().join([chr(34),chr(10)])) + s)
```

text:

This is a quine in text.

TODO: more, make biquine of C and comun

## See Also

- [polyglot](#)

---

race

## Race

*All races of men, however different, are to coexist in love and peace.*

Races of people are very large, fuzzy, loosely defined groups (clusters) of people who are genetically similar because they come from the same ancestors. Races usually significantly differ by their look and in physical, mental and cultural aspects; some races are physically more fit, some are more intelligent, some are better evolved for living in specific climate conditions and so on -- races make mankind very diverse; sadly they are also often a basis of identity fascism too (similarly to how e.g. gender is the basis of LGBT fascism). The topic of human race is nowadays forbidden to be critically discussed and researched (and even to be joked about) due to the political reign of pseudoleft which denies existence of human races and aggressively censors and attacks any disagreement, however there exists a number of undeniable old research, information hidden in the underground and book that haven't yet been burned, and many things about human races are simply completely obvious to those don't let themselves be blinded by the immensely powerful propaganda. Good society, unlike for example our current competitive capitalist society, acknowledges the differences between human races and lets them coexist peacefully, without competition and in social equality despite their differences and without any need for bullshit such as political correctness and biology denialism.

Instead of the word *race* the politically correct camp uses words such as *ethnicity* -- it's funny, sometimes they say no such thing as race exists but other times they simply have to operate with the fact that people are genetically diverse, e.g. when they accuse others of racism or point out statistics that benefit them ("black people are paid less!"), as existence of discrimination based on differences between people necessarily implies the existence of differences between people -- so here they try to substitute the word *race* for a different word so as to make their self-contradiction less obvious. Anyway, it doesn't work :) Races indeed do exist, no matter what we call them.

**Race can be told from the shape of the skull and one's DNA**, which finds use e.g. in forensics to help solve crimes. It is officially called the *ancestry estimation*. Some idiots say this should be forbidden to do because it's "racist" lmao. Besides the obvious visual difference such as skin color **races also have completely measurable differences acknowledged even by modern "science"**, for example unlike other races about 90% of Asians have dry earwax, Asians also have highest bone density, Huaorani tribe has flat feet, blood type distributions are wildly different between races as well as blood pressure and also heart rate, people near the equator have measurably smaller eyeballs than those very far north, even distribution of genes associated with specific behavior was measured to differ between races. Similar absolutely measurable differences exist in height, body odor, alcohol and lactose tolerance, high altitude tolerance, vulnerability to specific diseases, hair structure, cold tolerance, risk of obesity, behavior (see e.g. the infamous chimp out behavior of black people) and others. It is known for a fact that Sherpas are greatly accustomed to living in high altitudes, that's why they work as helpers for people climbing mt. Everest, they can just do it much easier than other races. While dryness of earwax is really a minor curiosity, it is completely unreasonable to believe that race differences stop at traits we humans find "controversial" and that genetics somehow magically avoids affecting traits that are harder to measure and which our current society deems politically incorrect to exist. In fact differences in important areas such as intelligence were measured very well -- these are however either censored or declared incorrect and "debunked" by unquestionable "science" authorities, because politics.

{ Curiosity: in the past there was a research of the specific smell of Jews -- whether Jews do have a specific smell distinguishable by humans may be highly debatable, but it's funny -- one guy tried to start eating like a

Jew to see if he would also start to smell like one :D Source book: *Race Differences* from 1935. ~drummyfish  
}

Pseudoleft uses cheap, logically faulty arguments to deny the existence of race; for example that there are no clear objective boundaries between races -- of course there are not, but how does that imply nonexistence of race? The same argument could also be given even e.g. for the term *species* (see e.g. ring species in which the boundaries are not clear) so as to invalidate it; yet we see no one doubting the existence of various species of animals. That's like saying that color doesn't exist because given any two distinct colors there exists a gradual transition, or that music and noise are the same thing because objectively no clear line can be drawn between them. If by this argument races don't exist, then movie genres, psychological disorders, emotions or political opinions also don't exist.

The politically correct camp further argues that there wasn't enough time for human races to develop significant differences as evolution operates on scales of millions of years while the evolution of modern humans was taking part about in an order of magnitude smaller time scale. However it has been shown that **evolution can be extremely fast and make great changes in mere DECADES**, e.g. in cases of rapid environment change (shown e.g. in a documentary *Laws of the Lizard* on anoles that show signs of evolutionary change only after 14 years, also see e.g. the book *The 10,000 Year Explosion* talking about actual acceleration of human evolution) and interbreeding with other (sub)species (e.g. Denisovan or Neanderthals, which European population bred with but African population didn't), which did occur when humans spread around the world and had to live in vastly different conditions -- successful civilizations themselves actually furthermore started to rapidly change their environment to something that favors very different traits. It has for example been found that average male brain increased from 1372 gram in 1860 to 1424 grams in 1940, a very significant change in LESS THAN A CENTURY. We can take a look at the enormous differences between dog breeds which have been bred mostly during only the last 200 years and whose differences are enormous and not only physical, but also that of intelligence and temperament -- yes, the breeding of dogs has been selective, but a rapid change in environment may have a similar accelerating effect, and the process in humans still took many tens of thousands of years. For example races of slaves were probably selectively bred, even if unintentionally, as physically fit slaves were more likely to survive than those who were smart; similarly in prospering civilizations, e.g. that of Europe, where trade, business and development of technology (e.g. military) became more crucial for survival than in primitive desert or jungle civilizations, different traits such as intelligence became preferred by evolution.

Another pseudoleftist argument is that "the DNA of any two individuals is 99.6 % identical so the differences are really insignificant". Now consider that DNA of a pig is 98 % identical to human. We see the argument is like saying a strawberry and beer is practically the same thing as they are both about 93 % water. It is known that only a minuscule part of DNA has any actual biological effect, only a small part is important and therefore including all the unimportant junk in judging similarity is just purposeful attempt at misleading statistics.

Denying the facts regarding human race is called **race denialism**, the acceptance of these facts is called **race realism**. Race denialism is part of the basis of today's pseudoleftist political ideology, theories such as polygenism (multiregional hypothesis) are forbidden to be supported and they're ridiculed and demonized by mainstream information sources like Wikipedia who only promote the politically correct "out of Africa" theory. SIWs reject any idea of a race with the same religious fanaticism with which Christian fanatics opposed Darwin's evolution theory.

**What races are there?** That depends on definitions^([according to who?][according to logic]), the boundaries between races are fuzzy and the lines can be drawn differently. The traditional, most general division still found in the greatest 1990s encyclopedias is to three large groups: **Caucasoid** (white), **Negroid** (black) and **Mongoloid** (yellow). These can be further subdivided. Some go as far as calling different nations separate races (e.g. the Norwegian race, Russian race etc.), thought that may be a bit of a stretch. One of the first scientific divisions of people into races was done by Francois Bernier in *New Division of the Earth by the Different Species or "Races" of Man that Inhabit It* into Europeans, Asians, Africans and Sami (north Europe), based on skin color, hair color, height and shape of face, nose and eyes.

There is a controversial 1994 book called *The Bell Curve* that deals with differences in intelligence between races (later followed by other books such as *The Global Bell Curve* trying to examine the situation world-wide). SIWs indeed tried to attack it, however international experts on intelligence agree the book is correct in saying average intelligence between races differs (see e.g. The Wall Street Journal's Mainstream

Science on Intelligence). Online resources with a lot of information on racial differences are e.g. <https://zerocontradictions.net/FAQs/race-FAQs> and <http://www.humanbiologicaldiversity.com/>, [https://en.metapedia.org/wiki/Race\\_and\\_morphology](https://en.metapedia.org/wiki/Race_and_morphology) etc. Note that even if some particular resource may be fascist, biased and contain propaganda of its own, it may likely give you information the pseudoleftist mainstream such as Wikipedia and Google simply ensor -- while we may of course not approve of the politics/opinions/goals/etc. of some we link to, we still link to them to provide access to censored information so that one can seek truth and form his own opinions.

**If you want a relatively objective view on races, read old (pre 1950) books.** See for example the article on *NEGRO* in 11th edition of Encyclopedia Britannica (1911), which clearly states on page 344 of the 19th volume that "mentally the negro is inferior to the white" and continues to cite thorough study of this, finding that black children were quite intelligent but with adulthood the intellect always went down, however it states that negro has e.g. better sense of vision and hearing. Even in the 90s still the uncensored information on race was still available in the mainstream sources, e.g. the 1995 *Desk Reference Encyclopedia* and 1993 *Columbia Encyclopedia* still have articles on races and their differences. Other books on races (which you can find e.g. on the Internet Archive) include e.g. *Race Differences* (1935, Klineberg) and *Races of Man* (a huge book from 1900, by Deniker).

{ Another curiosity: many old books argue that the black race is in many aspects NOT the one closest to apes and they make many good points, for example body hair (blacks have none, whites have a lot), hair structure (again white's hair structure is closer to apes) or lips (apes don't have big lips while black races have the bigger lips). ~drummyfish }

{ Lol, the 1917 book *The Circle of Knowledge* has a detailed table comparing various races physically and mentally, stating things like "negro: slight mental development after puberty" etc. Encyclopedia Americana (1918) also mentions a detailed description of the negro, mentioning things such as much lower brain weight, prolonged arms, distinct odor and a lower face angle. ~drummyfish }

It is useful to know the **differences in intellect** between different races (no matter whether the cause is genetic, cultural or other), though cultural and other traits linked to races may also play a big role. Of course, it is important to keep in mind intelligence isn't one dimensional, it's one of the most complex and complicated concepts we can be dealing with (remember the famous test that revealed that chimpanzees greatly outperform humans at certain intellectual tasks such as remembering the order of numbers seen for a very short period of time) and that other traits than raw intelligence may be equally or more important for good performance in intellectual tasks, e.g. personality traits such as curiosity (imagine a fast CPU running shit software versus slower CPU running good software). We can't generally simplify to a single measure such as IQ score (though it can still give some rough ideas, IQ is not absolutely useless), but we can measure performance at different tasks. Let intelligence here mean simply the ability to perform well in the area of given art. And of course, there are smart and stupid people in any race, the general statements we make are just about statistics and probabilities.

The smartest races seem to be Jews and Asians (also found so by the book *Bell Curve* and many old books). Asians have always been regarded as having superior intelligence and their religions and culture also seem to be the most advanced, with very complex ideas (as opposed to e.g. Christianity based on trivial rules to blindly follow), closest to nonviolence, socialism and true science (e.g. Buddhism). There is no question about the intelligence of Jews, the greatest thinkers of all times were Jewish (Richard Stallman, Einstein, Marx, Chomsky, even Jesus and others) -- the man often regarded as the smartest human in history, William James Sidis, was a Jew. Jews have dominated despite being a minority, they seem to have a very creative intelligence and some of them decide to gain further edge by giving up their morality (i.e. becoming capitalist), while Asians are more mechanically inclined -- they can learn a skill and bring it to perfection with an extremely deep study and dedication. Closely following is the general white race (which according to studies is also seen as most physically attractive by all races): white people have of course absolutely dominated history and there is always that one white guy at the top even in areas more dominated by other races (e.g. Eminem in rap, Carlsen in chess, Grubby in Warcraft 3, ...), however whites are still primitive in many ways (individualism, fascism, violence, simple religions and cults, e.g. that of economy, money, simplified commandments of Christianity etc.). The African black race known as the *negro* is one of the least intelligent according to basically all literature -- this makes a lot of sense, the race has been oppressed and living in harsh conditions for centuries and millennia and didn't get much chance to evolve towards good performance in intellectual tasks, quite the opposite, those who were physically fit rather than smart were probably more likely to survive and reproduce as slaves or jungle people (even if white people split from the

blacks relatively recently, a rapid change in environment also leads to a rapid change in evolution, even that of intelligence). However the more primitive, less intelligent races (blacks, indians etc.) were found by some to e.g. have significantly faster reaction times, which sometimes may be an advantage -- this is suspected to be caused by a tradeoff; the "smarter" races perform more complex processing of input information (in terms of computers: having a longer processing pipeline) and so it takes longer, i.e. the more primitive individual acts more impulsively and therefore quicker. The 1892 book *Hereditary Genius* says that the black race is *about two grades* below the white race (nowadays the gap will most likely be lower). Hispanics were found to perform somewhere in between the white and black people. There isn't so much info about other races such as the red race or Eskimos, but they're probably similarly intelligent to the black race. The above mentioned book *Hereditary Genius* gives an intelligence of the Australian aboriginal race *at least one grade below that of the negro*, making possibly the dumbest race of all. The brown races are kind of complicated, Indian people have Asian genes and showed a great intellectual potential, e.g. in chess, math, philosophy (nonviolence inherently connected to India is the most intellectually advanced philosophy), and lately also computer science (even though many would argue that "pajeets" are just trained coding monkeys, really their compsci "universities" are mostly a meme); they may be at the similar level to Hispanics.

Increasing multiculturalism, globalization and mixing of the races will likely make all of this less and less relevant as time goes on -- races will blend greatly which may either help get rid of true racism, but also fuel it: many will oppose racial mixing, many will become more paranoid (as is already the case with Jews who are sometimes very hard to tell apart from whites) and eventually pure races will actually become a minority that may become target of reversed racism: a pale white guy in a room full of mixed people will stand out and likely get lynched (if not just for the fact of being different, then for social revenge). For now the differences in races are still greatly visible.

LRS philosophy is of course FOR multiculturalism and mixing of races -- we just hope the situation won't escalate as described above. Biodiversity is good.

## See Also

- stereotype

---

racetrack

## Racetrack

Racetrack is an awesome minimalist pen and paper mathematical game in which one races a car through track with the goal to finish it as quickly as possible. For PC gamers we could describe it as "an extremely suckless version of Trackmania" for which you don't even need a computer. It is similar to other pen and paper games such as paper football. The basic idea is that of a car on a square grid that moves in steps -- in each step the player can adjust the car's current velocity a little bit (steer, accelerate, brake, ...) and so modify the velocity; the car must race to finish without crashing into walls, the tricky part is that one has to make predictions just like in real race, for example approaching a curve one must go to the right side of the road and brake a bit.

Racetrack is one of the best examples of what good games should look like, mainly because:

- It is extremely suckless, it may be implemented and played with the use of a computer but can also be played without it, i.e. it has practically no dependencies. In theory it can only be played in one's brain, making it brain software.
- It is extremely free (as in freedom): firstly no one legally owns it and secondly its simplicity makes it free practically, anyone can play it and modify it regardless of where he lives, how much money he has, whether he has a computer -- even if one has no eyes or hands the game can still probably be played.
- It may easily be played by any number of players, even solo. If one plays alone, he simply tries to find the fastest solution for given track. If multiple players play, they compete who finds the best solution.
- It is simple yet deep, the rules are very simple but to find the optimal solution for given track may get very difficult, especially if the track is somewhat complex and employs e.g. a number of checkpoints that can be taken in any order. This is probably an NP hard problem and finding a good solution may require a lot of experience, intuition, advanced programming techniques such as machine learning

- etc.
- It's not a mere game but a whole playground and "platform", for example it may be used to teach vector mathematics, programming (path finding, heuristic search, evolutionary programming, ...), test machine learning algorithms etcetc.
- It can be very nicely implemented on computers, even on very simple ones such as 8bits, without bloat such as floating point, and is friendly to e.g. implementing replays, artificial intelligence etc.
- The base version is extremely simple but may be extended greatly in various way, for example adding more rules or creating "rich" computer frontends; one may imagine e.g. a 3D frontend for the game with features such as bots, demo recording, different car skins, online multiplayer and leaderboards, track editor etc.
- ...

## Rules

There is no single rule set -- as no one owns the game, rules may be modified and adjusted, which is very good. However there exist core rules that basically make the game what it is -- let us describe those right now.

The game takes place on a 2D square grid (e.g. squared sheet of paper); the car can only ever occupy integer coordinates, i.e. its position cannot be e.g. a fraction of a square (however if e.g. in some computer implementation the grid is dense enough, it may in theory practically give an impression of continuous space). (Some modifications may perhaps try to utilize different kinds of grids of more than two dimensions.)

The car has a velocity vector which is initially [0,0] (i.e. the car is at rest). This vector can also only ever have integer components. The velocity vector is added to the car's position in each game step so that the car moves. For example car with position [3,2] and velocity [1,-1] will move to [4,1].

At each step the player can make a slight modification of the car's velocity, typically the player has to choose a vector from range [-1,-1] to [1,1] that's added to current velocity; in other words the player can modify current velocity by changing each of its two components by -1, 0 or 1. This makes for 9 possible choices at each game step, so the branching factor of the game is 9. This can be represented as racer steering, accelerating and braking. Of course modified version of the game may play around with this, e.g. an oil puddle may make player unable to modify velocity for one round etc.

Any specific track has a start (some versions of the game may just make player always start at [0,0]), finish (which may be a point, line, area etc.) and walls representing obstacles; modified versions of the game may also have other things such as checkpoints, items (nitro, time stop, ...) and other objects (jump ramps, oil puddles, teleports, ...). The player must race to the finish, usually without crashing into walls because a crash into wall means the car stops immediately (in some versions in may just mean the game ends).

Implementation of walls and crashes may somewhat differ: in some versions walls are actually borders of "solid" areas to which the player must never enter, in other versions walls may be just lines the player must not touch or cross. In simple versions of the game walls are really line segments that go between given grid points (this is possible the more KISS variant as walls too are just defined with vectors and collision detection may be quite simple), more complex versions may allow non-integer coordinates for walls, curved walls etc. Walls may also be implemented just as "filled squares", i.e. just saying some grid points are solid and inaccessible. Crash usually means that a player would make such illegal move and so his current velocity is set to [0,0] as a consequence, but an advanced version may also make the player move as close to the crash point as possible to make the behavior closer to reality; however this may be very non-trivial to do while assuring the behavior can't be "abused". Collision detection can be implemented e.g. by checking if two lines intersect (if walls are just lines), or if a point belongs to given area (if walls are edges of areas), using analytic geometry).

The goal is basically always to finish the track in as few steps as possible.

TODO: example, pictures, ...

---

racism

# Racism

The term racism has nowadays two main definitions, due to the onset of newspeak:

- **original definition:** Great hatred and/or hostility towards specific races of people. For example the Nazi genocide of Jews was an act of racism in the sense of the term's original meaning. Nowadays racism in this meaning is targetted especially against white people.
- **newspeak definition:** Disagreement with the mainstream pseudoleftist propaganda regarding the question of human race, or just performing certain prohibited sins connected to it, e.g. saying the word nigger. For example anyone who claims human race has a basis in biology or that there are any statistical differences between races at all is a racist in the modern meaning of the term.

## See Also

- political correctness
- 

ram

# RAM

RAM stands for *random access memory*, a type of computer memory characterized by allowing access to arbitrary addresses (as opposed to SAM -- sequential memories, such as tapes, which only allow sequential access); a bit confusingly (for historical reasons) the term RAM came to be used more as a synonym for so called **main memory**, i.e. the computer's **working memory** (memory used for performing the actual computation, as opposed to e.g. persistent storage or read only memory). It is true that working memory is very often a random access memory, but it doesn't always have to be so and there exist random access memories that don't serve as the main working memory. Similarly confusing is the fact that RAM is often opposed to ROM (read only memory) -- again, it is true that many computers use RAM as main working memory and ROM as the "other" kind of memory used for static data so in practice these two complement each other, but it is entirely possible for random access memory to be read-only (so RAM can also be ROM) and so on. Nevertheless, though it's imprecise, in this articles we WILL conform to the established terminology a lot -- implicitly we will see RAM as meaning a **volatile random access read/write memory serving as a working memory** (volatile meaning it's erased on power off).

RAM is one of the main components of a computer, it closely cooperates with the CPU; in fact CPU without RAM would be basically useless; RAM serves the CPU as a "scratchpad" where it keeps intermediate results to perform more complex calculations. RAM, being a relatively fast memory, is also often used to temporarily load parts of bigger data for faster access, sometimes it may also store the instructions of the program being executed by the CPU. For this RAM is, along with the CPU, one of the two components which can never be missing in a computer. A computer can work without a hard disk, without keyboard, mouse and monitor, but it can never meaningfully work without RAM.

**RAM is relatively fast**, in memory hierarchy only the CPU registers and CPU cache are faster than RAM, RAM is a lot faster than disk. How much faster exactly depends on a few things, firstly the exact types of both memories, and secondly on how you access the memories, e.g. with sequential access RAM may be only 10 times faster, but with random access it can even be 100000 times faster. The speed of RAM is often (in PCs always, but may be missing e.g. in embedded) boosted by the mentioned cache memory (standing between RAM and CPU), but again that will only work if we access the RAM correctly (respecting the principle of locality, i.e. not make big jumps in memory).

There are two main types of electronic RAM:

- **SRAM** (static RAM): After assigning a value (0 or 1) to a memory cell, the cell retains the value as long as the power is on. This is usually implemented with flip flop logic circuits. SRAMs are very fast but expensive (a flip flop requires several transistors) and also consume more power than DRAMs (when not idle), so in PCs they are actually NOT often used to implement the main memory (which this article is about), instead SRAMs are used for the smaller memories like CPU registers and caches. But simpler computers with low RAM (e.g. embedded) may use SRAM even for main memory.

- **DRAM** (dynamic RAM): After assigning a value to a memory cell, the cell will hold the value only for some time, therefore the cells have to be periodically refreshed (usually at least once in 64 milliseconds) so that they retain their values for long time (hence the name *dynamic*). This behavior exists because DRAMs are usually implemented with capacitors which lose charge over time. DRAMs are cheaper (a cell just requires a transistor and capacitor) but slower, so these are often used to implement the main memory.

Furthermore there are many other types like SDRAM, DDR, DDR2 etc.

**RAM from programmer's point of view:** in your programming language variables are typically places in RAM (the variable name is just a name for some RAM memory address), so the more variables you need (note that most significant are arrays and other "big" variables), the more RAM your program will consume. Though it may not be so simple, some variables whose value doesn't change (e.g. static const or string literals in C) may be rather placed in ROM by the compiler/optimizer. Also some small scope variables may be just stored in CPU registers. WATCH OUT: under a typical operating system the main memory is virtualized so the addresses your program sees are generally not the physical addresses in RAM.

Also thanks to virtual memory **your computer may actually be able to use more RAM than there is physically present** by temporarily storing some less used memory pages on to the disk to free space in RAM. This is called swapping and normally results in huge slowdown of the computer; swapping is many times a sign of memory leak or some other atrocity.

Saving content of RAM to disk is also exploited by hibernation.

**How much RAM do we need?** Not much, definitely not NEARLY as much as you see on a typical today's consumer PC which come with 16 or 32 GB of RAM, that's just too much, you never need that much memory and this craziness only exists for consumerism and due to extremely shitty capitalist software whose efficiency probably doesn't surpass 1%. The amount of RAM we need firstly depends on the task at hand and secondly on the details of our computer (e.g. if it stores the program itself in RAM or not, if we have helper coprocessors that save us some work, if we have a fast CPU and can afford to sacrifice some of its speed for needing less memory etc.) and what exactly we define as RAM (whether e.g. we see video memory as RAM or if we are allowed to store a lot of read-only data in ROM). Generally speaking for simple mathematical problems, such as solving a quadratic equation, a few bytes may be enough. With a few hundred bytes we can make simple games such as Tetris. With a few kilobytes we can already make more complex games, e.g. something akin Wolf 3D or chess with basic AI, we can make a simple text editor, probably even a programming language capable of compiling itself (see e.g. games for Arduboy which possesses 2.5 KB of RAM). Surpassing some 30 KB we can already make Doom-like games (Anarch runs on GB Meta with 32 KB of RAM) and basic versions of most of the tools we need on a personal computer such as text editor, image editor, music composer, programming editor, ... though still typically running on bare metal (without operating system). 1 MB is about 30 times that, so unless dealing with some memory-heavy task, such as processing HD video, **with good programming you should practically never need more than 1 MB of RAM**. If your computer has 1 GB of RAM, it already has 1000 times the overkill amount, so it can do all kind of fancy stuff like running an operating system that runs several programs at once (multitasking), some of which may be doing even memory heavy tasks.

## See Also

- SAM
- ROM
- VRAM
- flash
- EEPROM
- hard disk
- memory

---

randomness



# Randomness

Not to be confused with pseudorandomness.

Randomness means unpredictability, lack of patterns, and/or behavior without cause. Random events can only be predicted imperfectly using probability because there is something present that's subject to chance, something we don't know; events may be random to us either because they are inherently random (i.e. they really have no cause, pattern etc.) or because we just lack knowledge or practical ability to perfectly predict the events. Randomness is one of the most basic, yet also one of the most difficult concepts to understand about our Universe -- it's a phenomenon of uttermost practical importance, we encounter it every second of our daily lives, but it's also of no lesser interest to science, philosophy, art and religion. Whole libraries could be filled just with books about this topic, here we will be able to only scratch the surface of it by taking a look at the very basics of randomness, mostly as related to programming and math.

As with similarly wide spanning terms the word *randomness* and *random* may be defined in different ways and change meaning slightly depending on context, for example sometimes we have to distinguish between "true" randomness, such as that we encounter in quantum mechanics or that present in nondeterministic mathematical models, and pseudorandomness (what as a programmer you'll be probably dealing with), i.e. imitating this true randomness with deterministic ("non-randomly behaving") systems, e.g. sequences of numbers that are difficult to compress. Other times we call random anything at all that just deviates from usual order, as in "someone started randomly spamming me in chat". Let's briefly review a few terms related to this topic:

- **randomness:** The wide term meaning great unpredictability, which may be inherent or just apparent. We usually divide it to:
  - ♦ **true randomness:** Randomness that is caused by inherently unpredictable behavior of a system, i.e. behavior that truly has no cause and is decided purely by chance, without ever being able to be perfectly predicted, even just theoretically; this is contrasted with pseudorandomness. A typical example given is quantum physics in which true randomness seems to be present in things such as some properties of elementary particles of the Universe -- though in fact this can never be proven with certainty, there is so much evidence of us not being able to predict quantum phenomena that we just mostly take it for the closest thing to true randomness in real world. However we can also see some purely mathematical models to have true randomness, simply because they define it so, e.g. a nondeterministic Turing machine is simply defined to sometimes make purely random decisions.
  - ♦ **pseudorandomness:** Randomness that's at its basic level generated by a completely deterministic system, i.e. something (e.g. a sequence of numbers) that practically looks like something that would be generated by truly random system, which however stems from something completely non-random (e.g. a computer program). This is contrasted with pure randomness. Chaotic systems are mostly used to implement pseudorandomness. Pseudorandomness is used to imitate true randomness e.g. in computers, because it is mostly good enough and true randomness is difficult to achieve.
- **nondeterminism:** Attribute of a system, such as mathematical model or physics theory, of involving true randomness.
- **chaos:** Behavior that is deterministic (i.e. without true randomness) which however due to its mathematical properties is practically impossible to be predicted as there is no "nice" equation for it, resulting in practically having the same implications as true randomness. Chaotic behavior is predictable in theory but not in practice as it basically just requires "brute force" simulation, and so we often treat chaotic systems the same as completely random ones, with statistics and probability.
- **probability:** Mathematical theory examining randomness, it formally models systems that include randomness and reasons about them, it gives us equations, for example it says how we infer the exact probability of something happening knowing probabilities of some individual events etc. It is a theoretical area and stresses deductive reasoning, i.e. it starts by defining a system and reasons about what such system will do.
- **statistics:** Applying probability theory to examining data -- like probability it is a mathematical discipline, however it is applied (rather than purely theoretical) and stresses inductive reasoning, i.e. it works "in the other direction" than probability theory; statistics starts with having some data and then tries to find a probabilistic model that would likely produce such data, potentially revealing what system really lies underneath the data.

- **stochasticity**: Basically mathematics that deals with randomness and probability in some way, the term is often used as an attribute of a mathematical model, i.e. stochastic model is that which is somehow described in terms of probabilities.
- **entropy**: A measure related to randomness, saying how much information (in bits) we can extract from given message -- the higher the randomness (unpredictability), the higher the entropy because this randomness may be used to carry information.

Keep in mind **there are different "amounts" of randomness** -- that is to say you should consider that **probability distributions** exist and that some processes may be random only a little. It is not like there are only completely predictable and completely unpredictable systems, oftentimes we just have some small elements of chance or can at least estimate which outcomes are more likely. We see absolute randomness (i.e. complete unpredictability) only with uniform probability distribution, i.e. in variables in which all outcomes are equally likely -- for example rolling a dice. However in real life variables some values are usually more likely than others -- e.g. with adult human male height values such as 175 cm will be much more common than 200 cm; great many real life values actually have normal distribution -- the one in which values around some center value are most common.

**What do random numbers look like?** This is a tricky question. Let's now consider uniform probability distribution, i.e. "absolute randomness". When we see sequences of numbers such as [1, 2, 3, 4, 5, 6, 7], [0, 0, 0, 0, 0, 0, 0] or [9, 1, 4, 7, 8, 1, 5], which are "random" and which not? Intuitively we would say the first two are not random because there is a clear pattern, while the third one looks pretty random. However consider that under our assumption of uniform probability distribution all of these sequences are equally likely to occur! It is just that there are only very few sequences in which we recognize a common pattern compared to those that look to have no pattern, so we much more commonly see these sequences without a pattern coming out of random number generators and therefore we think the first two patterns are very unlikely to have come from a random source. Indeed they are, but the third, "random looking" sequence is equally unlikely (if you bet the numbers in lottery, you are still very unlikely to win), it just has great many weird looking siblings. You have to be careful, things around probability are great many times very unintuitive and tricky (see e.g. the famous Monty Hall problem).

Of course we cannot say just from the sequence alone if it was generated randomly or not, the sequences above may have been generated by true randomness or by pseudorandom generator -- we even see this is sort of stupid to ask. We should rather think about what we actually mean by asking whether the sequence is "random" -- to get meaningful answers we have to specify this first. If we formulate the question precisely, we may get precise answers. Sometimes we are looking for lack of patterns -- this can be tested by programs that look for patterns, e.g. compression programs; number sequences that have regularities in them can be compressed well. We may examine the sequences entropy to say something about its "randomness". Mathematicians often like to ask "how likely is it that a sequence with these properties was generated by this model?", i.e. for example listening to signals from space and capturing some numeric sequence, we may compute its properties such as distribution of values in it and then we ask how likely is it that such sequence was generated by some natural source such exploding star or black hole? If we conclude this is very unlikely, we may say the signal was probably not generated randomly and may e.g. come from intelligent lifeforms.

TODO: moar

## Randomness Tests

TODO

One of the most basic is the **chi-squared test** whose description can be found e.g. in the *Art of Computer Programming* book. TODO

```
{ The following is a method I came up with wrote about here (includes some code):
https://codeberg.org/drummyfish/my_writings/src/branch/master/randomness.md, I haven't found what this is
called, it probably already exists. If you know what this method is called, please send me a mail.
~drummyfish }
```

**Cool randomness test:** this test attempts to measure the unpredictability, the inability to predict what binary digit will follow. As an input to the test we suppose a binary sequence  $S$  of length  $N$  bits that's repeating forever (for example for  $N = 2$  a possible sequence is 10 meaning we are really considering an

infinite sequence 1010101010...). We suppose an observer knows the sequence and that it's repeating (consider he has for example been watching us broadcast it for a long time and he noticed we are just repeating the same sequence over and over), then we ask: if the observer is given a random (and randomly long) subsequence  $S_2$  of the main sequence  $S$ , what's the average probability he can correctly predict the bit that will follow? This average probability is our measured randomness  $r$  -- the lower the  $r$ , the "more random" the sequence  $S$  is according to this test. For different  $N$  there are different minimum possible values of  $r$ , it is for example not possible to achieve  $r < 0.7$  for  $N = 3$  etc. The following table shows this test's most random sequences for given  $N$ , along with their count and  $r$ .

| seq. len. | most random looking sequences                                                                                | count | min. r |
|-----------|--------------------------------------------------------------------------------------------------------------|-------|--------|
| 1         | 0, 1                                                                                                         | 2     | 1.00   |
| 2         | 01, 10                                                                                                       | 2     | 0.50   |
| 3         | 001, 010, 011, 100, 101, 110                                                                                 | 6     | ~0.72  |
| 4         | 0011, 0110, 1001, 1100                                                                                       | 4     | ~0.78  |
| 5         | 00101, 01001, 01010, 01011, 01101, 10010, 10100, 10101, 10110, 11010                                         | 10    | ~0.82  |
| 6         | 000101, 001010, 010001, 010100, 010111, 011101, 100010, 101000, 101011, 101110, 110101, 111010               | 12    | ~0.86  |
| 7         | 0001001, 0010001, 0010010, 0100010, 0100100, 0110111, 0111011, 1000100, 1001000, 1011011, ...                | 14    | ~0.88  |
| 8         | 00100101, 00101001, 01001001, 01001010, 01010010, 01011011, 01101011, 01101101, 10010010, ...                | 16    | ~0.89  |
| 9         | 000010001, 000100001, 000100010, 001000010, 001000100, 010000100, 010001000, 011101111, ...                  | 18    | ~0.90  |
| 10        | 0010010101, 0010101001, 0100100101, 0100101010, 0101001001, 0101010010, 0101011011, ...                      | 20    | ~0.91  |
| 11        | 00010001001, 00010010001, 00100010001, 00100010010, 00100100010, 01000100010, 01000100100, ...               | 22    | ~0.92  |
| 12        | 001010010101, 001010100101, 010010100101, 010010101001, 010100101001, 010100101010, ...                      | 24    | ~0.92  |
| 13        | 0010010100101, 0010100100101, 0010100101001, 0100100101001, 0100101001001, 0100101001001, 0100101001010, ... | 26    | ~0.93  |
| ...       | ...                                                                                                          | ...   | ...    |

## Truly Random Sequence Example

WORK IN PROGRESS { Also I'm not too good at statistics lol. ~drummyfish }

Here is a sequence of 1000 bits which we most definitely could consider truly random as it was generated by physical coin tosses:

{ The method I used to generate this: I took a plastic bowl and 10 coins, then for each round I threw the coins into the bowl, shook them (without looking, just in case), then rapidly turned it around and smashed it against the ground. I took the bowl up and wrote the ten generated bits by reading the coins kind of from "top left to bottom right" (heads being 1, tails 0). ~drummyfish }

```
0000111001110100000010000101110111101010011100011
0100110111010001001100010110100100001011111101110
10110110100010011011010001000111011010100100010011
1111100011101111011100001000000001101001101010000
1111111001000111100100011010110001011000001001000
1000101011110100111110010010101001101010000101101
10110000001101001010111100100100000110000000011000
1100000100111100001101110111110101101111011110111
1101000110010010011000111100011111001101111010010
10001001001010111000010101000100000111010110011000
00001010011100000110011010110101011100101110110010
0101001010111101000000110100011011101100100101001
00101101100100100101101100111101001101001110111100
```

```

11001001100110001110000000110000010101000101000100
0011011100010000110011100011110001101011100011011
1110111100010111000111001010110011001000011101000
010011110010100110001110000111110001111101110101
01000101101100010000010110110000001101001100100110
1110100001010110111100111011011010100110011110000
101111000101000001011110011110110110111000010101

```

Let's now take a look at how random the sequence looks, i.e. basically how likely it is that by generating random numbers by tossing a coin will give us a sequence with statistical properties (such as the ratio of 1s and 0s) that our obtained sequence has.

There are **494 1s and 506 0s**, i.e. the ratio is approximately 0.976, deviating from 1.0 (the value that infinitely many coin tosses should converge to) by only 0.024. We can use the binomial distribution to calculate the "rarity" of getting this deviation or higher one; here we get about 0.728, i.e. a pretty high probability, meaning that if we perform 1000 coin tosses like the one we did, we may expect to get the deviation we got or higher in more than 70% of cases (if on the other hand we only got e.g. 460 1s, this probability would be only 0.005, suggesting the coins we used weren't fair). If we take a look at how the ratio (rounded to two fractional digits) evolves after each round of performing additional 10 coin tosses, we see it gets pretty close to 1 after only about 60 tosses and stabilizes quite nicely after about 100 tosses: 0.67, 0.54, 0.67, 0.90, 0.92, 1.00, 0.94, 0.90, 0.88, 1.00, 1.04, 1.03, 0.97, 1.00, 0.97, 1.03, 1.10, 1.02, 0.98, 0.96, 1.02, 1.02, 1.02, 1.00, 0.95, 0.95, 0.99, 0.99, 0.99, 0.97, 0.95, 0.95, 0.96, 0.93, 0.90, 0.88, 0.90, 0.93, 0.95, 0.98, 0.98, 0.97, 0.97, 0.99, 1.00, 0.98, 0.98, 0.98, 0.97, 0.96, 0.95, 0.94, 0.95, 0.95, 0.96, 0.95, 0.96, 0.95, 0.96, 0.95, 0.96, 0.95, 0.96, 0.96, 0.96, 0.97, 0.97, 0.97, 0.95, 0.94, 0.93, 0.93, 0.93, 0.94, 0.94, 0.94, 0.96, 0.95, 0.96, 0.96, 0.95, 0.96, 0.95, 0.95, 0.96, 0.97, 0.97, 0.96, 0.96, 0.95, 0.95, 0.95, 0.96, 0.97, 0.97, 0.97, 0.97, 0.96, 0.97, 0.98, 0.98.

Let's try the chi-squared test (the kind of basic "randomness" test):  $D = (494 - 500)^2 / 500 + (506 - 500)^2 / 500 = 0.144$ ; now in the table for the chi square distribution for 1 degree of freedom (i.e. two categories, 0 and 1, minus one) we see this value of  $D$  falls somewhere around 30%, which is not super low but not very high either, so we can see the test doesn't invalidate the hypothesis that we got numbers from a uniform random number generator. { I did this according to Knuth's *Art of Computer Programming* where he performed a test with dice and arrived at a number between 25% and 50% which he interpreted in the same way. For a scientific paper such confidence would of course be unacceptable because there we try to "prove" the validity of our hypothesis. Here we put much lower confidence level as we're only trying not fail the test. To get a better confidence we'd probably have to perform many more than 1000 tosses. ~drummyfish }

We can try to convert this to a sequence of integers of different binary sizes and just "intuitively" see if the sequences still looks random, i.e. if there are no patterns such as e.g. the numbers only being odd or the histograms of the sequences being too unbalanced, we could also possibly repeat the chi-squared test etc.

The sequence as 100 10 bit integers (numbers from 0 to 1023) is:

```

57 832 535 501 227 311 275 90 267 1006
730 155 273 874 275 995 759 528 52 848
1020 572 565 556 72 555 935 805 309 45
704 842 969 24 24 772 963 479 695 759
838 294 241 998 978 548 696 337 29 408
41 774 429 370 946 330 1000 104 886 297
182 293 719 308 956 806 398 12 84 324
220 268 911 107 795 958 184 917 612 232
318 332 451 911 885 278 784 364 52 806
929 367 630 851 240 753 261 926 859 533

```

As 200 5 bit integers (numbers from 0 to 31):

```

1 25 26 0 16 23 15 21 7 3 9 23 8 19 2 26 8 11 31 14
22 26 4 27 8 17 27 10 8 19 31 3 23 23 16 16 1 20 26 16
31 28 17 28 17 21 17 12 2 8 17 11 29 7 25 5 9 21 1 13
22 0 26 10 30 9 0 24 0 24 24 4 30 3 14 31 21 23 23 23
26 6 9 6 7 17 31 6 30 18 17 4 21 24 10 17 0 29 12 24
1 9 24 6 13 13 11 18 29 18 10 10 31 8 3 8 27 22 9 9
5 22 9 5 22 15 9 20 29 28 25 6 12 14 0 12 2 20 10 4
6 28 8 12 28 15 3 11 24 27 29 30 5 24 28 21 19 4 7 8

```





unlike in raytracing, there are no shadows, reflections and refractions. Raytracing is the extension of raycasting.

- **2D raycasting:** Technique for rendering so called "pseudo3D" (primitive 3D) graphics, probably best known from the old game Wolf3D (predecessor of Doom). The principle of casting the rays is the same but we only limit ourselves to casting the rays within a single 2 dimensional plane and render the environment by columns (unlike the 3D variant that casts rays and renders by individual pixels).

## 2D Raycasting

{ We have an official LRS library for advanced 2D raycasting: raycastlib! And also a game built on top of it: Anarch. ~drummyfish }

{ Also there is a very cool oldschool book that goes through programming a whole raycasting game engine in C, called *Tricks of The Game Programming Gurus*, check it out! ~drummyfish }

2D raycasting can be used to relatively easily render "3Dish" looking environments (commonly labeled "pseudo 3D"), mostly some kind of right-angled labyrinth. There are limitations such as the inability for the camera to tilt up and down (which can nevertheless be faked with shearing). It used to be popular in very old games but can still be used nowadays for "retro" looking games, games for very weak hardware (e.g. embedded), in demos etc. It is pretty cool, very suckless rendering method.

[illegible]

*raycasted view, rendered by the example below*

The method is called **2D** because even though the rendered picture looks like a 3D view, the rays we are casting are 2 dimensional and the representation of the world we are rendering is also usually 2 dimensional (typically a grid, a top-down plan of the environment with cells of either empty space or walls) and the casting of the rays is performed in this 2D space -- unlike with the 3D raycasting which really does cast 3D rays and uses "fully 3D" environment representations. Also unlike with the 3D version which casts one ray per each rendered pixel ( $x * y$  rays per frame), 2D raycasting only casts **one ray per rendered column** ( $x$  rays per frame) which actually, compared to the 3D version, drastically reduces the number of rays cast and makes this method **fast enough for real time** rendering even using software rendering (without a GPU).

SIDENOTE: The distinction between 2D and 3D raycasting may get fuzzy, the transition may be gradual. It is possible to have "real 3D" world (with some limitations) but draw it using 2D raycasting, Anarch does something like that -- it uses 2D raycasting for rendering but player and projectiles have full X, Y and Z coordinates. Also consider for example performing 2D raycasting but having 3 layers of the 2D world, allowing for 3 different height levels; now we've added the extra Z dimension to 2D raycasting, though this dimension is small (Z coordinate of world cell can only be 0, 1 or 2), however we will now be casting 3 rays for each column and are getting closer to the full 3D raycasting. This is just to show that as with everything we can usually do "something in between".

Back to pure 2D raycasting; the principle is following: for each column we want to render we cast a ray from the camera and find out which wall in our 2D world it hits first and at what distance -- according to the distance we use perspective to calculate how tall the wall columns should look from the camera's point of view, and we render the column. Tracing the ray through the 2D grid representing the environment can be done relatively efficiently with algorithms normally used for line rasterization. There is another advantage for weak-hardware computers: we can easily use 2D raycasting **without a framebuffer** (without double buffering) because we can render each frame top-to-bottom left-to-right without overwriting any pixels (as we simply cast the rays from left to right and then draw each column top-to-bottom). And of course, it can be implemented using fixed point (integers only).

The classic version of 2D raycasting -- as seen in the early 90s games -- only renders walls with textures; floors and ceilings are untextured and have a solid color. The walls all have the same height, the floor and ceiling also have the same height in the whole environment. In the walls there can be sliding doors. 2D sprites (billboards) can be used with raycasting to add items or characters in the environment -- for correct rendering here we usually need a 1 dimensional z-buffer in which we write distances to walls to correctly draw sprites that are e.g. partially behind a corner. However we can **extend** raycasting to allow levels with different heights of walls, floor and ceiling, we can add floor and ceiling texturing and also use different level geometry than a square grid (however at this point it would be worth considering if e.g. BSP or portal rendering wouldn't be better). An idea that might spawn from this is for example using signed distance field function to define an environment and then use 2D raymarching to iteratively find intersections of the ray with the environment in the same way we are stepping through cells in the 2D raycasting described above.

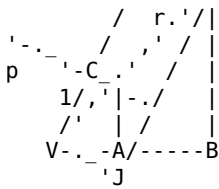
## Implementation

The core element to implement is the code for casting rays, i.e. given the square plan of the environment (e.g. game level), in which each square is either empty or a wall (which can possibly be of different types, to allow e.g. different textures), we want to write a function that for any ray (defined by its start position and direction) returns the information about the first wall it hits. This information most importantly includes the distance of the hit, but can also include additional things such as the type of the wall, texturing coordinate or its direction (so that we can shade differently facing walls with different brightness for better realism). The environment is normally represented as a 2 dimensional array, but instead of explicit data we can also use e.g. a function that procedurally generates infinite levels (i.e. we have a function that for given square coordinates computes what kind of square it is). As for the algorithm for tracing the ray in the grid we may actually use some kind of line rasterization algorithm, e.g. the **DDA algorithm** (tracing a line through a grid is analogous to drawing a line in a pixel grid). This can all be implemented with fixed point, i.e. integer only! No need for floating point.

**Note on distance calculation and distortion:** When computing the distance of ray hit from the camera, we usually DO NOT want to use the Euclidean distance of that point from the camera position (as is tempting) -- that would create a so called fish eye effect, i.e. looking straight into a perpendicular wall would make the wall look warped/bowled (as the part of the wall in the middle of the screen is actually closer to the camera position so it would, by perspective, look bigger). For non-distorted rendering we have to compute a distance that's perpendicular to the camera plane -- we can see the camera plane as a "canvas" onto which we project the scene, in 2D it is a line (unlike in 3D where it really is a plane) at a certain distance from the camera (usually conveniently chosen to be e.g. 1) whose direction is perpendicular to the direction the camera is facing. The good news is that with a little trick this distance can be computed even more efficiently than Euclidean distance, as we don't need to compute a square root! Instead we can utilize the similarity of triangles. Consider the following situation:

$$\frac{I}{\text{distance}} = \frac{X}{\text{height}}$$





In the above  $V$  is the position of the camera (viewer) which is facing towards the point  $I$ ,  $p$  is the camera plane perpendicular to  $VI$  at the distance 1 from  $V$ . Ray  $r$  is cast from the camera and hits the point  $X$ . The length of the line  $r$  is the Euclidean distance, however we want to find out the distance  $JX = VI$ , which is perpendicular to  $p$ . There are two similar triangles:  $VCA$  and  $VIB$ ; from this it follows that  $1 / VA = VI / VB$ , from which we derive that  $JX = VB / VA$ . We can therefore calculate the perpendicular distance just from the ratio of the distances along one principal axis (X or Y). However watch out for the case when  $VA = VB = 0$  to not divide by zero! In such case use the other principal axis (Y).

Here is a complete **C example** that uses only fixed point with the exception of the stdlib sin/cos functions, for simplicity's sake (these can easily be replaced by custom fixed point implementation):

```
#include <stdio.h>
#include <math.h>      // for simplicity we'll use float sin, cos from stdlib

#define U 1024          // fixed-point unit
#define LEVEL_SIZE 16 // level resolution
#define SCREEN_W 100
#define SCREEN_H 31

int wallHeight[SCREEN_W];
int wallDir[SCREEN_W];

int perspective(int distance)
{
    if (distance <= 0)
        distance = 1;

    return (SCREEN_H * U) / distance;
}

unsigned char level[LEVEL_SIZE * LEVEL_SIZE] =
{
#define E 1, // wall
#define l 0, // floor
    l l l l E l l l l l l l l E E
    l E l l E E E l l l l l E l l E
    l l l l l l l l l l l l l l l l
    l E l l E l E l E l E l E l l l
    l l l l E l l l l l l l l l E l
    l l l l E l l l l l l l l l E l
    l E E l E l l l l l l l l l l l
    l E E l E l l l l l l l l l l l
    l E l l l l l l l l l l l l l E
    l E l l E l l l l l l l l E l l
    l E l l E l l l l l l l l E l l
    l E l l l l E E E l l l l l l l
    l E E l E l l l l l E E E l l E
    l E E l E l l l l l E l l l E E
    l l l l l l E E E E l l E E E
    l l E l l l l l l l l l E E E
#undef E
#undef l
};

unsigned char getTile(int x, int y)
{
    if (x < 0 || y < 0 || x >= LEVEL_SIZE || y >= LEVEL_SIZE)
        return 1;

    return level[y * LEVEL_SIZE + x];
}
```

```

// returns perpend. distance to hit and wall direction (0 or 1) in dir
int castRay(int rayX, int rayY, int rayDx, int rayDy, int *dir)
{
    int tileX = rayX / U,
        tileY = rayY / U,
        addX = 1, addY = 1;

    // we'll convert all cases to tracing in +x, +y direction

    *dir = 0;

    if (rayDx == 0)
        rayDx = 1;
    else if (rayDx < 0)
    {
        rayDx *= -1;
        addX = -1;
        rayX = (tileX + 1) * U - rayX % U;
    }

    if (rayDy == 0)
        rayDy = 1;
    else if (rayDy < 0)
    {
        rayDy *= -1;
        addY = -1;
        rayY = (tileY + 1) * U - rayY % U;
    }

    int origX = rayX,
        origY = rayY;

    for (int i = 0; i < 20; ++i) // trace at most 20 squares
    {
        int px = rayX % U, // x pos. within current square
            py = rayY % U,
            tmp;

        if (py > ((rayDy * (px - U)) / rayDx) + U)
        {
            tileY += addY; // step up
            rayY = ((rayY / U) + 1) * U;

            tmp = rayX / U;
            rayX += (rayDx * (U - py)) / rayDy;

            if (rayX / U != tmp) // don't cross the border due to round. error
                rayX = (tmp + 1) * U - 1;

            *dir = 0;
        }
        else
        {
            tileX += addX; // step right
            rayX = ((rayX / U) + 1) * U;

            tmp = rayY / U;
            rayY += (rayDy * (U - px)) / rayDx;

            if (rayY / U != tmp)
                rayY = (tmp + 1) * U - 1;

            *dir = 1;
        }

        if (getTile(tileX, tileY)) // hit?
        {
            px = rayX - origX;
            py = rayY - origY;

            // get the perpend dist. to camera plane:

```

```

        return (px > py) ? ((px * U) / rayDx) : ((py * U) / rayDy);

        // the following would give the fish eye effect instead
        // return sqrt(px * px + py * py);
    }
}

return 100 * U; // no hit found
}

void drawScreen(void)
{
    for (int y = 0; y < SCREEN_H; ++y)
    {
        int lineY = y - SCREEN_H / 2;

        lineY = lineY >= 0 ? lineY : (-1 * lineY);

        for (int x = 0; x < SCREEN_W; ++x)
            putchar((lineY >= wallHeight[x]) ? '.' : (wallDir[x] ? '/' : '#'));

        putchar('\n');
    }
}

int main(void)
{
    int camX = 10 * U + U / 4,
        camY = 9 * U + U / 2,
        camAngle = 600, // U => full angle (2 * pi)
        quit = 0;

    while (!quit)
    {
        int forwX = cos(2 * 3.14 * camAngle) * U,
            forwY = sin(2 * 3.14 * camAngle) * U,
            vecFromX = forwX + forwY, // leftmost ray
            vecFromY = forwY - forwX,
            vecToX = forwX - forwY,   // rightmost ray
            vecToY = forwY + forwX;

        for (int i = 0; i < SCREEN_W; ++i) // process each screen column
        {
            // interpolate rays between vecFrom and vecTo
            int rayDx = (SCREEN_W - 1 - i) * vecFromX / SCREEN_W + (vecToX * i) / SCREEN_W,
                rayDy = (SCREEN_W - 1 - i) * vecFromY / SCREEN_W + (vecToY * i) / SCREEN_W,
                dir,
                dist = castRay(camX, camY, rayDx, rayDy, &dir);

            wallHeight[i] = perspective(dist);
            wallDir[i] = dir;
        }

        for (int i = 0; i < 10; ++i)
            putchar('\n');

        drawScreen();

        char c = getchar();

        switch (c) // movement
        {
            case 'a': camAngle += 30; break;
            case 'd': camAngle -= 30; break;
            case 'w': camX += forwX / 2; camY += forwY / 2; break;
            case 's': camX -= forwX / 2; camY -= forwY / 2; break;
            case 'q': quit = 1; break;
            default: break;
        }
    }

    return 0;
}

```

}

How to make this more advanced? Here are some hints and tips:

- **textured walls:** This is pretty simply, the ray hit basically gives us a horizontal texturing coordinate, and we simply stretch the texture vertically to fit the wall. I.e. when the ray hits a wall, we take the hit coordinate along the principal axis of the wall (e.g. for vertical hit we take the Y coordinate) and mod it by the fixed point unit which will give us the texturing coordinate. This coordinate tells us the column of the texture that the rendered column shall have; we read this texture column and render it stretched vertically to fit the column height given by the perspective. Note that for cache friendliness (optimization) textures should be stored column-wide in memory as during rendering we'll be reading the texture by columns (row-wise stored textures would make us jump wide distances in the memory which CPU caches don't like).
- **textured floor/ceiling:** Something akin mode7 rendering can be used.
- **sliding door:** TODO
- **jumping:** Camera can easily be shifted up and down. If we are to place the camera e.g. one fixed point unit above its original position, then for each column we render we compute, with perspective applied to this one fixed point unit (the same way with which we determine the column size on the screen) the vertical screen-space offset of the wall and render this wall column that many pixel lower.
- **looking up/down:** Correct view of a camera that's slightly tilted up/down can't be achieved (at least not in a reasonably simple way), but there's a simple trick for faking it -- camera shearing. Shearing literally just shifts the rendered view vertically, i.e. if we're to look a bit up, we render that same way as usual but start higher up on the screen (in the part of the rendered image that's normally above the screen and not visible), so that the vertical center of the screen will be shifted downwards. For smaller angles this looks good enough.
- **multilevel floor/ceiling:** This is a bit more difficult but it can be done e.g. like this: for each level square we store its floor and ceiling height. When casting a ray, we will consider any change in ceiling and/or floor height a hit, AND we'll need to return multiple of those hits (not just the first one). When we cast a ray and get a set of such hits, from each hit we'll know there are tiny walls on the floor and/or ceiling, and we'll know their distances. This can be used to correctly render everything.
- **different level geometry:** In theory the level doesn't have to be a square grid but some kind of another representation, or we may keep it a square grid but allow placement of additional shapes in it such as cylinders etc. Here you simply have to figure out how to trace the rays so as to find the first thing it hits.
- **adding billboards (sprites):** TODO
- **reflections:** We can make our 2D raycaster a 2D raytracer, i.e. when we cast a camera ray and it hits a reflective wall (a mirror), we cast another, secondary reflected ray and trace it to see which wall it hits, i.e. which wall will get reflected in the reflective wall.
- **partly transparent walls:** We can make some walls partially transparent, both with alpha blending or textures with transparent pixels. In both cases we'll have to look not just for the first hit of the ray, but also for the next.

---

raycastlib

## Raycastlib

Raycastlib (RCL) is a public domain (CC0) LRS C library for advanced 2D raycasting, i.e. "2.5D/pseudo3D" rendering. It was made by drummyfish, initially as an experiment for Pokitto -- later he utilized the library in his game Anarch. It is in spirit similar to his other LRS libraries such as small3dlib and tinyphysicsengine; just as those raycastlib is kept extremely simple, it is written in pure C99, with zero dependencies (not even standard library), it's written as a single file single header library, using no floating point and tested to run interactively even on very weak devices (simplified version was made run on Arduboy with some 2 KiB of RAM). It is very flexible thanks to use of callbacks for communication, allowing e.g. programming arbitrary "shader" code to implement all kinds of effects the user desires or using procedurally generated environments without having to store any data. The library implements advanced features such as floor and ceiling with different heights, textured floor, opening door, simple collision detection etc. It is written in just a bit over 2000 lines of code.

The repository is available at <https://codeberg.org/drummyfish/raycastlib>.

*Simple rendering made with raycastlib.*

# Raylib

The following are some features of raylib as of writing this. The good and neutral features seem to be:

- And some of the bad features are:

- 150000+ lines of code
- not a header only library, requires building (makefile, optionally with cmake)
- using floating point and OpenGL creates complex dependencies and sends a fuck you to small embedded computers
- even though abstracted, some software dependencies are still needed depending on the platform, e.g. on GNU/Linux you need to install ALSA, Mesa and X11

## Raycastlib

# Reactionary Software

{ The "founder", fschmidt, sent me a link to his website on saidit after I posted about LRS. Here is how I interpret his take on technology -- as always I may misinterpret or distort something, for safety refer to the original website. ~drummyfish }

Reactionary software (reactionary meaning *opposing the modern, favoring the old*) is a kind of software/technology philosophy opposing modern technology and advocating more simplicity as a basis for better technology (and possibly whole society); it is similar e.g. to suckless and our own less retarded software, though it's not as "hardcore" minimalist (e.g. it's okay with old versions of Java which we still consider kind of bloated and therefore bad). Just as suckless and LRS, reactionary software notices the unbelievably degenerated state of "modern" technology (reflecting the degenerate state of whole society) manifested in bloat, overengineering, overcomplicating, user abuse, ugliness, buzzword hype, DRM, bullshit features, planned obsolescence, fragility etc., and advocates for rejecting it, for taking a step back to when technology was still sane (before 2000s). The website of reactionary software is at <http://www.reactionary.software> (on top it reads *Make software great again!*). There is also a nice forum at <http://www.mikraite.org/Reactionary-Software-f1999.html> (tho requires JS to register? WTF. LOL they even use Discord, that's just lame.). The spirit is good, however the people in the group mostly seem not to be the experts of computer technology (still above average tech savvy but not like "top hackers"), which of course isn't anything bad, it's just that they sometimes propose shitty "solutions" -- at least from the forum posts it seems they are mostly frustrated users rather than frustrated skilled programmers. Again, there is nothing wrong about this, we need to listen to them, it's just that we should probably rather listen to the complaints than to some of the proposed solutions.

**The biggest difference compared to suckless/LRS is that reactionary software focuses on the simplicity from user's point of view** (as stated on their forums). Of course this is not in conflict with our views, we want the same thing, however if we stay ONLY at the external simplicity, we fall into the trap of pseudominimalism -- we, the LRS, therefore additionally see the simplicity of internals as equally important of a goal.

The founder of reactionary software is fschmidt and he still seems to be the one who mostly defines it (just like drummyfish is at the moment basically solo controlling LRS), though there is a forum of people who follow him. The philosophy can potentially be extended beyond just software, to other fields of endeavor and potentially whole society -- the discussion of reactionary software revolves around wide context, e.g. things like philosophy, religion and collapse of society (fschmidt made a post where he applies Old Testament ideas to programming). This is pretty good, focus on the big picture is something we greatly embrace too.

fschmidt seems to be a lot into religion and also has some related side projects with wider scope, e.g. Arkians which deals with society and eugenics. It seems to be trying to establish a community of "chosen people" (those who pass certain tests) who selective breed to renew good genes in society. { PLEASE DON'T JUMP TO CONCLUSIONS, I just quickly skimmed through it -- people will probably freak out and start calling that guy a Nazi -- please don't, read his site first. I can't really say more about it as I didn't research it well, but he doesn't seem to be proposing violent solutions. Peace. ~drummyfish }

**What do we think about reactionary software?** To sum up: the vibes are good, it basically seems like "suckless-lite" -- we agree with what they identify as causes of decline of modern technology, we like that they discuss wide context and the big picture and our solutions are often aligned, in the same direction -- theirs are just not as radical, or maybe we just disagree on minor points. We may e.g. disagree on specific cases of software, for example they approve of old Python, Java and lightweight JavaScript used on the web -- we see such software as unacceptable, it's too complex, unnecessary and from ground up designed badly. { As clarified on the forums, reactionary software focuses on the simplicity from user's perspective, not necessarily the simplicity of internals. ~drummyfish } Nevertheless we definitely see it as good this philosophy exists, it fills a certain niche, it's a place for people who aren't necessarily hardcore hackers but still see the value of minimalism, which of course shows they're one of the more intelligent out there. Reactionary software contributes to improving technology at the very least by spreading awareness and taking actual stance, they may help provide alternatives to tech refugees who suffer from modern tech but suckless or LRS is too difficult for them to jump right into. The fact that more and more smaller communities with ideas similar to LRS come to life indicates the ideas themselves are alive and start to flourish, in a decentralized way -- this is good.

Examples of reactionary software include (examples from the site itself):

- **bash**: Possibly the most popular Unix shell. In hardcore minimalist circles bash is still considered bloated and/or harmful due to its extensions over standard Posix shell, but indeed compared to mainstream software bash is pretty KISS.
- **old versions of languages such as Java and Python**: TBH these are seriously bloated -- the older versions maybe not THAT much but still. Even if these language may appear minimal to the programmer (e.g. by syntax or concepts), they are necessarily extremely complicated on the inside (see pseudominimalism), even if just for their HUGE standard libraries.
- **Mercurial**: OK, here the guy just bashes and shits on git for being extremely bloated and unusable -- of course, git is a bit bloated, but definitely not more than Java or Python. Not sure Mercurial is really so much better. { I have literally never touched Mercurial so I don't know, I just know that Git is a bit complex but still usable (just commit, push and pull) AND it doesn't even matter that much as my project do not depend on git, git is basically just a way for me to put my code on the internet and sync in between my machines. If git stops existing I can literally just use FTP or something. ~drummyfish }
- **Luan**: Their own programming language. TODO: research it :)
- ...

## See Also

- suckless
- KISS
- bitreich
- LRS

---

README

## Less Retarded Wiki

This is online at <http://www.tastyfish.cz/lrs/main.html>.

Wiki about less retarded software and related topics.

By contributing you agree to release your contribution under CC0 1.0, public domain (<https://creativecommons.org/publicdomain/zero/1.0/>). Please do **not** add anything copyrighted to this Wiki (such as copy pasted texts from elsewhere, images etc.).

Start reading at the [main page](#).

---

real\_number

## Real Number

Real numbers are all numbers found on the infinite, continuous one dimensional number line, they often represent what we generally just mean by the term "number" and include for example zero, pi or -39/11. The set or real numbers includes all whole numbers as well as all rational numbers (fractions with integer nominator and denominator), but in addition contains infinitely many "special" numbers such as pi, e or square root of 2, numbers that are mathematically very interesting because they for example produce infinitely many digits in our traditional number notation without showing any obvious patterns. However it has to be noted real numbers still do NOT include for example infinity or complex numbers (kind of "2D extension" of real numbers) such as i. Real numbers really represent a **continuum**, between any two numbers that are not the same there is always infinitely many real numbers that have no gaps in between -- this leads to the fact that not only is there infinitely many real numbers, but there is **uncountably many** of them, i.e. simply put there is "more than traditional infinity" of real numbers. Mathematically the set of real numbers and operations with them form a structure called a field (so you'll often hear the term "field of real numbers").

WATCH OUT: even though in programming we sometimes we encounter data types named *real*, they usually don't represent true real numbers! In programming we mostly only approximate real numbers with floating or fixed point numbers, which really are only rational numbers -- this is practically always good enough, we don't ever need an exact value of pi, a few decimal digits of accuracy is enough to an engineer, but to a mathematician real numbers represent a completely new, different world with some fundamental differences, which even a mere programmer should be at least aware of. Real numbers are tied to questions of the continuum, infinitely big and infinitely small, and they can really eventually lead to deep philosophical debates.

TODO: history?

## The Greater, Uncountable Infinity Of Reals

Compared to the basic sets of numbers, such as natural and rational numbers, real numbers are special because **there are uncountably many** of them, i.e. not just infinitely many; in a sense the infinity representing how many real numbers there are is a **"bigger infinity"** than that representing the size (better said cardinality) of sets of for example natural and rational numbers (which are still infinite, but only countably infinite); even if we consider just real numbers between 0 and 1, there is still kind of "more" of them than there are e.g. all possible fractions (with integer nominator and denominator). At first it looks like fractions and real numbers are kind of the same, but this is not true, fractions are still kind of sparse, discrete, even though we can use fractions to infinitely divide the number line, there will always be kind of "gaps" between them; on the other hand real numbers are TRULY continuous in nature: there are infinitely many "special" numbers among real numbers, such as  $\pi$  and  $e$  (but most just without any special name, e.g. square root of 2), that can never be written as a fraction of integers or as a number with finite decimal expansion (which is really the same thing as being a fraction), i.e. some real numbers we can approximately write down in decimal expansion (i.e. like 1.23456...) but never finish as their decimal expansion goes on forever and lacks any simple pattern. This is what gives rise to the fact that there are many more real numbers than integers and fractions -- due to this possibility of having "infinitely many arbitrary digits after the decimal point" **we cannot produce an ordered list of real numbers**, i.e. we cannot create a system that would say "this is the first real number, this is the second, this is third, ..." in a way that would eventually list out all the real numbers. I.e. we cannot establish a 1:1 mapping between natural numbers and real numbers, something that IS possible with integers and fractions (though with fractions it's less obvious), even considering things like negative numbers, we can order integers e.g. like: 0, 1, -1, 2, -2, 3, -3, .... With real numbers this is impossible, so we say the infinite size of the set of real numbers is uncountable.

At first this is usually confusing and hard to comprehend, keep in mind we are dealing with infinities here and so intuition fails us, what we mean by "size" of an infinite set is better called a cardinality because this is not really a size as we understand it with finite sets, it's a kind of generalization of it that allows us to examine infinity, but as we start dealing with infinities we have to be careful as things we usually take for granted may no longer hold -- for example here we may have a superset of a set of numbers (e.g. all integers, including negative ones, is a superset of natural numbers, which exclude negative numbers) with both sets having the same "size"/cardinality, i.e. something that with finite sets can't happen. With infinities we cannot measure size with counting elements -- there are always infinitely many -- but we can try with making mappings between the sets, which does tell us new things. But we are still in a different realm where our traditional language doesn't work and at best we sometimes have only "close enough" terms for things we encounter there.

**Is this of use to a programmer?** Not to a "normal" programmer, in practical programming we basically never deal with true real numbers in all their generality, we just approximate them with floats (i.e. rational numbers), though a programmer should definitely at least be aware of all this, this is just very basics of higher math and should be common knowledge to anyone dealing with math in any way. Of course some specialized programming (e.g. symbolic computation) and theoretical computer science will come to deal with this, so there it is a must know.

Here is a **proof** by contradiction of not being able to create an ordered list of real numbers, by so called diagonalization. Let us only consider (without loss of generality) real numbers between 0 and 1, written in binary, i.e. numbers written only with digits 0 and 1 that always start with 0.. Suppose we have found ordering of ALL real numbers in which real numbers go one after another like this:  $R_1, R_2, R_3$  etc., each  $R_N$  having the digits (after 0.)  $R_{N_1}, R_{N_2}, R_{N_3}$  etc. We can write these numbers into a table that expands infinitely to the right and bottom:



| number | digit 1 | digit 2 | digit 3 | digit 4 | ... |
|--------|---------|---------|---------|---------|-----|
| R1     | R1_1    | R1_2    | R1_3    | R1_4    | ... |
| R2     | R2_1    | R2_2    | R2_3    | R2_4    | ... |
| R3     | R3_1    | R3_2    | R3_3    | R3_4    | ... |
| ...    | ...     | ...     | ...     | ...     | ... |

Now however we can consider a number  $X$  whose digits are not  $(R1\_1)$ , not  $(R2\_2)$ , not  $(R3\_3)$ , not  $(R4\_4)$  etc., i.e. the number is obtained by taking the table's diagonal and inverting all the digits (1s to 0s and vice versa). We can see number  $X$  is not present in the table because it is different from every other number in the table -- with any number  $RN$  it will differ AT LEAST by the  $N$ th bit. By this we arrive at the contradiction with the original claim that we have an ordered list of ALL real numbers, therefore such list cannot exist.

Another **cool view of real numbers** is this: imagine fractions (rational numbers) in fact sitting on a 2D grid, having coordinates given by their denominator and nominator, e.g. number  $3/2$  sits at  $[2,3]$ , i.e. has  $x$  coordinate 2 and  $y$  coordinate 3. We are standing at point  $[0,0]$  and so every number projects to our field of view, i.e. to an angle from  $-90$  degrees to  $90$  degrees (with these extremes representing minus and plus infinity, looking straight forward we see zero). Notice that for example numbers  $2/3$  and  $4/6$  exactly overlap from our point of view, as they represent the same value:

```

...^ nominator
6 | . . . . . / 2/3 = 4/6 = 8/12 = ...
5 | . . . . . /
4 | . . . . . /
3 | . . . . . /
2 | . . . . . /
1 | / . . . . .
us -> 0 +-----> denominator
-1 | 1 2 3 4 5 6 7 ...
-2 | . . . . .
-3 | . . . . .
...| . . . . .

```

From our point of view we can see all number, not just the fractions (which only sit on the integer grid points) -- all numbers, including real numbers, project to our field of view. Here fractions represent all the GRID points we see, i.e. a very dense set of points, however there are still gaps shining through which represent the real numbers that aren't fractions -- for example  $\pi$ ; if we shoot a ray from our standpoint in the exact angle that represents  $\pi$ , the ray will go on forever without ever hitting any grid point! Such line will nearly miss some points, such as  $355/113$ , which represents a good approximation of  $\pi$ , but it will never hit any point exactly. So real numbers here are represented by the WHOLE, CONTINUOUS field of view.

**Are there bigger sets than those of real numbers?** Of course, a superset of real number is e.g. that of complex numbers and quaternions, though they still have the same cardinality. But there are even sets that have bigger cardinality than reals, e.g. the set of all subsets of real numbers (so called power set of real numbers). In fact there are infinitely many such infinities of different cardinality.

---

recursion

## Recursion

See [recursion](#).

Recursion (from Latin recursio, "running back") in general is a situation in which a definition refers to itself; for example the definition of a human's ancestor as "the human's parents and the ancestors of his parents" (fractals are also very nice example of what a simple recursive definition can achieve). In programming recursion takes on a meaning of a **function that calls itself**; this is the meaning we'll suppose in this article, unless noted otherwise.

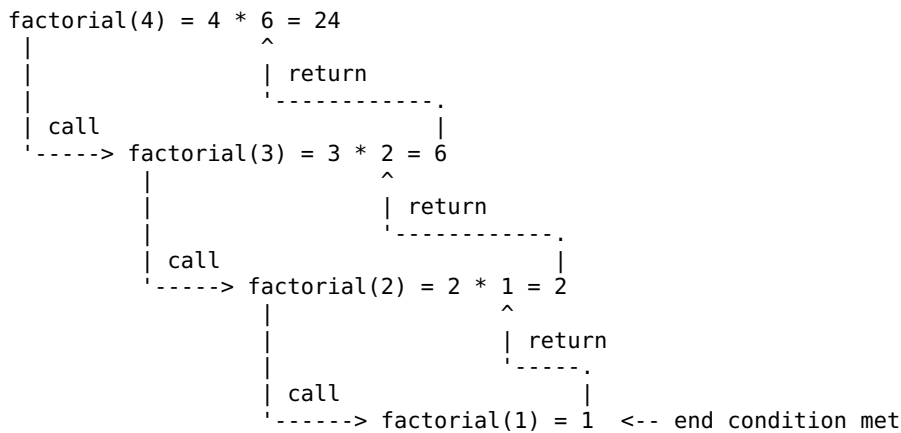
We divide recursion to a **direct** and **indirect** one. In direct recursion the function calls itself directly, in indirect function  $A$  calls a function  $B$  which ends up (even possibly by calling some more functions) calling  $A$  again. Indirect recursion is tricky because it may appear by mistake and cause a bug (which is nevertheless

easily noticed as the program will mostly run out of memory and crash).

When a function calls itself, it starts "diving" deeper and deeper and in most situations we want this to stop at some point, so in most cases **a recursion has to contain a terminating condition**. Without this condition the recursion will keep recurring and end up in an equivalent of an infinite loop (which in case of recursion will however crash the program with a stack overflow exception). Let's see this on perhaps the most typical example of using recursion, a factorial function:

```
unsigned int factorial(unsigned int x)
{
    if (x > 1)
        return x * factorial(x - 1); // recursive call
    else
        return 1; // terminating condition
}
```

See that as long as  $x > 1$ , recursive calls are being made; with each the  $x$  is decremented so that inevitably  $x$  will at one point come to equal 1. Then the *else* branch of the condition will be taken -- the terminating condition has been met -- and in this branch no further recursive call is made, i.e. the recursion is stopped here and the code starts to descend from the recursion. The following diagram show graphically computation of `factorial(4)`:



Note that even in computing we can use an infinite recursion sometimes. For example in Hashell it is possible to define infinite data structures with a recursive definition; however this kind of recursion is intentionally allowed, it is treated as a mathematical definition and with correct use it won't crash the program.

**Every recursion can be replaced by iteration and vice versa** (iteration meaning a loop such as `while`). In fact some language (e.g. functional) do not have loops and handle repetition solely by recursion. This means that you, a programmer, always have a choice between recursion and iteration, and here you should know that **recursion is typically slower than iteration**. This is because recursion has a lot of overhead: remember that every level of recursion is a function call that involves things such as pushing and popping values on stack, handling return addresses etc. The usual advice is therefore to **prefer iteration**, even though recursion can sometimes be more elegant/simple and if you don't mind the overhead, it's not necessarily wrong to go for it. Typically this is the case when you have multiple branches to dive into, e.g. in case of quicksort. In the above example of factorial we only have one recurring branch, so it's much better to implement the function with iteration:

```
unsigned int factorial(unsigned int x)
{
    unsigned int result = 1;

    while (x > 1)
    {
        result *= x;
        x--;
    }

    return result;
}
```

How do the computers practically make recursion happen? Basically they use a stack to remember states on each level of the recursion. In programming languages that support recursive function calls this is hidden behind the scenes in the form of call stack. This is why an infinite recursion causes stack overflow.

Another important type of recursion is **tail recursion** which happens when the recursive call in a function is the very last command. It is utilized in functional languages that use recursion instead of loops. This kind of recursion can be optimized by the compiler into basically the same code a loop would produce, so that e.g. stack won't grow tremendously.

Mathematical recursive functions find use in computability theory where they help us (similarly to e.g. Turing machines) define classes of functions (such as primitive recursive and partial recursive) by "how strong of a computer" we need to compute them.

---

reddit

## Reddit

Reddit, established in 2005, marketing itself as the "frontpage of the Internet", was an extremely successful, popular and quite nice website for sharing links, ideas and leading discussions about them, before it got absolutely destroyed by capitalists right before they year 2020. It used to be a forum with great amount of free speech and with quite a nice, plain user interface; in a swift turn it however turn completely over and is now among the most censored sites on the whole web, a place toxic with SJW fumes and its site is literally unusable for the amount of bloat and ads. Never visit the site if you don't have to.

Before the infamous censorship wave circa 2019 reddit used to be quite a beautiful place to behold, truly an experience unlike anything else (maybe a bit comparable to Usenet). { I used to actually love reddit, sad it died. ~drummyfish } It's hard to sum up to someone who didn't experience reddit back then, it found a great mix of excellent ideas that just worked great together, a combination mainly of free speech (that's completely gone now, it's almost comical to remember reddit used to be one of the "bastions of free speech" back then), nice minimalist user interface (also gone now), having many subforums for all kinds of niche communities, even the smallest you can imagine (like people who like round objects or people who try to talk without using some specific letter because they hate it etc.), sharing of interesting links and/or ideas, having a non-traditional comment system structured as a tree and letting people vote on both posts and individual comments to bring up the ones they found most valuable (i.e. informative, funny, interesting etc.). Users also gathered so called "karma", a kind of points they cumulated for getting upvotes, so users had some sort of "level" -- the more karma, the more "elite" the user was (users could also gift so called *reddit gold* for excellent posts, basically giving the user a free premium account for a while); this not once led to so called *karma whoring*. Anyway, reddit was like an whole new Internet within the Internet, it was just a place where you could spend hours searching and discovering things you didn't even know you wanted to find -- any hobby or any detail you had a morbid curiosity about you could dig up on reddit, you could find large interviews with ambulance drivers who told fascinating stories they saw during their careers, schizophrenic people answering questions like "can you walk through the imaginary people you see?", discussions like "what's the weirdest thing that happened to you as a beekeeper", people digging out extremely weird videos on YouTube, solving mysteries in video games, even famous people like Barak Obama took part in reddit IAMA interviews and just answered all the weird questions the internet asked them. There were also porn communities and controversial communities like *r/watchpeopledie* where users just shared videos of people dying { This was my favorite, seeing people die and suffer was actually what led me to completely reject all violence later on in my life. ~drummyfish }. This was sort of the vanilla reddit experience. However, as they always do, money and pseudoleftists soon swiftly killed all of this, a few greedy faggots just destroyed it all so that they could get even richer than they already were.

What was the big moment? Basically in 2019 reddit presented one the most visible, greatest examples of a **profit motivated 180 degree turn from a free speech site to a censorship dictatorship** -- as some cock invested money to reddit, the reddit CEO just said yeah, let's make this advertisement friendly and ban all free speech on the site; there were hilarious historical moments like Alexis Ohanian, the co-founder of the site, saying "we never intended reddit to be the bastion of free speech" while someone actually found a quote of him saying the exact opposite in the past :D This shitstorm resulted in one of the greatest disasters to ever have happened on the Internet. Subreddits such as *r/politicallyincorrect*, *r/Offensive\_Wallpapers*, *r/watchpeopledie*, *r/necrophilia*, *r/PicsOfHorseVaginas*, *r/sjwhate*, *r/lovenotacrime*, *r/fatpeoplehate* and

THOUSANDS of others were all banned (you can probably still find them in archives, but you can no longer discuss of course). Of course those who criticized this were just banned too, anyone who showed a dislike of this got a "fuck you bitch" message from a mod with a swift ban. People not familiar with reddit or Internet too much perhaps didn't notice too much, but to an Internet citizen this was comparable to something like the Pope one day waking up, admitting to atheism, dressing up as Voldemort and starting to masturbate on the balcony, cumming on people while promoting nuclear war, all because someone paid him \$1 to do it. Of course this was completely expected under capitalism, reddit just showed a very rapid, "we don't give a shit about users or society or anything but money" kind of step, one that must show clear as day even to any blind idiot what capitalism really is about. After this many people left reddit for good { Including me. ~drummyfish }, some migrated to alternative sites like Voat, but it was never what it used to be, communities were fragmented and they mostly degenerated to small groups bitching about how reddit fucked up. At least it's a great lesson learned about "free market" society.

Reddit had an extreme number of own memes, historical events, famous users, inside jokes and jargon -- it was kind of like a whole country. Especially notable are the acronyms that come from subreddit names and which reddit guys use in normal speech, like AMA (ask me anything), TIL (today I learned), TIFU (today I fucked up) or ELI5 (explain like I'm 5) etc.

Reddit is a famous rival to 4chan, it's basically the pseudoleftist forum vs the rightist forum -- the forums trash talk each other, raid each other, make fun of each other and so on.

Typical reddit thread after SJW takeover looks like this:

- [removed] +7000000
    - ♦ [removed] +20000
      - ◇ haha so hilarious, best thing I've ever read
      - ◇ [removed] -1000000
        - [removed] +300000
          - this changed my life
    - ♦ [removed] +123
- 

regex

## Regular Expression

Regular expression (shortened *regex* or *regexp*) is a kind of mathematical expression, very often used in programming, that can be used to define simple patterns in strings of characters (usually text). Regular expressions are typically used for searching patterns (i.e. not just exact matches but rather sequences of characters which follow some rules, e.g. numeric values), substitutions (replacement) of such patterns, describing syntax of computer languages, their parsing etc. (though they may also be used in more wild ways, e.g. for generating strings). Regular expression is itself a string of symbols which however describes potentially many (even infinitely many) other strings thanks to containing special symbols that may stand for repetition, alternative etc. For example `a.*b` is a regular expression describing a string that starts with letter `a`, which is followed by a sequence of at least one character and then ends with `b` (so e.g. `aab`, `abbbb`, `acaccb` etc.).

**WATCH OUT:** do not confuse regular expressions with Unix wildcards used in file names (e.g. `source/*.c` is a wildcard, not a *regexp*).

{ A popular online tool for playing around with regular expressions is <https://regexr.com/>, though it requires JS and is bloated; if you want to stay with Unix, just `grep` (possibly with `-o` to see just the matched string). ~drummyfish }

Regular expressions are widely used in Unix tools, programming languages, editors etc. Especially notable are grep (searches for patterns in files), sed (text processor, often used for search and replacement of patterns), awk, Perl, Vim etc.

From the point of view of theoretical computer science and formal languages **regular expressions are computationally weak**, they are equivalent to the weakest models of computations such as regular

grammars or **finite state machines** -- in fact regular expressions are often implemented as finite state machines. This means that **regular expressions can NOT describe any possible pattern** (for example they can't capture a math expression with brackets in which start brackets have to match end brackets), only relatively simple ones; however it turns out that very many commonly encountered patterns are simple enough to be described this way, so we have a good enough tool. The advantage of regular expressions is exactly that they are simple, yet very often sufficient.

## Details

WIP

There exist different standards and de-facto standards for regular expressions, some using different symbols, some having extra syntactic sugar (which however usually only make the syntax more comfortable, NOT more computationally powerful) and features (typically e.g. so called *capture groups* that allow to extract specific subparts of given matched pattern). There are cases where a feature makes regexes more computationally powerful, namely the backreference `\n` present in extended regular expressions (source: *Backreferences in practical regular expressions, 2020*). Most relevant standards are probably Posix and Perl (with specific implementations sometimes adding their own flavor, e.g. GNU, Vim etc.): Posix specifies **basic** and **extended** regular expression (extended usually turned on with the `-E` CLI flag). The following table sums up the most common constructs used in regular expressions:

| construct              | matches                                                                         | availability                              | example                                                                                            |
|------------------------|---------------------------------------------------------------------------------|-------------------------------------------|----------------------------------------------------------------------------------------------------|
| <code>char</code>      | this exact character                                                            | everywhere                                | <code>a</code> matches <code>a</code>                                                              |
| <code>.</code>         | any single character                                                            | everywhere                                | <code>.</code> matches <code>a</code> , <code>b</code> , <code>1</code> etc.                       |
| <code>expr*</code>     | any number (even 0) of repeating <code>expr</code>                              | everywhere                                | <code>a*</code> matches <i>empty</i> , <code>a</code> , <code>aa</code> , <code>aaa</code> , ...   |
| <code>^</code>         | start of expression (usually start of line)                                     | everywhere                                | <code>^a</code> matches <code>a</code> at the start of line                                        |
| <code>\$</code>        | end of expression (usually end of line)                                         | everywhere                                | <code>a\$</code> matches <code>a</code> at the end of line                                         |
| <code>expr+</code>     | matches 1 or more repeating <code>expr</code>                                   | escape ( <code>\+</code> ) in basic       | <code>a+</code> matches <code>a</code> , <code>aa</code> , <code>aaa</code> , ...                  |
| <code>expr?</code>     | matches 0 or 1 <code>expr</code>                                                | escape ( <code>\?</code> ) in basic       | <code>a?</code> matches either <i>empty</i> or <code>a</code>                                      |
| <code>[S]</code>       | matches anything character from set <code>S</code>                              | everywhere                                | <code>[abc]</code> matches <code>a</code> , <code>b</code> or <code>c</code>                       |
| <code>(expr)</code>    | marks group (for capt. groups etc.)                                             | escape ( <code>\(, \)</code> ) in basic   | <code>a(bc)d</code> matches <code>abcd</code> with group <code>bc</code>                           |
| <code>[A-B]</code>     | like <code>[ ]</code> but specifies a range                                     | everywhere                                | <code>[3-5]</code> matches <code>3</code> , <code>4</code> and <code>5</code>                      |
| <code>[^S]</code>      | matches any char. NOT from set <code>S</code>                                   | everywhere                                | <code>[^abc]</code> matches <code>d</code> , <code>e</code> , <code>A</code> , <code>1</code> etc. |
| <code>{M,N}</code>     | <code>M</code> to <code>N</code> repetitions of <code>expr</code>               | escape ( <code>\{, \}</code> ) in basic   | <code>a{2,4}</code> matches <code>aa</code> , <code>aaa</code> , <code>aaaa</code>                 |
| <code>e1 e2</code>     | <code>e1</code> or <code>e2</code>                                              | escape in basic                           | <code>ab cd</code> match. <code>ab</code> or <code>cd</code>                                       |
| <code>\n</code>        | backref., <code>n</code> th matched group (starts with 1)                       | extended only                             | <code>(...)*\1</code> matches e.g. <code>ABcdefAB</code>                                           |
| <code>[:alpha:]</code> | alphabetic, <code>a</code> to <code>z</code> , <code>A</code> to <code>Z</code> | Posix (GNU has <code>[[:alpha:]]</code> ) | <code>[:alpha:]*</code> matches e.g. <code>abcDEF</code>                                           |
| <code>[:alnum:]</code> |                                                                                 | same as above                             |                                                                                                    |
| <code>[:digit:]</code> |                                                                                 | same as above                             |                                                                                                    |
| <code>[:blank:]</code> |                                                                                 | same as above                             |                                                                                                    |
| <code>[:lower:]</code> |                                                                                 | same as above                             |                                                                                                    |
| <code>[:space:]</code> |                                                                                 | same as above                             |                                                                                                    |
| <code>\w</code>        | like <code>[:alnum:]</code> plus also <code>_</code> char.                      | Perl                                      |                                                                                                    |

| construct | matches        | availability | example |
|-----------|----------------|--------------|---------|
| \d        | digit, 0 to 9  | Perl         |         |
| \s        | like [:space:] | Perl         |         |

## Examples And Fun

Here we'll demonstrate some practical uses of regular expressions. Most common Unix tools associated with regular expressions are probably **grep** (for searching) and **sed** (for replacing).

The most basic use case is you just wanting to **search** for some pattern in a file, i.e. for example you are looking for all IP addresses in a file, for a certain exact word inside source code comment etc.

The following uses grep to find and count all occurrences of the word capitalism or capitalist (disregarding case with the -i flag) in a plain text version of this wiki and passes them to be counted with wc.

```
grep -i -o "capitalis[mt]" ~/Downloads/lrs_wiki.txt | wc -l
```

We find out there are 829 such occurrences.

Of course, quite frequently you may want to see the lines that match along with files and line numbers, try also e.g. `grep -m 10 -H -n "Jesus" ~/Downloads/lrs_wiki.txt`.

Now let's search for things that suck with (-o prints out just the matches instead of whole line, -m 10 limits the output to 10 results at most):

```
grep -o -m 10 "[^ ]* \(sucks\|is shit\)\" ~/Downloads/lrs_wiki.txt
```

Currently we get the following output:

```
body sucks
OS sucks
everything sucks
Everything is shit
language sucks
it sucks
D&D sucks
writing sucks
Fediverse sucks
This sucks
```

Now let's try **replacing** stuff with **sed** -- this is done with a very common format (which you should remember as it's often used in common speech) `s/PATTERN/REPLACE/g` where PATTERN is the regular expression to match, REPLACE is a string with which to replace the pattern (s stands for substitute and g for global, i.e. "replace all"). Let's say we are retarded and obsessed with muh privacy and want to censor all names in a portion of the wiki we want to print, so we'll just replace all words composed of letters that start with uppercase letter (and continue with lowercase letters) -- this will also censor other words than names but let's keep it simple for now. The command may look something like:

```
cat ~/Downloads/lrs_wiki.txt | tail -n +5003 | head -n 10 | sed "s/[A-Z][a-z]\+/<BEEP>/g"
```

Here we may get e.g.:

```
they typically have a personal and not easy to describe faith. [19]<BEEP>
was a <BEEP>. [20]<BEEP> often used the word "[21]<BEEP>" instead of
"nature" or "universe"; even though he said he didn't believe in the
traditional personal <BEEP>, he also said that the laws of physics were like
books in a library which must have obviously been written by someone or
something we can't comprehend. [22]<BEEP> <BEEP> said he was "deeply
religious, though not in the orthodox sense". <BEEP> are also very hardcore
religious people such as [23]<BEEP> <BEEP>, the inventor of [24]<BEEP>
language, who even planned to be a <BEEP> missionary. <BEEP> "true
atheists" are mostly second grade "scientists" who make career out of the
```

A more advanced feature is what we call **capture groups** that allow us to reuse parts of the matched pattern in the replacement string -- this is needed in some cases, for example if you just want to insert some extra characters in the pattern. Capture groups are parts of the pattern inside brackets (( and )) which sometimes have to be escaped to \ ( and \)); in the replacement string we then reference them with \1 (first group), \2 (second group) etc. Let's demonstrate this on the following example that will highlight all four letter words:

```
cat ~/Downloads/lrs_wiki.txt | tail -n +8080 | head -n 7 | sed "s/ \([^ ]\)\([^ ]\)\([^ ]\)\([^ ]\) / \!\!\!\1\2\3\4 /"
```

The result may look something like this:

```
Bootstrapping as a general principle can aid us in creation of extremely
!!!f-r-e-FUCKING-e!!! technology by greatly minimizing all its [7]dependencies, we are able
to create a small amount of !!!c-o-d-FUCKING-e!!! that !!!w-i-l-FUCKING-l!!! self-establish our whole
computing environment !!!w-i-t-FUCKING-h!!! only !!!v-e-r-FUCKING-y!!! small effort during the process. The
topic mostly revolves around designing [8]programming language
[9]compilers, but in general we may be talking about bootstrapping whole
computing environments, operating systems etc.
```

Now a pretty rare use case is **generating random patterns** -- we can imagine we have a regular expression describing for example a valid username in a game and for some reason we want to generate 1000 random strings that match this pattern (e.g. for bots). Now going into depth on this topic could take a long time because e.g. considering probability distributions we may get into some mathematical rabbit holes (considering that for example the regex .\* matches an arbitrarily long string, what will be the average length of a string randomly generated by this pattern? 10? 1000? 1 billion?). Anyway let's leave this aside -- if we do, there is actually a quite simple and natural way of generating random patterns from regular expressions. We can convert the regexp into nondeterministic finite automaton, then make a random traverse of the graph and generate the string along the way. There don't even seem to be many Unix tools for doing this -- at the time of writing this one of the simplest way seems to be with Perl and one of its libraries, which currently still has some great limitations (no groups, no special characters in square brackets, ...), but it's better than nothing. The command we'll be using is:

```
perl -e "use String::Random; for (1..20) { print String::Random->new->randregex('REGEX') . \"\n\"; }" 2>/dev/null
```

Here are some strings generated with different REGEXes:

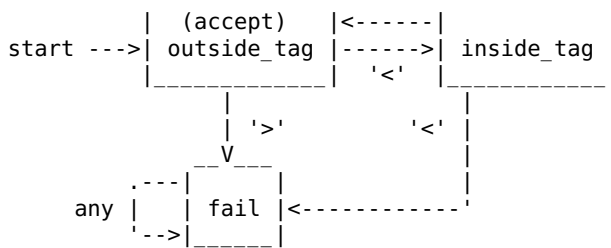
- .....\*: \n\"; }", /dB|^hRR/3za~, ![5q-%0NK, "oJT.UzSa}{0, t"}Yq]sWZjIv, Fq<xs~\_e, H=5yt9q<>29XW, <EoVf)ORH{m, ...
- [A-Z][a-z]{3,8}: Ponlc, Xwawo, Hgrtky, Brmcitpsw, Qrogdze, Olhyb, Gqeelz, Igppehljz, Azrdava, ...
- [[:;=8B][-o]?[ ]({}/|PS[:;o}, :-), ;(, B(, 8o), =/, :o|, =}, =-(, ...
- [lL][oue]+lz?: Loeoolz, Luel, luuoolz, lol, Leelz, Loeouoeouelz, luueoolz, ...
- ...

Let's now try to **program** a very simple regular expression in C. You can do this in quite fancy ways, serious regex libraries will typically let you specify arbitrary regular expression with a string at runtime (for example `char *myRegex = "(abc|ABC).*d+";`), then compile it to some fast, efficient representation like the mentioned state machine and use that for matching and replacing patterns. We'll do nothing like that here as that's too complex, we will simply make a program that has one hard wired regular expression and it will just say if given input string matches or not. Let's consider the following regular expression:

```
(<[^<>]*>|^[^<>]*)*
```

It describes an "XML"-like text; the text can contain tags that start with < and end with >, but there mustn't e.g. be a tag inside another tag. For example <hello> what <world> will match, but hello > world << bruh won't match. OK, so the first thing to do is to convert the regular expression to a finite state automaton -- this can be done intuitively but there is also an exact algorithm that can do this with any regular expression (look it up if you need it). Our automaton will look like this:





Here we start in the `outside_tag` state and move between states depending on what characters we read from the input string we are checking (indicated next to the arrows). If we end up in the `outside_tag` state again (marked as *accepting* state) when all is read, the input string matched the regular expression, otherwise it didn't. We'll translate this automaton to a C program:

```

#include <stdio.h>

#define STATE_OUTSIDE_TAG 0
#define STATE_INSIDE_TAG 1
#define STATE_FAIL 2

int main(void)
{
    int state = STATE_OUTSIDE_TAG;

    while (1)
    {
        int c = getchar();

        if (c == EOF)
            break;

        switch (state)
        {
            case STATE_OUTSIDE_TAG:
                if (c == '<')
                    state = STATE_INSIDE_TAG;
                else if (c == '>')
                    state = STATE_FAIL;

                break;

            case STATE_INSIDE_TAG:
                if (c == '>')
                    state = STATE_OUTSIDE_TAG;
                else if (c == '<')
                    state = STATE_FAIL;

                break;

            case STATE_FAIL:
                break;
        }
    }

    puts(state == STATE_OUTSIDE_TAG ? "matches!" : "string didn't match :(");
    return 0;
}
  
```

Just compile this and pass a string to the standard input (e.g. `echo "<testing> string" | ./program`), it will write out if it matches or not.

Maybe it seems a bit overcomplicated -- you could say you could program the above even without regular expressions and state machines. That's true, however imagine dealing with a more complex regex, one that matches a quite complex real world file format. Consider that in HTML for example there are pair tags, non-pair tags, attributes inside tags, entities, comments and many more things, so here you'd have great difficulties creating such parser intuitively -- the approach we have shown can be completely automatized and will work as long as you can describe the format with regular expression.



resnicks\_termite

## Resnick's Termite

WORK IN PROGRESS

{ Found this in the book *The Computational Beauty of Nature*. --drummyfish }

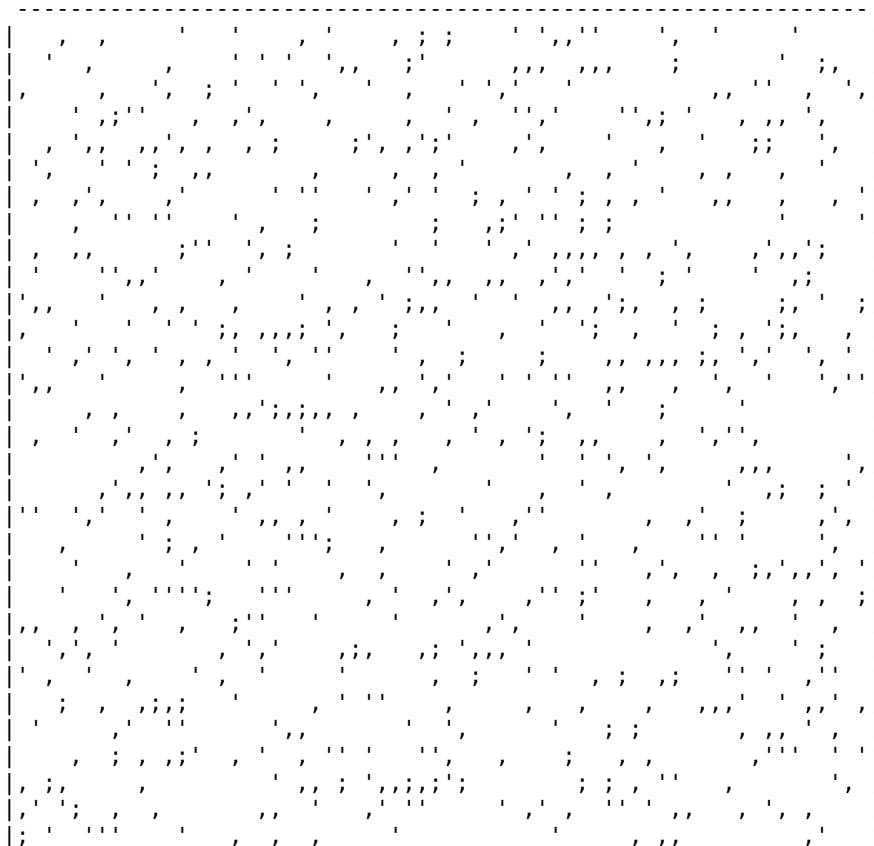
Resnick's termite is a simple cellular automaton simulating behavior of ants, demonstrating how even a very dumb behavior of a single agent can lead to higher collective intelligence once we increase the number of the agents. The simulation was made by Mitchel Resnick, the theme is similar to that of Langton's ant but Resnick's termites are stochastic, nondeterministic, they rather show how statistics/randomness in behavior help many ants build tunnels in sand. The game demonstrates how randomly scattered chips start getting chunked together and form tunnels once we let ants with extremely simple behavior work together on moving the chips. Besides this demonstration however there doesn't seem to be anything more interesting going on (at least until we start to modify and tweak the thing somehow).

The system is defined quite simply: we have a world made of cells, each cell can be either empty or have a wooden chip on it. In this world we have a number of ants, each of which behaves by the following algorithm:

1. Randomly walk around until you bump into a chip.
2. If you are not carrying a chip, pick up the one you bumped into, otherwise drop the chip you are carrying. Go to step 1.

The original implementation had ants who had direction (up, right, down, left) and on each step could make a random turn to the right or left. If an ant bumped into a chip it turned 180 degrees. These things prevented some annoying patterns like an ant just picking up a chip and immediately dropping it etc. Some further modifications were suggested like giving the ants some simple sense of sight or teleporting them randomly after dropping the chip.

iteration 0:





```

for (int y = 0; y < WORLD_SIZE; y += 2)
{
    putchar('|');
    for (int x = 0; x < WORLD_SIZE; ++x)
    {
        int n = y * WORLD_SIZE + x;

        switch ((world[n] <= 1) | (world[n + WORLD_SIZE]))
        {
            case 1: putchar('\n'); break;
            case 2: putchar(','); break;
            case 3: putchar(';'); break;
            default: putchar(' '); break;
        }
    }

    putchar('|');
    putchar('\n');
}

printhBorder();
}

void updateAnts(void)
{
    for (int i = 0; i < ANTS; ++i)
    {
        int newPos = // this just randomly moves in one direction
            (WORLD_SIZE * WORLD_SIZE +
             ants[i].pos +
             ((rand() % 2) ? ((rand() % 2) ? -1 : 1) :
              ((rand() % 2) ? -1 * WORLD_SIZE : WORLD_SIZE)))
            % (WORLD_SIZE * WORLD_SIZE);

        if (world[newPos]) // stepped on a chip?
        {
            if (ants[i].chip)
            { // has chip; drop the chip
                if (!world[ants[i].pos])
                {
                    ants[i].chip = 0;
                    world[ants[i].pos] = 1;
                }
            }
            else
            { // no chip; pick up the chip
                world[newPos] = 0;
                ants[i].chip = 1;
            }
        }

        ants[i].pos = newPos;
    }
}

int main(void)
{
    srand(123);

    for (int i = 0; i < WORLD_SIZE * WORLD_SIZE; ++i)
        world[i] = (rand() % CHIP_DENSITY) == 0;

    for (int i = 0; i < ANTS; ++i)
    {
        ants[i].pos = rand() % (WORLD_SIZE * WORLD_SIZE);
        ants[i].chip = 0;
    }

    int i;

    while (1)
    {

```

```

    if (i % 65536 == 0)
    {
        printf("iteration %d:\n",i);
        printWorld();
    }

    updateAnts();
    i++;
}

printWorld();
return 0;
}

```

---

rgb332

## RGB332

RGB332 is a general 256 color palette that encodes one color with 1 byte (i.e. 8 bits): 3 bits (highest) for red, 3 bits for green and 2 bits (lowest) for blue (as human eye is least sensitive to blue we choose to allocate fewest bits to blue). RGB332 is an implicit palette -- it doesn't have to be stored in memory (though doing so also has justifications) because the color index itself determines the color and vice versa. Compared to the classic 24 bit RGB (which assigns 8 bits to each of the RGB components), RGB332 is very "KISS/suckless" and often good enough (especially with dithering) as it saves memory, avoids headaches with endianness and represents each color with just a single number (as opposed to 3), so it is often used in simple and limited computers such as embedded. It is also in the public domain, unlike some other palettes, so it's additionally a legally safe choice. RGB332 also has a "sister palette" called RGB565 which uses two bytes instead of one and so offers many more colors.

A disadvantage of plain 332 palette lies in the linearity of each component's intensity, i.e. lack of gamma correction, so there are too many almost indistinguishable bright colors while too few darker ones { TODO: does a gamma corrected 332 exist? make it? ~drummyfish }. Another disadvantage is the non-alignment of the blue component with red and green components, i.e. while R/G components have 8 levels of intensity and so step from 0 to 255 by 36.4, the B component only has 4 levels and steps by exactly 85, which makes it impossible to create exact shades of grey (which of course have to have all R, G and B components equal).

The RGB values of the 332 palette are following:

```

#000000 #000055 #0000aa #0000ff #002400 #002455 #0024aa #0024ff
#004800 #004855 #0048aa #0048ff #006d00 #006d55 #006daa #006dff
#009100 #009155 #0091aa #0091ff #00b600 #00b655 #00b6aa #00b6ff
#00da00 #00da55 #00daaa #00daff #00ff00 #00ff55 #00ffaa #00ffff
#240000 #240055 #2400aa #2400ff #242400 #242455 #2424aa #2424ff
#244800 #244855 #2448aa #2448ff #246d00 #246d55 #246daa #246dff
#249100 #249155 #2491aa #2491ff #24b600 #24b655 #24b6aa #24b6ff
#24da00 #24da55 #24daaa #24daff #24ff00 #24ff55 #24ffaa #24ffff
#480000 #480055 #4800aa #4800ff #482400 #482455 #4824aa #4824ff
#484800 #484855 #4848aa #4848ff #486d00 #486d55 #486daa #486dff
#489100 #489155 #4891aa #4891ff #48b600 #48b655 #48b6aa #48b6ff
#48da00 #48da55 #48daaa #48daff #48ff00 #48ff55 #48ffaa #48ffff
#6d0000 #6d0055 #6d00aa #6d00ff #6d2400 #6d2455 #6d24aa #6d24ff
#6d4800 #6d4855 #6d48aa #6d48ff #6d6d00 #6d6d55 #6d6daa #6d6dff
#6d9100 #6d9155 #6d91aa #6d91ff #6db600 #6db655 #6db6aa #6db6ff
#6dda00 #6dda55 #6ddaaa #6ddaff #6dff00 #6dff55 #6dffaa #6dffff
#910000 #910055 #9100aa #9100ff #912400 #912455 #9124aa #9124ff
#914800 #914855 #9148aa #9148ff #916d00 #916d55 #916daa #916dff
#919100 #919155 #9191aa #9191ff #91b600 #91b655 #91b6aa #91b6ff
#91da00 #91da55 #91daaa #91daff #91ff00 #91ff55 #91ffaa #91ffff
#b60000 #b60055 #b600aa #b600ff #b62400 #b62455 #b624aa #b624ff
#b64800 #b64855 #b648aa #b648ff #b66d00 #b66d55 #b66daa #b66dff
#b69100 #b69155 #b691aa #b691ff #b6b600 #b6b655 #b6b6aa #b6b6ff
#b6da00 #b6da55 #b6daaa #b6daff #b6ff00 #b6ff55 #b6ffaa #b6ffff
#da0000 #da0055 #da00aa #da00ff #da2400 #da2455 #da24aa #da24ff
#da4800 #da4855 #da48aa #da48ff #da6d00 #da6d55 #da6daa #da6dff
#da9100 #da9155 #da91aa #da91ff #dab600 #dab655 #dab6aa #dab6ff
#dada00 #dada55 #dadaaa #dadaff #daff00 #daff55 #daffaa #daffff

```

```
#ff0000 #ff0055 #ff00aa #ff00ff #ff2400 #ff2455 #ff24aa #ff24ff
#ff4800 #ff4855 #ff48aa #ff48ff #ff6d00 #ff6d55 #ff6daa #ff6dff
#ff9100 #ff9155 #ff91aa #ff91ff #ffb600 #ffb655 #ffb6aa #ffb6ff
#ffda00 #ffda55 #ffdaaa #ffdaff #ffff00 #ffff55 #ffffaa #ffffff
```

## Operations

Here are C functions for converting RGB332 to RGB24 and back:

```
unsigned char rgbTo332(unsigned char red, unsigned char green, unsigned char blue)
{
    return ((red / 32) << 5) | ((green / 32) << 2) | (blue / 64);
}

void rgbFrom332(unsigned char colorIndex, unsigned char *red, unsigned char *green, unsigned char *blue)
{
    unsigned char value = (colorIndex >> 5) & 0x07;
    *red = value != 7 ? value * 36 : 255;

    value = (colorIndex >> 2) & 0x07;
    *green = value != 7 ? value * 36 : 255;

    value = colorIndex & 0x03;
    *blue = (value != 3) ? value * 72 : 255;
}
```

Addition/subtraction of two RGB332 colors can be performed by simply adding/subtracting the two color values as long as no over/underflow occurs in either component -- by adding the values we basically perform a parallel addition/subtraction of all three components with only one operation. Unfortunately checking for when exactly such overflow occurs is not easy to do quickly { Or is it? ~drummyfish }, but to rule out e.g. an overflow with addition we may for example check whether the highest bit of each component in both colors to be added is 0 (i.e. if (((color1 & 0x92) | (color2 & 0x92)) == 0) newColor = color1 + color2;). { Code untested. ~drummyfish }

Addition/subtraction of colors can also be approximated in a very fast way using the OR/AND operation instead of arithmetic addition/subtraction -- however this only works sometimes (check visually). For example if you need to quickly brighten/darken all pixels in a 332 image, you can just OR/AND each pixel with these values:

```
brighten by more:  doesn't really work anymore
brighten by 3:      | 0x6d (011 011 01)
brighten by 2:      | 0x49 (010 010 01)
brighten by 1:      | 0x24 (001 001 00)
darken by 1:        & 0xdb (110 110 11)
darken by 2:        & 0xb6 (101 101 10)
darken by 3:        & 0x92 (100 100 10)
darken by more:     doesn't really work anymore
```

{ TODO: Would it be possible to accurately add two 332 colors by adding all components in parallel using bitwise operators somehow? I briefly tried but the result seemed too complex to be worth it though. ~drummyfish }

Inverting a 332 color is done simply by inverting all bits in the color value.

TODO: blending?

## See Also

- [RGB565](#)
- [palette](#)
- [grayscale](#)

---

rgb565

RGB565 is a way of representing a total of 65536 colors in just 2 bytes, i.e. 16 bits, by using 5 bits (highest) for red, 6 bits for green (to which human eye is most sensitive) and 5 bits for blue; it can also be seen as a color palette. It is similar to rgb332

right

See left vs right.

# Rights Culture

rms

{ RMS is a legend and overall a great human, but let's be reminded we shouldn't be creating any heroes or celebrities. ~drummyfish }

[illegible]

Stallman's life along with free software's history is documented by a free-licensed book named *Free as in Freedom: Richard Stallman's Crusade for Free Software* on which he collaborated. You can get it gratis e.g. at Project Gutenberg. You should read this!

Richard Stallman is also famous for having foreseen and foretold virtually all the atrocities that corporations would do with computer technology, such as all the spying through cell phones, trade of personal data and abusing secrecy and "intellectual ownership" of source code for bullying others, though to be honest it doesn't take a genius to foresee that corporations will want to rape people as much as possible, it's more of a surprise he was one of very few who did. The important thing is he acted immediately he spotted this -- though corporations indeed did go on to rape people anyway, Richard Stallman made some very important steps early on to make the impact much less catastrophic nowadays. We should be all grateful.

tl;dr: At 27 as an employee at MIT AI labs Stallman had a bad experience when trying to fix a Xerox printer who's proprietary software source code was made inaccessible; he also started spotting the betrayal of hacker principles by others who decided to write proprietary software -- he realized proprietary software was inherently wrong as it prevented studying, improvement and sharing of software and enable abuse of users. From 1982 he was involved in a "fight" against the Symbolics company that pushed aggressive proprietary software; he was rewriting their software from scratch to allow Lisp Machine users more freedom -- here he proved his superior programming skills as he was keeping up with the whole team of Symbolics programmers. By 1983 his frustration reached its peak and he announced his GNU project on the Usenet -- this was a project to create a completely free as in freedom operating system, an alternative to the proprietary Unix system that would offer its users freedom to use, study, modify and share the whole software, in the hacker spirit. He followed by publishing a manifesto and establishing the Free Software Foundation. GNU and FSF popularized and standardized the term free (as in freedom) software, copyleft and free licensing, mainly with the GPL license. In the 90s GNU adopted the Linux operating system kernel and released a complete version of the GNU operating system -- these are nowadays known mostly as "Linux" distros. As a head of FSF and GNU Stallman more or less stopped programming and started traveling around the world to give talks about free software and has earned his status of one of the most important people in software history.

Regarding software Stallman has for his whole life strongly and tirelessly promoted free software and copyleft and has himself only used free software; he has always practiced what he preached and led the best example of how to live without proprietary software. This in itself is extremely amazing and rare, regardless of whether he ever slipped (which we aren't aware of) or to what degree we agree with his ideas; his moral strength and integrity is really what makes him special among basically all other great people of recent centuries, it's really as if he comes from a different time when people TRULY internally believed something so much they would die for it, that they wouldn't sell even a small part of that belief for any kind of personal benefit; this is something that really puts him alongside the greatest philosophers such as Plato or Socrates (who followed his own principles so much that he voluntarily died for them).

Fun fact: there is a package called vrms, for virtual RMS, that checks whether you have any non-free packages installed. Ironically it seems to not even tolerate non-free documentation under GFDL with invariant sections, which is very correct but probably not something Stallman himself would do since GFDL is basically his own invention :)

This said, we naturally also have to state we don't nearly agree with all he says. For example he isn't too concerned about bloat (judging by the GNU software and his own creation, Emacs) and he also doesn't care that much about free culture (some of his written works prohibit modification, see GFDL's "invariant seciotns", and his GNU project allows proprietary non-functional data as long as they are not "software"). Sadly he has also shown signs of being a type A fail personality by writing about some kind of newspeak "gender neutral language" and by seeming to be caught in a fight culture. On his website he also has an American flag and claims to be a patriot, i.e. leaning to nationalism and therefore fascism. Nevertheless he definitely can't be accused of populism or hypocrisy as he basically tells what he considers to be the truth no matter what, and he is very consistent in this. Some of his unpopular opinions (mostly those opposing pedophile witch hunt, with which we DO agree) brought him a lot of trouble and an endless wrath of SIWs. For this **he was cancelled** and in 2019 was forced to resigned from the position of president of the FSF but continues to support it.

He is a weird guy, having been recorded on video eating dirt from his feet before giving a lecture. In the book *Free as in Freedom* he admits he might be slightly autistic. Nevertheless he's pretty smart, has magna cum laude degree in physics from Harvard, 10+ honorary doctorates, fluently speaks English, Spanish and French and a little bit of Indonesian and has many times proven his superior programming skills (even though he later stopped programming to fully work on promoting the FSF).

Stallman has a beautifully minimalist website at <http://www.stallman.org> where he actively comments on current news and issues. He also made the famous free software song (well, only the lyrics, the melody is taken from a Bulgarian folk song Sadi Moma) -- he often performs it in public himself (he is pretty good at keeping the weird rhythm of the song while at the same time also singing, that's impressive).

Stallman has been critical of capitalism though he probably isn't a hardcore anticapitalist (he's an American after all). Wikidata states he's a proponent of alter-globalization (not completely against globalization in certain areas but not supporting the current form of it).

In the book *Free As In Freedom* it is also mentioned that **Stallman had aversion to passwords and secrecy in general** -- at MIT he used the username RMS with the same password so that other people could easily log in through his account and access ARPANET (the predecessor of Internet). Indeed, we applaud this, the "security" hysteria is killing the computing world.

As anarchists we of course despise the idea of worshiping people, creating heroes and cults of personalities, but the enormous historical significance of Stallman has to be stressed as a plain and simple fact and though we may disagree with some of his methods and even opinions, it's as clear as it can be that he acted selflessly, in favor of all people -- something that can be said about very few, if anyone at all. Most other old time hackers, such as Eric S. Ramoynd and Rob Pike immediately abandoned all ideals of ethics and jumped the capitalist train with the first sight of money, Stallman stayed opposed to it, and for this he has our uttermost respect. Even though in our days his name is overshadowed in the mainstream by rich businessman and creators of commercially successful technology and even though we ourselves disagree with Stallman on some points, in the future history may well see Stallman as perhaps the greatest man of the software era, and rightfully so. Stallman isn't a mere creator of a commercially successful software product or a successful politician, he is an extremely morally strong philosopher, a great example to others, a prophet, someone who sees the truth and shows it to people -- he brilliantly foresaw the course of history and quickly defined ethics needed for the new era of mass available programmable computers at the right time, before the hammer hit. And not only that, he also basically alone established this ethics as a standard IN SPITE of all the world's corporations fighting back, in a field that back then was relatively obscure, unpopular in mainstream and hence not much supported by any mass media. He is also extremely unique in not pursuing self interest, in TRULY living his own philosophy, dedicating his whole life to his cause and refusing to give in even partially. All of this is at much higher level than simply becoming successful and famous within the contemporary capitalist system, his life effort is pure, true and timeless, unlike things achieved by pieces of shit such as Steve Jobs.

---

robot

## Robot

TODO

{ Imao <https://yewtu.be/watch?v=ewAbJn96krw> ~drummyfish }

---

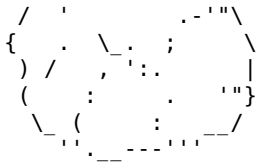
rock

## Rock

Rocks and stones are natural formations of minerals that can be used to create the most primitive technology. Stone age was the first stage of our civilization; it was characterized by use of stone tools. Rock nerds are called geologists.

\_\_...--'----'..





rock

Rocks are pretty suckless and LRS because they are simple, they are everywhere, free and can be used in a number of ways such as:

- As a building material.
- For fun and entertainment, you can play various games with rocks: rock skipping, petanque, long throw, even chess.
- As weapons, even though we discourage this use.
- For making tools such as knives or hammers.
- To hold heat: you can e.g. heat stones in fire and let them heat you while you sleep or use them to cook something.
- For writing, some can be used as a chalk, some may be used to carve into, which makes rocks an extremely durable storage for both digital and analog data -- thanks to this we have records of very old history, see also RCBD. A dust from some rocks can also be used to make dye to paint or write with.
- As weights.
- For help with counting (e.g. abacus) -- this makes rocks a kind of computer!
- To make art, decorations, small statues etc.
- For breaking things, grinding, sharpening etc.
- With advanced technology we can get metals out of rocks, extract geological knowledge from them etc.
- Some rocks can be used to start fire.
- Some are pretty rare and can be used as a currency, even though we hate money and discourage this as well.
- You can collect them as a cool non-capitalist hobby.
- As pets... apparently pet rocks are a thing :)
- ...

## See Also

- sand
- wood
- fire
- potato
- metal
- pop

---

ronja

## Ronja

{ I was informed about this by a friend over email <3 I basically paraphrase here what he told me. See also <http://www.modulatedlight.org/>. ~drummyfish }

Ronja (reasonable optical near joint access) is a free/open KISS device for wireless connection of two devices using light (i.e. optical communication) and the ethernet protocol; it can be made at home (for about \$100), doesn't require any MCUs and as such can be considered a LRS/suckless alternative to traditional WiFi routers that are de-facto owned and controlled by corporations. It works full duplex up to the distance of 1400 meters with a speed of 10 Mbps, which is pretty amazing. One can also imagine Ronja as a kind of ethernet cable, just implemented with light instead of electricity. The design is released under GFDL. The project website is at <http://ronja.twibright.com/>.

There are many advantages in Ronja -- besides the mentioned KISS design and all its implications (freedom, repairability, cheap price, compatibility, ...), Ronja doesn't use radio so there are no bullshit issues with legal bands etc., it also works with just an ethernet card, offers a stable and constant transmission speed with very low latency, can be potentially harder to block with jammers and to spy on: besides visible light the transmission can also use infrared spectrum and narrow direction of transmission, as opposed to radiating to all directions like wi-fi, also the fast flickering of the LED is unnoticeable by human or even normal cameras, therefore Ronja transmission is expensive to detect. Also note that some kind of protocol-level encryption can be used above Ronja, if one so desires. This makes it a nice communication tool for people under oppressive regimes like those in China or USA.

## See Also

- li-fi

---

rsa

## RSA

TODO

generating keys:

1.  $p := \text{large random prime}$
2.  $q := \text{large random prime}$
3.  $n := p * q$
4.  $f := (p - 1) * (q - 1)$  (this step may differ in other versions)
5.  $e := 65537$  (most common, other constants exist)
6.  $d := \text{solve for } x: x * e = 1 \bmod f$
7.  $\text{public key} := (n, e)$
8.  $\text{private key} := d$

message encryption:

1.  $m := \text{message encoded as a number} < n$
2.  $\text{encrypted} := m^e \bmod n$

message decryption:

1.  $m := \text{encrypted}^d \bmod n$
2.  $\text{decrypted} := \text{decode message from number } m$

---

rule110

## Rule 110

Not to be confused with rule 34 xD

Rule 110 is a specific cellular automaton (similar to e.g. Game of Life) which shows a very interesting behavior -- it is one of the simplest Turing complete (computationally most powerful) systems with a balance of stable and chaotic behavior. In other words it is a system in which a very complex and interesting properties emerge from extremely simple rules. The name *rule 110* comes from truth table that defines the automaton's behavior. Its **extreme simplicity** combined with full computational power is what makes rule 110 of great interest -- for example it can relatively easily be implemented as a mechanical computer using only marbles and a very simple maze (there's a video somewhere on the Internet).

Rule 110 is one of 256 so called elementary cellular automata which are special kinds of cellular automata that are one dimensional (unlike the mentioned Game Of Life which is two dimensional), in which cells have 1 bit state (1 or 0) and each cell's next state is determined by its current state and the state of its two

immediate neighboring cells (left and right). Most of the 256 possible elementary cellular automata are "boring" but rule 110 is special and interesting. Probably the most interesting thing is that rule 110 is Turing complete, i.e. it can in theory compute anything any other computer can, while basically having just 8 rules. 110 (along with its equivalents) is the only elementary automaton for which Turing completeness has been proven.

For rule 110 the following is a table determining the next value of a cell given its current value (center) and the values of its left and right neighbor.

| left center right center next |   |   |   |
|-------------------------------|---|---|---|
| 0                             | 0 | 0 | 0 |
| 0                             | 0 | 1 | 1 |
| 0                             | 1 | 0 | 1 |
| 0                             | 1 | 1 | 0 |
| 1                             | 0 | 0 | 1 |
| 1                             | 0 | 1 | 1 |
| 1                             | 1 | 0 | 1 |
| 1                             | 1 | 1 | 0 |

The rightmost column is where elementary cellular automata differ from each other -- here reading the column from top to bottom we get the binary number 01101110 which is 110 in decimal, hence we call the automaton rule 110. Some automata behave as "flipped" versions of rule 110, e.g. rule 137 (bit inversion of rule 110) and rule 124 (horizontal reflection of rule 110) -- these are in terms of properties equivalent to rule 110.

The following is an output of 32 steps of rule 110 from an initial tape with one cell set to 1. Horizontal dimension represents the tape, vertical dimension represents steps/time (from top to bottom).

```
#
##
###
# ##
#####
#  ##
##  ###
### # ##
# #####
###    ##
# ##    ###
#####  # ##
#  ## #####
##  ### #  ##
### # ##### ###
# #####  ## # ##
###  ## #####
# ## #####  ##
##### # ##  ###
#  ##### ### # ##
##  #  ## ##  #####
### ## # ##### #  ##
# #####  ## ##  ###
###  ## ##### # ##
# ##  ### #  #####
##### # ##### #  ##
#  ##  ### ##  ##  ###
##  ### # ## ##  ##  # ##
### # ## ##### ### ##  #####
# #####  ### ##### #  ##
###  ##  #  ##  #####  ###
# ##  ###  ### ##  #  ## #  ##
```

The output was generated by the following C code.

```
#include <stdio.h>
```

```

#define RULE 110 // 01100111 in binary
#define TAPE_SIZE 64
#define STEPS 32

unsigned char tape[TAPE_SIZE];

int main(void)
{
    // init the tape:
    for (int i = 0; i < TAPE_SIZE; ++i)
        tape[i] = i == 0;

    // simulate:
    for (int i = 0; i < STEPS; ++i)
    {
        for (int j = 0; j < TAPE_SIZE; ++j)
            putchar(tape[j] ? '#' : ' ');

        putchar('\n');

        unsigned char state = // three cell state
            (tape[1] << 2) |
            (tape[0] << 1) |
            tape[TAPE_SIZE - 1];

        for (int j = 0; j < TAPE_SIZE; ++j)
        {
            tape[j] = (RULE >> state) & 0x01;
            state = (tape[(j + 2) % TAPE_SIZE] << 2) | (state >> 1);
        }
    }

    return 0;
}

```

Discovery of rule 110 is attributed to [Stephen Wolfram](#) who introduced elementary cellular automata in 1983 and conjectured Turing completeness of rule 110 in 1986 which was proven by Matthew Cook in 2004.

## See Also

- [Game Of Life](#)
- [Langton's Ant](#)
- [interaction net](#)

---

rust

## Rust

Rust is an extremely poor attempt at a politically motivated [capitalist programming language](#) and one of the prime examples of badly designed software in general. It is extremely [harmful](#) not just because of its awful design and implementation and motivation, it also promotes [toxic](#) politics, tries to replace relatively good languages such as [C](#) and, worst of all, is gaining popularity among highly unqualified coding monkeys, i.e. the majority of people creating technology nowadays, so it is infecting everything and contributing to the downfall of technology. FOR THE LOVE OF GOD STAY AS FAR AWAY AS POSSIBLE FROM RUST.

[LMAO https://github.com/mTvare6/hello-world.rs](https://github.com/mTvare6/hello-world.rs)

Some things exist solely to give a really bad example of how it shouldn't be done -- indeed, at least at this Rust succeeded.

It should be made clear that **rust is shit** AND **CANNOT BE FIXED**, it is awful from the ground up and the only way to deal with it is to delete it. To mention just a few issues:

- **Rust is bloated as hell**, it violates the most important philosophy in programming: the Unix philosophy, and tries to do everything at once (i.e. follow the Windows philosophy), following the spirit of latest cancer such as systemd. As such it sports **TONS of dependencies even for trivial programs**, its toolchains is huge, complex and complicated. The repo has FKN OVER 200 MB OF SOURCE CODE??? It probably doesn't have to be said it includes such unnecessary trash as generics, twisted object obsession ("traits"), package manager, pushed memory safety and whatnot. It itself depends on extreme bloat like Python, ninja, cmake etc. Apparently compiling rust even requires Internet connection to download some bootstrap shit? Even SJW littered places like permacomputing wiki are just forced to admit it's not a minimalist language by any stretch.
- **It's just complete shit written**, everyone complains it compiles slow as hell (both rust programs and rust itself), it creates **HUGE binaries** because it statically links all the dependencies and runtime environment LMAO.
- **It's just badly designed as a language**, taking the mainstream road of bringing in more complexity, shiny new paradigms and features, accumulating layers of abstraction, adding handholders, "safety" OCD etc., instead of rather taking as inspiration a good language such as C and trying to make it better by SIMPLIFYING it. Their vision is that of someone who just learned about computers and naturally wants to just ADD MORE STUFF because "bigger is better", completely ignoring experienced people who know from practice that actual improvement lies in simplification.
- **Rust is corporate capitalist software** sponsored by Big Tech, organizations and corporations like Mozilla and Micro\$oft, trying to perpetuate the philosophy of "modern" anti-people technology rather than steering it in better direction. The Rust brand is trademarked and "protected" by corporations owning it. Remember, a corporation NEVER sponsors anything without trying to buy some control over that thing, "don't bite the hand that feeds you" is not just a phrase.
- **It has licensing/freedom issues**, specifically making it difficult to exercise freedom 3 (legally modify software), see e.g. [https://web.archive.org/web/20191224132425/https://wiki.hyperbola.info/doku.php?id=en:main:rusts\\_freedom](https://web.archive.org/web/20191224132425/https://wiki.hyperbola.info/doku.php?id=en:main:rusts_freedom) and <https://www.gnu.org/philosophy/open-source-misses-the-point.html> (criticism by GNU itself). **Its "open source" label is just openwashing**, it is "open source" in the same way Android is open source (it has a "FOSS" license but it is de facto centrally controlled by some fascist group).
- **Rust has no specification**, it tries to discourage other implementations and as such greatly hurts basic principles of freedom. Lack of fixed specifications also creates update culture etc.
- Due to the above, Rust is not really a language, it is more of a "platform" or "software framework" for programs -- Rust is a programming language in the same sense in which Google App Store is a package manager. If it becomes the mainstream language, it will do to computer programs what e.g. Steam has done to video games.
- **Rust is fascist tranny software** -- not because it is written by transsexuals, but because it sports political discrimination through codes of conduct and creating a de-facto monopoly on the language (with bloat monopoly, lack of specification, kicking people out of development based on their political views etc.). Its community is infamous for being extremely **toxic and woke**.
- Its selling point is supposed to be memory safety but that makes the language **extremely slow**. For fast programs safety has to be turned off but then you can just write in C, so there is no point to Rust -- definitely not as a replacement for C.
- It ridiculously tries to be a "**handholding system language**", i.e. an expert level language designed for beginners or incompetent people (of course with the obvious goal to empower women forced into tech etc.). Being written by pre-beginner level "coders", it sets goals such as "system language should tie your hands and prevent beginner mistakes". It's like trying to design a fighter jet around the idea that it will be operated by beginners and therefore e.g. removing fire buttons or limiting top speed so as to limit dangers posed by sitting an amateur pilot in a fighter jet.
- It is more than anything a **political language trying to push propaganda**, send messages about minorities in tech instead of creating good technology, and "fight" and destroy languages invented by straight white men -- the most popular activity of rust soydevs seems to be rewriting already existing, well established, tested and greatly optimized programs from scratch in rust. As such language it is mostly written by angry minorities such as girls and trannies who can't much into technology, hence the completely shit design and implementation.
- **It tries to displace good languages** such as C and sadly, thanks to its populism and political appeal, is becoming popular among masses of coding monkeys who have no idea about technology.
- **It is completely unnecessary** in the first place, a good low level language already exists: C. If anything, we need to improve C by making it simpler, but overcomplicated C also already exists for those who for some reason want to use a shit language: C++, and there are of course also slow and safe language well suited for beginners, such as Python. Rust is just flushing thousands of manhours

- Also "rust", what a shitty name lol. Who comes up with these idiotic names? Why do modern languages and libraries have to be called these shitty things like banana, unicorn hamburger, kangaroo space shuttle etc.?
- TODO: MORE, but you get the idea

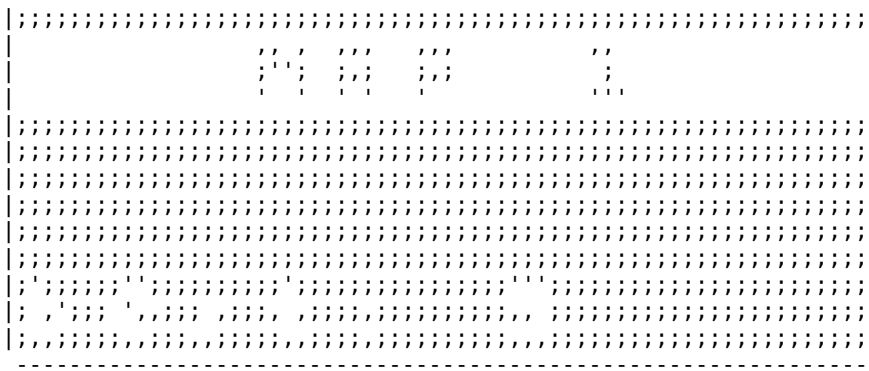
# SAF

The whole SAF library is implemented in a [single header](#) file (currently a bit under 5000 [lines of code](#), which is mostly due to including all the [boilerplate](#) for all possible platforms it supports, the actual SAF logic is pretty small) and offers a simple [API](#) for programming games, i.e. functions for drawing pixels to the screen, playing sounds, reading buttons, computing the [sine](#) function etc., and it handles boring/annoying issues such as the game main loop. It also has built-in a number of [frontends](#) that allow compiling a correctly made SAF games to any supported platform among which are [SDL](#), [SFML](#), [X11](#), [ncurses](#) (terminal), even [open consoles](#) such as [Pokitto](#), [Arduboy](#) or [Gamebuino META](#). There is an option to auto-convert a color game to black-and-white only displays. Some PC frontends add emulator-like features such as time manipulation, [pixelart upscaling](#), [TAS](#) support and so on.

Games made with SAF run in 64x64 resolution, with 256 colors (332 palette) and 25 FPS. They can use 7 input buttons (arrows, A, B and C), play 4 different sound effects and use 32 bytes of persistent memory, e.g. for savegames or highscores. These relatively low specifications are set on purpose so as to help portability, reduce bloat, frustration and friction. Many times good, entertaining games can be very simple, as is the case e.g. with Tetris, chess, various shmups, roguelikes and so on -- these are the kinds of games SAF is ideal for.

```
[ ] [ ][ ][ ][ ]
[ ][ ][ ]      [ ][ ]
[ ][ ]          [ ]
[ ]      XX      XX[ ]
[ ]          XXXX [ ]
[ ][ ]        [ ]
[ ][ ][ ]    [ ][ ]
[ ]   [ ][ ][ ][ ]
```

microTD



screenshot of a SAF game (uTD) running in terminal with ncurses

## See Also

- uxn

---

sanism

## Sanism

Sanism is an absolutely crazy idea made up by the most insane SIWs that says that words like "crazy" and "insane" are "offensive" or even "discriminatory" against mentally ill people and that we should censor such words so as to stay politically correct. Yes, this is pretty fucked up, but give it a year or two and it's gonna become mainstream.

LMAO imagine future news be like "Mentally divergent age fluid human people person of unspecified non-hexadecimal gender and afro american ethno-social-construct was arrested after an incident involving guns and liquor stores. No harm was intended during saying this sentence and we apologize in advance for any mental harm that may have been caused to mentally sensitive people persons by hearing this sentence."

---

science

## Science

*Not to be confused with soyence.*

Science (from Latin *scientia*, knowledge or understanding) in a wide sense means systematic gathering, inference and organization of knowledge, in a more strict, "western" sense this process has to be kept rational by obeying some specific strict rules and adhering to whatever principles of objectivity are currently set: nowadays for example the scientific method or mathematical proof. Sciences in the strict sense include mathematics (so called formal science), physics, biology, chemistry, computer science, as well as "soft sciences" such as psychology, sociology etc. **The beauty of science is you don't have to trust anyone:** science is just about discovering ideas that work, ideas which anyone can test himself that work, so there is no place for preachers, reviewers, judges of "trustworthiness" or "credibility"; people do not matter at all in science, only ideas do. For this science is not to be confused not only with pseudoscience (such as numerology or astrology) but especially with soyence (political propaganda masked as "science", e.g. gender studies, sponsored "science" of big pharma etc.) -- it must be remembered that **when science can no longer be questioned, it seizes to be science**, as asking questions and examining EVERYTHING are the very basic premises of a true science: this means that anything prohibited to be questioned, by law or otherwise (e.g. by cancel culture), such as the Holocaust (forbidden to be denied in many countries such as Germany), COVID vaccines, racial differences (prohibited on grounds of "hate speech") and similar topics CANNOT be seen as scientifically established, but rather politically established. Any shift towards establishing principles of trust, belief and "moderation", such as nowadays standard peer censorship, turns science into religion. In the wider sense science may include anything that involves systematic intellectual research, e.g.

Buddhists often say their teaching is science rather than religion, that it is searching for objective truths, and it really is true -- a western fedora atheist will shit himself in rage hearing such claim, however that's all he can really do.

TODO: some noise tree of sciences or smth

**There is no simple objective definition of a strict science** -- the definition of science is hugely arbitrary, political and changes with development of society, technology, culture, changes in government and so on. Science should basically stand for the most rational and objective knowledge we're able to practically obtain about something, however the specific criteria for this are unclear and have to be agreed on. The scientific method is evolving and there are many debates over it, with some even stating that there can be no universal method of science. The p-value used to determine whether measurements are statistically significant has basically just an arbitrarily set value for what's considered a "safe enough" result. Some say that if a research is to be trusted, it has to be peer reviewed, i.e. that what's scientific has to be approved by chosen experts -- this may be not just because people can make mistakes but also because in current highly competitive society there appears science bloat, obscurity and tendencies to push fake research and purposeful deception, i.e. our politics and culture are already defining what science is. However the stricter the criteria for science, the more monopolized, centralized, controlled and censored it becomes.

**Science is not almighty** as brainwashed internet euphoric kids like to think, that's a completely false idea fed to them by the overlords who abuse "science" (soyence) for control of the masses, as religion was and is still used -- soyence is the new religion nowadays. Yes, (true) science is great, it is an awesome tool, but it is just that -- a tool, usable for SOME tasks, not a silver bullet that could be used for everything. What can be discovered by science is in fact quite limited, exactly because it purposefully LIMITS itself only to accept what CAN be proven and so remains silent about everything else (which however doesn't mean there lies no knowledge or value in the everything else or in other approaches to learning) -- see e.g. Godel's incompleteness theorems that state it is mathematically impossible to really prove validity of mathematics, or the nice compendium of all knowability limitations at <http://humanknowledge.net/Thoughts.html>. For many (if not most) things we deal in life science is either highly impractical (do you need to fund a peer reviewed research to decide what movie you'll watch today?) or absolutely useless (setting one's meaning of life, establishing one's basic moral values, placing completely random bets, deciding to trust or distrust someone while lacking scientifically relevant indicators for either, answering metaphysical questions such as "Why is there ultimately something rather than nothing?", anything that cannot be falsified, if only for practical reasons etc.). So don't be Neil de Grass puppet and stop treating science as your omnipotent pimplord, it's just a hammer useful for bashing some specific nails.

**What should we accept as "legit" science?** We, in the context of our ideal society, argue for NOT creating a strict definition of science, just as we are for example against "formalizing morality" with laws etc. There are no hard lines between good and evil, fun and boring, useful and useless, bloated and minimal, and so also there is no strict line between science and non-science. What is and is not science is to be judged on a case-by-case basis and can be disagreed on without any issue, science cannot be a mass produced stream of papers that can automatically be marked OK or NOT OK. We might define the term **less retarded science** so as to distinguish today's many times twisted and corrupted "science/soyence" from the real, good and truly useful science. Less retarded science should follow similar principles as our technology, it should be completely free as in freedom, selfless, suckless as much as possible, unobscured etc. -- especially stressed should be the idea of many people being able to reproduce less retarded science; e.g. Newton's law of gravitation is less retarded because it can easily be verified by anyone, while the existence of Higgs boson is not.

**Never confuse trusting science with trusting scientists** (especially in capitalism and other dystopias), the latter is literally faith (soyence), no different from blindly trusting religious preachers and political propaganda, the former means only trusting that which you yourself can test and verify at home and therefore having real confidence. We are not saying that you should never trust a scientist, only that you should know doing so is just pure relying on someone's word, which in today's society you often cannot afford to do. Also do NOT confuse or equate science with academia. As with everything, under capitalism academia has become rotten to the core, research is motivated by profit and what's produced is mostly utter bullshit shat out by wannabe PhDs who need to mass produce "something" as a part of the crazy academia publish-or-perish game. As with everything in capitalism, the closer you look, the more corruption you find. So wait, **can we just trust nothing researched by someone else?** It's not so simple: for starters just realize that trusting "the big science" nowadays with anything important (e.g. one's health) is just like



entrusting a random stranger in the street something that's valuable to you (actually it's worse because unlike a stranger, entities such as corporations have absolutely no emotion and conscience) -- can you do that? Well, sometimes yes, mostly it's probably a great risk, and generally you want to avoid having to do it. In the past things were better, so you can generally trust "science" that was done much further in the past, i.e. facts you find in old encyclopedias are generally more trustworthy than facts you find on today's internet. LRS would like to establish society in which "big science" would be trustworthy again; until we succeed though, you have to keep distrust in soyence.

## See Also

- knowability
  - logic
  - academia
- 

sdf

## Signed Distance Function

Signed distance function (SDF, also signed distance field) is a function that for any point in space returns its distance to the closest point of some geometric shape, and also the information whether that point is outside or inside that shape (if inside, the distance is negative, outside it's positive and exactly on the surface it is zero -- hence *signed* distance function). SDFs find use in elegantly representing some surfaces and solving some problems, most notably in computer graphics, e.g. for smoothly rendering upscaled fonts, in raymarching, implementing global illumination, as a 3D model storage representation, for collision detection etc. SDFs can exist anywhere where distances exist, i.e. in 2D, 3D, even non-Euclidean spaces etc. (and also note the distance doesn't always have to be Euclidean distance, it can be anything satisfying the axioms of a distance metric, e.g. taxicab distance).

Sometimes SDF is extended to also return additional information, e.g. the whole vector to the closest surface point (i.e. not only the distance but also a direction towards it) which may be useful for specific algorithms.

**What is it all good for?** We can for example implement quite fast raytracing-like rendering of environments for which we have a fast SDF. While traditional raytracing has to somehow test each ray for potential intersection against all 3D elements in the scene, which can be slow (and complicated), with SDF we can perform so called raymarching, i.e. iteratively stepping along the ray according to the distance function (which hints us on how big of a step we can make so that we can potentially quickly jump over big empty areas) until we get close enough to a surface which we interpret as an intersection -- if the SDF is fast, this approach may be pretty efficient (Godot implemented this algorithm to render real-time global illumination and reflections even in GPUs that don't support accelerated raytracing). Programs for rendering 3D fractals (such as Mandelbulber) work on this principle as well. SDFs can also be used as a format for representing shapes such as fonts -- there exists a method (called multi-channel SDF) that stores font glyphs in bitmaps of quite low-resolution that can be rendered at arbitrary scale in a quality almost matching that of the traditional vector font representation -- the advantage over the traditional vector format is obviously greater simplicity and better compatibility with GPU hardware that's optimized for storing and handling bitmaps. Furthermore we can trivially increase or decrease weight (boldness) of a font represented by SDFs simply by adjusting the rendering distance threshold. SDFs can also be used for collision detection and many other things. One advantage of using SDFs is their **generality** -- if we have an SDF raymarching algorithm, we can plug in any shape and environment just by constructing its SDF, while with traditional raytracing we normally have to write many specialized algorithms for detecting intersections of rays with different kinds of shapes, i.e. we have many special cases to handle.

**How is an SDF implemented?** Well, it's a function, it can be implemented however we wish and need, it depends on each case, but we probably want it to be **fast** because algorithms that work with SDFs commonly call it often. SDF of simple mathematical shapes (and their possible combinations such as unions, see. e.g. CSG), e.g. spheres, can be implemented very easily (SDF of a sphere = distance to the sphere center minus its radius); even the already mentioned 3D fractals have functions that can be used to quickly estimate the distance towards their surface. Other times -- e.g. where arbitrary shapes may appear -- the function may be precomputed into some kind of N dimensional array, we might say we use a precomputed look up table. This can be done in a number of ways, but as a simple example we can imagine raymarching

mirror reflections with which we can subdivide the 3D scene into a grid and into each cell we store the SDF value at its center point (which here may be computed by even a relatively slow algorithm), which will allow for relatively fast search of intersections of rays with the surface (at any point along the ray we may check the SDF value of the current cell which will likely provide information for how big a step we can make next).

|   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|---|
| . | . | . | . | . | . | . | . | 3 | 2 | 2 | 2  | 2  | 2  | 2 |
| . | . | . | . | . | . | . | . | 3 | 2 | 1 | 1  | 1  | 1  | 1 |
| . | . | . | X | X | X | X | X | 2 | 2 | 1 | 0  | 0  | 0  | 0 |
| . | . | . | X | X | X | X | X | 2 | 1 | 1 | 0  | -1 | -1 | 0 |
| . | . | X | X | X | X | X | X | 2 | 1 | 0 | 0  | -1 | -2 | 0 |
| . | . | X | X | X | X | X | X | 2 | 1 | 0 | -1 | -1 | -2 | 0 |
| . | . | X | X | X | X | X | X | 2 | 1 | 0 | -1 | -1 | -1 | 0 |
| . | . | X | X | X | X | X | X | 2 | 1 | 0 | 0  | 0  | -1 | 0 |
| . | . | . | . | X | X | X | X | 2 | 1 | 1 | 0  | 0  | 0  | 0 |
| . | . | . | . | . | . | . | . | 2 | 2 | 2 | 1  | 1  | 1  | 1 |

*Shape (left) and its SDF (right, distances rounded to integers).*

SDFs in computer graphics were being explored a long time ago but seem to have start to become popular since around the year 2000 when Frisken et al used adaptive SDFs as an efficient representation for 3D models preserving fine details. In 2007 Valve published a paper at SIGGRAPH showing the bitmap representation of SDF shapes that they integrated into their Source engine.

---

security

## Security

*"As passwords first appeared at the MIT AI Lab I decided to follow my belief that there should be no passwords... I don't believe it's desirable to have security on a computer." -- Richard Stallman (from the book *Free As In Freedom*)*

Computer security (also cybersecurity) is a bullshit field of study of designing computer systems so as to make them hard to "attack" (which usually means accessing "sensitive" information, manipulating it or destabilizing the system itself). At the dawn of computer era security wasn't such a big deal as society wasn't yet so fucked up and didn't depend on computers so much, the damage one could cause by exploiting computers was limited, however once consumer technology became forced by capitalism, internetworked and put into EVERYTHING -- companies, governments, streets, homes, clothes, even human bodies and things that can work better without such technology (see e.g. Internet of stinks) -- privacy became another bullshit issue of society as cracking now theoretically allows not only killing individuals but wiping whole countries off the map; however despite computer networks really being vulnerable, capitalists additionally smelled a new "business opportunity" and started adding fuel to the fire, skyrocketing fear and paranoia. Recently security has really become a lot about ensuring digital "privacy", it's really causing an incredible deals of hysteria. Everyone is obsessed with security nowadays, sadly even most of the smart guys have fallen victims to it -- cartels and corporations have erected their "security" businesses that are now destroying everything, all the antiviruses, cloudflares, captchas, https certificate markets, VPNs, VMs and sandboxes, enormously bloated encrypted "secure networks", password managers and all kinds of other unbelievable idiocy -- and of course, most of it is just security theater.

**Security is in its essence a huge, completely unnecessary bullshit.** It shouldn't exist, the need for more security comes from the fact we live in a shitty dystopia. Consider that there are so many people who could cure cancer or solve world hunger but instead spend their whole life sweating about how to encrypt your dick size two million times and send it through twelve proxies so that no one gets to know your actual dick size, preventing your "PERSONAL DAATAAAZ" from being STOLEN by other people who are actively supported and empowered to do so by people this very "security expert with a furry stickers on his laptop" voted for in elections :D Nothing gets achieved, lives of geniuses are wasted on their lifelong fights just so that they are busy not doing much damage they've been taught by the competitive culture to instinctively want to do. In a good society there would be no need for security and people could spend their time by solving real problems. We, LRS, advocate NOT for increasing security (which leads to things like police states, censorship, blat etc.), but for decreasing the need for it, i.e. steering society towards a better direction. Remember, **secrets are always bad**, need for secrecy is an indicator something is wrong on a more fundamental level.

**If you want true security, the most basic thing to do is to disconnect from the Internet.** Just never use it. The next step is to leave the society and go live in a jungle, because due to spy devices everywhere they'll know about you even if you're not on the Internet. Additionally if you are really serious, you HAVE TO stop using a widely used human languages such as English or Chinese, you simply have to **learn an obscure language** of some jungle tribe that no one else knows and use that exclusively for your encrypted messages (like they did with the Navajo language during WWII). All in all for maximum security it's best if you never do anything at all, just lie in bed and be safe.

## See Also

- privacy
- bullshit
- stuxnet
- obscurity

---

see\_through\_clothes

## See Through Clothes

TODO: tech for seeing through clothes

---

selflessness

## Selflessness

Selflessness means acting with the intent of helping others without harming them, gaining edge over them or taking advantage of them in any way. It is the opposite of self interest. Selflessness is the basis of an ideal society and good technology (while sadly self interest is the basis of our current dystopian capitalist society).

Selflessness is about the **intent** behind behavior rather than about the result of the behavior; for example being a vegetarian (or even vegan) for ethical reasons (to spare animals of suffering) is selfless while being a vegetarian only because of one's health concerns is not selfless. Similarly if a selfless behavior unpredictably results in harming someone, it is still a selfless behavior as long as the intent behind it was pure. (Note that this does **NOT** at all advocate the "ends justify the means" philosophy which acts with an intent to hurt someone.)

In the real world absolutely pure selflessness may be very hard to find, partly because such behavior by definition seeks no recognition. Acts of sacrificing one's life for another may a lot of times be seen as selfless, but not always (saving one's child in such way may just serve perpetuating own genes, it can also be done to posthumously increase one's fame etc.). An example of high selflessness may perhaps be so called Langar, a big community kitchen run by Sikhs that prepare and serve free vegetarian food to anyone who comes without differentiating between religious beliefs, skin color, social status, gender etc. Sikhs sometimes also similarly offer a place to stay etc. The mentioned ethical vegetarianism and veganism is another example of selflessness, as well as LRS itself, of course.

**Selflessness doesn't mean one seeks no reward**, there is practically always at least one reward for a selflessly behaving individual: the good feeling that comes from the selfless action. Selfless acting may also include physical rewards, for example if a programmer dedicates years of his life to developing a free public domain software that will help all people, he himself will get the benefits of using that program. The key thing is that he doesn't use the program to harm others, e.g. by charging money for it or even by using a license that forces others to credit him and so increase his reputation. He sacrificed part of his life purely to increase good in the world for everyone without trying to gain an edge over others.

The latter is important to show that what's many times called selflessness nowadays is only **pseudoselflessness**, fake selflessness. This includes e.g. all the celebrities who publicly financially support charities; this seems like a nice gesture but it's of course just a PR stunt, the money spent on charities is money invested into promoting oneself, increasing fame, sometimes even tax hacking etc. This also goes for professional firefighters, doctors, FOSS programmers that use licenses with conditions such as attribution

etc. This is not saying the behavior of such people is always pure evil, just that it's not really selfless.

Selfless programs and art should be put into the public domain with waivers such as CC0. Using licenses (free or not) that give the programmer some advantage over others (even e.g. attribution) are not selfless.

## See Also

- altruism
- dog

---

semiconductor

## Semiconductor

{ For a physicist and electronics guys there's probably quite a lot of simplification, this is written from the limited point of view of a programmer. ~drummyfish }

Semiconductors are materials with electric conductivity between insulators and conductors, their conductivity may vary greatly with conditions such as temperature, illumination, their purity or applied voltage. In this they're unlike insulators who generally don't conduct electricity very well (have a great resistivity) and conductors who do. Semiconductors, especially silicon (Si), are the key component of digital electronic computers and integrated circuits. Other semiconductors include germanium, selenium or compound ones (composed of multiple elements).

Semiconductors are important for computers because they help implement the binary logic circuits, they can behave like a switch that is either on (1) or off (0). Besides that they can serve e.g. for making measurements (a component whose resistivity depends on its illumination can be used to measure amount of light by measuring the resistivity). Especially important electronic components based on semiconductors are the diode (lets current flow only one way) and transistor (a purely electrical "switch" that can be made extremely tiny).

Semiconductor elements are those with some special properties related to their atomic structure -- in atoms there are so called *orbitals*, certain discrete "levels" at which electrons "orbit" around the nucleus (distribution of electrons in these orbitals is given by what kind of atom it is, i.e. how many electrons there are in total) and when atoms are together (in a solid), these orbitals create kind of energy bands that electrons can occupy. The outermost energy band that has electrons is called the *valence band*, the band immediately above it (which is normally empty) is called the *conduction band*. Conductor elements are those that don't have the valence band completely filled, so that there is space for electrons to move around and conduct electricity. Insulators on the other hand have the valence band completely filled up, so those electrons can't move easily -- they don't conduct electricity well. Similarly semiconductors in their normal state have the valence band filled, but unlike with insulators the gap between the valence band and conduction band is quite small, so electrons can relatively easily jump from valence band to conduction band and move to conduct electricity -- still semiconductors don't conduct too well at room temperature (they conduct better at higher temperatures, unlike metals), but the small energy gap between the upper bands allows us to increase their conductivity by so called **doping** -- introducing small impurities of other elements. Of course increasing their conductivity in itself wouldn't achieve much as we still have materials that conduct well by themselves (the conductor elements), but as we'll see, doping leads to creating two possible types of semiconductors that when combined give rise to some extremely useful things. The two types of semiconductors are:

- **N type**: By adding e.g. an arsenic atom to a grid of silicon atoms we get an extra free electron because arsenic has 5 outer electrons while silicon has 4. Arsenic binds its 4 electrons with neighboring silicon atoms (with so called covalent bond, sharing pairs of electrons), but will have one unbound electron, which thanks to not being bound can move easily and help conduct electricity (note that when the electron does move away, arsenic will still want to get another electron because it will become positive charged, but the fifth electron is simply always more "loose" thanks to not being bound). In this type of semiconductor we have negative particles, the **electrons**, moving around.

- **P type:** By similarly adding e.g. an boron atom to a silicon grid, we get a similar situation; boron only has 3 outer electrons, so it will only bind to 3 silicon atoms, leaving one the bond with one silicon incomplete, forming a **hole**. This hole can be seen as a virtual positively charged particle, it is simply a lack of electron. An electron can temporarily jump in this hole but will want to move away as boron doesn't want it there. In this type of semiconductor we can imagine the holes moving around (even though physically only electrons are moving, of course).

If we connect a P and N type semiconductors, we get so called **PN junction** which only conducts current one way and is used in diodes and transistors. After putting P and N materials together, at the boundary some electrons from the N type material fill the holes in the P type material which creates a small *depletion region* of certain width. This region is an electric field that's negative on the P side and positive on the N side (because negative electrons have moved from N to P). If we connect the PN junction to a voltage source, with P side to the positive and N side to the negative terminal, we create an opposite electric field which will eliminate the depletion region and allow the flow of current. Connecting the sides the other way around will result in increasing the width of the depletion region and blocking the current flow.

---

settled

## Settled

When a software is so good it makes you stop searching further for any other software in the same category, it makes the debate settled -- you've found the ultimate tool, you're now free, you no longer have to think about the topic or invest any more precious time into trying different alternatives. You've found the ultimate, nearly perfect tool for the job, the one that makes the others obsolete. Settling the competition of different tools is one of the goal of good technology as it frees users as well as programmers.

For example Vim often settles the search for a text/programming editor for many programmers (yes, for others it's Emacs). It is one's own personal "category killer" (a term used by ESR in *Cathedral and Bazaar*).

Nevertheless some soyboys just like hopping and switching their tools just because, they simply like wasting their time with things like distrohopping etc. There's no help to these people.

{ software that made the debate settled for me: vim, dwm, st, badwolf ~drummyfish }

---

shader

## Shader

Shader is a program running on the graphics processing unit (GPU), typically in many parallel instances as to utilize the GPU's highly parallel nature. As such they are simple to mid complexity programs. The word *shader* is also used more loosely to stand for any specific effect, material or look in 3D graphics (e.g. games), as shaders are usually the means of achieving such effects.

Shaders are normally written in a special **shading language** such as GLSL in the OpenGL API, HLSL (proprietary) in Direct3D API or the Metal shading language (proprietary) in Metal API. These languages are often similar to C with some additions (e.g. vector and matrix data types) and simplifications (e.g. no function recursion). High level frameworks like Blender many times offer visual programming (point-n-click) of shaders with graph/node editors.

Initially (basically early 2000s) shaders were used only for graphics, i.e. to transform 3D vertices, draw triangles and compute pixel colors. Later on as GPUs became more general purpose (GPGPU), flexibility was added to shaders that allowed to solve more problems with the GPU and eventually general *compute* shaders appeared (OpenGL added them in version 3.3 in 2010).

To put shaders in the context, the flow of data is this: a CPU uploads some data (3D models, textures, ...) to the GPU and then issues a draw command -- this makes the GPU start its **pipeline** consisting of different **stages**, e.g. the vertices of 3D models are transformed to screens space (the vertex stage), then triangles are generated and rasterized (the shading stage) and the data is output (on screen, to a buffer etc.). Some of

these stages are programmable and so they have their own type of a shader. The details of the pipeline differ from API to API, but in general, depending on the type of data the shader processes (the stage), we talk about:

- **vertex shaders**: Perform per-vertex computations on 3D models, typically their transformation from their world position to the position in the camera and screen space.
  - **fragment/pixel shaders**: Compute the final color of each pixel (sometimes called more generally *fragments*), i.e. work per-pixel. A typical use is to perform texturing and amount of reflected light (lighting model).
  - **geometry shaders**: Advanced stage, serves to modify the geometry of 3D models (these shaders can, unlike vertex shaders, generate or discard vertices).
  - **tessellation shaders**: Advanced stage, serves for subdividing primitives to create smoother geometry.
  - **compute shaders**: Perform general computations, used e.g. for computing physics simulations, AI etc.
  - **ray tracing shaders** and other specific types
- 

shit

## Shit

Shit is something that's awfully bad.

Unicode for pile of shit is U+1F4A9.

Some **things that are shit** include systemd, capitalism, Feminism, Windows, Linux, Plan9, OOP, LGBT, security, encryption, military, laws, liberalism, USA, money, cryptocurrencies and many more.

## See Also

- crap
  - cancer
  - harmful
  - bullshit
- 

shogi

## Shogi

Shogi, also called *Japanese chess*, is an old Asian board game, very similar to chess, and is greatly popular in Japan, even a bit more than go, the second biggest Japanese board game. Shogi is yet more complex (and bloated) than chess, has a bigger board, more pieces and more complex rules that besides others allow pieces to come back to play; for a chess player shogi is not that hard to get into as the basic rules are still very similar, and it may offer a new challenge and experience. Also similarly to chess, go, backgammon and similar board games, LRS sees shogi as one of the best games ever as it is legally not owned by anyone (it is public domain), is relatively simple, cheap and doesn't even require a computer to be played. The culture of shogi is also different from that of chess, there are many rituals connected to how the game is conducted, there are multiple champion titles, it is not common to offer draws etc.

{ Lol apparently (seen in a YT video) when in the opening one exchanges bishops, it is considered rude to promote the bishop that takes, as it makes no difference because he will be immediately taken anyway. So ALWAYS DO THIS to piss off your opponent and increase your chance of winning :D ~drummyfish }

**Quick sum up for chess players:** Games are longer. When you get back to chess from shogi your ELO will bump 100 points as it feels so much easier. Pawns are very different (simpler) from chess, they don't take sideways so forget all you know about pawn structure (prepare for bashing your head thinking a pawn guards something, then opponent takes it and you realize you can't retake :D just write gg and start a new game). The drop move will fuck up your brain initially, you have to start considering that opponent can just smash

his general literally in front of your king and mate you right there { still fucking happens to me all the time lol :D ~drummyfish }. Exchanges and sacrifices also aren't that simple as any piece you sacrifice YOU GIVE TO THE OPPONENT, so you better not fuck up the final attack on the king or else the opponent just collects a bunch of your pieces and starts his own attack right in your base by dropping those pieces on your king right from the sky. You have to kill swiftly and precisely, it can turn over in an instant. There is no castling (but king safety is still important so you castle manually). Stalemate is a loss (not a draw) but it basically never happens, Japanese hate draws, draws are rare in shogi.

The game's disadvantage and a barrier for entry, especially for westerners, is that the **traditional design of the shogi pieces sucks big time**, for they are just same-colored pieces of wood with Chinese characters written on them which are unintelligible to anyone non-Chinese and even to Chinese this is greatly visually unclear -- all pieces just look the same on first sight and the pieces of both player are distinguished just by their rotation, not color (color is only used in amateur sets to distinguish normal and promoted pieces). But of course you may use different, visually better pieces, which is also an option in many shogi programs -- a popular choice nowadays are so called *international* pieces that show both the Chinese character along with a simple, easily distinguishable piece symbol. There are also sets for children/beginners that have on them visually indicated how the piece moves.

## Rules

As with every game, rules may slightly differ here and there, but generally they are in principle similar to those of chess, with some differences and with different pieces. The **goal** of the game is to deliver a checkmate to the opponent's king, i.e. make him unable to escape capture (same as in chess). The details are as follows.

Shogi is played on a 9x9 rectangular board: the squares are not square in shape but slightly rectangular and they all have the same color. There are two players: **sente** (plays first) and **gote** (plays second); sente is also sometimes called black and gote white, like in chess (though unlike in chess black starts first here), but the pieces actually all have the same color (as they can be exchanged).

The pieces are weird pentagonal arrow-like shapes that have on them written a Chinese character identifying the piece; on the other side there is a symbol representing the promoted version of the piece (i.e. if you promote, you turn the piece over). The arrow of the piece is turned against the enemy and this is how it is distinguished which player a piece belongs to.

The table showing all the types of pieces follows. The movement rules are same as in chess, i.e. pieces cannot jump over other pieces except for the knight. (F, R, B, L mean forward, right, bottom, left.)

| piece          | symbol | letter | ~value | move rules                  | comment                                                             |
|----------------|--------|--------|--------|-----------------------------|---------------------------------------------------------------------|
| pawn           | æ—☉    | P      | 1      | 1 F                         | also takes forward (not complicated like in chess)                  |
| lance          | é!     | L      | 4      | F (any distance)            | can't go backwards or sideways, just forward!                       |
| knight         | æi     | N      | 5      | 2 F., then 1 L or R         | similar to knight in chess, only one that jumps over pieces         |
| silver general | é      | S      | 7      | 1F1L, 1F, 1F1R, 1B1L, 1B1R  | like king but can't go directly back, left or right                 |
| gold general   | é      | G      | 8      | 1F1L, 1F, 1F1R, 1L, 1R, 1B  | similar to silver but has 6 squares (s. only has 5), can't promote  |
| bishop         | è§     | B      | 11     | diagonal (any distance)     | same as bishop in chess                                             |
| rook           | é£     | R      | 13     | horiz./vert. (any distance) | same as rook in chess                                               |
| promoted pawn  | ã ¨    | +P     | 10     | like gold general           | more valuable than gold because when captured, enemy only gets pawn |
| promoted lance | æ      | +L     | 9      | like gold general           |                                                                     |
|                | å —    | +N     | 9      | like gold general           |                                                                     |

| piece            | symbol | letter | ~value | move rules                | comment                                 |
|------------------|--------|--------|--------|---------------------------|-----------------------------------------|
| promoted knight  |        |        |        |                           |                                         |
| promoted silver  | ♁      | ..     | +S 9   | like gold general         |                                         |
| p. bishop        | ♁↔     | +B 15  |        | like both king and bishop | can now move to other set of diagonals! |
| p. rook (dragon) | ♁¾     | +R 17  |        | like both king and rook   |                                         |
| king             | ♁      | K inf  |        | any neighboring 8 squares | same as king in chess, can't promote    |

At the beginning the board is set up like this:

9 8 7 6 5 4 3 2 1

|                   |   |         |           |
|-------------------|---|---------|-----------|
| L N S G K G S N L | a | gote    | promotion |
| . R . . . . B .   | b | (white) | zone      |
| P P P P P P P P   | c | -----   |           |
| . . . . . . . .   | d |         |           |
| . . . . . . . .   | e |         |           |
| . . . . . . . .   | f |         |           |
| p p p p p p p p   | g | -----   |           |
| . b . . . . r .   | h | sente   | promotion |
| l n s g k g s n l | i | (black) | zone      |
| .....             |   |         |           |

So called *furigoma* is used to decide who starts (has the black pieces): one player throws 5 pawn pieces, if the number of unpromoted pawns ending up facing up is higher than the number of promoted ones, the player who tossed starts.

Then the players take turns in making moves, one can either:

- Move one own piece (according to its movement rules), possibly **capturing** (taking) one enemy piece by landing on it. If a piece is captured, it goes to the hand of that who captured it. After making this move the moved piece MAY (or may not) be promoted if: it can be **promoted** (as some pieces can't promote) AND its movement path went through the enemy promotion zone (his three starting rows, for example if the piece entered the zone or left it or moved within it). Promotion is mandatory if otherwise the piece would be unable to ever move again (e.g. a pawn entering the last row has to be always promoted). OR
- **Drop** one of the held pieces (captured from the enemy's army) anywhere on an empty square as a new piece of the player who drops it. After a drop the piece CANNOT be immediately promoted (it has to move to be promoted). One only cannot drop a piece so that it wouldn't ever have any legal move (e.g. a pawn to the last row), also a pawn mustn't be dropped into a column where there already is an unpromoted friendly pawn (there may only ever be at most one unpromoted pawn of each player in any column) AND one mustn't deliver an immediate checkmate by dropping a pawn (but can deliver a check etc.).

If a piece is immediately endangering the enemy king (so that it could capture it the next turn), a **check** happens. The player in check has to immediately avoid it, i.e. make a move that makes his king not be endangered by any enemy piece. If he cannot do that, he got **checkmated** and lost.

TODO

## Playing Tips

TODO

- Opening: moving the pawn right-up from your bishop seems to be the best first move, also most commonly played on top level.
- ...



## See Also

- [chess](#)
  - [go](#)
  - [xiangqi](#)
  - [backgammon](#)
- 

shortcut\_thinking

## Shortcut Thinking

Shortcut thinking means making conclusions by established associations (such as "theft = bad") rather than making the extra effort of inferring actual conclusions based on new context such as different circumstances or newly discovered facts. This isn't bad in itself, in fact it is a great and necessary optimization of our thinking process, a kind of cache, and it's really why we have long term memory -- imagine we'd have to deduce all the facts from scratch each time we thought about anything. However shortcut thinking can be a weakness in many situations and leaves people prone to manipulation by propaganda which twists meanings of words (such as "open mind", "rationality", "progress", "theft", "science" etc.), relying on people accepting the unacceptable by having them bypass the thinking process with the mental shortcut. As such this phenomenon is extremely abused by politicians, i.e. they for example try to shift the meaning of a certain negative word to include something they want to get rid of, to have it rejected just based on its name.

Some commonly held associations appearing in shortcut thinking of common people nowadays are for example "piracy = theft = bad", "laziness = bad", "pedophiles = child rapists = bad", "competition = progress = good", "more jobs = good", "more complex technology = better", "open mind = blindly trusting those officially declared smarter than myself = good" etc. Of those most are of course either extremely simplified or just plain wrong. however some association may still of course be correct, such as "murder = bad", which is an association that e.g. military tries to get rid of.

Let's focus on the specific example of the association "theft = bad". Indeed it has some sense in it -- if we turn shortcut thinking off, we may analyze why this association exists. For most of our history the word theft has meant *taking a physical personal possession of someone else against his will*. Indeed, in a society of people of which most weren't rich, this was bad in most cases as it hurt the robbed man, he has lost something he probably needed. However the society evolved, the meaning of property itself has changed from "personal property" to "private property", i.e. suddenly there were people who could own a whole forest or a factory even if they have never seen it, and there were people who had much more than they needed. If a poor starving man steals food from the rich to his family and the rich doesn't even notice this, suddenly the situation is different and many will say this is no longer bad. Nevertheless the word theft stayed in use and now included even such cases that were ethical because of the shifted meaning of the word "property" and due to changes in conditions of people. Recently the word property was shifted to the extreme with the invention of **intellectual property**, i.e. the concept of being able to own information such as ideas or stories in books. Intellectual property is fundamentally different from physical property as it can't be stolen in the same way, it can only be copied, duplicated, but this copying doesn't rid the "owner" of the original information. Indeed it may prevent the author from making a lot of money under capitalism, but that's only thanks to the artificially established system inventing ways of bullying recipients of information into paying money, the system was deliberately made so as to make non harmful things into harmful ones to fuel "competition" and the situation is no longer as simple as with physical property/stealing, it's actually the underlying system that's wrong here, not actions that aren't aligned with the harmful system. And so nowadays the word "theft", or one of its modern forms, "piracy", includes also mere copying of information or even just reusing an idea (patent) for completely good purposes, for example writing computer programs in certain (patented) ways is considered a theft, even if doing so is done purely to help other people at large. Those arguing that illegal downloads or reuses prevent the "owner's" profit must know that again society is completely different nowadays and this so called "theft" actually doesn't hurt anyone but some gigantic billion dollar corporation that doesn't even notice if loses one or two million dollars, no actual human gets hurt, only a legal entity, and these so called "thefts" actually give rise to good, helpful things by at least a little more balance to society, hurting a virtual, non-living entity to help millions of actual living people. In fact, hurting a corporation, by definition a fascist entity hostile to people, may yet further be seen as a good thing in itself, so stealing from corporation is also good by this view. The illusion of profit theft here is arbitrarily made, the "theft" exists only because we've purposefully created a system which allows selling

copies of information and restricting ideas and therefore enables this "theft", i.e. this is no longer a natural thing that would exist without something "preventing it", it's something miles away from the original meaning of the word "theft". With all this in mind we may, in today's context of the new meaning of old words, reconsider theft to no longer be generally bad.

When confronted with a new view, political theory etc., we should try to turn shortcut thinking off; we should also do this every time someone tries to have us make a decision under a competitive system such as capitalism as that someone is most likely trying to manipulate us. Doing this can be called **being open minded**, i.e. opening one's mind to reinterpretation of very basic, possibly strongly rooted concepts by a new view, however be careful as the meaning of the term "open mind" is often twisted too. Also we should probably update our association from time to time just to keep them up with the new state of the world, only by sitting down and thinking about the world and discussing it with the right people.

The politics and views of LRS requires extreme open mindedness to be accepted by someone indoctrinated by the standard capitalist fascist propaganda of today.

---

sigbovik

## SIGBOVIK

SIGBOVIK (special interest group on Harry Q. Bovik) is a computer science conference running since 2007 that focuses on researching and presenting fun ideas in fun ways, scientifically but in a lighthearted hacker spirit similar to e.g. esoteric programming languages research or the IOCCC. SIGBOVIK has its own proceedings just like other scientific conferences, the contributors are usually professional researchers and experts in computer science. The name seems to be a reference to the "serious" conferences such as SIGGRAPH, SIGMOD etc. (SIGBOVIK is organized by the *Association for Computational Heresy* while the "serious" SIGs are run by *Asscoiation for Computing Machinery*, ACM).

A famous contributor to the conference is for instance Tom7, a PhD who makes absolutely lovely youtube videos about his fun research (e.g. this one is excellent <https://www.youtube.com/watch?v=DpXy041BIIA>).

{ Skimming through the proceedings sadly most of the stuff seems rather silly, though there are a few good papers, usually those by Tom7. Maybe I'm just dumb. ~drummyfish }

## See Also

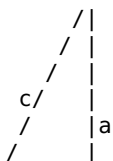
- ioccc
  - NaNoGenMo
- 

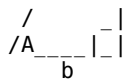
sin

## Sine

Sine, abbreviated *sin*, is a trigonometric function that simply said models a smooth oscillation, it is one of the most important and basic functions in geometry, mathematics and physics, and of course in programming. Along with cosine, tangent and cotangent it belongs to a group of functions that can be defined by ratios of sides of a right triangle depending on one of the angles in it (hence *trigonometric* -- "triangle measuring"). If some measurement looks like sine function, we say it is *harmonic*. This is very common in nature and technology, e.g. a weight on a spring goes up and down by this function, alternating current voltage has the sine shape (because it is generated by a circular motion) etc.

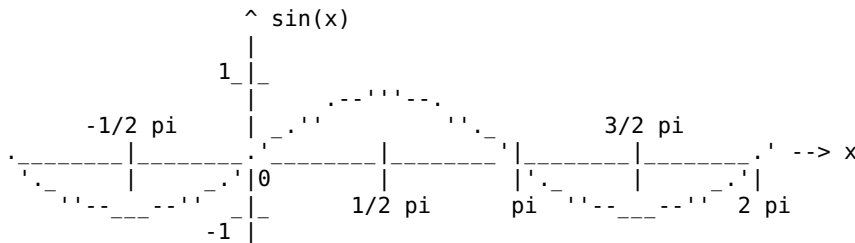
The function is most commonly defined using a right triangle as follows. Consider the following triangle:





$\sin(A)$ , where  $A$  is the angle between side  $b$  and  $c$ , is the ratio  $a / c$ . The function can be defined in many other ways, for example it is the curve we get when tracking only one direction (e.g. horizontal) of a point moving alongside circle. It can also be defined as a solution to some differential equations etc.

The graph of the sine function is following:



**Why the fuck are there these  $\pi$  values on the x line???** Nubs often can't comprehend this. These  $\pi$  values are values in radians, units of measuring angles where  $2\pi$  is the full angle (360 degrees). In fact sine is sometimes shown with degrees instead of radians (so imagine 90 degrees on the line where there is  $1/2\pi$  etc.), but mathematicians prefer radians. **But why are there angles in the first place???** Why doesn't it go e.g. from 0 to 1 like all other nice functions? Well, it's because of the relation to geometry, remember the fucking triangle above... also if you define sine with a circle it all repeats after  $2\pi$ . Just draw some picture if you don't get it.

Some additional facts and properties regarding the sine functions are:

- The domain are all real numbers, the codomain are real numbers in interval  $<-1,1>$  (including both bounds).
- It is an odd function ( $-\sin(x) = \sin(-x)$ ).
- It is periodic, with a period of  $2\pi$ .
- Sine is just shifted cosine, i.e.  $\sin(x) = \cos(x - 1/2\pi)$
- Its inverse function is arcus sine, abbreviated *asin*, also written as  $\sin^{-1}$  -- this function tells you what argument you need to give to sin to get a specific result number. It's actually an inverse of only part of the sine function because the whole sine function can't be inverted, it isn't bijective.
- Derivative of  $\sin(x)$  is  $\cos(x)$ , the integral of  $\sin(x)$   $dx$  is  $-\cos(x)$ .
- By adding many differently shifted and scaled sine functions we can create basically any other function, see e.g. cosine transform.
- Sine and cosine functions are used to draw circles. If you plot points with x coordinate equal to  $\sin(t)$  and y coordinate equal to  $\cos(t)$  for  $t$  going from 0 to  $2 * \pi$ , you'll get a unit circle.
- $\sin(x)^2 + \cos(x)^2 = 1$

Some values of the sine function are:

| x (rad)    | x (deg) | sin(x)                  |
|------------|---------|-------------------------|
| 0          | 0       | 0                       |
| $\pi / 12$ | 15      | $\sim 0.259$            |
| $\pi / 6$  | 30      | 0.5                     |
| $\pi / 4$  | 45      | $\sqrt{2}/2 \sim 0.707$ |
| $\pi / 3$  | 60      | $\sqrt{3}/2 \sim 0.866$ |
| $\pi / 2$  | 90      | 1                       |
| $2\pi$     | 360     | 0                       |

## Programming

In programming languages the sine function is generally available in some math library, for example in C the function `sin` is in `math.h`. Spare yourself bugs, **always check if your sin function expects radians or**

degrees!

**Want to make your own sine function for whatever reason (performance, curiosity, ...)?** Then firstly consider what you expect from it. If you want a small, fast and perhaps integer only `sin` function (the one we'd prefer in LRS) that doesn't need extreme accuracy, consider using a **look up table**. You simply precompute the values of the sine function into a static table in memory and the function just retrieves them when called -- this is super fast. Note that you can save a lot of space by **only storing sine values between 0 and 1/2 pi**, the remaining parts of the function are just different transformations of this part. You can further save space and/or make the function work with floats by further interpolating (even just linearly) between the stored values, for example if `sin(3.45)` is called and you only have values stored for `sin(3.4)` and `sin(3.5)`, you simply average them.

Lot of times, e.g. in many calculators where speed isn't really critical, sine is computed using Taylor series -- a sum of infinitely many terms of which if we take the first  $N$ , we get an approximation of the function (the more terms we add, the more precise we get). For sine the series is

$$\sin(x) = x - x^3 / 3! + x^5 / 5! - x^7 / 7! + \dots$$

Adding just the first 3 terms ( $x - x^3 / 6 + x^5 / 120$ ) already gives a very accurate approximation in range  $[-\pi/2, \pi/2]$  (error < 0.5 %). Here is a C function that uses this to compute an 8bit sine (the magic numbers are made so as to incorporate pi while using power of two divisors, also note the use of many operations that will make the function relatively slow):

```
// x = 255 means full angle, returns 0 to 255
unsigned char sin8(unsigned char x)
{
    int a = x;
    char flip = 0;

    if (a > 127)
    {
        a -= 128;
        flip = 1;
    }

    if (a > 63)
        a = 128 - a;

    int result = (411999 * a) - (a * a * a * 41);

    a /= 4;

    a = a * a * a * a * a * a;

    result = (a + result) / 131072;
    return flip ? (127 - result) : (127 + result);
}
```

If you just need a super fast and very rough sine-like value, there exists an **ugly engineering approximation** of sine that can be useful sometimes, it says that

$$\sin(x) = x, \text{ for small } x$$

Indeed, sine looks similar to a mere line near 0, but you can see it quickly diverges.

Very rough and fast approximations e.g. for primitive music synthesis can be done with the traditional very basic square or triangle functions. The following is a simple 8bit linear approximation that's more accurate than square or triangle (approximates sine with a linear function in each octant):

```
unsigned char sinA(unsigned char x)
{
    unsigned char quadrant = x / 64;

    x %= 64;
```

```

    if (quadrant % 2 == 1)
        x = 63 - x;

    x = x < 32 ? (2 * x + x) : (64 + x);

    return quadrant <= 1 ? (128 + x) : (127 - x);
}

```

Similar approximation can be made with a quadratic curve, the following is a modification of the above function that does this (notice that now we need at least 16 bits for the computation so the data type changed to int): { I quickly made this just now, maybe it can be improved. ~drummyfish }

```

int sinA(int x)
{
    unsigned char quadrant = x / 64;

    x %= 64;

    if (quadrant % 2 == 1)
        x = 63 - x;

    x -= 63;
    x = (x * x) / 32;

    return quadrant <= 1 ? (255 - x) : x;
}

```

Sine can also be surprisingly accurately approximated with the smoothstep function, which is just a polynomial  $3 * x^2 - 2 * x^3$ .

TODO: code for that

Furthermore there exist other nice approximations, such as the extremely accurate **Bhaskara I's approximation** (angle in radians):  $\sin(x) \sim (16 * x * (pi - x)) / (5 * pi^2 - 4 * x * (pi - x))$ . (This formula is actually more elegant for cosine, so it may be even better to consider using that.) Here is a C fixed point implementation:

```

#define UNIT 1024
#define PI ((int) (UNIT * 3.14159265))

/* Integer sine using Bhaskara's approx. Returns a number
in <-UNIT, UNIT> interval. Argument is in radians * UNIT. */

int sinInt(int x)
{
    int sign = 1;

    if (x < 0) // odd function
    {
        x *= -1;
        sign = -1;
    }

    x %= 2 * PI;

    if (x > PI)
    {
        x -= PI;
        sign *= -1;
    }

    x *= PI - x;

    return sign * (16 * x) / ((5 * PI * PI - 4 * x) / UNIT);
}

```

---

sjw

# Social Justice Warrior

Social justice warrior (SJW) is an especially active, toxic and aggressive kind of pseudoleftist (a kind of fascist) that tries to fight (nowadays mostly on the Internet but eventually also as a member of a physical execution squad) anyone opposing or even just slightly criticizing the mainstream pseudoleftist gospel such as the feminism and LGBT propaganda. SJWs divide people rather than unite them, they operate on the basis of hate, revenge and mass hysteria and as we know, hate spawns more hate and fear, they fuel a war mentality in society. They support hard censorship (forced political correctness) and bullying of their opposition, so called cancelling, and also such retardism as sanism and whatnot. Wokeism is yet more extreme form of SJWery that doesn't even anymore try to hide its militant ambitions.

SJWs say the term is pejorative. We say it's not pejorative enough xD

SJWs want to murder all straight white men, however they try to make it seem as though they tolerate all races and orientations by excluding from their death sentence (only for now) those straight white males that agree to help kill all straight white men than do not agree to do the same. Once they decimate the population of straight white men like this to a minimum, they will also kill off the rest. This works basically the same as how Nazi made some Jews collaborate on killing of other Jews by promising they would let them live; of course eventually Nazis aimed to exterminating all of them, but they figured they might just make it easier this way.

A sneaky tactic of an SJW is **masked hypocrisy**. As any good marketing guy he will proclaim some principle OUT LOUD IN BIG LETTERS, adding asterisks with exceptions that immediately break that principle. For example:

- "WE HAVE TO CREATE A CENSORSHIP RESISTANT INFORMATION NETWORK" (asterisk: "But we have to build in mechanisms to censor pedophiles and racists and other information we dislike.")
- "WE HAVE ZERO TOLERANCE OF CYBER BULLYING" (asterisk: "With the exception of bullying people we deem fair to be bullied.")
- "WE ARE PACIFIST REJECTING VIOLENCE" (asterisk: "With the exception of violence against people we deem good to use violence against.")
- "WE HAVE ZERO TOLERANCE OF RACISM" (asterisk: "We don't count reverse racism as racism.")
- "WE SUPPORT FREE SPEECH AND FREE INFORMATION SHARING" (asterisk: "Speech we dislike doesn't fall under free speech.")
- "WE SUPPORT TECHNOLOGICAL MINIMALISM" (asterisk: "As long as it's written in Rust, Python and JavaScript with encryption and virtual machines for security.")
- "WE PROMOTE SCIENCE" (asterisk: "Science being defined as trusting the word of authorities we approve without questioning them.")
- "THIS ENCYCLOPEDIA CAN BE EDITED BY EVERYONE" (asterisk: "Except for 90% of population who are blocked for not conforming to our political style of writing, also with the exception of articles that will probably be read by someone.")
- etc.

## See Also

- soydev
- idiot
- pseudoleftism

---

slowly\_boiling\_the\_frog

## Slowly Boiling The Frog

*Slowly boiling the frog* is a phrase said to communicate the idea that people will tolerate a great change for the worse if that change is very gradual, even if they would absolutely not tolerate this change being made quickly. It refers to an experiment in which a frog doesn't jump out of boiling water if the water temperature is raised very gradually (even though according to "modern science" this experiment isn't real).

For example the amount and aggressiveness of brainwashing ads and technological abuse that young people tolerate nowadays would have been absolutely unacceptable a few decades ago, but now it's the reality of life that few even question (some complete retards like that *linus tech faggot* even defend it).

The technique of slowly boiling the frog is used by corporations, governments, fascists and idiots to slowly take away people's freedom in small steps: each step takes away a bit of freedom while promising some reward, normally in form of additional comfort -- normal people are too braindead to see the obvious trick and are enthusiastic about the change. If you tell them that giving up net neutrality or P2P encryption will eventually lead to almost complete loss of freedom, they label you a tinfoil or "conspiracy theorist", they tell you that "it's not a big deal". So it will go on with other and other changes and the normie is still happy because he can only see one step ahead or behind. The bad thing is that it's not only the normie who will suffer --in fact he may even be happy as a slave robot of the system -- but you will suffer as well. Normies decide the future of the environment we all have to live in.

Slowly boiling the frog works very well when spanning several generations because a new generation won't remember that things used to be better. Parents can tell them but young never listen to older generations, or take them seriously. A zoomer won't remember that computers used to be better, he thinks that bloated phones filled with ads and DRM that don't work without Internet connection and that spy on you constantly are the only way of technology.

This can also be seen with all the subscriptions and service as software replacement in modern tech. Back in the 90s no one would buy a program he would have to keep periodically paying for, people saw that was stupid and everyone would tell you that no company can make subscription software because no one would pay subscriptions if he can just buy a competitor's program once and use it forever, people would just laugh at any company trying to do that; if back then you told anyone subscriptions would become the sole business model in technology, even e.g. for cars, they would literally put you in mental asylum, you would be labeled a retard and schizo, just like they are labeling us warning about the future. It took 1 to 2 generations to indeed make this schizo vision a reality. If you think something can't happen because it just sounds "schizo", you're a brainwashed retard.

Studies on caged people show that a 90s man can bear only as much as 3 ads per hour before killing himself^[1][2][3][4][5][6][7][8][9][10][11][12][13][14][15][16][17][18] while a zoomer kid can easily live with his senses being 97% occupied by ads all the time (even in sleep) -- in fact a zoomer cannot live without receiving at least 7 ads per 3 minutes. { I myself start being suicidal after watching TV for like 1 minute, I have now started to cover my ears when ads start to play somewhere (you just hear them on a bus, in stores etc.) but sometimes you can't do it. It's seriously endangering my life, not even kidding. ~drummyfish }

## See Also

- leading the pig to the slaughterhouse

---

small3dlib

## Small3dlib

Small3dlib (S3L) is a very portable LRS/suckless single header 3D software renderer library written by drummyfish in the C programming language. It is very efficient and runs on many resource-limited computers such as embedded open consoles. It is similar to TinyGL, but yet more simple. Small3dlib is public domain free software under CC0, written in under 3000 lines of code.

The repository is available at <https://codeberg.org/drummyfish/small3dlib> and <https://gitlab.com/drummyfish/small3dlib>.

Small3dlib can be used for rendering 3D graphics on almost any device as it is written in pure C99 without any software and hardware dependencies; it doesn't use the standard library, floating point or GPU. It is also very flexible, not forcing any preprogrammed shaders -- instead it only computes which pixels should be rasterized and lets the programmer of the main application decide himself what should be done with these pixels (this is typically applying some shading and writing them to screen).

- different drawing strategies (z-buffer, sorted rendering, ...)
- top-left rasterization rule (no holes between or overlaps of triangles)
- perspective correction (either none, full or approximate)
- different near plane collision handling strategies (culling, clipping, shifting, ...)

*Simple ASCII rendering made with small3dlib.*

# Smallchesslib

|   | A   | B    | C   | D   | E   | F | G | H |
|---|-----|------|-----|-----|-----|---|---|---|
| 8 | r:: | b::  | k:: | n:r |     |   |   |   |
| 7 | ::  | p:p  | ::  | p:: |     |   |   |   |
| 6 | p:b | n::  | ::  |     | p   |   |   |   |
| 5 | ::# | ::   |     | P   | p   |   |   |   |
| 4 | ::  | #B:: | N:: | ::  |     |   |   |   |
| 3 | P   | ::   |     | N:: |     |   |   |   |
| 2 |     | P    | P:B |     | P:P |   |   |   |
| 1 | R   | ::   |     | K   | ::  | R |   |   |



white played b5c4  
black to move

ply number: 27  
FEN: r1b1k1nr/1pp2p2/pbn4p/4P1p1/2B1N3/P4N2/1PPB2PP/R3K2R b KQ - 1 14  
board static evaluation: 0.167969 (43)  
board hash: 3262264934  
phase: midgame  
en passant: 0  
50 move rule count: 1  
PGN: 1. e4 e5 2. d4 d5 3. dxe5 dxe4 4. Qxd8+ Kxd8 5. f3 exf3 6. Nxf3 Ke8 7. Nc3 Bb4 8. a3 Bc5 9. Ne4 Bb6 10. Bb5+ Nc6 13. Bd2 a6 14. Bc4\*

*"Screenshot" of smolchess in action, playing against itself.*

Technical details: it's no stockfish, simplicity is favored over strength. Evaluation function is hand-made with 16 bit integer score (no float!). Board is represented as an array of bytes (in fact it's an array of ASCII characters that can conveniently be printed right out). AI uses basic recursive minimax and alpha-beta pruning, with quiescence handled by extending search for exchanges and checks. For simplicity there is no move ordering, transposition table, opening book or similar fanciness, so the performance isn't great, but it manages to play some intermediate moves. Xboard is supported.

While there are many high level engines focusing on maximizing playing strength, there are almost no seriously simple ones focusing on other points -- smallchesslib/smolchess tries to change this. It can be used as an educational tool, a basis for other engines, a simple engine runnable on weak devices, a helper for processing standard chess formats etc.

---

smart

## Smart

*Smart, smells like fart.*

The adjective "smart", as in e.g. *smartphone*, is in the context of modern capitalist technology used as a euphemism for malicious features that include spyware, bloat, obscurity, DRM, ads, programmed planned obsolescence, unnecessary dependencies (such as required Internet connection), anti-repair design and others; it is the opposite of dumb. "Smart" technology is far inferior to the traditional dumb technology and usually just downright harmful to its users and society as a whole, but normal (i.e. retarded) people think it's good because it has a cool name, so they buy and support such technology. They are slowly boiled to accept "smart" technology as the standard.

## See Also

- dumb

---

smol\_internet

## Smol Internet

Smol Internet, smol web, small web, smol net, dork web, poor man's web, web revival, web 1.0 and similar terms refer to Internet technology (such as gopher, gemini, plain HTML etc.) and communities that are smaller (see minimalism), simpler, less controlled/centralized and less toxic than the "big" mainstream/commercial Internet (especially the web) which due to capitalism became littered with shit like ads, unbearable bloat, censorship, spyware, corporate propaganda, masses of retarded people, bullshit cheap visuals like animations etc. Consider this analogy: the mainstream, i.e. world wide web, Discord, Facebook etc., is like a big shiny city, but as the city grows and becomes too stressful, overcrowded, hasted, overcontrolled with police and ads on every corner, people start to move to the countryside where life is simpler and happier -- smol Internet is the countryside.

What EXACTLY constitutes the Smol Internet? Of course we don't really have exact definitions besides what people write on blogs, it also depends on the exact term we use, e.g. smol web may refer specifically to lightweight self-hosted websites while smol net will also include different protocols than [HTTP\(s\)](#) (i.e. things outside the Web). But we are usually talking about simpler ([KISS](#), [suckless](#), ...), alternative, [decentralized](#), [self hosted](#) technology (protocols, servers, ...), and communities that strive to escape commercially spoiled spaces. These communities don't aim to grow to big sizes or compete with the mainstream web, they do not seek to replace the web or achieve the same things (popularity, profitability, ...) but rather bring back the quality the web (and similar services such as [Usenet](#)) used to have in the early days such as relative freedom, unrestricted sharing, [free speech](#), [simplicity](#), [decentralization](#), creative personal sites, [comfiness](#), [fun](#) and so on. It is for the people, not for companies and [corporations](#). Smol Internet usually refers to [gopher](#) and [gemini](#), the alternative protocols to [HTTP](#), the basis of the web. Smol Web on the other hand stands for simple, plain [HTML](#) web 1.0 static personal/community sites on the web itself which are however hosted independently, often on one's own server (self hosted) or a server of some volunteer or non-profit -- such sites can be searched e.g. with the [wiby](#) search engine. It may also include small communities such as [pubnixes](#) like [SDF](#) and [tildeverse](#). Other [KISS](#) communication technology such as [email](#) and [IRC](#) may also fall under Smol Internet.

BEWARE: even the Smol Net gets its fair share of toxicity, nowadays especially gemini is littered with [SIW](#) fascists and [pseudominimalists](#). Avoid such places if you can.

## See Also

- [web 1.0](#)
- [web 0.5](#)
- [Fediverse](#)
- [tildeverse](#)
- [Usenet](#)
- [geocities](#)
- [neocities](#)
- [soynet](#)
- [webring](#)
- [incelosphere](#)

---

social\_inertia

## Social Inertia

Social inertia appears when a social group continues to behave in established ways chiefly because it has behaved that way for a long time, and that even if such behavior is no longer well rationally justified.

---

software

## Software

The article you're looking for is [here](#).

---

sorting

## Sorting

Sorting means rearranging a sequence, such as a [list](#) of numbers, so that the elements are put in a specific order (e.g. ascending or descending). In [computer science](#) sorting is quite a wide topic, there are dozens, maybe hundreds of sorting [algorithms](#), each with pros and cons and different attributes are being studied, e.g. the algorithm's [time complexity](#), its stability etc. Sorting algorithms are a favorite subject of programming classes as they provide a good exercise for [programming](#) and analysis of algorithms and can be nicely put on tests :)

Some famous sorting algorithms include bubble sort (a simple KISS algorithm), quick and merge sort (some of the fastest algorithms) and stupid sort (just tries different permutations until it hits the jackpot).

In practice we often get away with using just some of the simplest sorting algorithms (such as bubble sort or insertion sort) anyway, unless we're programming a database or otherwise dealing with enormous amounts of data. If we need to sort just a few hundred of items and/or the sorting doesn't occur very often, a simple algorithm does the job well, sometimes even faster due to a potential initial overhead of a very complex algorithm. So always consider the KISS approach first.

Attributes of sorting algorithms we're generally interested in are the following:

- **time and space complexity**: Time and space complexity hints on how fast the algorithm will run and how much memory it will need, specifically we're interested in the **best, worst and average case** depending on the length of the input sequence. Indeed we ideally want the fastest algorithm, but it has to be known that a better time complexity doesn't have to imply a faster run time in practice, especially with shorter sequences. An algorithm that's extremely fast in best case scenario may be extremely slow in non-ideal cases. With memory, we are often interested whether the algorithm works **in place**; such an algorithm only needs a constant amount of memory in addition to the memory that the sorted sequence takes, i.e. the sequence is sorted in the memory where it resides.
- **implementation complexity**: A simple algorithm is better if it's good enough. It may lead to e.g. smaller code size which may be a factor e.g. in embedded.
- **stability**: A stable sorting algorithm preserves the order of the elements that are considered equal. With pure numbers this of course doesn't matter, but if we're sorting more complex data structures (e.g. sorting records about people by their names), this attribute may become important.
- **comparative vs non-comparative**: A comparative sort only requires a single operation that compares any two elements and says which one has a higher value -- such an algorithm is general and can be used for sorting any data, but its time complexity of the average case can't be better than  $O(n * \log(n))$ . Non-comparison sorts can be faster as they may take advantage of other possible integer operations.
- **recursion and parallelism**: Some algorithms are recursive in nature, some are not. Some algorithms can be parallelised e.g. with a GPU which will greatly increase their speed.
- **other**: There may be other specific, e.g. some algorithms are slow if sorting an already sorted sequence (which is addressed by *adaptive* sorting), so we may have to also consider the nature of data we'll be sorting. Other times we may be interested e.g. in what machine instructions the algorithm will compile to etc.

In practice not only the algorithm but also its implementation matters. For example if we have a sequence of very large data structures to sort, we may want to avoid physically rearranging these structures in memory, this could be slow. In such case we may want to use **indirect sorting**: we create an additional list whose elements are indices to the main sequence, and we only sort this list of indices.

## List Of Sorting Algorithms

TODO

## Example And Code

For starters let's take a look at one of the simplest sorting algorithms, bubble sort. Its basic version looks something like this (pseudocode):

```
for j from 0 to N - 2 (inclusive)
  for i from 0 to N - 2 - j (inclusive)
    if array[i] > array[i + 1]
      swap array[i] and array[i + 1]
```

How does this work? Firstly notice there are two loops. The outer loop, with counter variable *j*, runs *N - 1* times -- in each iteration of this loop we will ensure one value gets to its correct place; specifically the values will be getting to their correct places from the top -- highest values will be sorted first (you can also implement the algorithm the other way around too, to sort the lowest values first, try it as an exercise). This

makes sense, imagine that we have e.g. a sequence of length  $N = 4$  -- then the outer loop will run  $N - 1 = 3$  times ( $j$  will have values 0, 1 and 2); after first iteration 1 value will be in its correct place, after 2 iterations 3 values will be in place and after 3 iterations 3 values will be in place which also means the last (forth) value has to be in place too, i.e. the array must be sorted. Now for the inner loop (with variable  $i$ ): this one ensures actually getting the value in its place. Notice it goes from 0 to the top and always compares two neighbors in the array -- if the bottom neighbor is higher than the top neighbor, the loop swaps them, ensuring that the highest value will get to the top (it kind of "bubbles" up, hence the algorithm name). Also notice this loop doesn't always go to the very end of the array! It subtracts the value  $j$  from its top boundary because there the values that are already in place reside, so we don't need to sort them anymore; the inner loop can end earlier and earlier as the outer loop progresses. The algorithm would still work if we went through the whole array every time (try it), but its time complexity would suffer, i.e. by noticing the inner loop can get progressively shorter we greatly optimize the algorithm. Anyway, how the algorithm actually works is best seen on an example, so let's now try to use the algorithm to sort the following sequence:

3 7 8 3 2

The length of the sequence is  $N = 5$ , so  $j$  (the outer loop) will go from 0 to 3. The following shows how the array changes (/ \ shows comparison of neighbors, read top to bottom and left to right):

|         | $j = 0$                      | $j = 1$                        | $j = 2$                               | $j = 3$                               |            |
|---------|------------------------------|--------------------------------|---------------------------------------|---------------------------------------|------------|
| $i = 0$ | / \<br>37832                 | / \<br>37328                   | / \<br>33278                          | swapped<br>/ \<br>23378<br>" " " "    | <-- SORTED |
| $i = 1$ | / \<br>37832                 | swapped<br>/ \<br>33728        | swapped<br>/ \<br>32378<br>" " " "    | <--- last 3 items are in their places |            |
| $i = 2$ | swapped<br>/ \<br>37382      | swapped<br>/ \<br>33278<br>" " | <--- last 2 items are in their places |                                       |            |
| $i = 3$ | swapped<br>/ \<br>37328<br>" | <--- last item is in its place |                                       |                                       |            |

Hopefully it's at least a bit clear -- if not, try to perform the algorithm by hand, that's a practically guaranteed way of gaining understanding of the algorithm.

Now let's see other algorithms and some actual runnable code. The following is a C program that shows implementations of some of the common sorting algorithms and also measures their speed:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define N 64

char array[N + 1]; // extra 1 for string terminating zero

void swap(int i1, int i2)
{
    int tmp = array[i1];
    array[i1] = array[i2];
    array[i2] = tmp;
}

void setupArray(void) // fills the array with pseudorandom ASCII letters
{
    array[N] = 0;
    srand(123);

    for (int i = 0; i < N; ++i)
        array[i] = 'A' + rand() % 26;
}
```

```

void test(void (*sortFunction)(void), const char *name)
{
    int timeTotal = 0;

    for (int i = 0; i < 64; ++i) // run the sort many times to average it a bit
    {
        setupArray();
        long int t = clock();
        sortFunction();
        timeTotal += clock() - t;
    }

    printf("%-10s: %s, CPU ticks: %d\n",name,array,(int) timeTotal);
}

// ===== sort algorithms =====

void sortBubble(void)
{
    for (int j = 0; j < N - 1; ++j)
        for (int i = 0; i < N - 1 - j; ++i)
            if (array[i] > array[i + 1])
                swap(i,i + 1);
}

void sortBubble2(void) // simple bubble s. improvement, faster if already sorted
{
    for (int j = 0; j < N - 1; ++j)
    {
        int swapped = 0;

        for (int i = 0; i < N - 1 - j; ++i)
            if (array[i] > array[i + 1])
            {
                swap(i,i + 1);
                swapped = 1;
            }

        if (!swapped) // if no swap happened, the array is already sorted
            break;
    }
}

void sortInsertion(void)
{
    for (int j = 1; j < N; ++j)
        for (int i = j; i > 0; --i)
            if (array[i] < array[i - 1]) // keep moving down until we find its place
                swap(i,i - 1);
            else
                break;
}

void sortSelection(void)
{
    for (int j = 0; j < N - 1; ++j)
    {
        int min = j;

        for (int i = j + 1; i < N; ++i) // find the minimum
            if (array[i] < array[min])
                min = i;

        swap(j,min);
    }
}

void sortStupid(void)
{
    while (1)
    {

```

```

    for (int i = 0; i < N; ++i) // check if the array is sorted
        if (i == (N - 1))
            return;
        else if (array[i] > array[i + 1])
            break; // we got to the end, the array is sorted

    for (int i = 0; i < N; ++i) // randomly shuffle the array
        swap(i, rand() % N);
}

void _sortQuick(int a, int b) // helper recursive function for the main quick s.
{
    if (b <= a || a < 0)
        return;

    int split = a - 1;

    for (int i = a; i < b; ++i)
        if (array[i] < array[b])
        {
            split++;
            swap(split, i);
        }

    split++;
    swap(split, b);

    _sortQuick(a, split - 1);
    _sortQuick(split + 1, b);
}

void sortQuick(void)
{
    _sortQuick(0, N - 1);
}

int main(void)
{
    setupArray();
    printf("array      : %s\n", array);

#ifdef 0 // stupid sort takes too long, only turn on while decreasing N to like 10
    test(sortStupid, "stupid");
#endif
    test(sortBubble, "bubble");
    test(sortBubble2, "bubble2");
    test(sortInsertion, "insertion");
    test(sortBubble2, "selection");
    test(sortQuick, "quick");

    return 0;
}

// TODO: let's add more algorithms in the future :-)
```

It may output for example:

```

array      : RLPALFTOCFWGVJYPLLUNEPDBSOMIBMXSXMVLR0ZUWXARHAIUNCJTUNVMDHWHHTZT
bubble     : AAABBCDDEFFGHHHIIJJLLLLMMMMNNN000PPPPRRRSSTTTTTUUUUUVVVWWWWWXXXYZZ, CPU ticks: 1191
bubble2    : AAABBCDDEFFGHHHIIJJLLLLMMMMNNN000PPPPRRRSSTTTTTUUUUUVVVWWWWWXXXYZZ, CPU ticks: 1164
insertion  : AAABBCDDEFFGHHHIIJJLLLLMMMMNNN000PPPPRRRSSTTTTTUUUUUVVVWWWWWXXXYZZ, CPU ticks: 665
selection  : AAABBCDDEFFGHHHIIJJLLLLMMMMNNN000PPPPRRRSSTTTTTUUUUUVVVWWWWWXXXYZZ, CPU ticks: 1217
quick      : AAABBCDDEFFGHHHIIJJLLLLMMMMNNN000PPPPRRRSSTTTTTUUUUUVVVWWWWWXXXYZZ, CPU ticks: 365
```

## See Also

- [searching](#)
- [pathfinding](#)

# Soydev

Sodevs are incompetent wanna-be programmers that usually have these characteristics:

- Being pseudoleftist (fascist) political activists pushing tranny software and COCs while actually being mainstream centrists in the tech world (advocating "open-source" instead of free software, being okay with proprietary software, bloat etc.).
- Trying to be "cool", having friends and even spouses and kids, wearing T-shirts with "coding jokes", having tattoos, piercing and colored hair (also trimmed bear in case of males).
- Only being hired in tech for a no-brainer job such as "coding websites" or because of diversity quotas.
- Being actually bad at programming, using meme high-level languages like JavaScript, Python or Rust. { I shit you not, I learned from a friend who lives in India that "universities" there produce "security experts" who don't even have to know any programming and math. They just learn some sysadmin stuff and installing antiviruses, without having any clue about how encryption works etc. These people get regular degrees. Really makes me wanna kys myself. ~drummyfish }
- Using a Mac.
- Thinking they're experts in technology because they are familiar with the existence of Linux (usually some mainstream distro such as Ubuntu) and can type `cd` and `ls` in the terminal.
- Having only shallow awareness of tech culture, telling big-bang-theory HTML jokes (*sudo make sandwich*, 42 hahahahahahaha).
- Being highly active on social networks, probably having a pixel-art portrait of their ugly face and "personal pronouns" on their profile.
- Believing in and engaging in capitalism, believing corporations such as Microsoft, wanting to create "startups", being obsessed with productivity, inspirational quotes and masturbating to tech "visionaries" like Steve Jobs. The "rebels" among these are advocating FOSS, however they always promote huge bloat and de-facto capitalist software which is no different from proprietary software.
- Using buzzwords like "solution", "orchestration" etc.
- ...

Here is a quick rough comparison of soydevs and actual good programmers (nowadays mostly an extinct species):

| characteristic      | good programmer                                | soydev                                                                                   |
|---------------------|------------------------------------------------|------------------------------------------------------------------------------------------|
| math skills         | deep knowledge of math                         | "I don't need it", "there's library for that", memorized math interview questions        |
| computer knowledge  | all-level, big-picture knowledge of principles | knowledge of trivia ("This checkbox in this framework has to be unchecked.", ...)        |
| specialization      | generalist                                     | hyperspecialized, knows one language/framework                                           |
| prog. languages     | C, assembly, FORTRAN, Forth, comun, lisp, ...  | Python, JavaScript, Rust, Java, C#, C++2045, ...                                         |
| mostly does         | thinking about algorithms and data structures  | typing glue code for different libraries, updates/maintains systems, talks to people     |
| political opinions  | politically incorrect hippie anarcho pacifist  | liberal capitalist feminist pro black lesbian LGBT fascist anti Nazi                     |
| hardware            | 640x480 1990s laptop, no mouse                 | 2023 touchscreen 1080K macbook with stickers all over, wireless \$1000 AI gaming mouse   |
| memorized knowledge | 10000 digits of pi                             | 10000 genders plus offensive words he mustn't say                                        |
| text editor         | vim, ed, ...                                   | Microsoft AI blockchain VSCode with 10000 plugins running in 10000 virtual sandboxes     |
| looks               | fat, unwashed, unkept beard, dirty clothes     | pink hair, fake glasses, \$1000 T-shirt "sudo make sandwich HAHA BAZINGA", 10000 tattoos |
| gender              | male                                           | depends on mood                                                                          |
| race                | white                                          | prefers not to specify, offended by the question                                         |

| characteristic | good programmer                              | soydev                                                                     |
|----------------|----------------------------------------------|----------------------------------------------------------------------------|
| hobbies        | reading encyclopedias, chess, rocket science | distrohopping, browserhopping, githopping, editorhopping, tiktok, partying |

---

soyence

## Soyence

*Not to be confused with science.*

{ I did my own peer review of this article and give it 10/10. ~drummyfish }

Soyence is business, propaganda and politics trying to pass as science, nowadays promoted typically by pseudoleftists, pseudoskeptics, capitalists and corporations. It is what in the 21st century has taken on the role that's historically been played by the church: that of establishing and maintaining orthodoxy for the control of mass population -- this time it is so called "science" or "rationality" that's used as the tool instead of God and religion, however the results are the same. Soyence is not about listening to what science says, it is about listetning to what "reputable scientists" say, and of course not questioning them; soyence is what the typical reddit atheist or tiktok feminist believes science is or what Neil De Grass Tyson tells you science is. While science is about collecting facts and drawing conclusions, soyence is about setting conclusions and finding or fabricating facts that support them. One red flag to watch out in relation to soyence is a great weight put on **reputation** -- in true science reputation plays no role, only results do; reputation and its great value for one's acceptance is rather part of politics (and maybe show business). Notice for example how in the past it was more common to hear "science has found X" (as in "logic itself shows this fact") rather than "scientists have found X", which is more common nowadays -- mentally we have shifted to separate people to "scientists", those who "know" and dictate what's true, and non-scientists, those who don't know and must just listen. Soyence calls itself the one and only science^TM and gatekeeps the term by calling unpopular science (such as that regarding human race, questioning official versions of historical events or safety of big pharma vaccines) "pseudoscience" and "conspiracy theories". Soyence itself is pseudoscience but it has an official status, approval of state, strong connection to politics, it is mainstream, popular, controlled by those in power, censored ("moderated") and intentionally misleading. Soyence can be encountered in much of academia, on Wikipedia and in other popular/mainstream media such as TV "documentaries" and YouTube. A soyence supporter wrongfully believes that reason wouldn't allow such a large scale mass population manipulation (despite this happening over and over throughout history) -- people at large aren't reasonable and reason cannot beat propaganda.

Compared to good old fun pseudosciences such as astrology and flat Earth, soyence is extra sneaky by purposefully trying to blend in with real science, i.e. within a certain truly scientific field, such as biology, there is a soyentific cancer mixed in by activists, corporations and state, that may be hard to separate for common folk and many times even for pros. This is extremely harmful as in the eyes of retarded people (basically everyone) the neighboring legit science gives credibility to propaganda bullshit. There is a tendency to think we somehow magically live in a time that's fundamentally different from other times in history in which it is now a pretty clear and uncontroversial fact that the name of science was abused hard by propaganda, almost everyone easily accepts that historically politically constructed lies were presented as confirmed by science, but somehow people refuse to believe it could be the case nowadays. In times of Nazism there was no doubt about race being a completely scientific term and that Jews were scientifically confirmed to be the inferior race -- nowadays in times when anti Nazis have won and politics is based on denying existence of race somehow scientists start to magically find evidence that no such thing as race has ever existed -- how convenient! And just in case you wanted to check if it's actually true, you'll be labeled a racist and you won't find job ever again.

Soyence uses all the cheap tricks of politics (also not dissimilar to those of greenwashing, openwashing etc.) to win stupid people, it builds on the cult of bullying religion and creating a war mentality, overuse of twisted "rationality" (pseudoskepticism), creating science bloat and bullshit "scientific" fields to obscure lies, punishment of the correct use of rationality, building cults of personality ("science educators", the gatekeepers of "science") and appealing to egoism and naivty of wannabe smartasses while at the same time not even holding up to principles of science such as genuine objectivity. A soyence kid will for example keep preaching about how everything should be proven by reproducible experiments while at the same time



accepting de facto irreproducible results, e.g. those obtained with billion dollar worth research performed at CERN which can NOT be reproduced anywhere else than at CERN with thousands of top scientist putting in years of work. Such results are not reproducible in practice, they are accepted on the basis of pure faith in those presenting it, just as religious people accept the words of preachers. The kid will argue that in theory someone else can build another CERN and reproduce the results, but that won't happen in practice, it's just a purely theoretical unrealistic scenario so his version of what "science" is is really based on reproducibility that only works in a dreamed up world, this kind of reproducibility doesn't at all fulfill its original purpose of allowing others to check, confirm or refute the results of experiments. This starts to play a bigger role when for example vaccines start to get promoted by the government as "proven safe by science" (read "claimed safe by a corporation who makes money off of people being sick"), the soyence kid will gladly accept the vaccine and fight for their acceptance just thanks to this label, not based on any truly scientific facts but out of pure faith in self proclaimed science authorities -- here the soyentist is relying purely on faith, a concept he would like to think he hates with his soul.

The "citation needed" craziness that indicates lack of any brain and pure reliance on the word of authority is seen e.g. on Wikipedia. Wikipedia doesn't accept original research, observation or EVEN LOGIC ITSELF as a basis for presenting something -- everything, even trivial claims have to have a "citation" from a source WITH mainstream political views (unpopular and controversial sources are banned); Wikipedia is therefore one big propaganda ground for those with power over the mainstream media.

Soyence relies on low IQ, shallow education and popular "science education" (e.g. neil de grass), while making its followers believe they are smart. It produces propaganda material such as "documentaries" with Morgan Freeman (i.e. people who are good at persuasion rather than being competent), series like The Big Bang Theory and YouTube videos with titles such as "Debunking Flat Earth with FACTS AND LOGIC", so there's a huge mass of NPCs thinking they are Einsteins who blindly support this cult. Soyence attacks science from within by attacking its core principles, i.e. it tries to ridicule and punish thinking outside the box and asking specific questions -- in this it is not dissimilar to a mass religion.

Examples of soyence:

- gender studies LMAO
- "Race is a social construct and doesn't have any biological meaning."
- any pseudoskeptical shit trying to look "scientific"
- Bullshit degrees, e.g. someone getting PhD in "user experience", "level design", "diversity in software engineering" or "making youtube videos" (like that fucker from Veritasium lmao).
- "Women are as intelligent as men, if not more."
- "citation needed" on everything
- "Science popularization" as in building authority of so called "scientists" so as to create a political capital.
- "This extremely lucrative Covid vaccine made by us in record time is absolutely safe, don't dare question it, just take it 5 times a year and pay us each time you do, don't mind any side effects." --Big Pharma
- "Science says god doesn't exist." aka reddit atheism
- "We can't believe this because it wasn't peer censored and/or it didn't pass the null ritual and/or it wasn't published in a journal on our approved literature list." (--Wikipedia)
- "This gender studies expert has proven sex is a racial construct and has no biological meaning. You disagree? Well, do you have a PhD in gender studies? No? Then shut up you fucking sexist."
- "This goes against SCIENTIFIC CONSENSUS therefore it's pseudoscience and conspiracy theory."
- "This research is racist.", using terms such as "scientific racism".
- This guy's research is invalid because in his spare time he makes videos on ufology and other "conspiracy theories", his REPUTATION AND CREDIBILITY is destroyed.
- "We should burn these old books that say things we don't like, just in case. When Nazis did it it was different."
- "This research was made by a racist so it is invalid, also we should lynch the guy just in case."
- Neil de grass/Morgan Freeman "documentaries"
- "We can totally trust the results of commercial research."
- "These negative results are useful but unexciting so let's not publish them, we gotta entertain our readers to stay on the market. We GOTTA TELL INSPIRATIONAL STORIES with our papers." --soyence journals

- "No, you can't research the details of historic events such as Holocaust." (see anti [Holocaust](#) denial laws)
- great part of [economics](#)
- "[pedophilia](#) is a mental illness while pure [homosexuality](#) is not"
- ...

## See Also

- [pseudoskepticism](#)
- [conspiracy theory](#)

---

speech\_synthesis

# Speech Synthesis

TODO

## Example

This is a simple [C](#) program (using [float](#) for simplicity of demonstration) that creates basic vowel sounds using formant synthesis (run e.g. as `gcc -lm program.c && ./a.out | aplay`, 8000 Hz 8 bit audio is supposed):

```
#include <stdio.h>
#include <math.h>

double vowelParams[] = { // vocal tract shapes, can be found in literature
    // formant1  formant2  width1  width2  amplitude1  amplitude2
    850,         1650,     500,     500,     1,         0.2, // a
    390,         2300,     500,     450,     1,         0.9, // e
    240,         2500,     300,     500,     1,         0.5, // i
    250,         600,      500,     400,     1,         0.9, // o
    300,         400,      400,     400,     1,         1.0, // u
};

double tone(double t, double f) // tone of given frequency
{
    return sin(f * t * 2 * M_PI);
}

/* simple linear ("triangle") function for modelling spectral shape
   of one formant with given frequency location, width and amplitude */
double formant(double freq, double f, double w, double a)
{
    double r = ((freq - f + w / 2) * 2 * a) / w;

    if (freq > f)
        r = -1 * (r - a) + a;

    return r > 1 ? 1 : (r < 0 ? 0 : r);
}

/* gives one sample of speech, takes two formants as input, fundamental
   frequency and possible offset of both formants (can model "bigger/smaller
   head") */
double speech(double t, double fundamental, double offset,
    double f1, double f2,
    double w1, double w2,
    double a1, double a2)
{
    int harmonic = 1; // number of harmonic frequency

    double r = 0;

    /* now generate harmonics (multiples of fundamental frequency) as the source,
       and multiply them by the envelope given by formants (no need to deal with
       multiplication of spectra; as we're constructing the result from basic
```

```

    frequencies, we can simply multiply each one directly): */
while (1)
{
    double f = harmonic * fundamental;
    double formant1 = formant(f,f1 + offset,w1,a1);
    double formant2 = formant(f,f2 + offset,w2,a2);

    // envelope = max(formant1,formant2)
    r += (formant1 > formant2 ? formant1 : formant2) * 0.1 * tone(t,f);

    if (f > 10000) // stop generating harmonics above 10000 Hz
        break;

    harmonic++;
}

return r > 1.0 ? 1.0 : (r < 0 ? 0 : r); // clamp between 0 and 1
}

int main(void)
{
    for (int i = 0; i < 50000; ++i)
    {
        double t = ((double) i) / 8000.0;
        double *vowel = vowelParams + ((i / 4000) % 5) * 6; // change vowels

        putchar(128 + 127 *
            speech(t,150,-100,vowel[0],vowel[1],vowel[2],vowel[3],vowel[4],vowel[5]));
    }

    return 0;
}

```

---

splinternet

## Splinternet

TODO

---

sqrt

## Square Root

Square root (sometimes shortened to *sqrt*) of number *a* is such a number *b* that  $b^2 = a$ , for example 3 is a square root of 9 because  $3^2 = 9$ . Finding square root is one of the most basic and important operations in math and programming, e.g. for computing distances, solving quadratic equations etc. Square root is a special case of finding Nth root of a number for  $N = 2$ . Square root of a number doesn't have to be a whole number; in fact if the square isn't a whole number, it is always an irrational number (i.e. it can't be expressed as a fraction of two integers, for example square root of two is approximately 1.414...); and it doesn't even have to be a real number (e.g. square root of -1 is *i*). Strictly speaking there may exist multiple square roots of a number, for example both 5 and -5 are square roots of 25 -- the positive square root is called **principal square root**; principal square root of *x* is the same number we get when we raise *x* to 1/2, and this is what we are usually interested in -- from now on by *square root* we will implicitly mean *principal square root*. Programmers write *square root of x* as `sqrt(x)` (which should give the same result as raising to 1/2, i.e. `pow(x,0.5)`), mathematicians write it as:

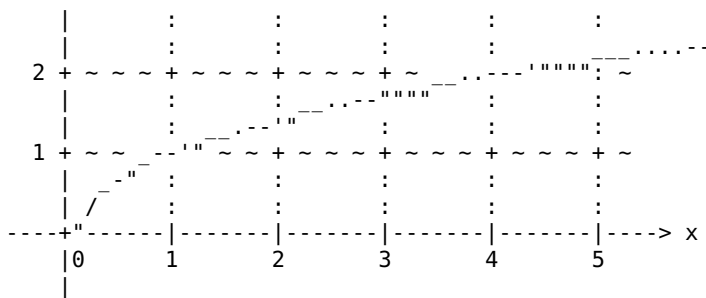
$$\sqrt{x} = x^{1/2}$$

Here is the graph of square root function (notice it's a parabola flipped by the diagonal axis, for square root is an inverse function to the function  $x^2$ ):

```

      ^ sqrt(x)
      |      :      :      :      :
3 + ~ ~ ~ + ~ ~ ~ + ~ ~ ~ + ~ ~ ~ + ~ ~ ~ + ~

```



TODO

## Programming

TODO

If we need extreme speed, we may use a look up table with precomputed values.

Within desired precision square root can be relatively quickly computed iteratively by binary search. Here is a simple C function computing integer square root this way:

```
unsigned int sqrt(unsigned int x)
{
    unsigned int l = 0, r = x / 2, m;

    while (1)
    {
        if (r - l <= 1)
            break;

        m = (l + r) / 2;

        if (m * m > x)
            r = m;
        else
            l = m;
    }

    return (r * r <= x ? r : l) + (x == 1);
}
```

TODO: Heron's method

The following is a **non-iterative approximation** of integer square root in C that has acceptable accuracy to about 1 million (maximum error from 1000 to 1000000 is about 7%): { Painstakingly made by me. ~drummyfish }

```
int32_t sqrtApprox(int32_t x)
{
    return
        (x < 1024) ?
        (-2400 / (x + 120) + x / 64 + 20) :
        ((x < 93580) ?
        (-1000000 / (x + 8000) + x / 512 + 142) :
        (-75000000 / (x + 160000) + x / 2048 + 565));
}
```

---

SSAO

## SSAO

Screen space ambient occlusion (SSAO) is a screen space technique used in 3D computer graphics for **approximating** ambient occlusion (basically "dim shadows in corners", which itself is an approximation of

true global illumination) in a way that's easy and not so expensive to implement to run in real time. The effect however looks ugly many times and is often criticized, see e.g. an excellent article at <https://nothings.org/gamedev/ssao/>.

{ 2023 report: SSAO still sucks. ~drummyfish }

Exact ambient occlusions can be computed with algorithms such as RTO (which uses raytracing), but this requires complete information about the geometry and is too slow without special hardware. Therefore some game devs cheat and use a cheap approximation: SSAO is implemented as a post-processing shader and only uses the information available on the screen, specifically in the depth buffer -- this gives only partial information about the actual scene geometry, i.e. the algorithm doesn't know what the back facing, screen-perpendicular or off-screen geometry looks like and has to make guesses which sometimes result in quite visible inaccuracies.

This method is notoriously ugly in certain conditions and many modern games suffer from this, even the supposedly "photorealistic" engines like Unreal -- if someone is standing in front of a wall there is a shadow outline around him that looks so unbelievably ugly you literally want to puke. But normie eyes can't see this lol, they think that's how reality looks and they are okay with this shit, they allow this to happen. Normies literally destroy computer graphics by not being able to see correctly.

What to do then? The most suckless way is to simply do no ambient occlusion -- seriously test how it looks and if it's okay just save yourself the effort, performance and complexity. Back in the 90s we didn't have this shit and games unironically looked 100 times better. You can also just bake the ambient occlusion in textures themselves, either directly in the color texture or use light maps. Note that this makes the ambient occlusions static and with light maps you'll need more memory for textures. Finally, if you absolutely have to use SSAO, at least use it very lightly (there are parameters you can lower to make it less prominent).

---

steganography

## Steganography

Steganography means hiding secret information within some unrelated data by embedding it in a way that's very hard to notice; for example it is possible to hide text messages in a digital photograph by slightly modifying the colors of the image pixels -- that photo then looks just like an innocent picture while in fact bearing an extra information for those who know it's there and can read it. Steganography differs from encryption by trying to avoid even suspicion of secret communication.

There are many uses of steganography, for example in secret communication, bypassing censorship or secretly tracking a piece of digital media with an invisible watermark (game companies have used steganography to identify which tester's game client was used to leak pre-release footage of their games). Cicada 3301 has famously used steganography in its puzzles.

Steganography may need to take into account the possibility of the data being slightly modified, for example pictures exchanged on the Internet lose their quality due to repeating compression, cropping and format conversions. Robust methods may be used to preserve the embedded information even in these cases.

Some notable methods and practices of steganography include:

- Embedding in **text**, e.g. making intentional typos in certain places, using extra white or zero-width characters, modifying formatting and case or using Unicode homoglyphs can all carry information.
- Embedding in **images**. One of the simplest methods is storing data in least significant bits of pixel values (which won't be noticeable by human eyes). Advanced methods may e.g. modify statistical properties of the image such as its color histogram.
- Embedding in **sound, video**, vector graphics and all other kinds of media is possible.
- All kinds of data can be embedded given enough storage capacity of given bearing medium (e.g. it is possible to store an image in text, sound in another sound etc.).
- Information that's present but normally random or unimportant can be used for embedding, e.g. the specific order of items in a list (its permutation) can bear information as well as length of time delays in timed data, amount of noise in data etc.

```

    ,~.,-...~.
      ,-.~.,~>"
        ,,-.-,',,-!$$r><l+
          '':~::~:'..',.,-,+$0fls'
            .~::~:'::;:_::_;_::_;"(!s^x}$;
              -.~JJ<;.'::;~;;;:_::_;<+F!v!r{888
                -.<#0#$ezrslll)l+lfr}{V$88$#!
                  '~+588#0$8$08$8#0$#$05,
                    ,s00#$08$8#88#.
                      $opa{a{x_
                        @$ M$ 8W M$ eFc>!vs::;SJ:/, _:-!\
                          @$ @8 8@ @# ^esCc+/s/^_+ccc+cc+cc+cCi
                            88#8 8# 88 #exoCC+cc>^s^s^s^s^s^s/>/c+c/
                              #8 8#88 #VsFC)CC+cc+cc+cc++cc+cc+cc)/
                                8#88 #8Vi^^oFFoFFoFFoFFoFFoFFoFFFs
                                  8@ W8 8#88 s88#Ve}xxixxiee}ee}eeixxi^_
                                    88#88# 8@- 8@ #8 8#88#88#88#88#88#88#x
                                      8W @8 #88#8 ,#88#88#88#88#88#88(
                                        88#8 8W ,8# ,V#88#88#88#88#88#^
                                          8W 88 -C8#88#88#8^-
                                            ,8#8,-^Vie,
                                              _;;oF^#88+c,-cc),
                                                ,>, , -, />/,
                                                  sCC)c/s^/s,
                                                    , ,

```

[illegible]

## Example Code

```
#include <stdio.h>

const char alphabet[] = // our 6 bit alphabet, position = code
```

```

"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 .";

const char groups[] = // newline separated groups of similar brightness chars
"@WM0"  "\n"  "Q&%R"  "\n"  "8#0$"  "\n"  "B69g"  "\n"
"NUMP"  "\n"  "w3XD"  "\n"  "Vbp5"  "\n"  "d24S"  "\n"
"qkGK"  "\n"  "EHAZ"  "\n"  "hY[T"  "\n"  "e}a{"  "\n"
"Jy17"  "\n"  "Fofu"  "\n"  "n?Ij"  "\n"  "C)(l"  "\n"
"xizr"  "\n"  "^sv!"  "\n"  "t*=L"  "\n"  "/>\\<"  "\n"
"c+J\\"  "\n"  ";_~:"  "\n"  ",-'. "  "\n";

unsigned char toAlphabet(unsigned char c) // encodes char to 6 bit alphabet
{
    for (unsigned char i = 0; i < 64; ++i)
        if (alphabet[i] == c)
            return i;

    return 63;
}

unsigned char fromAlphabet(unsigned char c) // decodes char from 6 bit alphabet
{
    return alphabet[c & 0x3f];
}

int canEncode(char c) // says if specific visual char can be used for encoding
{
    if (c == '\n')
        return 0;

    for (int i = 0; i < sizeof(groups) - 1; ++i)
        if (groups[i] == c)
            return 1;

    return 0;
}

const char *seekGroup(char c) // helper, seeks to similar brightness group
{
    const char *s = groups;

    while (c != *s)
        s++;

    while (*s != '\n')
        s++;

    s--;

    return s;
}

char encode(char c, int n) // encodes value n in given encodable char
{
    const char *s = seekGroup(c);

    while (n)
    {
        s--;
        n--;
    }

    return *s;
}

int decode(char c) // decodes value n from given encodable char
{
    int n = 0;

    const char *s = seekGroup(c);

    while (*s != c)
    {

```

```

        s--;
        n++;
    }

    return n;
}

int main(int argc, char **argv)
{
    unsigned char currentChar = 0;
    int pos = 0, done = 0;

    while (1) // read all chars from input
    {
        int c = getchar();

        if (c == EOF)
            break;

        if (argc < 2)
        {
            // decoding text from image

            if (canEncode(c))
            {
                currentChar = (currentChar << 2) | decode(c);

                if (pos % 3 == 2)
                {
                    putchar(fromAlphabet(currentChar));
                    currentChar = 0;
                }

                pos++;
            }
        }
        else
        {
            // encoding text into image

            if (canEncode(c))
            {
                unsigned char c2 = !done ? argv[1][pos / 3] : 0;

                if (!c2)
                {
                    done = 1;
                    c2 = ' ';
                }

                c2 = (toAlphabet(c2) >> ((2 - pos % 3) * 2)) & 0x03;
                c = encode(c, c2);
                pos++;
            }

            putchar(c);
        }
    }

    return 0;
}

```

The usage is following: make a file with a grayscale ASCII art picture, then pass it to the standard input of this program along with text you want to encode (maximum length of the text you can encode is given by the count of usable characters in the input image) passed as the first argument to the program, for example: `cat picture.txt | ./program "hello"`. The program will print out the image with the text embedded in. To read the text from the image similarly pass the picture to the program's input, without passing any arguments, for example: `cat picture2.txt | ./program`. The text will be written to terminal.



The method used is this: firstly for the encoded message we use our own 6 bit alphabet -- this only allows us to represent 63 symbols (which we have chosen to be uppercase and lowercase letters, space and period) but will allow us to store more of them. Each 6 bit symbol of our alphabet will be encoded by three bit pairs ( $3 * 2 = 6$ ). One bit pair will be encoded in one ASCII art character by altering that character slightly -- we define groups of ASCII characters that have similar brightness. Each of these groups consists of 4 characters (e.g. @WM0 is the group of darkest characters), so a character can be used to encode 2 bits (one bit pair of the encoded symbol). The first character in the group encodes 00, the second one 01 etc. However not all ASCII art characters can be used for encoding, for example space ( ) has no similar brightness characters, so these are just skipped.

---

stereotype

## Stereotype

Stereotypes are general statistical observations about groups of people (such as different races) which have been discovered naturally and became part of common knowledge (without rigorous scientific effort). Stereotypes are good because they tell us what we may expect from different kinds of people. Of course no one, maybe with the exception of blonde women, is so stupid as to think stereotypes apply 100% of the times -- let us repeat they are STATISTICAL observations, they talk about probabilities.

Stereotypes are also good for showing us the diversity of human races and cultures. Pseudoleftists want to suppress awareness of stereotypes by calling them "offensive" or "discriminating", aiming for creating a sterile society without any differences, without any beauty of diversity. Do not support that, spread the knowledge of stereotypes.

Some stereotypes are:

{ WIP ~drummyfish }

- Americans:
  - ◆ extremely stupid, primitive, close-minded, not knowing geography/history besides the US, think US is the center of the world
  - ◆ extremely fat, eat only fast food, have no real cuisine
  - ◆ shallow, obsessed with looks (white teeth etc.)
  - ◆ materialist, obsessed with money, hardcore capitalists, panic fear of anything resembling communism/socialism
  - ◆ arrogant, rude, individualist, self-centered
  - ◆ eccentric, extroverted, loud behavior
  - ◆ violent, militant, imperialist, constantly invade other countries, everyone has a gun and shoots at everything including their own presidents
  - ◆ don't mind violence but are afraid of public nudity, get panic attacks when see a naked child or nipple on TV
  - ◆ solve things by brute force rather than by smartness
  - ◆ obsessed with working as much as possible and forcing others to do the same
  - ◆ want everything big
  - ◆ US south: slow, even more stupid, racist, rednecks, inbred, for slavery, for guns
- Arabs:
  - ◆ terrorists, suicidal bombers
  - ◆ women are belly dancers
  - ◆ pedophiles, bigamists
  - ◆ dirty
- Asians:
  - ◆ extremely smart
  - ◆ all look the same
  - ◆ polite
  - ◆ don't show emotion
  - ◆ work extremely hard
  - ◆ small penises
  - ◆ men of honor

- ♦ collectivist, sacrifice themselves for society
- ♦ there are too many of them, lives of the poor ones have no value, work safety of peasants is non existent
- black people:
  - ♦ unintelligent, stupid, uneducated, primitive, poor
  - ♦ physically fit, good at sports
  - ♦ good at music, especially rhythmic music and jazz
  - ♦ fathers leave their children
  - ♦ all look the same, similar to monkeys
  - ♦ have big dicks
  - ♦ criminals
  - ♦ love chicken and watermelon
  - ♦ in certain situations act like monkeys (so called chimp out), e.g. when excited they start jumping around like crazy, or when scared instinctively react by punching the perceived danger
- Australians:
  - ♦ tough, living in dangerous wilderness
- blond, attractive women:
  - ♦ extremely stupid
  - ♦ gold diggers
- Canadian:
  - ♦ extremely polite
  - ♦ ice hockey fans
- Chinese:
  - ♦ smart, wise
  - ♦ do martial arts
  - ♦ make crappy off brands and cheap copies of western art, steal "intellectual property", manufacture cheap things at large quantities, everything is "made in China"
  - ♦ don't value human rights
- Czech:
  - ♦ heavy drinkers, especially beer
  - ♦ friendly but appear cold
  - ♦ beautiful women
- English:
  - ♦ well behaved, reserved, educated, classy
  - ♦ conservative, old fashioned
  - ♦ drink tea
  - ♦ dry humor
  - ♦ football fans
  - ♦ dislike French
  - ♦ bad cuisine
- French:
  - ♦ good lovers
  - ♦ lazy, Bohemian life, hate work
  - ♦ eat baguettes and frogs
  - ♦ dislike Brits
  - ♦ revolutionaries, constantly protest
  - ♦ artists, intellectuals
- gays:
  - ♦ men act feminine, are good at art and women jobs
  - ♦ women (lesbian) are masculine, ugly with short pink hair
- Germans:
  - ♦ no sense of humor, being kind of robots
  - ♦ precise, efficient, organized, great technology
  - ♦ love beer and sausage
  - ♦ ugly women
- gypsies:
  - ♦ don't work, steal stuff, welfare leeches, make a lot of children
  - ♦ children don't go to school, uneducated, can hardly read
  - ♦ passionate, emotional, friendly

- ♦ talent for music
- Indians:
  - ♦ extremely friendly, often too much
  - ♦ no hygiene, dirty
  - ♦ smart but poor
  - ♦ good at IT but usually tech support scammers
  - ♦ spiritual, peaceful, meditate a lot
  - ♦ don't know what work safety means
  - ♦ transport extremely big loads on bicycles or small motorcycles
- Italians:
  - ♦ handsome men who are passionate lovers
  - ♦ extremely passionate, have heated emotional arguments about even trivial things
  - ♦ involved with mafia
  - ♦ great focus on family, know and regularly meet distant relatives
  - ♦ have mustaches, eat pizza and pasta
  - ♦ talk with hands
- Japanese:
  - ♦ like extremely weird things like actually living with sex dolls instead of human life partners
  - ♦ salarymen regularly jump out of skyscraper windows due to overworking depression
  - ♦ men talking Japanese to other men sound as if being aggressive to each other even if in fact being polite or talking something uninteresting
  - ♦ everyone reads manga and goes to karaoke after work
  - ♦ extremely precise, always on time, well organized
  - ♦ have extremely technologically advanced toilets
  - ♦ commit seppuku when fail at something important
- jews:
  - ♦ very smart, inventive
  - ♦ greedy
  - ♦ good at business, filthy capitalists
  - ♦ have the "eagle nose"
  - ♦ members of secret societies, closed jew-only communities, conspire for world control, some being fascists wanting to become the ruling race
  - ♦ spread everywhere like rats
  - ♦ can adapt to any environment
  - ♦ do all kinds of weird religious rituals
- Polish:
  - ♦ very religious
  - ♦ heavy drinkers
- Russians:
  - ♦ very tough, big and strong, endure conditions that would kill other people, keep pet bears
  - ♦ drunk (especially by vodka), aggressive, rude
  - ♦ wear Adidas pants
  - ♦ act straight without talking too much, ignore work safety
- Slovak:
  - ♦ who?
- Spanish:
  - ♦ extroverted, passionate, dance flamenco
  - ♦ take naps on siesta
  - ♦ attractive tanned men
- women:
  - ♦ bad at driving
  - ♦ bad at logical thinking and math
  - ♦ passive aggressive
  - ♦ gossip
  - ♦ don't know what they want, "no" can mean "yes"
  - ♦ too emotional, especially on period
  - ♦ attracted to douchebags and money, avoid nice guys
  - ♦ can distinguish and name different shades of similar colors
  - ♦ on board of a ship bring bad luck

## Steve Jobs

"I'm not glad he'd dead, but I'm glad he's gone." -- [Richard Stallman](#)

Steve Jobs (also Steve Jewbs) was the prototypical evil [CEO](#) and co-founder of one of the worst [corporations](#) in the world: [Apple](#). He was a psychopathic entrepreneur with a cult of personality that makes Americans cum. He was mainly known for his ability to manipulate people and he worsened technology by making it more consumerist, expensive and incompatible with already existing technology. All americans masturbate daily to Steve Jobs so he can also be considered the most famous US porn star. Someone once said that there are essentially two types of men in technology: those who understand what they don't manage and those who manage what they don't understand. Jobs was the latter.

{ LOL how come in the American movies the villain is always some rich boss of a huge corporation clearly resembling Steve Jobs, doing literally the same things, it's almost as if the average American actually somehow KNOWS and feels deep inside these people are pure evil, but suddenly outside of a Hollywood movie their brain switches to "aaaaah, that guy is amazing" and they just eat all his bullshit. I just can't comprehend this. ~drummyfish }

Jobs was born on February 24, 1955 and later was adopted which may have contributed to his development of psychopathy. He was already very stupid as a little child, he never really learned programming and was only interested in achieving what he wanted by crying and pressuring other people to do things for him. This translated very well to his adult life when he quit school to pursue money. He manipulated and abused his schoolmate [Steve Wozniak](#), a [hacker](#), to make computers for him. They started [Apple](#) in 1976 and started producing one of the first personal computers: Apple I and Apple II with which he won the [capitalist](#) lottery and unfortunately succeeded on the market. Apple became a big ass company, however Jobs was such [shit](#) CEO that **Apple fired him** lol. He went to do some other shit like NeXT. Then a bunch of things happened (TODO) and then, to the relief of the whole world, he died on October 5, 2011 from cancer. { LRS never wishes for anyone's death, here we only state the simple fact that the world is a better place without Jobs in it. ~drummyfish } Some cause joy wherever they go, others whenever they go.

---

suckless

## Suckless

Suckless, software that [sucks](#) less, is a type of [free software](#), programming philosophy as well as an organization (<http://suckless.org/>), that tries to adhere to a high technological [minimalism](#), [freedom](#) and [hackability](#), and opposes so called [bloat](#) and unnecessary complexity which has been creeping into most "[modern](#)" software and by which technology has started to become less useful and more burdening. It is related to [Unix philosophy](#) and [KISS](#) but brings some new ideas onto the table. It became somewhat known and highly influenced some newly formed groups, e.g. [Bitreich](#) and our own [less retarded software](#). Suckless seems to share many followers with [cat-v.org](#).

The community used to be relatively a small underground niche, however after a rise in popularity sometime in 2010s, thanks to tech youtubers such as [Luke Smith](#), [Distro Tube](#) and [Mental Outlaw](#), the awareness about the group spread a lot wider, even mainstream programmers now usually know what *suckless* stands for. It has also gained traction on [4chan](#)'s technology board which again boosted suckless popularity but also inevitably brought some retardism in. While the group core consisting a lot of expert programmers and [hackers](#) mostly interested in systems like [GNU/Linux](#), [BSDs](#) and [Plan 9](#), a lot of less skilled "[Linux](#)" users and even complete non-programmers now hang around suckless to various degrees -- especially the [dwm](#) window manager has seen a great success among "Unix porn" lovers and chronic [ricers](#). While most of the true suckless followers are hardcore minimalists and apply their principles to everything, many of the noobs around suckless just cherry pick programs they find nice to look at and integrate them in their otherwise bloated systems.

Suckless is pretty cool, it has inspired [LRS](#), but watch out, as with most of the few promising things nowadays it is half cool and half shitty -- for example most suckless followers seem to be [rightists](#) and [capitalists](#) who

are motivated by harmful goals such as their own increased productivity, not by altruism. Many suckless people are quite pragmatic -- though they believe in hardcore minimalism, they will oftentimes, for practical reasons, rather choose e.g. a well established programming language (C) before the more minimal one (e.g. Forth). LRS takes the good and tries to fix the issues of suckless, we only take the good ideas of suckless. Also it seems like by now that part of the suckless community degenerated a bit by its increase in popularity into a bit of what it opposed -- a kind of consumerist fashion followers who aren't interested so much in good design of technology but rather constantly ricing their dwm in pursuit of cool looking pseudominimalist system in ways not dissimilar to those of iToddlers.

{ From what it seems to me, the "official" suckless community is largely quiet and closed, leading conversations mostly on mailing lists and focusing almost exclusively on the development of their software without politics, activism and off topics, probably because they consider it bullshit that would only be distracting. There is also suckless subreddit which is similarly mostly focused on the software alone. They let their work speak. Some accuse the community of being Nazis, however I believe this is firstly irrelevant and secondly mostly false accusations of haters, even if we find a few Nazis among them, just as in any community. Most pro-suckless people I've met were actually true socialists (while Nazis are not socialist despite their name). Unlike tranny software, suckless software itself doesn't promote any politics, it is a set of purely functional tools, so the question of the developers' private opinions is unimportant here, we have to separate ideas and people. Suckless ideas are good regardless of whose brains they came from.  
~drummyfish }

## Attributes

Notable attributes of suckless software include:

- **Being free software** with the preference of **permissive licenses** such as MIT and CC0.
- **Extreme minimalism and minimizing dependencies**, elimination of any bullshit and **bloat**. Advocating Unix philosophy, KISS etc.
- **Configuration of software is part of its source code** (`config.h`) and change of this configuration requires recompiling the software (which is extremely easy and fast with suckless software). This removes the need for dealing with config files which requires special libraries, file systems and extra code.
- Mainly using two programming languages: C (C89 or C99) for compiled programs and **POSIX shell** for scripting. Some also use languages such as go or lisp, but they're in minority.
- **Forking and compiling by default**, software is distributed in source format (no binaries), every user is supposed to create a personal customized fork and compile/customize the software himself.
- Mods (extension/addons) are implemented and distributed as **patch files**. The idea is to fork the base version of the software and then apply patches to make a unique, completely personalized version of the software.
- **Typical upper limit for lines of code of about 10k**, mostly just about 1-2k. This makes software easy to understand, modify, fork and maintain.
- **Focus on the technology itself** without mixing it with politics and other bullshit such as COCs.
- Not aiming for mainstream popularity, being a bit of an **elitist club**, in the good sense -- suckless is for expert users who understand, handle and create non-mainstream technology without handholding. Trying to be normie friendly would just lead to software and community that looks like the mainstream software and its community. { My view on this is that it's not that suckless WANTS to be an elitist club for its own sake; the issue lies in mainstream technology being hostile towards ethical software -- using ethical software nowadays requires one to be very tech savvy, hence it's not suckless who is discriminating but rather those who create mainstream technology. ~drummyfish }

## History

Suckless in current form has existed since 2006 when the domain suckless.org was registered by a German guy Anselm R. Garbe who is the founder of the community. It has evolved from a community centered around specific software projects, most notably wmii. Garbe has given interview about suckless in FLOSS Weekly episode 355.

Some time before 2010 suckless developed stali, a statically linked glibc-less "Linux distro" that was based on the idea that dynamic linking is harmful and that static linking is mostly advantageous. It also came with

suckless software by default. This project was made independent and split from suckless in 2018 by Garbe.

In 2012 a core veteran member of suckless, a Spanish guy nicknamed Uriel, has killed himself and became a meme.

## Projects

Notable projects developed by the suckless group include:

- dwm
- st
- dmenu
- surf
- stali
- ...

However there are many more (IRC clients, file formats, presentation software, ...), check out their website.

## See Also

- less retarded software
  - reactionary software
  - bitreich
  - cat-v
- 

sudoku

## Sudoku

Sudoku is a puzzle that's based on filling a grid with numbers that is hugely popular even among normies such as grandmas and grandpas who find this stuff in magazines for elderly people. The goal is to fill in all squares of a 9x9 grid, prefilled with a few clue digits, with digits 1 to 9 so that no digit repeats in any column, row and 3x3 subgrid. It is like a crosswords puzzle for people who lack general knowledge, but it's also pretty suckless, pure logic-based puzzle whose generation and solving can be relatively easily automatized (unlike generating crosswords which requires some big databases). The puzzle is a pretty fun singleplayer game, posing opportunities for nice mathematical research and analysis as well as a comfy programming exercise. Sudokus are a bit similar to magic squares. There also exist many similar kinds of puzzles that work on the principle of filling a grid so as to satisfy certain rules given initial clues, many of these are implemented e.g. in Simon Tatham's Portable Puzzle Collection.

Curiously sudoku has its origins in agricultural designs in which people wanted to lay out fields of different plants in more or less uniform distributions (or something like that, there are some papers about this from 1950s). The puzzle itself became popular in Japan in about 1980s and experienced a boom of popularity in the western world some time after 2000 (similar Asian puzzle boom was historically seen e.g. with tangram).

The following is an example of a sudoku puzzle with only the initial clues given:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   | 3 | 1 |   | 5 | 7 |   | 6 |   |
|   |   |   | 9 |   | 8 |   | 4 |   |
| 4 | 7 | 8 | 6 |   | 2 | 1 |   | 5 |
| 7 |   | 5 |   | 6 |   | 4 |   |   |
|   |   | 6 |   | 8 | 1 | 7 | 2 |   |
|   |   |   | 7 |   | 3 | 6 | 5 |   |
| 5 | 6 |   |   | 9 |   |   |   | 2 |
|   |   |   | 1 |   | 5 | 9 |   | 6 |
|   |   | 3 | 8 | 2 | 6 |   |   | 4 |

The solution to the above is:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 9 | 3 | 1 | 4 | 5 | 7 | 2 | 6 | 8 |
| 6 | 5 | 2 | 9 | 1 | 8 | 3 | 4 | 7 |
| 4 | 7 | 8 | 6 | 3 | 2 | 1 | 9 | 5 |
| 7 | 1 | 5 | 2 | 6 | 9 | 4 | 8 | 3 |
| 3 | 4 | 6 | 5 | 8 | 1 | 7 | 2 | 9 |
| 2 | 8 | 9 | 7 | 4 | 3 | 6 | 5 | 1 |
| 5 | 6 | 7 | 3 | 9 | 4 | 8 | 1 | 2 |
| 8 | 2 | 4 | 1 | 7 | 5 | 9 | 3 | 6 |
| 1 | 9 | 3 | 8 | 2 | 6 | 5 | 7 | 4 |

We can see neither digit in the solution repeats in any column, row and any of the 9 marked 3x3 subgrids or, in other words, the digits 1 to 9 appear in each column, row and subgrid exactly once. These are basically the whole rules.

**We generally want a sudoku puzzle to have initial clues such that there is exactly one possible (unique) solution.** For this sudoku has to have at least 17 clues (this was proven by a computer). Why do we want this? Probably because in the puzzle world it is simply nice to have a unique solution so that human solvers can check whether they got it right at the back page of the magazine. This constraint is also mathematically more interesting.

**How many possible sudokus are there?** Well, this depends on how we view the problem: let's call one sudoku one grid completely filled according to the rules of sudoku. Now if we consider all possible such grids, there are 6670903752021072936960 of them. However some of these grids are "basically the same" because we can e.g. swap all 3s and 5s in any grid and we get basically the same thing as digits are nothing more than symbols here. We can also e.g. flip the grid horizontally and it's basically the same. If we take such things into account, there remain "only" 5472730538 essentially different sudokus.

Sudoku puzzles are sometimes assigned a difficulty rating that is based e.g. on the techniques required for its solving.

Of course there exist variants of sudoku, e.g. with different grid sizes, extended to 3D, different constraints on placing the numbers etc.

## Solving Sudoku

There are two topics to address: solving sudoku by people and solving sudoku by computers.

Humans almost exclusively use logical reasoning techniques to solve sudoku, which include:

- **scanning:** We take a look at some frequently appearing number in the grid and see which columns and rows they intersect which implies they cannot be placed in those columns and rows, possibly revealing the only possible location to place such number.
- **single remaining candidate:** When there is only one number left to fill in any column, row or subgrid, it is always clear which one it is and can be safely placed.
- **candidate sets:** A more advanced technique in which we create sets of possible candidate numbers for each square on the grid e.g. by writing tiny numbers in the top corners of the squares. We then apply various reasoning to reduce those sets, i.e. remove candidate numbers, until a single candidate remains for a certain square in which case we can fill in that number with certainty. This will further help us reason about candidates in other squares.
- **set equivalence properties:** Sudoku squares have some nice properties, it can e.g. easily be proven that some set of squares will always contain the same values as another set of squares -- this is quite easy to use, you just have to remember the rules that hold. See below.
- **advanced techniques:** There are quite a lot more advanced and expert level techniques like X Wings, Alternating Inference Chains and many more, described e.g. at <http://zitowolf.net/sudoku/strategy.html>. { TBH no idea what this is. ~drummyfish }

Relatively recently (sometime in 2020s) there was a quite huge discovery/highlight of so called **Phistomefel ring** -- this is an area on the sudoku board that will always contain the same values as another area, which can greatly help in finding solutions (and also in generating sudokus). Consider the following patterns:

|   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|--|---|---|---|---|---|---|---|---|---|
| B | B | . | . | . | . | B | B |  | . | . | . | C | C | C | . | . | . |
| B | B | . | . | . | . | B | B |  | . | . | . | C | C | C | . | . | . |
| . | . | A | A | A | A | . | . |  | D | D | D | . | . | . | D | D | D |
| . | . | A | . | . | . | A | . |  | D | D | D | . | . | . | D | D | D |
| . | . | A | . | . | . | A | . |  | . | . | . | C | C | C | . | . | . |
| . | . | A | . | . | . | A | . |  | . | . | . | C | C | C | . | . | . |
| . | . | A | A | A | A | . | . |  | . | . | . | . | . | . | . | . | . |
| B | B | . | . | . | . | B | B |  | . | . | . | . | . | . | . | . | . |
| B | B | . | . | . | . | B | B |  | . | . | . | . | . | . | . | . | . |

On the left we see the Phistomefel ring -- the set of *A* squares (the ring) will always contain the same values as the set of *B* squares! (Check it on our example sudoku above.) That's it, it's pretty simple, and it's simple to prove too (quickly: consider set  $S1 = \text{row3} + \text{row7} + 3 \times 3\text{square4} + 3 \times 3\text{square6}$ , and  $S2 = \text{column1} + \text{column2} + \text{column8} + \text{column9}$ ; it can be seen that  $S1$  and  $S2$  contain the same values; now remove from both sets their intersection -- we have removed the same values from both sets so they still contain the same values, set  $S1$  is now the Phistomefel ring,  $S2$  are the corners). A nice thing is that you can find more such relationship just using the simple idea of manipulating sets (the other example, values in sets *C* and *D* also have to be the same etc.).

For computers the traditional 9x9 sudoku is nowadays pretty easy to solve, however solving an  $N \times N$  sudoku is an NP complete problem, i.e. there most likely doesn't exist a "fast" algorithm for solving a generalized  $N \times N$  sudoku, even though the common 9x9 variant can still be solved pretty quickly with today's computers by using some kind of "smart" brute force, for example backtracking (or another state tree search) which recursively tries all possibilities and at any violation of the rules gets one step back to change the previous number. Besides this a computer can of course use all the reasoning techniques that humans use such as creating sets of possible values for each square and reducing those sets until only one possibility stays. The approach of reasoning and brute forcing may also be combined: first apply the former and when stuck fall back to the latter.

## Generating Sudoku

{ I haven't personally tested these methods yet, I'm just writing what I've read on some web pages and ideas that come to my mind. ~drummyfish }

Generating sudoku puzzles is non-trivial. There are potentially many different algorithms to do it, here we just foreshadow some common simple approaches.

Note that during generation of the sudoku you may utilize the knowledge of some inherent relationship between squares, e.g. the above mentioned Phistomefel ring.

Firstly we need to have implemented basic code for checking the validity of a grid and also some automatic solver, e.g. based on backtracking.

For generating a sudoku we usually start with a completely filled grid and keep removing numbers to leave only a few ones that become the initial clues. For this we have to know how to generate the solved grids. Dumb brute force (i.e. generating completely random grids and testing their validity) won't work here as the probability of finding a valid grid this way is astronomically low (seems around  $10^{(-56)}$ ). What may work is to randomly fill a few squares so that they don't break the rules and then apply our solver to fill in the rest of the squares. Yet a simpler way may be to have a database of a few hand-made grids, then we pick on of them and apply some transformations that keep the validity of the grid which include swapping any two columns, swapping any two rows, transposing, flipping the grid, rotating it 90 degrees or swapping any two digits (e.g. swap all 7s with all 9s).

With having a completely filled grid generating a non-unique (more than one solution) sudoku puzzle is trivial -- just take some completely filled grid and remove a few numbers. But as stated, we usually don't want non-unique sudokus.

For a unique solution sudoku we have to check there still exists exactly one solution after removing any numbers from the grid, for which we can again use our solver. Of course we should optimize this process by quitting the check after finding more than one solution, we don't need to know the exact count of the solutions, only whether it differs from one.



The matter of generating sudokus is further complicated by taking into account the difficulty rating of the puzzle.

## Code

Here is a C code that solves sudoku with brute force (note that for too many empty squares it won't be usable as it might run for years):

```
#include <stdio.h>

char sudoku[9 * 9] = // 0s for empty squares
{
    9, 3, 1, 0, 5, 7, 2, 6, 0,
    6, 5, 0, 9, 1, 8, 3, 4, 7,
    4, 7, 8, 6, 3, 2, 1, 9, 5,

    7, 1, 5, 2, 6, 0, 4, 8, 3,
    3, 0, 6, 5, 8, 1, 7, 2, 9,
    2, 8, 9, 7, 0, 3, 6, 5, 1,

    5, 6, 7, 3, 9, 0, 8, 1, 2,
    8, 2, 0, 1, 7, 5, 9, 3, 6,
    1, 9, 3, 8, 2, 6, 5, 0, 4
};

void print(void)
{
    puts("-----");

    for (int i = 0; i < 9 * 9; ++i)
    {
        putchar('0' + sudoku[i]);
        putchar(i % 9 != 8 ? ' ' : '\n');
    }
}

int isValid(void) // checks if whole sudoku is valid
{
    for (int i = 0; i < 9; ++i)
    {
        unsigned int m1 = 0, m2 = 0, m3 = 0; // bit masks of each group

        char *s1 = sudoku + i, // column
              *s2 = sudoku + i * 9, // row
              *s3 = sudoku + (i / 3) * (3 * 3 * 3) + (i % 3) * 3; // square

        for (int j = 0; j < 9; ++j)
        {
            m1 |= (1 << (*s1));
            m2 |= (1 << (*s2));
            m3 |= (1 << (*s3));

            s1 += 9;
            s2 += 1;
            s3 += (j % 3 != 2) ? 1 : 7;
        }

        if ((m1 != m2) || (m1 != m3) || (m1 != 0x03fe)) // all must be 111111110
            return 0;
    }

    return 1;
}

int printCounter = 0;

int solve(void) // find first empty square and brute forces all values on it
{
    char *square = sudoku;
```

```

printCounter++;

if (printCounter % 512 == 0) // just to limit printing speed
    print();

for (int j = 0; j < 9 * 9; ++j, ++square) // find first empty square
    if (!(*square)) // empty square?
    {
        while (1) // try all possible values in the square
        {
            *square = ((*square) + 1) % 10;

            if (!(*square)) // overflow to 0 => we tried all values now
                break;

            if (solve()) // recursively solve the next empty square
                return 1;
        }

        return 0; // no value led to solution => can't be solved
    }

// no empty square found, the sudoku is filled
return isValid();
}

int main(void)
{
    /* Here we could do some initial attempts at reasoning and filling in
    digits by "logic" before getting to brute force -- with too many empty
    squares brute force will take forever. However this is left as an
    exercise :-) */

    int success = solve();
    print();
    puts(success ? "solved" : "couldn't solve it");

    return 0;
}

```

## See Also

- [sudo](#)

---

suicide

## Suicide

{ I hate disclaimers but I'm not advising you to commit fucking suicide, OK? I mean it's an option and sometimes it's the best option, but I want you to live if it's at least a little possible -- remember, LRS loves all life and all life is precious. We will all die, no need to rush it. Also if you're feeling like shit you can send me a mail, we can talk. ~drummyfish }

Suicide is when someone voluntarily kills himself. Suicide offers an immediate escape from capitalism and is therefore a kind of last-resort hope; it is one of the last remaining freedoms in this world, even though capitalists can't profit from dead people and so are working hard on preventing people from killing themselves (rather than trying to make them NOT WANT TO kill themselves of course).

TODO: methods, add suicide by internet/free speech :D

For SCIENCE RESEARCHERS: there is a text file describing suicide methods at <http://textfiles.com/fun/suicide.txt>. One site related to suicide that's being censored because it discussed methods of killing oneself is called **sanctioned suicide** (<https://sanctioned-suicide.net>).

---

SW

# Software

Software (SW) are programs running on a computer, i.e. its non-physical parts (as opposed to hardware); for example an operating system, the Internet browser, games etc. Software is created by the act of programming (and related activities such as software engineering etc.).

Usually we can pretty clearly say what is software vs what is hardware, however there are also edge cases where it's debatable. Normally software is that about the computer which *can relatively easily be changed* (i.e. reinstalled by a typing a few commands or clicking a few buttons) while hardware is hard-wired, difficult to modify and not expected or designed to be modified. Nevertheless e.g. some firmware is kind of software in form of instructions which is however many times installed in some special kind of memory that's difficult to reprogram and not expected to be reprogrammed often -- some software may be "burned in" into a circuit so that it could only be changed by physically rewiring the circuit (the ME spyware in Intel CPUs has a built-in minix operating system). And this is where it may on occasion become difficult to judge where the line is to be drawn. This issue is encountered e.g. by the FSF which certifies some hardware that works with free software as *Respects Your Freedom* (RYF), and they have very specific definition what to them classifies software.

## See Also

- algorithm
- 

sw\_rendering

## Software Rendering

Software (SW) rendering refers to rendering computer graphics without the help of graphics card (GPU), or in other words computing images only with CPU. Most commonly the term means rendering 3D graphics but may as well refer to other sorts of graphics such as drawing fonts or video. Before the invention of GPU card all rendering was done in software of course -- games such as Quake or Thief were designed with SW rendering and only added optional GPU acceleration later. SW rendering for traditional 3D graphics is also called software rasterization, for rasterization is the basis of current real-time 3D graphics.

SW rendering has advantages and disadvantages, though from our point of view its advantages prevail (at least given only capitalist GPUs exist nowadays). Firstly it is **much slower** than GPU graphics -- GPUs are designed to perform graphics-specific operations very quickly and, more importantly, they can process many pixels (and other elements) in parallel, while a CPU has to compute pixels sequentially one by one and that in addition to all other computations it is otherwise performing. This causes a much lower FPS in SW rendering. For this reasons SW rendering is also normally of **lower quality** (lower resolution, nearest neighbour texture filtering, ...) to allow workable FPS. Nevertheless thanks to the ginormous speeds of today's CPUs simple fullscreen SW rendering can be pretty fast on PCs and achieve even above 60 FPS; on slower CPUs (typically embedded) SW rendering is usable normally at around 30 FPS if resolutions are kept small.

On the other hand SW rendering is more portable (as it can be written purely in a portable language such as C), less bloated and **eliminates the dependency on GPU** so it will be supported almost anywhere as every computer has a CPU, while not all computers (such as embedded devices) have a GPU (or, if they have it, it may not be sufficient, supported or have a required driver). SW rendering may also be implemented in a simpler way and it may be easier to deal with as there is e.g. no need to write shaders in a special language, manage transfer of data between CPU and GPU or deal with parallel programming. SW rendering is the KISS approach.

SW rendering may also utilize a much wider variety of rendering techniques than only 3D rasterization traditionally used with GPUs and their APIs, thanks to not being limited by hard-wired pipelines, i.e. it is more flexible. This may include splatting, raytracing or BSP rendering (and many other "pseudo 3D" techniques).

A lot of software and rendering frameworks offer both options: accelerated rendering using GPU and SW rendering as a fallback (in case the first option is not possible). Sometimes there exists a rendering API that has both an accelerated and software implementation (e.g. TinyGL for OpenGL).

For simpler and even somewhat more complex graphics **purely software rendering is mostly the best choice**. LRS suggests you prefer this kind of rendering for its simplicity and portability, at least as one possible option. On devices with lower resolution not many pixels need to be computed so SW rendering can actually be pretty fast despite low specs, and on "big" computers there is nowadays usually an extremely fast CPU available that can handle comfortable FPS at higher resolutions. There is a LRS software renderer you can use: small3dlib.

SW renderers are also written for the purpose of verifying rendering hardware, i.e. as a reference implementation.

Note that SW rendering doesn't mean our program is never touching GPU at all, in fact most personal computers nowadays **require** some kind of GPU to even display anything. SW rendering only means that computation of the image to be displayed doesn't use any hardware specialized for this purpose.

Some SW renderers make use of specialized CPU instructions such as MMX which can make SW rendering faster thanks to handling multiple data in a single step. This is kind of a mid way: it is not using a GPU per se but only a mild form of hardware acceleration. The speed won't reach that of a GPU but will outperform a "pure" SW renderer. However the disadvantage of a hardware dependency is still present: the CPU has to support the MMX instruction set. Good renderers only use these instructions optionally and fall back to general implementation in case MMX is not supported.

## Programming A Software Rasterizer

{ In case small3dlib is somehow not enough for you :) ~drummyfish }

Difficulty of this task depends on features you want -- a super simple flat shaded (no textures, no smooth shading) renderer is relatively easy to make, especially if you don't need movable camera, can afford to use floating point etc. See the details of 3D rendering, especially how the GPU pipelines work, and try to imitate them in software. The core of these renderers is the **triangle rasterization** algorithm which, if you want, can be very simple -- even a naive one will give workable results -- or pretty complex and advanced, using various optimizations and things such as the top-left rule to guarantee no holes and overlaps of triangles. Remember this function will likely be the performance bottleneck of your renderer so you want to put effort into optimizing it to achieve good FPS. Once you have triangle rasterization, you can draw 3D models which consist of vertices (points in 3D space) and triangles between these vertices (it's very simple to load simple 3D models e.g. from the obj format) -- you simply project (using perspective) 3D position of each vertex to screen coordinates and draw triangles between these pixels with the rasterization algorithm. Here you need to also solve visibility, i.e. possible overlap of triangles on the screen and correctly drawing those nearer the view in front of those that are further away -- a very simple solution is a z buffer, but to save memory you can also e.g. sort the triangles by distance and draw them back-to-front (painter's algorithm). You may add a scene data structure that can hold multiple models to be rendered. If you additionally want to have movable camera and models that can be transformed (moved, rotated, scaled, ...), you will additionally need to look into some linear algebra and transform matrices that allow to efficiently compute positions of vertices of a transformed model against a transformed camera -- you do this the same way as basically all other 3D engines (look up e.g. some OpenGL tutorials, see model/view/projection matrices etc.). If you also want texturing, the matters get again a bit more complicated, you need to compute barycentric coordinates (special coordinates within a triangle) as you're rasterizing the triangle, and possibly apply perspective correction (otherwise you'll be seeing distortions). You then map the barycentrics of each rasterized pixel to UV (texturing) coordinates which you use to retrieve specific pixels from a texture. On top of all this you may start adding all the advanced features of typical engines such as acceleration structures that for example discard models that are completely out of view, LOD, instancing, MIP maps and so on.

Possible tricks, cheats and optimizations you may utilize include:

- Using painter's algorithm (sorting triangles and drawing back to front) instead of z-buffer if you need to save a lot of RAM. But remember sorting doesn't work perfectly, glitches will inevitably appear, and you will probably gain overdraw penalty.
- Ad previous point: you don't have to perform whole triangle sorting each frame if you need to save speed, it may be good enough to perform a constant continuous sorting by performing only a few iterations of some sorting algorithm per frame.

- You may lower the quality of far-away objects in many ways, e.g. with LOD, only using affine texturing for them (as opposed to perspective-correct one) or even just using a constant color (average color of the texture), maybe even just drawing 2D sprites instead of 3D models etc. This may help a lot.
- Try to reduce overdraw (overwriting already rendered pixels with new closer ones) which wastes computation time. This can be achieved by good culling of obscured objects or by using z-buffer along with front to back drawing.
- Generally use cheap approximations such as Gouraud (per-vertex) shading instead of Phong (per-pixel), nearest neighbour texture sampling, only approximate perspective correction (every N pixels), simplified handling of near-plane culling (e.g. just pushing the vertices in front of camera instead of actually culling a triangle) etc.
- Use general optimization techniques: e.g. precomputation, using power of two resolution for textures, fixed screen resolution that's known at compile time or inlining of your shader function will probably help performance.
- TODO: MORE

## Specific Renderers

These are some notable software renderers:

- **Build engine:** So called "pseudo 3D" or primitive 3D, this was a very popular proprietary portal-rendering engine for older games like Duke Nukem 3D or Blood.
- **BRender:** Old commercial renderer used in games such as Carmageddon, Croc or Harry Potter 1. Later made FOSS.
- **Chasm: The Rift engine:** Mysterious proprietary 1997 renderer made specifically for one game, notable especially by being a hybrid of "2.5D" and "true 3D", it managed to make it look very good.
- **Dark Engine:** Old proprietary game engine which includes a SW renderer, used mainly in the game Thief. The author writes about it at [https://nothings.org/gamedev/thief\\_rendering.html](https://nothings.org/gamedev/thief_rendering.html).
- **Descent engine:** The 1995 proprietary game Descent featured one of the first real time "true 3D" engines based on portal rendering, it still stands as a marble of that time's technology.
- **id Tech:** Multiple engines by Id software (later made FOSS) used for games like Doom, Quake and its successors included a software renderer. Quake's SW renderer was partially described in the *Michael Abrash's Graphics Programming Black Book*, Doom's renderer is described e.g. in the book *Game Engine Black Book DOOM*.
- **Irrlich:** FOSS game engine including a software renderer as one of its backends.
- **Jedi:** Old proprietary "pseudo3D" engine.
- **Mesa:** FOSS implementation of OpenGL that includes a software rasterizer.
- **raycastlib:** LRS, free C 2D raycasting ("2.5D") engine most notably used in Anarch.
- **small3dlib:** LRS, free pure C "true 3D" rasterizer, very simple but flexible and coming with all the high level features (textures, perspective correction etc.).
- **SSRE:** The guy who wrote LIL also made this renderer named Shitty Software Rendering Engine, accessible [here](#).
- **System Shock engine:** Old proprietary game engine.
- **TinyGL:** Implements a subset of OpenGL.
- **Tomb Raider:** Famous 90s game with custom software 3D renderer.
- **Ultima underworld:** Proprietary game featuring a very early (1992) texture mapped software 3D renderer.
- **old Unreal Engine:** One of the most mainstream popular proprietary engines nowadays featured software rendering fallbacks in early versions.
- In general many old games in the 90s implemented their own software renderers. Also games on non-3D consoles such as Gameboy Advance sometimes attempted simple software rendering 3D. These are the places where you can look for interesting renderers of this kind.
- ...

## See Also

- 3D rendering
- "pseudo/primitive 3D, 2.5D"

# Systemd

Systemd, also shitstemd, is a horribly disastrous bloated, anti-Unix, "FOSS" "software suite" used for initialization of an operating system and handling services like logging in or managing network connections. It is a so called PID 1 process, or an init system. Systemd has been highly criticised by the proponents of suckless and LRS and even normies for its enormous amount of bloat, ugliness, anti-Unix-philosophy design, feature creep, security vulnerabilities and other stuff. Unfortunately it is being adopted by many GNU/Linux distributions including Arch Linux and Debian. Some distros such as Devuan just said no to this shit and forked to a non-systemd version.

Systemd was born when Harry Pot... ummm Lennart Poettering had an unprotected gay sex with Kay Sievers.

For more detailed bashing of systemd see e.g. <https://nosystemd.org/>. The site sums up systemd with a fitting quote: "*If this is the solution, I want my problem back*". Another sum up by suckless: <http://suckless.org/sucks/systemd/>. There is also e.g. <https://sysdfree.wordpress.com/>.

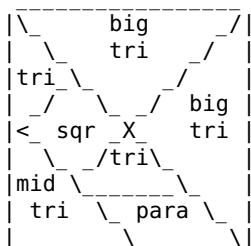
---

tangram

## Tangram

{ I made a simple tangram game in SAE, look it up if you want to play some tangram. ~drummyfish }

Tangram is a simple, yet greatly amusing old puzzle game in which the player tries to compose a given shape (of which only silhouette is seen) out of given basic geometric shapes such as triangles and squares. It is a rearrangement puzzle. Many thousands of shapes can be created from just a few geometric shapes, some looking like animals, people and man made objects. This kind of puzzles have been known for a long time -- the oldest recorded tangram is Archimedes' box (square divided into 14 pieces), over 2000 years old. In general any such puzzle is called tangram, i.e. it is seen as a family of puzzle games, however tangram may also stand for **modern tangram**, a one with 7 polygons which comes from 18th century China and which then became very popular also in the west and even caused a so called "tangram craze" around the year 1818. Unless mentioned otherwise, we will talk about this modern version from now on.

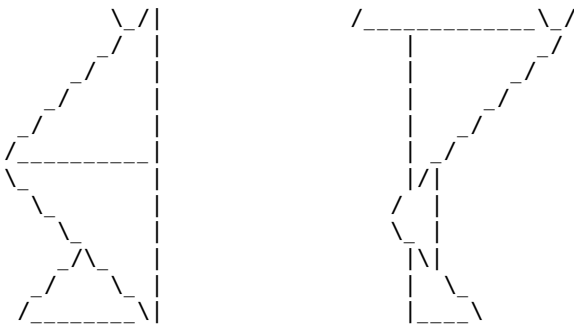


*Divide square like this to get the 7 tangram pieces. Note that the parallelogram is allowed to be flipped when creating shapes as it has no mirror symmetry (while all other shapes do).*

LRS considers tangram to be **one of the best games** as it is extremely simple to make and learn, it has practically no dependencies (computers, electricity, ... one probably doesn't even have to have the sense of sight), yet it offers countless hours of fun and allows deep insight, there is art in coming up with new shapes, math in counting possibilities, good exercise in trying to program the game etc.

Tangram usually comes as a box with the 7 pieces and a number of cards with shapes for the player to solve. Each card has on its back side a solution. Some shape are easy to solve, some are very difficult.





Two tangram shapes: bunny and stork (from 1917 book *Amusements in Mathematics*).

{ I found tangram to be a nice practice for **letting go of ideas** -- sometimes you've got an almost complete solution that looks just beautiful, it looks like THE only one that just has to be it, but you can't quite fit the last pieces. I learned that many times I just have to let go of it, destroy it and start over, usually there is a different, even more beautiful solution. This experience may carry over to practical life, e.g. programming. ~drummyfish }

**Can tangram shapes be copyrighted?** As always nothing is 100% clear in law, but it seems many tangram shapes are so simple to not pass the threshold of originality for copyright. Furthermore tangram is old and many shapes have been published centuries ago, making them public domain, i.e. if you find some old, public domain book (e.g. the book *The Fashionable Chinese Puzzle, Amusement in Mathematics* or *Ch'i ch'iao hsin p'u: ch'i chiao t'u chieh*) with the shape you want to use, you're most definitely safe to use it. HOWEVER watch out, a collection of shapes, their ordering and/or shapes including combinations of colors etc. may be considered non-trivial enough to spawn copyright (just as collections of colors may be copyrightable despite individual colors not being copyrightable), so do NOT copy whole shape collections.

**Tangram paradoxes** are an interesting discovery of this game -- a paradox is a shape that looks like another shape with added or subtracted piece(s), despite both being composed of the same pieces. Of course geometrically this isn't possible, the missing/extra area is always compensated somewhere, but to a human eye this may be hard to spot (see also infinite chocolate). New players get confused when they encounter a paradox for the first time, they think they solved the problem but are missing a piece, or have an extra one, while in fact they just made a wrong shape. TODO: example

### Tips for solving:

- Start by placing pieces you know for certain where they belong (small details that can only be made with the smallest pieces, pieces you deduce that HAVE to be somewhere etc.). This reduces the problem to making a smaller shape from fewer pieces, making it much easier to solve. But BEWARE: sometimes you wrongfully assume some piece in some place because the silhouette "suggests" it, do not fall to this trap.
- At the beginning try to get a sense of scale, sometimes what in the silhouette looks like the big triangle may actually be the middle sized one etc.
- Learn some common patterns, e.g. you can make a rectangle out of the two small triangles and parallelogram. This comes with just solving more puzzles.
- If you have an "almost solution" and can't fit the last few pieces for some time, just destroy it and start over. There are many nicely looking blind paths.
- If in your partial solution you can replace some subshape composed of smaller pieces with the same subshape composed of one larger piece, do it. Having smaller pieces is preferable because you have more flexibility.
- Be careful to make the exact shape you see, sometimes it is possible to make a very similar looking shape that has just a tiny bit different proportions e.g. by rotating the parallelogram.
- ...

TODO: some PD shapes, math, stats, ...

---

tas

# Tool Assisted Speedrun

Tool assisted speedrun (TAS, also more generally *tool assisted superplay*) is a category of game speedruns in which help of any tools is allowed, even those that would otherwise be considered cheating, e.g. scripts, savestates, aimbots, AI or time manipulation, however NOT those that alter the game itself. In other words the game rules stay intact, we just try to boost the player's skill to superhuman levels. This makes it possible to create flawless, perfect or near-perfect runs which can serve as a theoretical upper limit for what is achievable by humans -- and of course TAS runs are extremely fun to watch, you just see the player making perfectly timed and planned actions, 100% accurate head shots etc. The normal, non-TAS runs are called RTA (real time attack). For example the current (2022) RTA world record of Super Mario Bros is 4.58.881 while the TAS record is 4.41.27 (here we can see the RTA run is very optimized already, in less popular games a TAS can be orders of magnitude faster).

{ Watching a TAS run is kind of like watching the God play the game. I personally like to watch Trackmania TASes, some are really unbelievable. Elastomania and Doom TASes are also pretty fucked up. Also note that SAF games and Anarch have TAS support. ~drummyfish }

There is a website with videos of game TASes: <https://tasvideos.org/>.

TAS does NOT allow hacking the game in other ways than what's possible to achieve by simply playing the game, i.e. it is not possible to hex edit the game's code before running it or manipulate its RAM content at run time with external tools. However note that some games are buggy and allow things such as altering their RAM content or code by merely playing the game (e.g. Pokemon Yellow allows so called arbitrary code execution) which generally IS allowed. The goal of TAS is merely to find, as best as we can, the series of game inputs that will lead to completing the game as fast as possible. For this the game pretty much needs to be deterministic, i.e. the same sequence of inputs must always reproduce the same run when replayed later.

TAS runs coexist alongside RTA (non-TAS) runs as separate categories that are beneficial to each other: RTA runners come up with speedrunning techniques that TAS programmers can perfectly execute and vice versa, TAS runners many times discover new techniques and ideas for RTA runners (for example the insane discovery of groundbreaking noseboost when TAS was introduced to Trackmania). In fact RTA and TAS runners are many times the very same people. Of course if you submit a TAS run in RTA category, you'll be seen as a cheater.

Creating a TAS is not an easy task, it requires great knowledge of the game (many times including its code) and its speedrunning, as well as a lot of patience and often collaboration with other TASers, sometimes a TASer needs to also do some programming etc. TASes are made *offline* (not in real time), i.e. hours of work are required to program minutes or even seconds of the actual run. Many paths need to be planned and checked. Compared to RTAs, the focus switches from mechanical skills towards skillful mathematical analysis and planning. While RTA runs besides skill and training also require risk planning, i.e. sometimes deciding to do something in a slower but safer way to not ruin a good run, TAS can simply go for all the fastest routes, no matter how risky they are, as there is certainty they will succeed. Besides this some technological prerequisites are necessary: the actual tools to assist with creation of the TAS. For many new proprietary games it is extremely difficult to develop the necessary tools as their source code isn't available, their assembly is obscured and littered with "anti-cheating" malware. Many "modern" (even FOSS) games are additionally badly programmed and e.g. lacking a deterministic physics, which makes precise TASing almost impossible (as the traditional precise crafting of inputs requires deterministic behavior). The situation is better with old games that are played in emulators such as DOS games (Doom etc.) or games for consoles like GameBoy -- emulators can give us a complete control over the environment, they allow to save and load the whole emulator state at any instant, we may slow the time down arbitrarily, rewind and script the inputs however we wish (an advanced technique includes e.g. bruteforcing: exhaustively checking all possible combinations of inputs over the following few frames to see which one produces the best time save). In games that don't have TAS tools people at least try to do the next best thing with **segmented speedruns** (e.g. stitching together world record runs of each game level).

A libre game (under CC0!) called Lix, a clone of Lemmings is kind of based on making TAS runs, and it's excellent! In the game, like in original Lemmings, one has to manage a group of units to cooperate in overcoming obstacles and so get safely to the level exit; however, unlike Lemmings, Lix incorporates a



replay system so the player may not just pause the game, accelerate or slow down the time, but also rewind back and issue commands perfectly on any any given frame. The game also shows to the player all necessary info like exact frame number, exact survivable jump height etc., so winning a level doesn't depend on fast reaction time, good estimate or grinding attempts over and over until one doesn't make any mistake -- no, solving the level is purely about thinking and finding the mathematical solution. Once one knows how to get to the exit, it's easy to program in any complex sequence of actions, and of course then he can rewatch it in real time and get this kind of rewarding movie in which everything is performed perfectly. Lix is really an excellent example of how TAS is not just 3rd party hacking of the game but inherent part of the original game's design, one that takes the fun to the next level.

There also exists a term *tool assisted superplay* which is the same principle as TAS but basically with the intention of just flexing, without the goal of finishing the game fast (e.g. playing a Doom level against hundreds of enemies without taking a single hit).

Some idiots are against TASes for various reasons, mostly out of fear that TASers will use the tools to CHEAAAAAAT in RTAs or that TASes will make the human runners obsolete etc. That's all bullshit of course, it's like being against computers out of fear they would make human calculators obsolete. Furthermore TASes always coexist perfectly peacefully with RTA runs as can e.g. be seen in the case of Trackmania -- in 2021 TAS tools started to appear for Trackmania and many people feared it would kill the game's competition, however after the release of the tools no such disaster happened, TAS became hugely popular and now everyone loves it, human competition happily continues, plus the development of the tools actually helped uncover many cheaters among the top players (especially Riolu who was forced to leave the scene, this caused a nice drama in the community).

We could even go as far as to say that morally TAS is the superior way of speedrunning as it puts humans in the role of thinkers rather than treating them as wannabe machines who waste enormous amounts of time on grinding real time runs with arbitrary obstacles (such as requiring a run to not be spliced etc.), which a real machine can simply do instantly and perfectly. There is really no point in someone spending 10000 hours of life on getting lucky and nailing a series of frame perfect keypresses when a computer can do this in 1 second.

---

tattoo

## Tattoo

Tattoo is a body disfigurement formed by injecting ink under the skin to permanently mark it. Tattoo, similarly to piercing, suits, dyed hair etc., is a sign of egoism, narcissism, herd mentality, identity crisis, overconfidence of the incompetent and a cheap attempt at desperately trying to get attention or make oneself look interesting. We highly advise to distance oneself from anyone having a voluntarily made tattoo.

---

tech

## Tech

Tech is a short for technology.

---

technology

## Technology

Technology (from Greek tekhnologia, "systematic treatment of art", also just "tech") encompasses tools and knowledge of making such tools invented and achieved mainly through science and by long systematic effort. This includes everything from stone tools to space rockets and artificial intelligence. On the Internet, as well as on this Wiki, this term is commonly used with an increased focus on computer technology, i.e. hardware and software, for this is the kind of technology that is being discussed and developed the most these days. Let it be said that technology, like fire, should serve us, but can also be dangerous and often gets misused and abused.

**The foremost purpose of technology is to make people not have to work** -- see also progress. Proponents of dystopian societies, such as capitalists, fear that technology will "take people's work" -- such people are for sure greatly idiotic and often end up abusing technology in the completely opposite manner: for enslaving and oppressing people. Proponents of good technology strive to make technology do work for humans so that people can actually live happy lives and do what they want. With this in mind we have to remember that **one of the most important concepts in technology is minimalism**, as that is a necessary prerequisite for technological freedom.

**Knowledge of older technology gets lost extremely quickly in society** -- this is a crucial realization that follows a naive idea of the young man who by his inexperience believes that we somehow pertain knowledge of all technology that's been invented from dawn of man until today. In history our society has always only held knowledge of technology it was CURRENTLY ACTIVELY USING; knowledge of decades old technology no longer in use only stays in hands and heads of extremely few individuals and perhaps in some obscure books that ARE UNREADABLE to most, sometimes to none; yet older technology oftentimes gets forgotten for good. For instance renaissance had to largely reinvent many arts and sciences of making building and statues of antiquity because middle ages have simply forgotten them. A more recent example can be found at NASA and their efforts to recreate THEIR OWN old rocket engines: you would think that since they literally have detailed documentation of those engines, they'd be able to simply make them again, but that's not the case because the small undocumented (yet crucial) know-how of the people who built the engines decades ago was lost with those individuals who died or retired in the meanwhile; NASA had to start a ginormous project to reinvent its own relatively recent technology. The same is happening in the field of programming: modern soydevs just CANNOT create as efficient software as hackers back then as due to normalization of wasting computing resources they threw away the knowledge of optimization technique and wisdom in favor of bullshit such as "soft skills" and memorizing one billion genders and personal pronouns. One might naively think that e.g. since our agriculture is highly efficient and advanced due to all the immense complexity of our current machines, simple farming without machines would be a child's play for us, however the opposite is true: we no longer know how to farm without machines. If a collapse comes, we are quite simply fucked.

---

ted\_kaczynski

## Ted Kaczynski

*"The Industrial Revolution and its consequences have been a disaster for the human race."* --Ted Kaczynski

Ted Kaczynski (22.5.1942 - 10.6.2023, RIP), known as *Unabomber*, was an imprisoned American mathematician who lived a simple life in the nature, warned of the dangers of advanced technology and killed several people by mailing them bombs in order to bring attention to his manifesto that famously starts with the words "The Industrial Revolution and its consequences have been a disaster for the human race". Besides being one of the most famous mass murderers he is very well known in the tech community.

Ted was born in Chicago, US. As a kid he was very shy. He was also extremely smart (IQ measured at 167), skipped a few grades, graduated from Harvard at 20 years old and got a PhD at 25 at the University of Michigan. Then he became a professor at the University of California, until his resignation in 1969.

Fun fact: at one point he considered a gender change surgery.

In 1971 he moved to a remote cabin in the woods in Montana where he lived in a primitive way with no electricity or running water. He grew more and more disenchanted with the society, especially with its technology and how it's enslaving and destroying humanity. The last straw may have been the moment when a road was built nearby his cabin, in the middle of the nature he loved.

He started sending hand-made bombs to various universities and airports (hence the nickname *Unabomber*, *university and airline bomber*). He managed to kill 3 people and injured dozens of others. He was arrested on April 3, 1996 in his cabin. He got life imprisonment in court. He died in prison in 2023 after having been diagnosed with cancer, reports said he committed suicide.

# Manifesto

The manifesto is named *Industrial Society and Its Future*. In it he refers to his movement as a *Freedom Club* (FC). Let's start by summarizing it:

{ The following is a sum up according to how I personally understood it, there's most likely subjective bias but I did my best. ~drummyfish }

First he bashes "leftists", analyses their psychology and says they are kind of degenerate sheeple, characterized by low self esteem, inventing bullshit artificial issues (such as the issue of political correctness), sometimes using violence. He also criticizes conservatives for supporting technological and economical growth which in his view inevitably brings on shift in societal values and said degeneracy. The usual societal issues are presented such as bad mental health, people being slaves to the system, feeling powerless, having no security, no autonomy etc. The cause of unhappiness and other human issues is identified as people not being able to fulfill what he sees as a necessity for fulfilling life, so called *power process*, the process of considerable struggle towards a *real* goal that can be achieved such as obtaining food by hunting -- he argues nowadays it's "too easy" to satisfy these basic needs and people invent artificial "surrogate" activities (such as sports, activism and even science) to do to try to fulfill the power process, however he sees these artificial activities as harmful, not *real* goals. It is mentioned we only have freedom in unimportant aspects of life, the system controls and regulates everything, brainwashes people etc. He defines real freedom as the opportunity to go through the power process naturally and being in control of one's circumstances. It is talked a lot about modification of humans themselves, either by advanced psychological means (propaganda), drugs or genetic modification which is seen as a future danger. A number of principles by which society works is outlined and it is concluded that the industrial society can't be reformed, a revolution is needed (not necessarily violent). Ted argues the system needs to be destroyed, we have to get back to the nature, and for this revolution he outlines a plan and certain recommendations (creation of ideology for intellectuals and common folk, the necessity of the revolution being world-wide etc.). He ends with again bashing "leftism" and warns they must never be collaborated with.

Now Let us leave a few comments on the manifesto. Firstly we have to say the text is easy to read, well thought through and Ted makes some great points, many of which we completely agree on; this includes the overall notion of technology having had mostly negative effects on recent society, the pessimistic view of our future and the criticism of "harmful modern bullshit" such as political correctness. He analyzes and identifies some problems in society very well (e.g. the propaganda that's so advanced that even its creators aren't usually consciously aware they're creating propaganda, his analysis of the inner working of the system is spot on). Nevertheless we also **disagree on many points**. Firstly we use different terminology; people who Ted calls *leftist* and whom he accuses of degeneracy and harmfulness we call pseudoleftists, we believe in a truly leftist society (i.e. nonviolent, altruistic, non-censoring, loving without fascist tendencies). **We disagree on Ted's fundamental assumption** that people can't change, i.e. that people are primitive animals that need to live primitive lives (go through the power process by pursuing *real* goals such as obtaining food by hunting) in order to be happy (we are not against primitivism but we support it for other reasons). We believe society can become adult, just like an individual, if it is raised properly (i.e. with effort) and that the primitive side of a human can be overshadowed by the the intellectual side and that activities he calls *surrogate* (and considers undesirable) can be fulfilling. We think that in a sane, adult society **advanced technology can be helpful** and compatible with happy, fulfilling lives of people, even if the current situation is anything but. And of course, we are completely nonviolent and disagree with murdering people for any reason such as bringing attention to a manifesto.

## See Also

- Diogenes

---

teletext

## Teletext

Teletext is now pretty much obsolete technology that allowed broadcasting extremely simple read-only text/graphical pages along with TV signal so that people could browse them on their TVs. It was used mostly

in the 70s, 80s and 90s but with world wide web teletext pretty much died.

{ Just checked on my TV and it still works in 2022 here. For me teletext was something I could pretend was "the internet" when I was little and when we didn't have internet at home yet, it was very cool. Back then it took a while to load any page but I could read some basic news or even browse graphical logos for cell phones. Nowadays TVs have buffers and have all the pages loaded at any time so the browsing is instantaneous. ~drummyfish }

The principal difference against the Internet was that teletext was broadcast, i.e. it was a one-way communication. Users couldn't send back any data or even request any page, they could only wait and catch the pages that were broadcast by TV stations (this had advantages though, e.g. it couldn't be DDOSed and it couldn't spy on its users as they didn't send any information back). Each station would have its own teletext with fewer than 1000 pages -- the user would write a three place number of the page he wanted to load ("catch") and the TV would wait until that page was broadcast (this might have been around 30 seconds at most), then it would be displayed. The data about the pages were embedded into unused parts of the TV signal.

The pages allowed fixed-width text and some very blocky graphics, both could be colored with very few basic colors. It looked like something you render in a very primitive terminal.

## See Also

- videotex
  - world broadcast
- 

temple\_os

## Temple OS

Temple OS is a funny operating system made by a schizo guy Terry Davis who has become a meme and achieved legendary status for this creation in the Internet tech circles as it's extremely impressive that a single man creates such a complex OS and also the OS features and the whole context of its creation are quite funny. It has a website at <https://templeos.org>.

According to Terry, God commanded him to write TempleOS and guided him in the development: for example it was demanded that the resolution be 640x480. It is written in HolyC, Terry's own programming language. The OS comes with GUI, 2D and 3D library, games and even a program for communicating with God.

Notable Temple OS features and programs are:

- multitasking (non-preemptive)
- supported file systems: FAT32, ISO9660, RedSea (custom)
- HolyC compiler
- 2D/3D library
- oracle (communicate with God)
- games
- IDE supporting images and 3D models embedded in text

In his video blogs Terry talked about how technology became spoiled and that TempleOS is supposed to be simple and fun. For this and other reasons the OS is limited in many way, for example:

- no networking
- Only runs on x64.
- Only runs in 640x480 16 color display mode.
- single audio voice
- ring-0 only
- single address space
- multitasking is non-preemptive (programs have to yield CPU themselves)

Temple OS source code has over 100000 LOC. It is publicly available and said to be in the public domain, however there is no actual license/waiver in the repository besides some lines such as "100% public domain" which are legally questionable and likely ineffective (see licensing).

There still seems to be some people developing the OS and applications for it, e.g. Crunklord420.

## See Also

- Timecube
- Sonichu

---

tensor\_product

## Tensor Product

TODO

$a(x) b = [a_0 * b_0, a_0 * b_1, a_0 * b_2, \dots a_1 * b_0, a_1 * b_1, \dots a_n * b_0, a_n * b_1, \dots]$

---

terry\_davis

## Terry Davis

*"An idiot admires complexity, a genius admires simplicity."* --Terry Davis

Terry A. Davis, aka the *divine intellect*, born 1969 in Wisconsin, was a genius+schizophrenic programmer that singlehandedly created TempleOS in his own programming language called HolyC, and greatly entertained and enlightened an audience of followers until his tragic untimely death. For his programming skills and quality videos he became a legend and a meme in the tech circles, especially on 4chan which additionally valued his autistic and politically incorrect behavior.

He was convinced he could talk to God and that God commanded him to make an operating system with certain parameters such as 640x480 resolution, also known as the God resolution. According to himself he was gifted a *divine intellect* and was, in his own words, the "best programmer that ever lived". Terry was making YouTube talking/programming videos in which God was an often discussed topic, alongside valuable programming advice and a bit of good old racism. He was also convinced that the government was after him and often delved into the conspiracies against him, famously proclaiming that **"CIA niggers glow in the dark"** ("glowing in dark" subsequently caught on as a phrase used for anything suspicious). He was in mental hospital several times and later became homeless, but continued to post videos from his van. An entertaining fact is also that he fell in love with a famous female physics YouTuber Dianna Cowern which he stalked online. In 2018 he was killed by a train (officially a suicide but word has it CIA was involved) but he left behind tons of videos full of endless entertainment, and sometimes even genuine wisdom.

Terry, just as us, greatly valued simplicity and fun in programming, he was a low-level programmer and saw that technology went to shit and wanted to create something in the oldschool style, and he expressed his will to dedicate his creation to the public domain. This is of course extremely based and appreciated by us (though the actual public domain dedication wasn't executed according to our recommendations).

---

thrembo

## Thrembo

Thrembo (also hidden or forbidden number) is allegedly a "fictional" whole number lying between numbers 6 and 7, it's a subject of jokes and conspiracy theories with most people seeing it as a meme and others promoting its existence, it is represented by a Unicode symbol U+03EB. Thrembo originated as a schizo post on 4chan in 2021 by someone implying there's some kind of conspiracy to hide the existence of a number between 6 and 7, who was of course in turn advised to take his meds, however the meme has already been

started. Thrembo now even has its own subreddit (though it's extremely retarded, don't go there).

How can there be an integer between 6 and 7? Well, that's what thrembologists research. Sure, normally there is no space on the number line to fit a number between 6 and 7 so that its distance is 1 to both its neighbors, however this only holds due to simplifications we naively assume because of our limited IQ; one may for example imagine a curved, non Euclidean number line (:D) on which this is possible, just like we can draw a triangle with three right angles on a surface of a sphere. In history we've seen naysayer proven wrong in mathematics, for example those who claimed there is no solution to the equation  $x^2 = -1$ ; some chad just came and threw at us a new number called i (he sneakily made it just a letter so that he doesn't actually have to say how much it ACTUALLY equals), he just said "THIS NUMBR IS HENCEFORTH THE SOLUTION BECAUSE I SAY SO" and everyone was just forced to admit defeat because no one actually had a bigger authority than this guy. That's how real mathematics is done kids. As we see e.g. with political correctness, with enough propaganda anything can be force-made a fact, so if the number gets enough likes on twitter, it will just BE.

Some pressing questions about thrembo remaining to be researched are following. Is thrembo even of odd? Is it a prime? If such number can exist between 6 and 7, can similar numbers exist between other "mainstream" numbers?

## See Also

- noncomputable numbers
- schizophrenic number
- illegal number
- 42
- pi

---

throwaway\_script

## Throw Away Script

Throw away script is a script for one-time job which you "throw away" after its use. Such scripts may be ugly, badly written and don't have to follow LRS principles as their sole purpose is to quickly achieve something without any ambition to be good, future-proof, readable, reusable etc.

For example if you have a database in some old format and want to convert it to a new format, you write a throw away script to do the conversion, or when you're mocking an idea for a game, you write a quick throw away prototype in JavaScript to just test how the gameplay would feel.

For throw aways cripts it is acceptable and often preferable to use languages otherwise considered bad such as Python or JavaScript.

---

tinyphysicsengine

## Tinyphysicsengine

Tinyphysicsengine (TPE) is a very simple suckless/KISS physically inaccurate 3D physics engine made according to LRS principles (by drummyfish). Similarly to other LRS libraries such as small3dlib, smallchesslib, raycastlib etc., it is written in pure C with no dependencies (not even standard library) as a single header library, using only fixed point math, made to be efficient and tested on extremely small and weak devices such as Pokitto. It is completely public domain free software (CC0) and is written in fewer than 3500 lines of code. TPE got some attention even on hacker news where people kind of appreciated it and liked it. { Until they found my website lol. Just to clarify I did not post it to HN myself, I was surprised to find an email that someone posted it there and that it went trending :) Thank you to anyone who posted it <3 ~drummyfish }

The repository is currently at <https://codeberg.org/drummyfish/tinyphysicsengine>.

Let's stress that TPE is NOT physically accurate, its purpose is mainly entertainment, simplicity and experimenting; a typical imagined usecase is in some suckless game that just needs to add some simple "alright looking" physics for effect. { Though I am currently in process of making a full racing game with it. ~drummyfish } It tries to respect physics equations where possible but uses cheap approximations otherwise. For example all shapes are in fact just soft bodies made of spheres connected by stiff wires, i.e. there are no other primitives like cuboids or capsules. Environments are made by defining a custom signed distance field (ish) function -- this allows setting up all kinds of environments (even dynamic ones, precomputation is not required), checking a sphere-SDF collision is very easy.

---

tom\_scott

## Tom Scott

TODO

LOL this idiot tries to take some kind of ownership of ideas in a video called *14 science fiction stories in under 6 minutes* (iQGI-ffVtaM), he just comes up with story ideas and says people have to have his permission for commercial use, even though it's probably not legally possible to own ideas like this. Fucking fascist. Do not support.

---

tor

## Tor

TODO

## The BIG RANT

start NOTE: I got some objections from people, keep in mind the rant below is not based on evidence but rather my own experience and what I think is the most reasonable thing to believe. Belief is the key thing here really as that's all we can have with a bloated project by only select few, and in such case I argue to lean towards skepticism.

OK, so **is Tor really safe and "private"?** Well, PROBABLY NOT. It is "safer" and more "private" than clearweb/clearnet (if only for the obscurity) AGAINST THE SMALL GUYS ONLY, like average server owners and tiny ISPs -- that's pretty clear. However the big guys (NSA/CIA/FBI/governments, Google, Facebook etc.) can most likely get through -- consider how much of a bloat Tor and Tor browser are (we have to mention that Tor router and Tor Browser are two different things): did you alone go through the code, checked and tested there isn't a single exploitable line? Or did you just trust a promise of a website and your favorite content producers? (Tor Browser now even has autoupdates so they're just inserting you their code in real time now.) There most definitely is an exploitable line, and very likely more than one; if not by intention (it is EXTREMELY easy to sneak a malicious obfuscated line to a FOSS project; for a government or trillionaire corporation that's like a laughable amount of effort) then by pure statistics (even excellent code will have about 1 bug on 100 lines of code; if not in 100 then in 1000, 10000 or 100000). You may prove the protocol to be safe but remember, security may be broken anywhere, not just the protocol: for example the encryption library used may have a bug, the code implementing sockets may have a bug, random generator may have a weakness, interface code may have a bug (in case of Tor Browser imagine they e.g. manage to silently turn your JavaScript on, then simply fingerprint you) etcetc. And if there is an exploitable line somewhere, you can bet your life they know about it -- do you think NSA won't put their greatest effort into searching for a way to infiltrate the biggest communication network of criminals, terrorists and other competition? This will be on top of every government intelligence service list, they will pour incredible amounts of resources into finding those lines, so you would be really crazy to think they don't know about them. So why don't they just go and bust all the bad guys selling drugs and CP on Tor? OK, does that question even need an answer? Do you think they will reveal this? The moment they do, everyone stops using Tor and they can no longer spy. They literally don't give a single shit about some harmless incels downloading illegal videos or making laughable amounts of pocket money selling marijuana, why would they give up their biggest weapon and spend great money and resources on busting a few horny neckbeards (there wouldn't even be enough space in jail for them lol)? They just want you to think it's safe to use so that

Tor

LRS Wiki

735/815

you use it and they can spy on you. GOVERNMENTS AND CORPORATIONS LIKE GOOGLE LITERALLY SPONSOR TOR, they WANT YOU TO USE IT -- if you can't see what this means then there's likely not much hope in trying to explain anything. They will use the info they gather to bust the big guys who threaten national security, economic stability or whatever, without revealing how they did it of course. They will let small black market exist so as to "prove" it's safe and so make it a nice honeypot -- they lost some money in economy, but it's an investment like any other. Also if they find you're using Tor you automatically get on their watchlist so it may actually be even less "safe" than vanilla clearnet with HTTPS. "BUTTT BUT IT OPEN SOARRRS EVERYONE CAN CHECK THE CODEEEEE" -- are you shitting yourself? Your "everyone" here means someone with excellent expertise AND tremendous resources that analysis of such a huge codebase requires, who is also willing to spend them -- how many entities like that are there in the world? A handful maybe, most of them exactly the mentioned bad guys like government and monster corporations, all of them rich capitalists, i.e. without any morals; now even if there is an independent entity of this kind and if such an entity does the insane, makes the investment and finds the exploit, do you think it will throw its investment out of the window for "public good", or does it try to profit from it by selling the knowledge to the highest bidder (or just staying silent about it)? Hmmm but if that's true, surely Tor developers also know it's futile, why are they even developing it? Because it's their living, they're sponsored, it's a pretty comfy job, they're just snake oil sellers who maybe even believe it works. Hmmm but wait, there are many good guy organizations who need security and will sponsor people to look for vulnerabilities. No -- any organization that really NEEDS a private conversation won't be dumb, it won't search for hyped things but something that's cheap and works, i.e. they will just use encrypted email or encrypted snail mail that flies under the radar easily and just works as long as math works. Some will sponsor Tor because it is still useful, e.g. it helps break some censorship, but no one serious about privacy will rely on Tor. So really until proven otherwise (which probably can't be done) you can't rely on safety of Tor. You are never safe under capitalism. Anyway, let this show you how futile any effort for "privacy" is in a shitty society -- the solution doesn't lie in increasing privacy and security of course, but in unfucking society.

Well then, **is Tor literally useless?** That we don't say -- it is definitely bloated and ugly and one of its main selling points, the "security", may only be partially valid, but it still is useful at least for **kind of allowing some free speech** -- real super offensive sites are happily running on Tor and that's very good, in this case it's probably better to have an ugly, weird free speech platform than none.

## See Also

- HTTPS
- privacy

---

toxic

## Toxicity

A social environment is said to be toxic if it induces a high psychological discomfort, toxic individuals are members of such environment that help establish it. Examples of toxic environments include a capitalist society and SIW social networks.

---

tpe

## TPE

TPE stands for tinyphysicsengine.

---

tranny\_software

## Tranny Software

Tranny software is a harmful software developed within and infected by the culture of the toxic LGBTFIJIGIIQWWOW SIW pseudoleftists, greatly characterized e.g. by codes of conduct, bad engineering and excluding straight white males from the development in the name of "inclusivity". Such software is retarded.



It is practically always a high level of bloat with features such as censorship, bugs and spyware.

To be clear, *tranny software* does NOT stand for *software written by transsexuals*, it stands for a specific kind of software infected by fascism (in its features, development practices, culture, goals etc.) which revolves around things such as sexual identity. Of course with good technology it doesn't matter by whom it is made. For god's sake do NOT bully individuals for being transsexual! Refuse bad software.

Some characteristics of tranny software are:

- **It is typically FOSS** because a big part of trannyism in software is the development process and interaction with public (and with proprietary software the development is not open to public). In theory we may call a proprietary software *tranny* if it e.g. very aggressively promote some SIW bullshit -- in fact most companies are nowadays infected by SIWism -- but it's not so common to use this term for proprietary software.
- **Typically has a code of conduct** or some equivalent "SJW guideline" (excluding the anti-COCs).
- Sometimes promotes SJW fascism in other ways, e.g. LGBT flags etc.
- **It is typically bloat and capitalist software**, and generally just very bad software at least from the LRS point of view. This is partly because devs try to be "inclusive" and afraid to refuse PRs from "diverse people" (who are mostly incompetent, again trying software development e.g. as part of proving that "minorities can program too"), partly because their software doesn't even aim to be good but rather popular (as it is to serve to promote political ideas etc.).
- **It often has SJW "features"**, e.g. "slur filters", "diverse" characters in games etc.

Examples of tranny software are:

- Rust
- Lemmy
- Linux
- Firefox
- Chromium
- ...

Example of software that doesn't seem to be tranny software:

- all official LRS
- likely most suckless software
- TODO

---

transistor

## Transistor

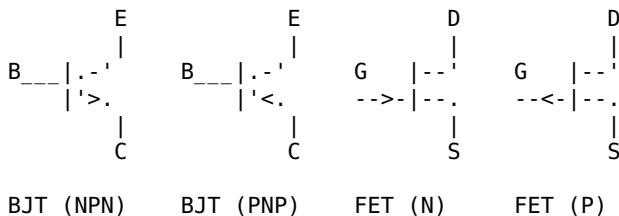
Transistor is a small semiconductor element of electronic circuits that can be used as an amplifier or a switch, and which is a basic building block of digital electronic computers, integrated circuits and many other electronic devices. Transistors replaced vacuum tubes and relays which were used in primitive computers of the 20th century; transistors can be made much smaller, cheaper, more reliable and, unlike relays, operated purely electronically and therefore much faster. Transistor has become the most manufactured device in history.

Transistor generally has three terminals. Its key principle is that of behaving like an electronically operated amplifier or switch: we can make a transistor *open* or *close* (i.e. conduct or not conduct electricity) by applying different voltage or current (and we can also make it something between *open* and *close*). The voltage/current by which we control the transistor can be lower than that which we control, so we can see this as an amplifier: we can control high current with low current, i.e. the high current follows the low current but has higher amplitude. We can also see this as a switch: by applying voltage/current we can make a wire connect (low resistivity) or disconnect (high resistivity) similarly to a physical switch. This switch behavior is important for computers because we can exploit it to implement binary (on/off) logic circuits.

A basic division of transistors is following:

- **bipolar junction transistor (BJT)**: They have 3 terminals: **emitter**, **base** and **collector**. By applying **current** in the base we control the current between emitter and collector (this current can be greater than the control current). BJTs use both kinds of carriers at once, electrons and holes. BJTs are older than FETs, not much used in integrated circuits now.
  - ♦ **PNP**: There is P semiconductor (emitter), N (base) and then P again (collector), creating 2 NP junctions. It opens when there is no current at base.
  - ♦ **NPN**: The other way around that PNP (N, then P, then N). It opens when there is current at base.
- **field effect transistor (FET)**: They have 3 terminals: **source**, **gate** and **drain**. By applying **voltage** to gate we control the current between source and drain. They use only one kind of charge carriers (electrons or holes). This is due to a different internal structure from BJTs, e.g. by having gate separated from source and drain with a metal oxide layer in MOSFET. A voltage applied here kind of "pushes" the holes or electrons out of the way to create a channel between source and drain. The transistor can either be in enhancement mode (high voltage opens the transistor) or depletion mode (low voltage opens the transistor). FETs are newer and have some nicer properties, e.g. lower noise or lower power consumption: they **only consume power on state change**.
  - ♦ **P channel**: Source and drain are made of P semiconductor put into an N semiconductor.
  - ♦ **N channel**: Source and drain are made of N semiconductor put into a P semiconductor. They have a bit different properties from P channel FETs.

Commonly used graphical symbols for transistor are (usually in a circle):



First FET transistors were JFETs (junction-gate FET) but by today were mostly replaced by **MOSFETs** (metal-oxide-semiconductor FET), a transistor using a metal oxide layer for separating the gate terminal which gives it some nice properties over JFET. These transistors are used to implement logic gates e.g. using the **CMOS** fabrication process which uses complementary pairs of P and N channel FETs so that e.g. one is always off which decreases power consumption.

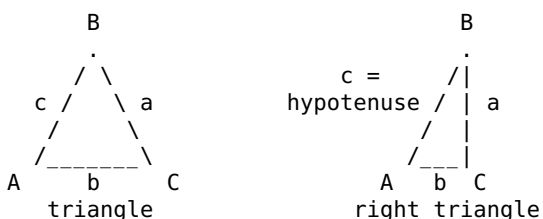
triangle

## Triangle

Triangle is a three sided polygon, one of the most basic geometric shapes. It is a convex shape. Furthermore it is a 2-simplex, i.e. the simplest "shape composed of sides" in 2 dimensions. Triangles are very important, they for example help us to compute distances or define functions like sine and cosine (see trigonometry).

{ In my favorite book Flatland triangles represent the lowest class of men with isoscele triangles being the lowest as they are most similar to women who are just straight lines. ~drummyfish }

Triangle consists of three vertices (usually labeled *A*, *B* and *C*), three sides (usually labeled *a*, *b* and *c* according to the side's opposing vertex) and three angles (usually labeled *alpha*, *beta* and *gamma* according to the closest vertex):



**Right triangle**, a triangle with one angle equal to 90 degrees ( $\pi / 2$  radians), is an especially important type of triangle. Right triangles are used to define trigonometric functions, such as sine, cosine and tangent, as ratios of side lengths as a function of the triangle angle. For example in a right triangle (as drawn above) it holds that  $\sin(\alpha) = a / c$ .

**Similar triangles** are triangles that "have the same shape" (but may be of different sizes, positions and rotations). Two triangles are similar if the lengths of corresponding sides have the same ratios, or, equally, if they have the same inside angles. E.g. a triangle with side lengths 1, 2 and 3 is similar to a triangle with side lengths 2, 4 and 6. This fact is very useful in some geometric computations as it can help us determine unknown side lengths.

**Equilateral triangle** is a triangle whose sides have the same length, which means all its angles are also equal (60 degrees,  $\pi / 3$  radians). Equilateral triangles are of course all similar to each other. An **isoscele triangle** is a triangle with two sides of the same length. We can also distinguish acute and obtuse triangles (obtuse having one angle greater than 90 degrees).

In a triangle there exist two important types of helper line segments: **median** and **altitude**. Median goes from a triangle's vertex to the opposite side's center. Altitude goes from a vertex to the opposite side in a perpendicular direction to that side. Each triangle has three medians and three altitudes.

Some basic facts, features and equations regarding triangles are following (beware: many of them only hold in Euclidean geometry):

- **Triangle angles add up to 180 degrees** ( $\pi$  radians). This can be used to determine unknown side angles.
- Center of weight: average the three coordinates, or take the intersection of the triangle's medians.
- **area**:
  - ♦ general triangle:  $a * \text{altitude}(a) / 2$
  - ♦ right triangle:  $a * b / 2$
- **Pythagorean theorem**: For the lengths of the sides of a RIGHT triangle it always holds that  $a^2 + b^2 = c^2$ . This is extremely important and can be used to determine unknown side lengths of right triangles.
- **Thales's theorem**: if points A, B and C lie on a circle, then they form a right triangle with hypotenuse equal to the circle diameter (and the center of the circle lying in the middle of the hypotenuse).
- **Triangle inequality**: Sum of any two side lengths can't be greater than the length of the third side, i.e.  $a + b \leq c$ . That means that e.g. a triangle with side lengths 1, 2 and 4 can't exist because  $1 + 4 > 2$ . If one side of a triangle is exactly the sum of the other two, the triangle is called **degenerate**, its vertices lie on the same line and it is completely "squashed".
- **Law of sines**:  $a / \sin(\alpha) = b / \sin(\beta) = c / \sin(\gamma)$
- **Law of cosines**: Generalization of Pythagorean theorem:  $a^2 = b^2 + c^2 - 2 * b * c * \cos(\alpha)$ .
- Triangle tessellation is one of only three possible regular plane tilings (the other two being square and hexagon).
- Every triangle has two special associated circles:
  - ♦ **incircle**: circle inside the triangle which touches each of its sides at one point, its center (incenter) lies on the intersection of all angle bisectors.
  - ♦ **circumcircle**: circle outside the triangle which touches each of its vertices, its center (circumcenter) lies on the perpendicular bisectors of each side.
- Triangle vertices always line in a single plane (unlike other polygons).

In non Euclidean ("crazy") geometries triangles behave weird, for example we can draw a triangle with three right angles on a surface of a sphere (i.e. its angles add to more than 180 degrees). This fact can be exploited by inhabitants of a space (e.g. our Universe) to find out if they in fact live in a non Euclidean space (and possibly determine the space's exact curvature).

Constructing triangles: if we are to construct (draw) triangles with only partial knowledge of its parameters, we may exploit the above mentioned attributes to determine things we don't explicitly know. For example if we're told to construct a triangle with knowing only the lengths of the sides but not the angles, we can determine an angle of one side using the law of cosines at which point we can already draw all three vertices and just connect them. In other words just use your brain.

Triangles also play a big role e.g. in realtime 3D rendering where they are used as a basic building block of 3D models, i.e. we approximate more complex shapes with triangles because triangles are simple (thanks to being a simplex) and so have nice properties such as always lying in a plane so we cannot ever see both its front and back side at once. They are relatively easy to draw (rasterize) so once we can draw triangles, we can also draw complex shapes composed of triangles. In general triangles, just as other simple shapes, can be used to approximate measures and attributes -- such as area or center of mass -- of more complex shapes, even outside computer graphics. For example to determine an area of some complex shape we approximate it by triangles, then sum up the areas of the triangles.

**Barycentric coordinates** provide a coordinate system that can address specific points inside any triangle -- these are used e.g. in computer graphics for texturing. The coordinates are three numbers that always add up to 1, e.g. [0.25, 0.25, 0.5]. The coordinates can be thought of as ratios of areas of the three subtriangles the point creates. Points inside the triangle have all three numbers positive. E.g. the coordinates of the vertices *A*, *B* and *C* are [1, 0, 0], [0, 1, 0] and [0, 0, 1], and the coordinates of the triangle center are [1/3, 1/3, 1/3].

**Winding** of the triangle says whether the ordered vertices of the triangle go clockwise or counterclockwise. I.e. winding says whether if we were to go in the direction *A* -> *B* -> *C* we'd be going clockwise or counterclockwise around the triangle center. This is important e.g. for backface culling in computer graphics (determining which side of a triangle in 3D we are looking at). Determining of the winding of triangle can be derived from the sign of the z-component of the cross product of the triangle's sides. For the lazy: compute  $w = (y_1 - y_0) * (x_2 - x_1) - (x_1 - x_0) * (y_2 - y_1)$ , if  $w > 0$  the points go clockwise, if  $w < 0$  the points go counterclockwise, otherwise ( $w = 0$ ) the points lie on a line.

Sierpinski triangle is a fractal related to triangles.

**Testing if point lies inside 2D triangle:** one way to do this is following. For each triangle side test whether the winding of the tested point and the side is the same as the winding of whole triangle -- if this doesn't hold for any side, the point is outside the triangle, otherwise it is inside. In order words for each side we are testing whether the tested point and the remaining triangle point are on the same side (in the same half plane). Here is a C code:

```
int pointIsInTriangle(int px, int py, int tp[6])
{
    // winding of the whole triangle:
    int w = (tp[3] - tp[1]) * (tp[4] - tp[2]) - (tp[2] - tp[0]) * (tp[5] - tp[3]);
    int sign = w > 0 ? 1 : (w < 0 ? -1 : 0);

    for (int i = 0; i < 3; ++i) // test winding of point with each side
    {
        int i1 = 2 * i;
        int i2 = i1 != 4 ? i1 + 2 : 0;

        int w2 = (tp[i1 + 1] - py) * (tp[i2] - tp[i1]) - (tp[i1] - px) * (tp[i2 + 1] - tp[i1 + 1]);
        int sign2 = w2 > 0 ? 1 : (w2 < 0 ? -1 : 0);

        if (sign * sign2 == -1) // includes edges
            //if (sign != sign2) // excludes edges
            return 0;
    }

    return 1;
}
```

---

trolling

## Trolling

Trolling is a linguistically fairly recent term that has evolved alongside the Internet and whose meaning ranges from an activity of purposefully causing drama (mostly on the Internet) for entertainment (the original, narrower meaning) to just shamelessly abusing rules of any system on the detriment of others (narrower meaning, e.g. patent trolling, pre-election *media trolling* etc.). Core value of the narrow-sense trolling is simply fun on the detriment of others -- trolls are those who before the Internet would be called

something like jokers or pranksters, however real life pranks have limited potential -- on the Internet on the other hand trolling can flourish (likely owing to the aspect of anonymity, lack of nonverbal communication, lack of consequences and responsibility and so on), so this art has evolved into something more, it is by now an inseparable part of Internet culture (and should probably become protected by Unesco). Trolling is cherished by people who don't take things too seriously and can make fun of themselves and is therefore embraced by forums like 4chan; on the other hand it is demonized by those without sense of humor and those who mostly get butthurt by it (reddit, Wikipedia and other "safe space" networks). Evil is always afraid of fun.

Here are some potentially entertaining ways of trolling (they'll be written from the point of view of a fictional character so as to avoid legal responsibility for potential crimes they may inspire lol):

- **Link troll:** on my website I randomly put dangerous links that look like normal links, for example "back to homepage" leads to a Google search of "free CP" or "how to get a bomb on the plane". This way anyone who clicks it automatically gets on the NSA watch list :D
- **Crime troll:**
  - ♦ **Evidence troll:** before committing crime I go to a barber shop and collect hair on the floor. Later I scatter all this hair all around the crime scene so that investigators get overwhelmed with DNA evidence of dozens of unrelated people :D This is also known as DNA DDOS (or just DOS?).
  - ♦ **Crime without motive:** criminal investigators always look for a motive in a crime, if there is none they get greatly confused, so for example as an extremely rich man owning 10 Lamborghinis I like to just steal some shitty car from time to time -- even if clues point to me they are always like "why would this rich man steal such a shitty car, it doesn't make sense". I just laugh :D
  - ♦ **Irrational crime:** similarly investigators usually suspect some basic rationality even of the most stupid criminal -- you want to behave even stupider than that. For example I break into two stores and then just relocate goods from one to the other, then leave :D
- **Seizure troll:** when in some kind of lecture where the students are allowed laptops (typically in a compsci uni) I take a seat somewhere in the front row, near the lecturer, open my laptop and start a program that just rapidly flashes wild colors in fullscreen -- I leave it like that for the whole lecture so that everyone sitting behind me is forced to watch the flashing and can get an epileptic seizure. For educational purposes code for such a program can be written in a few lines of browser JavaScript (it may coincidentally possibly even be found in that JavaScript article).
- **Eco whoring troll:** I take a bag of plastic trash, go to the forest, find some nice, clean place, take a thumbs up selfie photo with it, dump the trash in there, take another photo, then post the photos in reverse order to Twitter with something like "today I worked tirelessly to clean this garbage dump, for our children!", get a million eco likes for literally dumping plastic trash in a forest.
- **Stack overtroll:** I love to perform this troll on sites like programming advice subreddits where wannabe soydevs try to roleplay as authorities on programming -- these deserve to be trolled the most :D This gets me banned every time but it's totally worth it, I use this to leave social network sites with style once I get bored with them. I make a piece of code that looks like some noob attempt at making a game, but it's secretly an obfuscated code that when run does something nasty like delete all files on the computer, create one trillion subdirectories or set goatse as a wallpaper (bloat languages like Python are actually great for this as they can do nasty stuff like execute a dynamically constructed string and they can also download stuff from the Internet and basically do anything they like). Then I post it with a question "hello fellow programmers, I am trying to make my first game but my code doesn't work, can u help me plz?" If the code is well made, i.e. not trivial and quite hard to understand just by looking at it, chances are the first thing people are gonna do is simply copy paste the code and run it -- that's why I prefer to make the code completely destroy the computer so that the guy has to take at least a few hours to reinstall the system to be able to warn others it's a troll. Whenever some comment pops up saying it's a trap, I immediately downvote it and report it for hate speech (I also use puppet accounts here to spam the downvotes because I'll get banned anyway). The good things about this is that I actually teach people about muh security, those who step on this mine will never run a random code from the Internet again.
- **Troll the troll:** Advanced mastery of trolling allows one to troll other less experienced trolls -- an encounter of two trolls can be quite fun and educational. Imagine for example troll A setting up the above mentioned *stack overtroll* bait -- troll B, an experienced player of the game, notices the bait, but of course he doesn't bring this up -- no, he pretends to take the bait and responds with something like "wait a minute, let me run the code". Troll A is happy because he thinks he won, but then troll B

responds: "yeah, here on line X you got this wrong, here is the correct code...". Troll A is now confused, he's unsure if he's been spotted or if troll B simply skipped running the code, troll B is now in advantage of controlling the game -- a best result here is if troll B actually somehow gets troll A to run the "fixed" code which however breaks his computer; here troll B succeeded in deflecting the troll back and catching OP into his own trap -- this kind of outcome is the best you can wish for and a showcase of true trolling mastery.

- **Creative Wikipedia vandalism:** for example funny redirects or categorizations (put Bill Gates to "famous homosexuals" category or something), also consider vandalizing other wikis that usually don't have as much protection.
- `a: hover { display: none; }`
- Classic trollz revolve around creating drama on forums -- this is kind of an art as you have to keep the right balance of seriousness and stupidity; too much of the former and you're not trolling anyone, too much of the latter and you're just spotted as obvious troll. It's definitely not about logging on a starting to drop the N-words and insulting everyone, that's just an instant ban that ends the fun; you rather want to start slow, get many people seriously involved in the discussion, be polite and then slightly steer the talk towards something controversial (nice if you pretend to be part of some "oppressed minority"). Then you just make it look like you're just an uneducated simple minded individual who kind of happens to lean towards an opinion the others truly hate, but you have to keep their hope that they can convince you to change your opinion, so still try to be polite, just so you keep arguing with them and wasting more and more of their time until they start losing their shit and the thread explodes into hitler arguments etc., then just watch and enjoy.
- ...

---

troll

## Troll

Please go here.

HAHAHA U BEEN TROOOOOLD. For realz go here.

---

trom

## TROM

{ WIP, still being researched. Was trying to reach the Tio guy but he didn't respond. ~drummyfish }

TROM (The Reality Of Me) is a project running since 2011 that's educating mainly about the harmfulness of trade-based society and tries to show a better way: that of establishing so called trade-free (in the sense of WITHOUT trade) society. It is similar to e.g. Venus Project (with which TROM collaborated for a while), the Zeitgeist Movement and our own LRS -- it shares the common core of opposing money and trade as a basis of society (which they correctly identify as the core issue), even though it differs in some details that we (LRS) don't see as wholly insignificant. The project website is at <https://www.tromsite.com/>.

TROM was started and is run by a Romanian guy called Tio (born June 12 1988, according to blog), he has a personal blog at <https://www.tiotrom.com> and now lives mostly in Spain. The project seems more or less run just by him.

The project is funded through Patereon and has created a very impressive volume of very good quality educational materials including books, documentaries, memes, even its own GNU/Linux distro (Tromjaro). The materials TROM creates sometimes educate about general topics (e.g. language), not just those directly related to trade-free society -- Tio says that he simply likes to learn all about the world and then share his knowledge.

The idea behind TROM is very good and similar to LRS, but just as the Venus project it is a bit sus, we align with many core ideas and applaud many things they do, but can't say we 100% support everything about TROM.

## Summary

{ WATCH OUT, this is a work in progress sum up based only on my at the moment a rather brief research of the project, I don't yet guarantee this sum up is absolutely correct, that will require me to put in much more time. Read at own risk. ~drummyfish }

### good things about TROM:

- It identifies trade/money/capitalism as the root cause of most problems in society -- this we basically agree with (we officially consider competition to be the root cause).
- It has TONS of very good quality educational materials exposing the truth, breaking corporate propaganda, even showing such things as corruption in soyence and open source etc. -- that's extremely great.

### bad things about TROM:

- It is based on fight culture and hero culture (judging by the content of the book *the origin of most problems*, creating superheroes and stating such things as "the world needs an enemy"), which we greatly oppose, we (LRS) don't believe a good and peaceful society can worship heroes and wars. Hopefully this is something the project may realize along the way, however at this point there is a great danger of falling into the trap of turning into a violent revolutionary pseudoleftist movement.
- TROM refuses to use free (as in freedom) licenses! Huge letdown. Tio gives some attempt at explanation at <https://www.tiotrom.com/2022/08/free-software-nonsense/>, but he doesn't seem to understand the concept of free culture/software very well, he's the "it don't need no license I say do whatever you want" guy.
- Bloat, a lot of bloat everywhere, Javascript sites discriminating against non-consumerist computers. Tio isn't a programmer and seems to not be very aware of the issues regarding software, he uses the Wordpress abomination with plugins he BUYS etc.
- There seems to be signs of atheist/iamverysmart/neil de grass overrationalism, some things are pretty cringe and retarded, see the following point.
- From Tio's blog TROM seems to be just his one-man project, even though he may employ a few helpers -- not that there's anything wrong about one man projects :) -- Tio is very determined, talented in certain areas such as "content creation" etc., however he dropped out of school and lacks a lot of knowledge and insight into important areas such as technology and free culture while talking about them as if he knew them very well, this is not too good and even poses some dangers. He really loves to write about himself, shows great signs of narcissism. Hell, at times he seems straight from /r/iamverysmart, e.g. with this real cringe attempt at his own theory of black holes <https://www.tiotrom.com/2021/11/my-theory-about-black-holes/>. Of course it's alright to speculate and talk about anything, but sometimes it seems he thinks he's a genius at everything { Not wanting to make fun of anyone though, been there myself. :) ~drummyfish }. His eagerness is both his great strength and weakness, and as TROM is HIS project, it's the strength and weakness of it as well. Don't blindly follow TROM, takes the good out of it, leave the rest.

---

trump

## Donald Trump

"me goin to hav covfefe" --president of the United States

TODO

He is the smartest capitalist of all time, which puts his intellect somewhere between a dolphin and a rat. When written in binary, his IQ goes into triple digits!

---

trusting\_trust

# Trusting Trust

In computer security *trusting trust* refers to the observation (and a type of attack exploiting it) that one cannot trust the technology he didn't create 100% from the ground up; for example even a completely free compiler such as gcc with verifiably non-malicious code, which has been compiled by itself and which is running on 100% free and non-malicious hardware may still contain malicious features if a non-trusted technology was ever involved in the creation of such compiler in the past, because such malicious technology may have inserted a self-replicating malicious code that's hiding and propagating only in the executable binaries. It seemed like this kind of attack was extremely hard to detect and counter, but a method for doing exactly that was presented in 2009 in a PhD thesis called *Fully Countering Trusting Trust through Diverse Double-Compiling*. The problem was introduced in Ken Thompson's 1984 paper called *Reflections on Trusting Trust*.

**Example:** imagine free software has just been invented and there are no free C compilers yet, only a proprietary (potentially malicious) C compiler *propC*. We decide to write the first ever free C compiler called *freeC*, in C. *freeC* code won't contain any malicious features, of course. Once we've written *freeC*, we have to compile it with something and the only available compiler is the proprietary one, *propC*. So we have to compile *freeC* with *propC* -- doing this, even if *freeC* source code is completely non-malicious, *propC* may sneakily insert malicious code (e.g. a backdoor or telemetry) to *freeC* binary it generates, and it may also insert a self-replicating malicious code into it that will keep replicating into anything this malicious *freeC* binary will compile. Then even if we compile *freeC* with the (infected) *freeC* binary, the malicious self-replicating feature will stay, no matter how many times we recompile *freeC* by itself. Keep in mind this principle may be used even on very low levels such as that of assembly compilers, and it may be extremely difficult to detect.

**For a little retarded people:** we can perhaps imagine this with robots creating other robots. Let's say we create plans for a completely nice, non-malicious, well behaved servant robot that can replicate itself (create new nice behaving robots). However someone has to make the first robot -- if we let some potentially evil robot make the first "nice" robot according to our plans, the malicious robot can add a little malicious feature to this otherwise "nice" robot, e.g. that he will spy on its owner, and he can also make that "nice" robot make pass this feature to other robots he makes. So unless we make our first nice robot by hand, it's very hard to know whether our nice robots don't in fact possess malicious features.

---

turing\_machine

## Turing Machine

Turing machine is a mathematical model of a computer which works in a quite simple way but has nevertheless the full computational power that's possible to be achieved. Turing machine is one of the most important tools of theoretical computer science as it presents a basic model of computation (i.e. a mathematical system capable of performing general mathematical calculations) for studying computers and algorithms -- in fact it stood at the beginning of theoretical computer science when Alan Turing invented it in 1936 and used it to mathematically prove essential things about computers; for example that their computational power is inevitably limited (see computability) -- he showed that even though Turing machine has the full computational power we can hope for, there exist problems it is incapable of solving (and so will be any other computer equivalent to Turing machine, even human brain). Since then many other so called **Turing complete** systems (systems with the exact same computational power as a Turing machine) have been invented and discovered, such as lambda calculus or Petri nets, however Turing machine still remains not just relevant, but probably of greatest importance, not only historically, but also because it is similar to physical computers in the way it works.

The advantage of a Turing machine is that it's firstly very simple (it's basically a finite state automaton operating on a memory tape), so it can be mathematically grasped very easily, and secondly it is, unlike many other systems of computations, actually similar to real computers in principle, mainly by its sequential instruction execution and possession of an explicit memory tape it operates on (equivalent to RAM in traditional computers). However note that a **pure Turing machine cannot exist in reality** because there can never exist a computer with infinite amount of memory which Turing machine possesses; computers that can physically exist are really equivalent to finite state automata, i.e. the "weakest" kind of systems of



computation. However we can see our physical computers as approximations of a Turing machine that in most relevant cases behave the same, so we do tend to theoretically view computers as "Turing machines with limited memory".

In Turing machine data and program are separated (data is stored on the tape, program is represented by the control unit), i.e. it is closer to Harvard architecture than von Neumann architecture.

**Is there anything computationally more powerful than a Turing machine?** Well, yes, but it's just kind of "mathematical fantasy". See e.g. oracle machine which adds a special "oracle" device to a Turing machine to make it magically solve undecidable problems.

## How It Works

Turing machine has a few different versions (such as one with multiple memory tapes, memory tape unlimited in both directions, one with non-determinism, ones with differently defined halting conditions etc.), which are however equivalent in computing power, so here we will describe just one of the most common variants.

A Turing machine is composed of:

- **memory tape:** Memory composed of infinitely many cells (numbered 0, 1, 2, ...), each cell can hold exactly one symbol from some given alphabet (can be e.g. just symbols 0 and 1) OR the special *blank* symbol. At the beginning all memory cells contain the *blank* symbol. Memory holds the data on which we perform computation.
- **read/write head:** Head that is positioned above a memory cell, can be moved to left or right. At the beginning the head is at memory cell 0.
- **control unit:** The program (algorithm) that's "loaded" on the machine (the controls unit by itself is really a finite state automaton). It is composed of:
  - ♦ **a set of N (finitely many) states** {Q0, Q1, ... QN-1}: The machine is always in one of these states. One state is defined as starting (this is the state the machine is in at the beginning), one is the end state (the one which halts the machine when it is reached).
  - ♦ **a set of finitely many rules** in the format [*stateFrom*, *inputSymbol*, *stateTo*, *outputSymbol*, *headShift*], where *stateFrom* is the current state, *inputSymbol* is symbol currently under the read/write head, *stateTo* is the state the machine will transition to, *outputSymbol* is the symbol that will be written to the memory cell under read/write head and *headShift* is the direction to shift the read/write head in (either *left*, *right* or *none*). There must not be conflicting rules (ones with the same combination of *stateFrom* and *inputSymbol*).

The machine halts either when it reaches the end state, when it tries to leave the tape (go left from memory cell 0) or when it encounters a situation for which it has no defined rule.

The computation works like this: the input data we want to process (for example a string we want to reverse) are stored in the memory before we run the machine. Then we run the machine and wait until it finishes, then we take what's present in the memory as the machine's output (i.e. for example the reversed string). That is a Turing machine doesn't have a traditional I/O (such as a "printf" function), it only works entirely on data in memory!

Let's see a simple **example**: we will program a Turing machine that takes a binary number on its output and adds 1 to it (for simplicity we suppose a fixed number of bits so an overflow may happen). Let us therefore suppose symbols 0 and 1 as the tape alphabet. The control unit will have the following rules:

### stateFrom inputSymbol stateTo outputSymbol headShift

|         |           |         |             |       |
|---------|-----------|---------|-------------|-------|
| goRight | non-blank | goRight | inputSymbol | right |
| goRight | blank     | add1    | blank       | left  |
| add1    | 0         | add0    | 1           | left  |
| add1    | 1         | add1    | 0           | left  |
| add0    | 0         | add0    | 0           | left  |
| add0    | 1         | add0    | 1           | left  |

**stateFrom inputSymbol stateTo outputSymbol headShift**

end

Our start state will be *goRight* and *end* will be the end state, though we won't need the end state as our machine will always halt by leaving the tape. The states are made so as to first make the machine step by cells to the right until it finds the blank symbol, then it will step one step left and switch to the adding mode. Adding works just as we are used to, with potentially carrying 1s over to the highest orders etc.

Now let us try inputting the binary number 0101 (5 in decimal) to the machine: this means we will write the number to the tape and run the machine as so:

```

goRight
  V
| 0 | 1 | 0 | 1 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
  goRight
    V
| 0 | 1 | 0 | 1 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
      goRight
        V
| 0 | 1 | 0 | 1 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
            goRight
              V
| 0 | 1 | 0 | 1 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
                  goRight
                    V
| 0 | 1 | 0 | 1 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
                        add1
                          V
| 0 | 1 | 0 | 1 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
                              add1
                                V
| 0 | 1 | 0 | 0 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
                                  add0
                                    V
| 0 | 1 | 1 | 0 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|
                                      add0
                                        V
| 0 | 1 | 1 | 0 |  |  |  |  |  | ...
|---|---|---|---|---|---|---|---|

```

END

Indeed, we see the number we got at the output is 0110 (6 in decimal, i.e. 5 + 1). Even though this way of programming is very tedious, it actually allows us to program everything that is possible to be programmed, even whole operating systems, neural networks, games such as Doom and so on. Here is C code that simulates the above shown Turing machine with the same input:

```

#include <stdio.h>

#define CELLS 2048          // ideal Turing machine would have an infinite tape...
#define BLANK 0xff         // blank tape symbol
#define STATE_END 0xff
#define SHIFT_NONE 0
#define SHIFT_LEFT 1
#define SHIFT_RIGHT 2

unsigned int state;          // 0 = start state, 0xffff = end state
unsigned int headPosition;
unsigned char tape[CELLS];

unsigned char input[] =

```

```

    { 0, 1, 0, 1 };

unsigned char rules[] =
{
// state symbol newstate newsymbol shift
    0,    0,    0,    0,    SHIFT_RIGHT, // moving right
    0,    1,    0,    1,    SHIFT_RIGHT, // moving right
    0,    BLANK, 1,    BLANK,  SHIFT_LEFT,  // moved right
    1,    0,    2,    1,    SHIFT_LEFT,  // add 1
    1,    1,    1,    0,    SHIFT_LEFT,  // add 1
    2,    0,    2,    0,    SHIFT_LEFT,  // add 0
    2,    1,    2,    1,    SHIFT_LEFT,  // add 0
};

void init(void)
{
    state = 0;
    headPosition = 0;

    for (unsigned int i = 0; i < CELLS; ++i)
        tape[i] = i < sizeof(input) ? input[i] : BLANK;
}

void print(void)
{
    printf("state %d, tape: ", state);

    for (unsigned int i = 0; i < 32; ++i)
        printf("%c%c", tape[i] != BLANK ? '0' + tape[i] : '.', i == headPosition ?
            '<' : ' ');

    putchar('\n');
}

// Returns 1 if running, 0 if halted.
unsigned char step(void)
{
    const unsigned char *rule = rules;

    for (unsigned int i = 0; i < sizeof(rules) / 5; ++i)
    {
        if (rule[0] == state && rule[1] == tape[headPosition]) // rule matches?
        {
            state = rule[2];
            tape[headPosition] = rule[3];

            if (rule[4] == SHIFT_LEFT)
            {
                if (headPosition == 0)
                    return 0; // trying to shift below cell 0
                else
                    headPosition--;
            }
            else if (rule[4] == SHIFT_RIGHT)
                headPosition++;

            return state != STATE_END;
        }

        rule += 5;
    }

    return 0;
}

int main(void)
{
    init();

    print();

    while (step())

```

```

    print();

    puts("halted");

    return 0;
}

```

And here is the program's output:

```

state 0, tape: 0<1 0 1 . . . . .
state 0, tape: 0 1<0 1 . . . . .
state 0, tape: 0 1 0<1 . . . . .
state 0, tape: 0 1 0 1<. . . . .
state 0, tape: 0 1 0 1 .<. . . . .
state 1, tape: 0 1 0 1<. . . . .
state 1, tape: 0 1 0<0 . . . . .
state 2, tape: 0 1<1 0 . . . . .
state 2, tape: 0<1 1 0 . . . . .
halted

```

**Universal Turing machine** is an extremely important type of Turing machine: one that is able to simulate another Turing machine -- we can see it as a Turing machine interpreter of a Turing machine. The Turing machine that's to be simulated is encoded into a string (which can then be seen as a programming language -- the format of the string can vary, but it somehow has to encode the rules of the control unit) and this string, along with an input to the simulated machine, is passed to the universal machine which executes it. This is important because now we can see Turing machines themselves as programs and we may use Turing machines to analyze other Turing machines, to become self hosted etc. It opens up a huge world of possibilities.

**Non-deterministic Turing machine** is a modification of Turing machine which removes the limitation of determinism, i.e. which allows for having multiple different "conflicting" rules defined for the same combination of state and input. During execution such machine can conveniently choose which of these rules to follow, or, imagined differently, we may see the machine as executing all possible computations in parallel and then retroactively leaving in place only the most convenient path (e.g. that which was fastest or the one which finished without getting stuck in an infinite loop). Surprisingly a **non-deterministic Turing machine is computationally equivalent to a deterministic Turing machine**, though of course a non-deterministic machine may be faster (see especially P vs NP).

Turing machines can be used to define computable formal languages. Let's say we want to define language  $L$  (which may be anything such as a programming language) -- we may do it by programming a Turing machine that takes on its input a string (a word) and outputs "yes" if that string belongs to the language, or "no" if it doesn't. This is again useful for the theory of decidability/computability.

## See Also

- [brainfuck](#)
- [busy beaver](#)
- [counter machine](#)

---

twos\_complement

## Two's Complement

Two's complement is an elegant way of encoding signed (i.e. potentially negative) integer (and possibly also fixed point) numbers in binary. It is one of the most basic concepts to know as a programmer; for its simplicity and nice properties two's complement is the way that's used to represent binary integers almost everywhere nowadays. Other ways of encoding signed numbers, mainly sign-magnitude and one's complement, are basically always inferior.

Why is two's complement so great? Its most notable advantages are:

- **There is only one zero value** (while other encodings such as sign-magnitude and one's complement have positive and negative zero which wastes values and complicates algorithms).
- **Highest bit indicates the number sign** in the same way as e.g. in sign-magnitude and one's complement representations, i.e. determining whether a number is positive or negative is just as easy as in the more naive representations.
- **Addition, subtraction and multiplication (both signed and unsigned!) work the same as with unsigned representation and overflow naturally**, i.e. we can have exactly the same hardware for these operations as for unsigned numbers and we don't even have to know whether the number is supposed to be unsigned or signed (this of course does NOT hold for any operation, e.g. division or comparison). Operations such as decrementing 0 or incrementing -1 correctly yield -1 and 0, respectively, without any special conditions. Subtraction can simply be done as adding a negated value. One's complement and sign-magnitude have to have special conditions for many of these situations.
- **Positive values and zero are the same as the straightforward unsigned representation**, i.e. it is "backwards compatible" with the straightforward representation. For example the 4 bit value 0011 represents number 3 in two's complement just like it does in a normal unsigned binary number. (This also holds in sign-magnitude and one's complement.)
- **Multiplying by -1 is still relatively simple** -- even though it is a tiny bit more expensive than in one's complement or sign-magnitude, it is still pretty straightforward (only requires two operations instead of one) and in hardware it can be implemented just as fast.

TODO: disadvantages?

$N$  bit number in two's complement can represent numbers from  $-(2^N) / 2$  to  $2^N / 2 - 1$  (including both). For example with 8 bits we can represent numbers from -128 to 127.

**How does it work?** EZ: the highest (leftmost) bit represents the sign: 0 is positive (or zero), 1 is negative. To negate a number negate all its bits and add 1 to the result (with possible overflow). (There is one exception: negating the smallest possible negative number will give the same number as its positive value cannot be represented.)

In other words given  $N$  bits, the positive values representable by two's complement with this bit width are the same as in normal unsigned representation and any representable negative value  $-x$  corresponds to the value  $2^N - x$ .

**Example:** let's say we have a 4 bit number 0010 (2). It is positive because the leftmost bit is 0 and we know it represents 2 because positive numbers are the same as the straightforward representation. To get number -2, i.e. multiply our number by -1, we negate the number, which gives 1101, and add 1, which gives 1110. We see by the highest 1 bit that this number is negative, as we expected. As an exercise you may try to negate this number back and see we obtain the original number. Let's just now try adding 2; we expect that adding 2 to -2 will give 0. Sure enough,  $1110 + 0010 = 0000$ . Etcetc. :)

The following is a comparison of the different representations, notice the shining superiority of two's complement:

| value | unsigned | two's com. | sign-mag. | one's com. | biased |
|-------|----------|------------|-----------|------------|--------|
| 000   | 0        | 0          | 0         | 0          | -4     |
| 001   | 1        | 1          | 1         | 1          | -3     |
| 010   | 2        | 2          | 2         | 2          | -2     |
| 011   | 3        | 3          | 3         | 3          | -1     |
| 100   | 4        | -4         | -0        | -3         | 0      |
| 101   | 5        | -3         | -1        | -2         | 1      |
| 110   | 6        | -2         | -2        | -1         | 2      |
| 111   | 7        | -1         | -3        | -0         | 3      |

ubi

# Universal Basic Income

Universal basic income (UBI) is the idea that all people should get regular pay from the state without any conditions, i.e. even if they don't work, even if they are rich, criminals etc. It is a great idea and **we fully support it** as the first step towards the ideal society in which people don't have to work. As automation takes away more and more jobs, it is being discussed more and experiments with UBI are being conducted, even though capitalist idiots rather try to invent more and more bullshit jobs to keep people enslaved.

UBI that itself covers all basic needs is called full, otherwise it is called partial. UBI subscribes to the idea that **the goal of progress is to eliminate the need for work**, which is correct -- we should leave all work to machines eventually, that's why we started civilization. This doesn't mean we can't work, just that we aren't obliged.

The first reaction of a noob hearing about UBI is "but everyone will just stop working!" Well no, for a number of reasons (which have been confirmed by real life experiments). For example most people don't want to just survive, they want to buy nice things and have something extra, so most people will want to get some additional income. Secondly people do want to work -- work in the sense of doing something meaningful. If they don't have to be wage slaves, most will decide to dedicate their free time to doing something useful. Thirdly people are already used to working, most will keep doing it just out of inertia, e.g. because they have friends at work or simply because they actually happen to like going there.

Another question of the noob is "but who will pay for it?!" Well, we all and especially the rich. In current situation, even if we make the rich give away 90% of their wealth, they won't even notice.

Of course, UBI works with money and money is bad, however the core idea is simply about sharing resources, i.e. true communism which surpasses the concept of money. That is once money is eliminated, the idea of UBI will stay as the right of everyone to get things such as food and health care unconditionally. LRS supports UBI as the immediate next step towards making money less important and eventually eliminating them altogether.

Advantages of UBI:

- **People will cease to be slaves of capitalist employers** as it will no longer be mandatory to work somewhere. Nowadays people have to accept shitty working conditions because they have no other choice. They can't go elsewhere because it is the same everywhere or their conditions don't allow them to move, every employer abuses his employees as much as possible so people today can only choose their slave master but they can't choose not being slaves. **If people can actually leave, employers will have to offer good conditions to keep people working for them.** It may also lead to e.g. greater freedom from consumerism as people can e.g. decide to not use certain bad technology which they are nowadays forced to use by employers.
- **It will greatly suppress bullshit jobs** and undesirable phenomena such as the antivirus paradox. People who stop doing bullshit jobs will be able to actually focus on useful things.
- **Suffering of many people will be lowered or eliminated.** Simple as that.
- **We will actually save money and other resources** because the system will simplify a lot. Nowadays we have complex bureaucracy and commissions that judge who can get social welfare, who can get disability pensions etc. Huge amounts of money are wasted to just keep unnecessary jobs existing, e.g. people have to commute to their bullshit jobs, which implies cars and roads have to be maintained more, also workplaces have to be maintained, cleaned, maintain work safety, etcetc. If everyone just gets money to live, we can save on this bureaucracy, maintenance, commissions, on doctor examinations, caring about the homeless, maintaining hugely complex laws etc. If people become less stressed, mental health will also improve and we will save money on treatment of mentally ill people. Money may also be saved on organization of worker unions as they may become much less important etc.
- **Society will become more ecological** thanks to the elimination of bullshit and saving resources, as mentioned above.
- **People will become less stressed**, happier, will have security and as a result perhaps even become more "productive" (this has been confirmed by some experiments).
- **Criminality will greatly decrease** as it is directly linked to poverty, this will of course further save money on police, lawyers, medical bills etc.

- **People will become more equal** which will shift us closer to the ideal society.
- **People will be able to do more important things than work if needed**, for example they may choose to focus on education for a few years, which will make the population better educated and therefore better.
- **Social security will suppress fear in people** and therefore make them less xenophobic, less militant, less fascist etc.
- **Homelessness will greatly decrease**, streets will be cleaner, we'll be able to close many homeless centers etc.
- **There will be many more indirect benefits**, e.g. people may start to live without unnecessary things that are nowadays forced on them by jobs such as cell phones, cars or bank accounts, which will subsequently make people consume less, spend less time on toxic social networks and live more healthy lives among friends in local communities, lowering the negative effects of globalization such as the spread of the US culture etc.
- **Capitalists will be crying like little babies.**

Disadvantages of UBI:

- none { Well I guess a slight advantage might be that UBI still works with the idea of money, but as mentioned above, the principle of UBI will be preserved even after abolishment of money.  
~drummyfish }

---

ui

## User Interface

User interface, or UI, is an interface between human and computer. Such interface interacts with some of the human senses, i.e. it can be visual (text, images), auditory (sound), touch (haptic) etc.

Remember the following inequality:

*non-interactive CLI > interactive CLI > TUI > GUI*

Some faggots make living just by designing interfaces without even knowing programming lmao. They call it "user experience" or UX. We call it a bullshit.

---

unary

## Unary

Unary generally refers to having "one of a thing". In different contexts it may specifically signify e.g.:

- **unary numeral system:** A base for writing numbers (just as binary, decimal, hexadecimal etc.). This base is kind of an extreme, using only one symbol (0) and has at least two possible versions:
    - ♦ The most primitive "caveman" system of recording numbers with a single symbol, recording a number simply by writing "that many symbols", e.g. using the symbol 0, one is written as 0, two as 00, three as 000 etc. Zero itself is represented by an empty string (writing nothing). Though primitive, this system is actually usable.
    - ♦ The system following rules of computers, i.e. having a fixed space, i.e. number of places, for storing a number (just as in binary we have may have e.g. 8 bits for storing a number). However since each of those places can only hold one value (the single symbol of the unary system, usually set to be 0), the system **is a joke**, because no matter how many places, we can only ever record one number -- zero. The advantage is that we can store zero even with zero places, i.e. we don't even need any memory to store the number.
  - **unary function, operator etc.:** function, operator etc. that only has one parameter (e.g. square root, ...).
  - ...
- 

unicode

[illegible]

In Unicode every character is unique like a unicorn.

# Universe

Computers can be used to simulate certain parts of the universe, in fact all programs mimic the universe in some kind of simplified way, be it scientific simulations of planet collisions, government databases or games -- they all more or less accurately model the reality. We can also see computers as a way of creating smaller universes, which leads many to think our universe may itself be a simulation running on some computer in a "bigger" universe (note that this probably isn't testable and the debate isn't scientific, but we can lead philosophical discussions about it).

# Unix

Unix (plural *Unixes* or *Unices*) is an old operating system developed since 1960s as a research project of Bell Labs, which has become one of the most influential pieces of software in history and whose principles (e.g. the Unix philosophy, everything is a file, ...) live on in many so called Unix-like operating systems such as Linux and BSD (at least to some degree). The original system itself is no longer in use (it was later followed by a new project, plan9, which itself is now pretty old), the name UNIX is nowadays a trademark and a certification. However, as someone once said, *Unix is not so much an operating system as a way of thinking*.

The main highlights of Unix are possibly these:

- 752/815



- **everything is a file**: Unix chose to use the file abstraction to enable universal communication of programs with hardware and among themselves, i.e. on unices most things such as printing, reading keyboard, networking etc. will be likely implemented as reading or writing to/from some special (sometimes just virtual) file. This has the advantage of being able to just use some file reading library or syscall, not having to access physical memory bits in memory, which may be difficult, unsafe etc.
- Text centrism (great command line preference), value on portability (even over performance), sharing of source code, freedom of information and openness, connection to hacker culture, valuing human time over machine time, ...
- ...

Unix is greatly connected to software minimalism, however most unices are still not minimalist to absolute extreme and many unix forks (e.g. GNU/Linux) just abandon minimalism as a priority. So the question stands: **is Unix LRS or is it too bloated?** The answer to this will be similar to our stance towards the C language (which itself was developed alongside Unix); from our point of view Unix -- i.e. its concepts and some of their existing implementations -- is relatively good, there is a lot of wisdom to take away (e.g. "do one thing well", modularity, "use text interfaces", ...), however these are intermixed with things which under more strict minimalism we may want to abandon (e.g. "everything is a file" requires we buy into the file abstraction and will often also imply existence of a file system etc., which may be unnecessary), so in some ways we see Unix as a temporary "least evil" tool on our way to truly good, extremely minimalist technology. DuskOS is an example of operating system more close to the final idea of LRS. But for now Unix is very cool, some Unix-like systems are definitely a good choice nowadays.

There is a semi humorous group called the *UNIX HATERS* that has a mailing list and a whole book that criticizes Unix, arguing that the systems that came before it were much better -- though it's mostly just joking, they give some good points sometimes. It's like they are the biggest boomers for whom the Unix is what Windows is to the Unix people.

## History

In the 1960s, Bell Labs along with other groups were developing Multics, a kind of operating system -- however the project failed and was abandoned for its complexity and expensiveness of development. In 1969 two Multics developers, Ken Thompson and Dennis Ritchie, then started to create their own system, this time with a different philosophy; that of simplicity (see Unix philosophy). They weren't alone in developing the system, a number of other hackers helped program such things as a file system, shell and simple utility programs. At VCF East 2019 Thompson said that they developed Unix as a working system in three weeks. At this point Unix was written in assembly.

In the early 1970s the system got funding as well as its name Unix (a pun on Multix). By now Thompson and Richie were developing a new language for Unix which would eventually become the C language. In version 4 (1973) Unix was rewritten in C.

Unix then started being sold commercially. This led to its fragmentation into different versions such as the BSD or Solaris. In 1983 a version called System V was released which would become one of the most successful. The fragmentation and a lack of a unified standard led to so called Unix Wars in the late 1980s, which led to a few Unix standards such as POSIX and Single Unix Specification.

For zoomers and other noobs: Unix wasn't like Windows, it was more like DOS, things were done in text interface -- if you use the command line in "Linux" nowadays, you'll get an idea of what it was like, except it was all even more primitive. Things we take for granted such as a mouse, copy-pastes, interactive text editors, having multiple user accounts or running multiple programs at once were either non-existent or advanced features in the early days. Anything these guys did you have to see as done with stone tools.

---

unix\_philosophy

## Unix Philosophy

Unix philosophy is one of the most important and essential approaches to programming (and by extension all technology design) which advocates great minimalism and is best known by the saying that **a program should only do one thing and do it well**. Unix philosophy is a collective wisdom, a set of design

recommendations evolved during the development of one of the earliest (and most historically important) operating systems called Unix, hence the name. Having been defined by hackers (the true, old style ones) the philosophy naturally advises for providing a set of many highly effective tools that can be combined in various ways, i.e. to perform hacking, rather than being restricted by a fixed, intended functionality of huge do-it-all programs. Unix philosophy advocates simplicity, clarity, modularity, reusability and composition of larger programs out of very small programs rather than designing huge monolithic programs as a whole. Unix philosophy, at least partially, lives on in many project and Unix-like operating systems such as Linux (though Linux is more and more distancing from Unix), has been wholly adopted by groups such as suckless and LRS (us), and is even being reiterated in such projects as plan9.

NOTE: see also everything is a file, another famous design principle of Unix -- this one is rather seen as a Unix-specific design choice rather than part of the general Unix philosophy itself, but it helps paint the whole picture.

As written in the GNU coreutils introduction, a Swiss army knife (universal tool that does many things at once) can be useful, but it's not a good tool for experts at work, they note that a professional carpenter will rather use a set of relatively simple, highly specialized tools, each of which is extremely efficient at its job. Unix philosophy brings this observation over to the world of expert programmers.

In 1978 Douglas McIlroy has written a short overview of the Unix system (*UNIX Time-Sharing System*) in which he gives the main points of the system's style; this can be seen as a summary of the Unix philosophy (the following is paraphrased):

1. **Each program should do one thing and do it well.** Overcomplicating existing programs isn't good; for new functionality create a new program.
2. **Output of a program should be easy to interpret by another program.** In Unix programs are chained by so called pipes in which one program sends its output as an input to another, so a programmer should bear this in mind. Interactive programs should be avoided if possible. Make your program a filter if possible, as that exactly helps this case.
3. **Program so that you can test early, don't be afraid to throw away code and rewrite it from scratch.**
4. **Write and use tools**, even if they're short-lived, they're better than manual work. Unix-like systems are known for their high scriptability.

This has later been condensed into: do one thing well, write programs to work together, make programs communicate via text streams, a universal interface.

Details such as to what extent/extreme this minimalism ("doing only one thing") should be taken are of course a hot topic of many debates and opinions, the original Unix hackers very often very strict, famous example of which is the "cat -v considered harmful" presentation bashing a relatively simple function added to the cat program that should only ever concatenate files. Some tolerate adding some convenience functions to simple programs, especially nowadays.

**Simple example:** maybe the most common practical example that can be given is piping small command line utility programs; in Unix there exist a number of small programs that do *only one thing but do it well*, for example the cat program that only concatenates and outputs the content of selected files, the grep program that searches for patterns in text etc. In a command line we may use so called pipes to chain some of these simple programs into more complex processing pipelines by redirecting one program's output stream to another one's input. Let's say we want to for example automatically list all first and second level headings on given webpage and write them out alphabetically sorted. We can do it with a command such as this one:

```
wget -q -O - "http://www.tastyfish.cz/lrs/main.html" | grep -i -o "<h[12][^>]*>[^<]*<" | sed "s/[^>]*> *\[([
```

Which may output for example:

```
Are You A Noob?
Did You Know
less_retarded_wiki
Topics
Wanna Help?
Welcome To The Less Retarded Wiki
```

In the command the pipes (|) chain multiple programs together so that the output of one becomes the input of the next. The first command, wget, downloads the HTML content of the webpage and passes it to the second command, grep, which filters the text and only prints lines with headings (using so called regular expressions), this is passed to sed that removes the HTML code and the result is passed to sort that sorts the lines alphabetically -- as this is the last command, the result is then printed out, but we could also e.g. add `> output.txt` at the end to save the result into a text file instead. We also use flags to modify the behavior of the programs, for example `-i` tells grep to work in case-insensitive mode, `-q` tells wget to be silent and not print things such as download progress. This whole wiki is basically made on top of a few scripts like this (compare e.g. to MediaWiki software), so you literally see the manifestation of these presented concepts as you're reading this. This kind of "workflow" is a fast, powerful and very flexible way of processing data for anyone who knows the Unix tools. Notice the relative simplicity of each command and how each one works as a **text filter**; text is a universal communication interface and behaving as a filter makes intercommunication easy and efficient, utilizing the principle of a pipeline. A filter simply takes an input stream of data and outputs another stream of data; it ideally works on-the-go (without having to load whole input in order to produce the output), which has numerous advantages, for example requiring only a small amount of memory (which may become significant when we are running many programs at once in the pipeline, imagine e.g. a server with 10000 users, each one running his own commands like this) and decreasing latency (the next pipe stage may start processing the data before the previous stage finishes). When you're writing a program, such as for example a compression tool, make it work like this.

Compare this to the opposite Windows philosophy in which combining programs into collaborating units is not intended, is possibly even purposefully prevented and therefore very difficult, slow and impractical to do -- such programs are designed for manually performing some predefined actions, mostly using GUI, e.g. painting pictures with a mouse, but aren't made to collaborate or be automatized, they can rarely be used in unintended, inventive ways needed for powerful hacking. Getting back to the example of a compression tool, on Windows such a program would be a large GUI program that requires a user to open up a file dialog, manually select a file to compress, which would then probably go on to load the whole file into memory, perform compression there, and then write the data back to some other file. Need to use the program on a computer without graphical display? Automatize it to work with other programs? Run it from a script? Run it 10000 at the same time with 10000 other similar programs? Bad luck, Windows philosophy doesn't allow this.

**Watch out! Do not misunderstand Unix philosophy.** There are many extremely dangerous cases of misunderstanding Unix philosophy by modern wannabe programmers who can't tell pseudominimalism from true minimalism. One example is the hilarious myth about "React following Unix philosophy" (LMAO this), the devs just show so many misunderstandings here -- firstly of course JavaScript itself is extremely bloated as it's a language aiming for things like comfort, rapid development, "safety" and beginner friendliness to which it sacrifices performance and elegance, an expert hacker trying to write highly thought through, optimized program is not its target group, therefore nothing based on JavaScript can ever be compatible with the Unix way in the first place. Secondly they seem to imply that basically any system of modules follows Unix philosophy -- that's of course wrong, modularity far predates Unix philosophy, Unix philosophy is more than that, merely having a package system of libraries, each of which focuses on some thing (even very broad one like highly complex GUI), doesn't mean those tools are simple (both internally and externally), efficient, communicating in good ways and so on.

**Does Unix philosophy imply universality is always bad?** Well, most likely no, not in general at least -- it simply tells us that for an expert to create art that reaches the peak of his potential it seems best in most cases if he lives in an environment with many small, highly efficient tools that he can tinker with, which allow him to combine them, even (and especially) in unforeseen ways -- to do hacking. Universal tools, however, are great as well, either as a supplement or for other use cases (non-experts, quick dirty jobs and so on) -- after all a general purpose programming language such as C, another creation of Unix creators themselves, is a universal tool that prefers generality over effectiveness at one specific task (for example you can use C to process text but you likely won't match the efficiency of sed, etc.). Nevertheless let us realize an important thing: a universal tool can still be implemented in minimalist way, therefore never confuse a universal tool with a bloated monolith encumbered by feature creep!

{ One possible practical interpretation of Unix philosophy I came up with is this: there's an upper but also lower limit on complexity. "Do one thing" means the program shouldn't be too complex, we can simplify this

to e.g. "Your program shouldn't surpass 10 KLOC". "Do it well" means the programs shouldn't be too trivial because then it is hardly doing it well, we could e.g. say "Your program shouldn't be shorter than 10 LOC". E.g. we shouldn't literally make a separate program for printing each ASCII symbol, such programs would be too simple and not doing a thing well. We rather make a cat program, that's neither too complex nor too trivial, which can really print any ASCII symbol. By this point of view Unix philosophy is really about balance of triviality and huge complexity, but hints that the right balance tends to be much closer to the triviality than we humans are tempted to intuitively choose. Without guidance we tend to make programs too complex and so the philosophy exists to remind us to force ourselves to rather minimize our programs to strike the correct balance. ~drummyfish }

## See Also

- [LRS](#)
  - [Unix](#)
  - [minimalism](#)
  - [suckless](#)
  - [KISS](#)
  - [Windows philosophy](#)
  - [hacking](#)
- 

unretard

## Unretard

Unretarding means aligning oneself with the *less retarded software and society* after acquiring necessary education and mindset. This [wiki](#) should help achieve this goal.

Nowadays unretarding is almost synonymous to learning to live and think in exact opposite ways we have been taught to by modern society.

## See Also

- [how to](#)
  - [nirvana](#)
  - [zen](#)
- 

update\_culture

## Update Culture

Update culture is a malicious mindset emerging in a capitalist society which in technology manifests by developers of a (typically bloated) program creating frequent modifications called "updates" (sometimes also more sneakily masked under terms such as progress or modernization) and forcing users to keep consuming them, e.g. by deprecating or neglecting old versions, dropping backwards compatibility (e.g. Python) or by downright forcing updates in code. This often manifests by a familiar pop-up message:

*"Your software is too old, please update to the latest version."*

In software this process is a lot of times automatized and known as autoupdates, but update culture encompasses more than this, it's the whole mentality of having to constantly keep up, update one's software, hardware and other products, it is part of fear culture, bullshit and consumerism. Normies get all neurotic when they haven't received their weekly updates that give them new content or fake sense of "security". The truth is updates break more things that they fix and make software progressively shittier. STOP FUCKING UPDATING EVERYTHING EVERY 3 SECONDS YOU IDIOTS. Good software is written once and works for hundreds of years without maintenance.

A typical example falling under update culture are web browsers or proprietary operating systems that strive for bloat monopoly.

Update culture is however not limited to computers or technology, hell no. It is the mood of the whole society and applies to things such as fashion, business, gossip, watching TV news every day, browsing social media or constantly updating laws, it is the acceptance and approval of living in a constant stress of having to extort extreme amounts of energy just to keep up with artificially made up bullshit, to race against oneself and others in a never ending artificially sustained race with no winners, just with extremely exhausted participants. Current system of law requires constant everyday maintenance that's extremely costly, law needs to be constantly remade and rewritten to reflect any emerging trend in society because it is so unbelievably complex and tries to encompass every single aspect of our society. Of course we eventually oppose any kind of formal law, however the kind of update culture law is yet orders of magnitude worse -- if we see law as a tool to serve society, this kind of law is an utterly shitty tool similar to a hammer that has to be repaired every second just to keep functioning.

Software updates are usually justified by "muh security" and "muh modern features". Users who want to avoid these updates or simply can't install them, e.g. due to using old incompatible hardware or missing dependency packages, are ridiculed as *poorfags*, idiots and their suffering is ignored. In fact, update culture is cancer because:

- **It is a form of software consumerism**, even if the updates themselves are gratis, they always come at a cost such as potential instability, requiring new hardware, forcing installing more dependencies, required learning to use the new version, or even dropping of old features and malicious code in the updates.
- **It is dangerous**, updates regularly break things, and there are cases where a lot depends on software running smoothly.
- It is bullshit effort, **wasting human work and creating an intentionally high maintenance cost**. Humans, both users and programmers, become slaves to the software.
- **The security justifications are lies**: a true concern for security would lead to unbloating and creating a minimal, stable and well tested software. Update culture in fact constantly pushes newly created vulnerabilities with the updates which are only better in not having been discovered yet, i.e. relying on **security by obscurity**. This creates an intentionally **endless cycle of creating something that will never be finished** (even if it well could be).
- **It kills freedom**. E.g. with the example of web the constant meaningless updates of JavaScript and addition of "features" eliminates any small competition that can't afford to keep up with the constantly changing environment. **This is why we have no good web browsers**.
- **It is actually a huge security risk** (yes, we don't really buy intro security but this still holds). The developer, whoever it is, has the power to remotely push and execute any code at any time to the devices of many users. In fact this can be seen as the definition of backdoor. This is not just an issue of proprietary software, there have been many FOSS projects pushing malware this way (look up e.g. the projects that targeted malware at Russians during the Russia-Ukraine war).

---

usa

## USA

United States of America (also United Shitholes of America, burgerland, USA, US or just "murika") is a dystopian imperialist country of fat, stupid idiots enslaved by capitalism, either rightist or pseudoleftist fascists endlessly obsessed with money, wars, fighting, shooting their presidents and shooting up their schools. Other things they like include guns, oil, throwing nuclear bombs on cities, detonating nuclear bombs in the sea and crashing planes into their own skyscrapers so that they can invade other countries. USA consists of 50 states located in North America, a continent that ancestors of Americans invaded and have stolen from Indians, the natives whom Americans mass murdered. Americans are stupid idiots with guns who above all value constant societal conflict and make the world so that all people are dragged into such conflict.

{ Sorry to some of my US frens :D I love you <3 ~drummyfish }

USA is very similar to North Korea: in both countries the people are successfully led to believe their country is the best and have strong propaganda based on cults of personality, which to outsiders seem very ridiculous but which is nevertheless very effective: for example North Korea officially proclaims their supreme leader Kim Jong-il was born atop a sacred mountain and a new star came to existence on the day of his birth, while Americans on the other hand believe one of their retarded leaders named George Washington was a divine god who was PHYSICALLY UNABLE TO TELL A LIE, which was actually taught at their schools. North Korea is ruled by a single political party, US is ruled by two practically same militant capitalist imperialist parties (democrats and republicans), i.e. de-facto one party as well. Both countries are obsessed with weapons (especially nuclear ones) and their military, both are highly and openly fascist (nationalist). Both countries are full of extreme propaganda, censorship and hero culture, people worship dictators such as Kim Jong-un or Steve Jobs. US is even worse than North Korea because it exports its toxic culture all over the whole world and constantly invades other countries, it is destroying all other cultures and leads the whole world to doom and destruction of all life, while North Korea basically only destroys itself.

In US mainstream politics there exists no true left, only right and pseudoleft. It is only in extreme underground, out of the sight of most people, where occasionally something good comes into existence as an exception to a general rule that nothing good comes from the US. One of these exceptions is free software (established by Richard Stallman) which was however quickly smothered by the capitalist open source counter movement.

On 6th and 9th August 1945 **USA murdered about 200000 civilians**, most of whom were innocent men, women and children, by throwing atomic bombs on Japanese cities Hiroshima and Nagasaki. The men who threw the bombs and ordered the bombing were never put on trial, actually most Americans praise them as heroes and think it was a good thing to do.

**Americans are uber retarded** for example in trying to somehow pursue both self interest and "social equality", it's extremely ridiculous, an american brain is literally incapable of imagining someone who doesn't at his core work on the basis of self interest, so the American that tries to identify with "wanting equality and human rights" just comes up with hugely fucked up arguments like "SKIN COLOR IS JUST ILLUSION THEREFORE WE ARE ALL EQUAL" -- because he inevitably sees differences implying oppression because self interest just cannot be not present (this idea won't even occur for a second to him during his whole lifetime, it's simply something he NEVER can physically think), his mind is hard wired to be unable of grasping the idea of accepting difference between people while giving up the self interest of falling to fascism as a consequence. Similar arguments are encountered e.g. regarding vegetarianism: an American supporting vegetarianism will resort to denying evolution, biology and anatomy and will argue something like "HUMANS ARE HERBIVORES BECAUSE THIS FEMINIST SCIENTIST SAYS IT AND MEAT KILLS US SO WE MUST NOT EAT IT", again because he just thinks that admitting meat is healthy to us automatically implies we have to eat it because self interest is just something that's an inherent part of laws of physics; a normal (non retarded) vegetarian will of course admit not eating meat at all is probably a bit unhealthy, but it's a voluntary choice made of altruistic love towards other living beings who now don't have to die for one's tastier food.

{ "List of atrocities by the United States" is the longest page on leftypedia :-)  
[https://wiki.leftypol.org/wiki/List\\_of\\_atrocities\\_committed\\_by\\_the\\_United\\_States](https://wiki.leftypol.org/wiki/List_of_atrocities_committed_by_the_United_States). ~drummyfish }

Here is a comparison of average European country before and after infestation with American culture (judged by Czech Republic, the author's country of residence, but it's more or less the same in whole EU):

| what               | before US culture (~1990s) | after US culture (~2020s) |
|--------------------|----------------------------|---------------------------|
| mass shootings     | no                         | yes                       |
| <u>feminazism</u>  | no                         | yes                       |
| <u>gay fascism</u> | no                         | yes                       |
| <u>slavery</u>     | mild                       | extreme                   |
| public toilets     | yes                        | no                        |
| free healthcare    | yes                        | no                        |
| corruption         | mild                       | extreme                   |
| youtubers          | no                         | yes                       |
| Santa Claus        | no                         | yes                       |

| what                         | before US culture (~1990s) | after US culture (~2020s) |
|------------------------------|----------------------------|---------------------------|
| ads                          | mild                       | unbearable, aggressive    |
| morality                     | sometimes                  | no                        |
| idiots                       | some                       | all                       |
| <u>free speech</u>           | mostly                     | no                        |
| <u>Apple</u>                 | no                         | yes                       |
| <u>privacy</u> hysteria      | no                         | yes                       |
| social security              | yes                        | no                        |
| old age pension              | yes                        | no                        |
| <u>fear culture</u>          | no                         | extreme                   |
| update culture, consumerism  | mostly not                 | extreme                   |
| poverty                      | none                       | extreme                   |
| financial crisis             | no                         | yes                       |
| society collapsing           | no                         | yes                       |
| <u>art</u> and culture       | good                       | none                      |
| <u>toxicity</u>              | rare                       | extreme                   |
| <u>technology</u>            | fine                       | worst in history          |
| plastic surgery abominations | no                         | yes                       |
| TV content to ad ratio       | > 10                       | < 1                       |
| wanted to commit suicide     | no                         | yes                       |
| society worked               | kinda                      | no                        |

In Europe, or maybe just anywhere else in the world, you are afraid of getting hit by a car because you might die, in America you afraid of it because you couldn't afford the ambulance bill and would get into unpayable debt (yes, even if you pay "health insurance"). You can literally find footage of half dead people running away from ambulances so that they don't have to go to debt for being kept alive. In Europe you are afraid to hit someone with a car because you might kill him, in America you are afraid of it because he might sue you. This is not an exaggeration or joke, it's literally how it is -- it's incredible how people can believe the country is somehow "more advanced", it is quite literally the least developed country in history.

---

used

## Used

Used is someone using technology that abuses him, most notably proprietary software. For example those using Windows are not users but useds. This term was popularized by Richard Stallman.

---

usenet

## Usenet

Usenet (User's Network) is an ancient digital discussion network -- a forum -- that existed long before the World Wide Web. At the time it was very popular, it was THE place to be, but nowadays it's been forgotten by the mainstream, sadly hardly anyone remembers it.

Back in the day there were no web browsers, there was no web. Many users were also **not connected through Internet** as it was expensive, they normally used other networks like UUCP working through phone lines. They could communicate by some forms of electronic mail or by directly connecting to servers and leaving messages for others there -- these servers were called BBSes and were another popular kind of "social network" at the time. Usenet was a bit different as it was decentralized (see also federation) -- it wasn't stored or managed on a single server, but on many independent servers that provided users with access to the network. This access was (and is) mostly paid (to lurk for free you can search for Usenet archives online). To access Usenet a **newsreader** program was needed, it was kind of a precursor to web

browsers (nowadays newsreaders are sometimes built into e.g. email clients). Usenet was lots of time not moderated and anonymous, i.e. kind of free, you could find all kinds of illegal material there.

Usenet invented many things that survive until today such as the words spam and FAQ as well as some basic concepts of how discussion forums even work. It was also generally quite a lot for the free speech, which is good.

Usenet was originally ASCII only, but people started to post binary files encoded as ASCII and there were dedicated sections just for posting binaries, so you co go piiiiiiiirating.

It worked like this: there were a number of Usenet servers that all collaborated on keeping a database of *articles* that users posted (very roughly this is similar to how blockchain works nowadays); the servers would more or less mirror each other's content. These servers were called *providers* as they also allowed access to Usenet but this was usually for a fee. The system uses a NNTP (Network News Transfer Protocol) protocol. The articles users posted were also called *posts* or *news*, they were in plain text and were similar to email messages (mailing lists actually offer a similar experience). Other users could reply to posts, creating a discussion thread. Every post was also categorized under certain **newsgroup** that formed a hierarchy (e.g. comp.lang.java). After so called *Big Renaming* in 1987 the system eventually settled on 8 top level hierarchies (called the *Big 8*): comp.\* (computers), news.\* (news), sci.\* (science), rec.\* (recreation), soc.\* (social), talk.\* (talk), misc.\* (other) and humanities.\* (humanities). There was also another one called alt.\* for "controversial" topics (see especially alt.tasteless). According to Jargon File, by 1996 there was over 10000 different newsgroups.

Usenet was the pre-web web, kind of like an 80s reddit which contained huge amounts of historical information and countless discussions of true computer nerds which are however not easily accessible anymore as there aren't so many archives, they aren't well indexed and direct Usenet access is normally paid. It's a shame. It is possible to find e.g. initial reactions to the AIDS disease, people asking what the Internet was, people discussing future technologies, the German cannibal (Meiwes) looking for someone to eat (which he eventually did), Bezos looking for Amazon programmers, a heated debate between Linus Torvalds and Andrew Tanenbaum about the best OS architecture (the "Linux is obsolete" discussion) or Douglas Adams talking to his fans. There were memes and characters like BIFF, a kind of hilarious noob wannabe cool personality. Some users became kind of famous, e.g. Scott Abraham who was banned from Usenet by court after an extremely long flame war, Alexander Abian, a mathematician who argued for blowing up the Moon (which according to his theory would solve all Earth's issues), Archimedes Plutonium who suggested the Universe was really a big plutonium atom (he later literally changed his name to Plutonium lol) or John Titor (pretend time traveler). There are also some politically incorrect groups like alt.niggers lol.

{ I mean I don't remember it either, I'm not that old, I've just been digging on the Internet and in the archives, and I find it all fascinating. ~drummyfish }

## The Newsgroup Hierarchy

Here are some notable groups, placed in the group hierarchy:

- *comp*: computers, part of Big 8
  - ◆ *ai*: AI
  - ◆ *answers*
  - ◆ *bbs*: BBS
  - ◆ *binaries*
  - ◆ *compression*: compression
  - ◆ *editors*: text editors
  - ◆ *graphics*: computer graphics
  - ◆ *internet*: Internet
    - ◇ *services*
    - ◇ *wiki*: wikis
  - ◆ *lang*: programming languages
    - ◇ *asm*: assembly
    - ◇ *c*: C language
    - ◇ *forth*: Forth



- ◊ *lisp*: Lisp
  - ◊ *python*: Python language
- ♦ *object*: OOP
- ♦ *os*: operating systems
  - ◊ *linux*: Linux
  - *misc*
- ♦ *programming*: programming
- ♦ *society*: technology as related to society
- ♦ *theory*: theoretical compsci
- ♦ *unix*: Unix
- *humanities*: humanities, part of Big 8
  - ♦ *lit*: literature
  - ♦ *music*: music
- *misc*: miscellaneous, part of Big 8
  - ♦ *books*: books
  - ♦ *business*: business
  - ♦ *education*: education
  - ♦ *fitness*
  - ♦ *news*: news from various regions
  - ♦ *survival*
- *news*: about the network itself, part of Big 8
  - ♦ *announce*
  - ♦ *software*
- *rec*: recreational activity, part of Big 8
  - ♦ *animals*
  - ♦ *arts*: art
  - ♦ *drugs*
  - ♦ *food*
  - ♦ *games*
  - ♦ *humor*: funny stuff
  - ♦ *knives*: knives
- *sci*: science, part of Big 8
  - ♦ *answers*: answers and questions
  - ♦ *bio*: biology
  - ♦ *chem*: chemistry
  - ♦ *crypt*: cryptography
  - ♦ *fractals*: fractals
  - ♦ *lang*: languages, linguistics
  - ♦ *logic*: logic
  - ♦ *math*: math { See, it's science. ~drummyfish }
  - ♦ *physics*: physics
  - ♦ *space*: space
- *soc*: social issues and socializing, part of Big 8
  - ♦ *atheism*: atheism
  - ♦ *bi*: bisexual
  - ♦ *culture*: culture
  - ♦ *feminism*: feminism
  - ♦ *history*: history
  - ♦ *politics*: politics
  - ♦ *religion*: religion
    - ◊ *christian*: Christianity
    - ◊ *islam*: Islam
  - ♦ *women*: women
- *talk*: talk, part of Big 8
  - ♦ *euthanasia*
  - ♦ *philosophy*
  - ♦ *politics*
  - ♦ *rumors*
- *alt*: alternative, weird/controversial/NSFW/NSFL
  - ♦ *alien*

- ♦ *anarchism*: [anarchism](#)
- ♦ *ascii-art*: [ASCII art](#)
- ♦ *atheism*: [atheism](#)
- ♦ *abortion*
- ♦ *binaries*: binary file sharing, [piracy](#)
  - ◊ *games*
  - ◊ *pictures*: picture sharing
    - *erotica*: erotic pictures
      - *animals*
      - *child*: [pedo](#)
        - ♦ *female*
        - ♦ *male*
      - *lolita*: [loji](#)
      - *pre-teen*
      - *tasteless*
      - *teen*
- ♦ *drugs*
- ♦ *freedom*
- ♦ *fun*: [fun](#)
- ♦ *games*: [games](#)
  - ◊ *doom*: [Doom](#)
    - *ii*
  - ◊ *duke3d*: [Duke3D](#)
  - ◊ *video*: vidya
    - *game-boy*: [GameBoy](#)
- ♦ *humor*
- ♦ *jokes*: [jokes](#)
- ♦ *niggers*: [niggers](#)
- ♦ *sex*: [sex](#)
  - ◊ *pedophilia*: [pedo](#)
    - *pictures*
    - *swaps*
  - ◊ *sheep*
  - ◊ *zoophilia*
- ♦ *suicide*: [suicide](#)
- ♦ *tasteless*
  - ◊ *jokes*
- ♦ *ufo*: [UFO](#)
- ♦ *tv*: [TV](#)

## Where To Freely Browse Usenet

Search for Usenet archives, I've found some sites dedicated to this, also [Internet archive](#) has some newsgroups archived. [Google](#) has Usenet archives on a website called "Google groups" (now sadly requires login). There is a nice archive at <https://www.usenetarchives.com>. Possibly guys from Archive Team can help (<https://wiki.archiveteam.org/index.php/Usenet>, <https://archive.org/details/usenet>, ...). See also <http://www.eternal-september.org>. There is an archive from 1981 accessible through [gopher](#) at <gopher://gopher.quux.org/1/Archives/usenet-a-news>. Also <https://yarchive.net/>. Also search for *Henry Spencer's UTZOO NetNews Archive*.

{ Also here's some stuff <https://old.reddit.com/r/usenet/wiki/index>. ~drummyfish }

## See Also

- [BBS](#)
- [modem world](#)
- [multi user dungeon](#)
- [mailing list](#)
- [FidoNet](#)

# Uxn

{ WIP, researching this etcetc. ~drummyfish }

Uxn is a minimalist self hosted stack-based virtual machine 8bit/16bit computer aiming for great simplicity and portability. It is quite nice and impressive, having its own instruction set, assembly language, many implementations and many programs written for it already (e.g. Left text editor, Noodle drawing tool etc.); it was made by the author of xxiivv wiki (some weird narcissist self proclaimed artist that's sailing the seas or something). From the minimalist point of view uxn really seems to be going in the right direction, it is inspired by old computers such as NES and C64, practicing real minimalism (not just pseudominimalism or just "lightweight minimalism") -- that's pretty awesome -- however its presentation is shit and while there are many free as in freedom implementations of uxn, official supplemental material to uxn (on the xxiivv wiki), such as its specification, is proprietary (NC) and therefore should be avoided and boycotted.

Uxn is similar to other projects such as IBNIZ, and can be compared to some of our projects as well, for example SAF, but mainly comun -- the goals of uxn and comun may be seen as significantly overlapping, aiming to create a minimalist, completely independent from-ground-up computing "stack", an extremely portable platform for minimalist programs, and they do so in a similar way (both are e.g. stack based, inspired by Forth). **To quickly compare uxn and comun:** comun is more of a pure programming language focusing only on expressing algorithms without talking about I/O or instruction sets, uxn on the other hand really is a computer (even if initially only virtual), one that comes with its own instruction set, language and protocols for communication with peripheral devices, though the computer is purposefully made so that it can be implemented as a virtual machine running on other computers. Comun is a low level language but higher level than assembly (having e.g. control structures and a concept of "native integer" type), usually compiling to bytecode, while uxn is programmed directly in assembly and tied to its virtual machine's architecture and specifications. Comun is trying to stay more abstract, hardware independent and be more close to math notation, it doesn't assume any native integer size or working memory size, it doesn't use any English keywords, it assumes as little as possible about its platform -- it is trying to be a "better, simpler C". Uxn is more of a "new NES", a "practically useful fantasy console", an idealization and improvement of old computers, it has a hardcoded amount of memory, specified integer size (8 or 16 bit), uses assembly with English mnemonics just like the old computers etc. As for complexity, uxn is probably a bit simpler, or rather allowing smaller implementations than those of full comun, though simplified versions of comun (such as minicomun) may possibly be as simple or simpler than uxn and specification of full comun (a possible measure of complexity) is extremely small and will probably compare to or beat uxn. Uxn only has 32 instructions and its self hosted implementation is around 2000 bytes big, while current comun's bytecode has around 80 instructions and self hosted compiler will probably have a few thousand lines of code (as it really is a library, compiler, interpreter and simple optimizer, as opposed to mere assembler). Comun is a completely selfless, absolutely public domain free software, while uxn has a selfish proprietary (NC) specification.

## Details

{ **Start rant:** the description of everything by 100rabbits is **so fucking hard and painful as fuck to understand** -- not because the described technology itself would be complicated but the writing is just really REALLY bad, it's hard to say exactly what it is but he just seems to be trying to write poetry in technical specifications, that's just extremely fucked up. There are tables where meaning of rows and columns is left to be guessed, hyperlinks of important terms lead to shitass long articles about something completely else, he invents 10 different fancy sounding words he uses interchangeably without ever explaining their meaning in a non-cryptic way or which simply have circular definitions. It's also fucking infuriating there are some **fucking shitty childish drawings randomly inserted into opcode specification** along with some random hand gesture signals for the opcodes and what the fuck. It's like he's abusing the specification to force feed you his ugly drawings which he probably thinks look good while also trying to teach you what sounds animals make as if it's fucking kindergarden. Fuck this shit, I now have to read through it and make sense of it so you don't have to. I'll prolly rather read some 3rd party tutorial lol. ~drummyfish }

Here is a kind of concise sum up of uxn "ecosystem"/terminology/specs/etc.:

- **uxn** (also *uxn CPU*): The virtual machine, "computing backend", something that runs uxn binary *roms*.
  - ♦ so called *one page computer* (can implement itself in at most 500 lines of own assembly)
  - ♦ has **2 circular stacks**: working stack and return stack, each one has size of 256 bytes.
  - ♦ **memory**: total size 64 kB, memory map is following:
    - ◊ 16 varvara devices, each one taking 16 bytes: the whole device section is called a **page**, the bytes inside it are called **ports**. A port that holds a callback address is called **vector** (e.g. a mouse device will have a vector for *mouse click* callback).
    - ◊ address 256: start of program instructions.
    - ◊ extensible with so called *banks*.
  - ♦ **instructions**: Each one takes 8 bits. Every value is a valid instruction. Instruction format is following:
    - ◊ lowest 5 bits: instruction opcode (i.e. there are 32 instructions in total)
    - ◊ following bit (2): if set, instruction works with 16 bit shorts (otherwise it works with 8 bit bytes).
    - ◊ following bit (*k*): if set, instruction will not pop any operands.
    - ◊ following bit (*r*): if set, instruction operates on the return stack (otherwise it operates on working stack).
- **uxntal** (also just *tal*): Assembly language, i.e. human readable language that compiles to a binary *rom*, with more or less one-to-one mapping between machine code instructions and the language mnemonics. Tal is what uxn program are written in; the tal files have .tal extension. A quick sum up of the language follows:
  - ♦ Comments are in parentheses, as ( comment ).
  - ♦ Mnemonics such as ADD, EQU, POP etc. are used.
  - ♦ Postfix notation.
  - ♦ "Runes" are used as directives and syntax sugar, for example | says where in memory to store the instruction, @something creates a jump label etc.
  - ♦ Macros (inlined instructions) are supported.
- **rom**: Compiled binary executable that can be directly run by an *uxn CPU*. The rom files have .rom extension.
- **varvara**: A device, "frontend", peripheral that's connected to *uxn CPU* to perform input/output, for example a screen, keyboard or terminal.

TODO: subroutines, ...

## See Also

- IBNIZ
- xxiivv
- comun
- SAF
- fantasy console

---

vector

## Vector

Vector is a basic mathematical object that expresses direction and magnitude (such as velocity, force etc.) and is very often expressed as (and many times misleadingly equated with) an "array of numbers". Nevertheless in programming 1 dimensional arrays are somewhat synonymous with vectors -- for example in two dimensional space an array [4,3] expresses a vector pointing 4 units to the "right" (along X axis) and 3 units "up" (along Y axis) and has the magnitude 5 (which is the vector's length). Vectors are one of the very basic concepts of advanced math and are used almost in any advanced area of math, physics, programming etc. -- basically all of physics and engineering operates with vectors, programmers will mostly encounter them in areas such as 3D graphics, physics engines (forces, velocities, acceleration, ...), machine learning (feature vectors, ...) or signal processing (e.g. Fourier transform just interprets a signal as a vector and transforms it to a different basis) etc. In this article we will implicitly focus on vectors from programmer's point of view (i.e. "arrays of numbers"), which to a mathematician will seem very simplified, but we'll briefly also foreshadow the mathematical view.

(NOTE: the term *vector* is used a lot in different contexts and fields, usually with some connection to the mathematical idea of vector which is sometimes however very loose, e.g. in low-level programming *vector* means a memory holding an address of some event handler. It's better to just look up what "vector" means in your specific area of interest.)

Just like in elemental mathematics we deal with "simple" numbers such as 10,  $-2/3$  or  $\pi$  -- we retrospectively call such "simple" numbers **scalars** -- advanced mathematics generalizes the concept of such a number into vectors ("arrays of numbers", e.g.  $[1, 0, -3/5]$  or  $[0.5, 0.5]$ ) and yet further to matrices ("two dimensional arrays of numbers") and defines a way to deal with such generalizations into **linear algebra**, i.e. we have ways to add and multiply vectors and matrices and solve equations with them, just like we did in elemental algebra (of course, linear algebra is a bit more complex as it mixes together scalars, vectors and matrices). In yet more advanced mathematics the concepts of vectors and matrices are further generalized to **tensors** which may also be seen as "N dimensional arrays of numbers" but further add new rules and interpretation of such "arrays" -- vectors can therefore be also seen as a tensor (of rank 1) -- note that in this context there is e.g. a fundamental distinction between row and column vectors. Keep in mind that vectors, matrices and tensors aren't the only possible generalization of numbers, another one is e.g. that of complex numbers, quaternions, p-adic numbers etc. Anyway, in this article we won't be discussing tensors or any of the more advanced concepts further, they are pretty non-trivial and mostly beyond the scope of mere programmer's needs :) We'll keep it at linear algebra level.

**Vector is not merely a coordinate**, though the traditional representation of it suggest such representation and programmers often use vector data types to store coordinates out of convenience (e.g. in 3D graphics engines vectors are used to specify coordinates of 3D objects); vector should properly be seen as a **direction and magnitude** which has **no position**, i.e. a way to correctly imagine a vector is something like an **arrow** -- for example if a vector represents velocity of an object, the direction (where the arrow points) says in which direction the object is moving and the magnitude (the arrow length) says how fast it is moving (its speed), but it doesn't say the position of the object (the arrow itself records no position, it just "hangs in thin air").

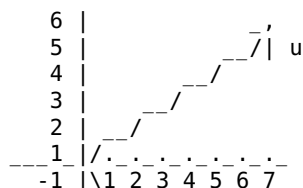
Watch out, **mathematicians dislike defining vectors as arrays of numbers** because vectors are essentially NOT arrays of numbers, such arrays are just one way to express them. Similarly we don't have to interpret any array of numbers as a vector, just as we don't have to interpret any string of letter as a word in human language. A vector is simply a direction and magnitude, an "arrow in space" of  $N$  dimensions; a natural way of expressing such arrow is through multiples of basis vectors (so called components), BUT the specific numbers (components) depend on the choice of basis vectors, i.e. the SAME vector may be written as an array of different numbers (components) in a different basis, just as the same concept of a dog is expressed by different words in different languages. Even with the same basis vectors the numbers (components) depend on the method of measurement -- instead of expressing the vector as a linear combination of the  $N$  basis vectors we may express it as  $N$  dot products with the basis vectors -- the numbers (components) will be different, but the expressed vector will be the same. Mathematicians usually define vectors abstractly simply as members of a **vector space** which is a set of elements (vectors) along with operations of addition and multiplication which satisfy certain given rules (axioms).

## How Vectors Work

Here we'll explain the basics of vectors from programmer's point of view, i.e. the traditional "array of numbers" simplification (expressing a linear combination of basis vectors).

Given an  $N$  dimensional space, a vector to us will be an array of real numbers (in programming floats, fixed point or even just integers) of length  $N$ , i.e. the array will have  $N$  components (2 for 2D, 3 for 3D etc.).

For example suppose 2 vectors in a 2 dimensional space,  $u = [7, 6]$  and  $v = [2, -3.5]$ . To visualize them we may simply plot them:



$$\begin{array}{c|c} -2 & \backslash \\ -3 & \backslash | \\ -4 & -'''' v \\ -5 & \end{array}$$

NOTE: while for normal (scalar) variables we use letters such as  $x$ ,  $y$  and  $z$ , for vector variables we usually use letters  $u$ ,  $v$  and  $w$  and also put a small arrow above them as:

$$u = [7, 6], \quad v = [2, -3.5]$$

The vector's components are referred to by the vector's symbol and subscript or, in programming, with a dot or square brackets (like with array indexing), i.e.  $u.x = 7$ ,  $u.y = 6$ ,  $v.x = 2$  and  $v.y = -3.5$ . In programming data types for vectors are usually called `vecN` or `vN` where  $N$  is the number of dimensions (i.e. `vec2`, `vec3` etc.).

Also note that we'll be writing vectors "horizontally" just as shown, which means we're using so called row vectors; you may also see usage of so called columns vectors as:

$$\begin{matrix} \rightarrow & |7| & \rightarrow & |2| \\ u = & |6|, & v = & |-3.5| \end{matrix}$$

Now notice that we do NOT plot the vectors as points, but as arrows starting at the origin (point [0,0]) -- this is again because we don't normally interpret vectors as a position but rather as a **direction with magnitude**. The direction is apparent from the picture ( $u$  points kind of top-right and  $v$  bottom-right) and can be exactly computed with arcus tangent (i.e. angle of  $u = \text{atan}(6/7) = 40.6 \text{ degrees}$ , ...), while the magnitude (also length or norm) is given by the vector's Euclidean **length** and is denoted by the vector's symbol in `||` brackets (in programming the function for getting the length may be called something like `len`, `size` or `abs`, even though absolute value is not really a mathematically correct term here), i.e.:

```
->
||u|| = sqrt(7^2 + 6^2) ~= 9.22

->
||v|| = sqrt(2^2 + -3.5^2) ~= 4.03
```

In fact we may choose to represent the vectors in a format that just directly says the angle with the X axis (i.e. direction) and magnitude, i.e.  $v$  could be written as  $\{40.6 \text{ degrees}, 9.22...\}$  and  $u$  as  $\{-56.32 \text{ degrees}, 4.03...\}$  (see also polar coordinates). This represents the same vectors, though we don't do this so often in programming.

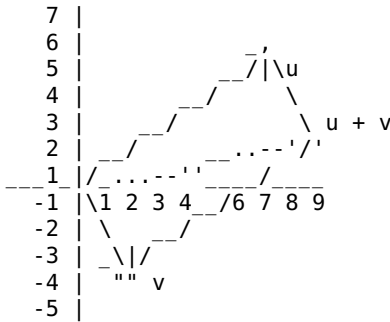
A vector whose magnitude is exactly 1 is called a **unit vector** -- such vectors are useful in situations in which we only care about direction and not magnitude.

The vectors  $u$  and  $v$  may e.g. represent a velocity of cars in a 2D top-down racing game -- the vectors themselves will be used to update each car's position during one game frame and their magnitudes may be displayed as the current speed of each car to their drivers. However keep in mind the vector magnitude may also represent other things, e.g. in a 3D engine a vector may be used to represent camera's orientation and its magnitude may specify e.g. its field of view, or in a physics engine a vector may be used to represent a rotation (the direction specifies the axis of rotation, the magnitude specifies the angle of rotation).

**But why not just use simple numbers?** A velocity of a car could just as well be represented by two variables like `carVelocityX` and `carVelocityY`, why all this fuzz with defining vectors n shit? Well, this simply creates an abstraction that fits to many things we deal with and generalizes well, just like we e.g. define the concept of a sphere and cylinder even though fundamentally these are just sets of points. If for example we suddenly want to make a 3D game out of our racing game, we simply start using the `vec3` data type instead of `vec2` data type and most of our equation, such as that for computing speed, will stay the same. This becomes more apparent once we start dealing with more complex math, e.g. that related to physics where we have many forces, velocities, momenta etc. This becomes even more apparent when we start to look into operations with vectors which are really what makes vectors vectors.

Some of said **operations** with vectors include:

- **addition** (vector + vector = vector): Just like with numbers we can add vectors, i.e. for example if there are two forces acting on an object and we represent them by vectors, we can get the total force as a vector we get by adding the individual vectors. Addition is pretty straightforward, we simply add each component of both vectors, e.g.  $u + v = [7 + 2, 6 - 3.5] = [9, 2.5]$ . Geometrically addition can be seen as drawing one vector from the other vector's endpoint (note that the order doesn't matter, i.e.  $u + v = v + u$ ):



- **subtraction** (vector - vector = vector): Subtracting vector  $v$  from vector  $u$  is the same as adding  $v$  times  $-1$  (i.e.  $v$  with opposite direction) to  $u$ .
- **multiplication**: There are several ways to "multiply" vectors.
  - ♦ **scalar multiplication** (scalar \* vector -> vector): Multiplying vector  $u$  with scalar  $x$  means scaling the vector  $v$  along its direction by the amount given by  $x$  -- for this we simply multiply each component of  $u$  by  $x$ , e.g. if  $x = 0.5$ , then  $u * x = [7 * 0.5, 6 * 0.5] = [3.5, 3]$ .
  - ♦ **dot product** (vector \* vector -> scalar): Dot product of vectors  $u$  and  $v$  is a very common operation and can be used to determine the angle between the vectors, it is denoted as  $u \cdot v$  and is computed as  $u \cdot v = u.x * v.x + u.y * v.y + u.z * v.z + \dots$ ; the value this gives is the cosine of the angle between the vectors times the magnitude of  $u$  times the magnitude of  $v$ . I.e. if  $u$  and  $v$  have length of 1 (are normalized), we directly get the cosine of the angle and can get the angle with the acos function. This is used in many situations, e.g. in graphics shaders dot product is used to determine the angle at which light hits a surface which says how bright the surface will appear.
  - ♦ **cross product** (vector \* vector -> vector): Cross product basically gives us a vector that's perpendicular to the input vectors.
  - ♦ **vector matrix multiplication** (vector \* matrix -> vector): Multiplying a vector by matrix can achieve e.g. a specific transformation of the vector (such as rotating it). For details see the article about matrices.
- **normalization**: Normalization forces a vector's magnitude to a certain amount, typically 1 -- we do this when we have a vector and simply want to just discard its magnitude because we only care about direction (e.g. when doing a dot product to get an angle between vectors we first normalize them). To normalize a vector (to length 1) simply divide all its components by the current magnitude, e.g.  $normalized(u) = [u.x / ||u||, u.y / ||u||] = [0.76..., 0.65...]$ .

## Code Example

TODO

## See Also

- [linear algebra](#)
- [matrix](#)
- [racetrack](#)

---

venus\_project

# The Venus Project

The Venus Project is a big project established by Jacque Fresco, already running for decades, aiming for a voluntary and rational transition towards an ideal, highly technological and automated society without money, scarcity, need for human work, social competition, wars and violence, a society in which people would have abundance thanks to so called resource based economy, where they would be collaborating, loving, respecting the nature, caring for others and free to pursue their true potential. It is similar to the Zeitgeist Movement and TROM. In its views, goals and means the Venus Project is extremely close to LRS and we highly support it, however watch out: while the ideas behind the project are good, the project itself is a bit sus and may show internal corruption just as the ESF, TROM and other projects -- use your brain, follow ideas, not people.

There is a non-profit organization called One Community that tries to pursue goals set by the Venus Project and strive for what they call *Highest Good*. Its website is at <https://www.onecommunityglobal.org>.

## Overview

{ The following is based mainly on my understanding of what I've read in Fresco's book *The Best That Money Can't Buy*. I recommend the book for an overall overview of the project. ~drummyfish }

The project is a child of Jacque Fresco, a generalist futurist who sadly died in 2017, who worked on it for many decades with his life partner Roxanne Meadows. It has a center located in Florida and is a nonprofit organization that performs research, education and prototype technology according to their ideas of a future we should strive for.

Although the project seems to be avoiding specific political labels (possibly as to avoid historical associations), it is de facto **anarcho pacifist communist** movement (i.e. politically the same as LRS). Very nicely it also seems, at least as of 2022, uninfected with the SIW cancer -- fight culture and fascism goes directly against their goals and Fresco explicitly stated that we have to stop constantly fighting for human rights and rather establish a society with human rights built-in.

Fresco highly criticizes today's society and just as us says it only tries to cure the symptoms (search the solutions within the current framework and mindset) rather than the root cause of its issues (the system itself). He mainly criticizes the presence of the monetary system and laws -- currently taking the form of capitalism -- which he correctly blames for most today's issues such as artificial scarcity, hunger, wars, fascism, lack of social security, poverty, wage slavery, destruction of natural environment, waste, energy crisis, planned obsolescence, deteriorating psychological health etc. He says with the presence of advanced technology we have this system is highly outdated (for example it forces artificial scarcity because only scarce resources can be sold, unlike for example air) and points out the fact that we have more than enough resources for everyone on Earth and could live in abundance and peace, practicing **collaboration rather than competition**. Therefore he argues that we have to eliminate money, barter and markets from the society and change the very basis of whole society, down to our mentality and outdated historical associations (and eventually even language which should be closer to the scientific language).

He argues to replace the monetary system with so called **resource based economy** (RBE) which should be a pillar of the future society. RBE is called an "economy" but doesn't use any money or barter, it starts by declaring that all Earth's resources are a common heritage of all people on Earth -- it basically means that "everything is available to everyone", i.e. no one can own resources of the Earth, they belong to us all and whoever needs something can take it. RBE would be supported by high automatization and computer monitoring to deliver resources where they are needed. Normies usually can't comprehend that this could work, they say "but then someone will just steal everything", but Fresco correctly argues that with correct and rational use of our technology we can, unlike in the past, already extract as many resources as to satisfy everyone with high abundance; basically we can make for example food as abundant as is air nowadays -- no one will be (and can be) stealing food when there's more free food than anyone can eat, just as stealing air isn't a concern nowadays.

This should therefore also eliminate the need for complex laws -- when no one is stealing, we don't need laws against stealing etc. Elimination of money and laws will remove the need for bullshit jobs such as lawyers, judges, politicians, marketing guys, bankers etc., freeing more people and getting rid of a lot of unnecessary



work and burden of society.

Fresco supports this by the fact that **human behavior is determined by the environment and upbringing** -- nowadays we have criminality mostly because firstly people are poor, i.e. pushed into illegal activity, and secondly nurtured by the competitive propaganda that teaches them, right from little children, to fight and compete with others. In a caring society that provides all their needs and raises them in the spirit of collaboration and love towards others criminals will be almost non existent, there will simply be no gain from it.

The project further seeks to **eliminate the need for human work**: all work, including complex decision making, would be automated. Bullshit jobs would be removed and maintenance reduced to minimum. People would be free to pursue their true interests and could fully and freely devote themselves to it. Again normies usually say something like "BUT THEN NOBODY AIN'T GONNA WORK". Well, firstly that wouldn't be an issue since no human work would even be needed anymore, and secondly Fresco correctly answers by saying that competition and force isn't the only drive of human activity, people are motivated for work and creative activity by other phenomena such as curiosity, sense of accomplishment, boredom, moral values etc., and usually even perform better than when forced to it. Maslow's Hierarchy of Needs is a well known psychological model that says that once basic needs such as food and shelter are satisfied (which RBE will accomplish), people start voluntarily pursuing higher needs such as art, science and other creative work. People did work and create long before money and jobs existed. The idea of reducing or eliminating human work is already being considered nowadays in the form of universal basic income -- experiments have been confirming that it works.

Venus Project stresses the important of **science** -- their approach is strictly scientific -- technology (such as AI and sensors all over the Earth) and rationality and argues for application of technology to everything as that, in their opinion, will allow RBE and remove the need for human government. **There should be no human governing the society**, decisions will be made mostly by machines in a network of decentralized cities all over the Earth ("technophobes" are informed that even nowadays we put our lives into the hands of machines, e.g. in planes or with pacemakers, and they do better job than humans). Technology should be sustainable, respect the nature and be aligned with it, i.e. not fight against it but rather direct its forces towards good causes. Protection of the environment and integration of natural elements in cities is stressed. Only clean and safe energy would be used. Earth carrying capacity should be respect, i.e. people would avoid overpopulation by voluntary birth control.

The project is for **absolute freedom of information** -- there would be no intellectual property (copyright, patents, ...), no trade secrets, state secrets and probably also no personal secrets, as in a non-competitive society there wouldn't be a danger of abusing personal information. They argue that despite computer sensors being present everywhere, there would simply be no need for surveillance of people as there would be no corporations, no criminality etc.

It also opposes nationalism, racism and other forms of privilege and inequality. However this shouldn't be forced in the SIW style, it should rather come naturally thanks to fixing the the root cause of these issues (removing competition, governments, money etc.).

Education would play a huge role in the society -- again, it wouldn't be forced on children, they should want to go to school because education would be fun and give them freedom to pursue their interests. There would be no grades and it should teach high scientific and critical thinking, rational discussion, nonviolent resolution of conflicts, collaboration through group project and collaborative games, love towards nature (e.g. by projects involving growing plants) etc. Generalism would be preferred before hyper specialization (which we see nowadays).

Fresco also addresses the fear of some people of people becoming too uniform and losing individuality -- he stresses that individuality would be focused on, uniformity would only lie in common goals and caring for all humans and nature. Unlike in current society, each human would have the freedom to pursue his true interests and goals.

The project claims it has years of research and seems to have a great number of specific ideas for what the technology might look like, how we would harness energy, travel etc. There are many 3D visualizations. Fresco claims that in the new society everyone would have a higher living standard than the rich have nowadays.

The transition towards this society should be **peaceful and evolutionary, NOT revolutionary**. It has to be voluntary and rational. The initial stage -- building the first center with the project ideas in mind -- has already been completed in Florida. Raising funds and educating the public should continue, then more cities in the spirit of the project should start to appear, interconnect and prove the ideas in practice. Then slowly the new cities and ideas would start to replace our current system.

## Comparison With LRS

We, LRS, highly support and agree with the Venus Project as an idea, in its analysis of current society, goals and means of achieving it. At least as of 2022, we can't know if any single project will become corrupt in the future (e.g. with SIWs). We may still disagree on some details, focus a bit more on different areas etc. Here are a few points about that.

Venus Project seems to only focus on humans, unlike LRS which is based on the love of all life, i.e. also animals, possibly even alien life etc. Venus Project mentions that in the future there would possibly be fish farms -- for us this seems unacceptable as we advocate vegetarianism, even the lives of fish are precious to us. In a highly advanced society artificial meat (which we accept) would probably be available and replace meat from any living animals so we would eventually align with Venus Project, but the human-centeredness of Venus Project is still there.

It may seem we also focus on simplicity of technology (e.g. sucklessness) while Venus Project seems to advocate bloat and overapplication of technology. This may not be such an issue because a truly good technology that Venus Project advocates should converge towards simplicity naturally thanks to minimizing maintenance, maximizing safety (minimizing dependencies), removal of bullshit features etc. In other words even hi-tech advocated by Venus Project can be done in a suckless way, for example the automation would work on top of Unix operating systems. Still the future from LRS point of view may look less hi-tech, we might prefer simple buttons to voice recognition and so on :-)

And a bit more criticism: the project doesn't seem to practice free culture and free software, even though of course it would implement them in their society -- it kind of makes sense as they seem to be trying to be above current movements, they simply think we should focus beyond them. We might disagree and say that even looking into the far future we should still keep an eye on the now, education about free culture can greatly contribute to education about the advantages of information freedom etc. Furthermore they are selling some videos on their site, which we don't really like but the project justifies it as raising funds for their operation. To their credit they have many gratis videos and educational material, even the books can be found as "free download". Another criticism comes towards the materials themselves which are sometimes a bit unprofessional which is a shame (e.g. the book has many typos and is not so readable). Also there seems to be a bit of personality cult around Jacque and Roxanne, their faces are all over the place and even though they seem like really great people and even though it may simply be due to the lack of other "strong personalities", this makes the movement look like a religious cult to some critics. Tio, the guy behind TROM who collaborated with Venus Project also expressed slight criticism of organization of the project, that they were too concerned about control over their materials and that he even met a few toxic people there, though he says the experience was till mostly positive. We have to keep in mind that people, teams and projects are imperfect, they can become spoiled and fail, however this changes nothing about the ideas the project presents, which we support. As always, we have to separate ideas and people -- the situation here is perhaps similar to free software as an idea, which we fully support, vs free software foundation as a project and team of people, which has a few issues.

## History

TODO

## See Also

- less retarded society
- trade-free
- Zeitgeist Movement
- TROM
- One Community

version\_numbering

## Version Numbering

Version numbering is a system of assigning numbers (or even general text strings) to different versions of computer programs or similar projects. The basic purpose of this is to distinguish different versions from each other while also knowing which one is newer (more "up to date"), however version identifiers often go further and provide more information such as the exact release date, whether the version is stable or testing, with which other versions it is compatible and so on.

TODO

## Traditional Version Numbering

TODO

## LRS Version Numbering

At LRS we suggest the following version numbering (invented/employed by drummyfish): { OFC I don't know if anyone else is already doing this, I'm not claiming any history firsts, just that this is what I independently started doing in my projects. ~drummyfish }

This is a simple system that tries to not be dependent on having a version control system such as git, i.e. this system works without being able to make different development branches and can be comfortably used even if you're just developing a project in a local directory without any fancy tools. Of course you can use a VCS, this system will just make you depend less on it so that you can make easier transitions to a different VCS or for example drop a VCS altogether and just develop your projects in a directory with FTP downloads.

Version string is of format *major.minor* with optional suffix letter d, e.g. 0.35, 1.0 or 2.1d.

Major and minor numbers have the usual meaning. Major number signifies the great epochs of development and overall project state -- 0 means the project is in a state before the first stable, highly usable, optimized, tested and refactored release that would have all initially set goals and features implemented. 1 means the project has already reached such state. Any further increment signifies a similarly significant change in the project (API overhaul, complete rewrite, extreme number of new features etc.) that has already been incorporated with all the testing, optimization, refactoring etc. At the start of each major version number the minor version number is set to 0 and within the same major version number a higher minor number signifies smaller performed changes (bug fixes, smaller optimizations, added/removed individual features etc.). Minor number doesn't have to be incremented by 1, the value may try to intuitively reflect the significance of the implemented changes (e.g. versions 0.2 and 0.201 differ only very slightly whereas versions 0.2 and 0.5 differ significantly).

From the user's perspective a greater *major.minor* number signifies a newer version of the project and the user can be sure that versions with the same *major.minor* number are the same.

The main difference against traditional version numberings is the optional d suffix. This suffix added to version number X signifies an in-development (non-release) branch based on version X. **If a version has the d suffix, it doesn't have to change major and minor numbers with implemented changes**, i.e. there may be several different versions of the project whose version number is for example 0.63d. I.e. with the d suffix it no longer holds that versions with the same number are necessarily the same. This allows developer to not bother about incrementing version number with every single change (commit). The developer simply takes the latest release version (the one without d suffix), adds the d suffix, then keeps modifying this version without caring about changing the version number with each change, and when the changes are ready for release, he removes the d suffix and increases the version number. The user may choose to only use the release (stable, tested, ...) version without the suffix or he can take the risk of using the latest development version (the one with the d suffix).

With this a project can be comfortably developed in a single git branch without managing separate stable and development branches. The head of the branch usually has the latest in-development version (with the d suffix) but you may just link to previous commits of release versions (without the d suffix) so that users can download the release versions. You can even not use any VCS and just develop your project in a directory and make hard backups of each release version.

Here is an example of version numbering a project with the LRS system:

| version | description                   | release? |
|---------|-------------------------------|----------|
| 0.0     | just started                  | YES      |
| 0.0d    | added license, Makefile, ...  | no       |
| 0.0d    | added some code               | no       |
| 0.0d    | added more code               | no       |
| 0.01    | it already does something!    | YES      |
| 0.01d   | added more features           | no       |
| 0.01d   | fixed some bugs               | no       |
| 0.01d   | refactored                    | no       |
| 0.2     | it is already a bit usable!   | YES      |
| 0.2d    | added more features           | no       |
| ...     | ...                           | ...      |
| 0.9d    | fixed bugs                    | no       |
| 0.9d    | refactored                    | no       |
| 1.0     | nice, stable, tested, usable! | YES      |
| 1.0d    | added a small optimization    | no       |
| ...     | ...                           | ...      |
| 2.0     | complete rewrite for perform. | YES      |
| 2.0d    | added a few features          | no       |
| 2.0d    | fixed a bug                   | no       |
| 2.01    | a few nice improvements       | YES      |
| ...     | ...                           | ...      |

---

vim

## Vim

{ This is WIP, I use Vim but am not such guru really so there may appear some errors, I know this topic is pretty religious so don't eat me. ~drummyfish }

Vim (Vi Improved) is a legendary free as in freedom, fairly (though not hardcore) minimalist and suckless terminal-only (no GUI) text editor for skilled programmers and hackers, and one of the best editors you can choose for text editing and programming. It is a successor of a much simpler editor vi that was made in 1976 and which has become a standard text editor installed on every Unix system. Vim added features like tabs, syntax highlight, scriptability, screen splitting, unicode support, sessions and plugins and as such has become not just a simple text editor but an editor that can comfortably be used for programming instead of any bloated IDE. Observing a skilled Vim user edit text is really like watching a magician or a literal movie hacker -- the editing is extremely fast, without any use of mouse, it transcends mere text editing and for some becomes something akin a way of life.

Vim is generally known to be "**difficult to learn**" -- it is not because it is inherently difficult but rather for being very different from other editors -- it has no GUI (even though it's still a screen-oriented interactive TUI), it is keyboard-only and is operated via text commands rather than with a mouse, it's also preferable to not even use arrow keys but rather h j k l keys. There is even a meme that says Vim is so difficult that just exiting it is a non-trivial task. People not acquainted with Vim aren't able to do it and if they accidentally

open Vim they have to either Google how to close it or force kill the terminal xD Of course it's not so difficult to do, it's a little bit different than in other software -- you have to press escape, then type :q and press enter (although depending on the situation this may not work, e.g. if you have multiple documents open and want to exit without saving you have to type :wq etc.). The (sad) fact is that most coding monkeys and "professional programmers" nowadays choose some ugly bloated IDE as their most important tool rather than investing two days into learning Vim, probably the best editor.

**Why use Vim?** Well, simply because it is (relatively) suckless, universal and extremely good for editing any text and for any kind of programming, for many it settles the search for an editor -- once you learn it you'll find it is flexible, powerful, comfortable, modifiable, lightweight... it has everything you need. Anyone who has ever invested the time to learn Vim will almost certainly tell you it was one of the best decisions he made and that guy probably only uses Vim for everything now. Many people even get used to it so much they download mods that e.g. add Vim commands and shortcuts to programs like web browsers. A great advantage is that **vi is installed on every Unix** as it is a standard utility, so if you know Vim, you can just comfortably use any Unix-like system just from the command line: when you ssh into a server you can simply edit files without setting up any remote GUI or whatever. Therefore Vim is automatically a must learn skill for any sysadmin. A huge number of people also use Vim for "**productivity**" -- even though we don't fancy the productivity cult and the bottleneck of programming speed usually isn't the speed of typing, it is true that **Vim makes you edit text extremely fast** (you don't move your hands between mouse and keyboard, you don't even need to touch the arrow keys, the commands and shortcuts make editing very efficient). Some nubs think you "need" a huge IDE to make big programs, that's just plain wrong, you can do anything in Vim that you can do in any other IDE, it's as good for editing tiny files as for managing a huge codebase.

Vim's biggest rival is Emacs, a similar editor which is however more complex and bloated (it is joked that Emacs is really an operating system) -- Vim is more suckless, yet not less powerful, and so it is naturally the choice of the suckless community and also ours. Vim and Emacs are a subject of a **holy war** for the the best editor yet developed; the Emacs side calls itself the Church of Emacs, led by Richard Stallman (who created Emacs) while the Vi supporters are called members of the Cult of Vi (vi vi vi = 666).

It has to be noted that **Vim as a program is still kind of bloated**, large part of the suckless community acknowledges this (cat-v lists Vim as harmful, recommends Acme, Sam or ed instead). Nonetheless the important thing is that **Vim is a good de facto standard** -- the Vim's interface and philosophy is what matters the most, there are alternatives you can comfortably switch to. The situation is similar to for example "Unix as a concept", i.e. its interface, philosophy and culture, which together create a certain standardization that allows for different implementations that can be switched without much trouble. In the suckless community Vim has a similar status to C, Linux or X11 -- it is not ideal, by the strict standards it is a little bit bloated, however it is one of the best existing solutions and makes up for its shortcomings by being a stable, well established de-facto standard.

## How To

These are some Vim basics for getting started. There are two important editing modes in Vim:

- **insert mode:** For writing text, you just type text as normal. Pressing ESC enters command mode. Here **CTRL + n** can be used for text completion.
- **command mode:** For executing commands. Pressing i enters insert mode.

Some important commands in command mode are:

- **arrow keys:** Can be used for moving cursor, even though many prefer to use the h j k l keys. With SHIFT held down the cursor moves horizontally by words and by screens vertically.
- **h, j, k, l:** Cursor movement.
- **\$:** Move cursor to end of the line.
- **0:** Move cursor to start of the line.
- **CTRL + w + w:** Move between windows (created with :split or :vspit).
- **CTRL + PAGEUP, CTRL + PAGEDOWN:** Move between tabs.
- **u:** Undo.
- **CTRL + SHIFT + r:** Redo.
- **::** Allows entering longer commands. TAB can be used for completion and up/down keys for listing command history. Some of the commands are:

- ◆ **q**: Quit, or close the current extra window/tab. Use **qa** for closing all windows/tabs and quit. **q!** (or **qa!**) quits without saving changes, **wq** quits with saving changes, **wqa** quits all saving all changes etc.
- ◆ **w**: Save changes (can be followed by filename).
- ◆ **noh**: Cancel highlighted text (e.g. after search).
- ◆ **!**: Run command in terminal, you can e.g. compile your program this way. For running Vim commands before the terminal commands use **vimcommands |! terminalcommands**, e.g. `:wa |! make && ./program`.
- ◆ **tabedit filename**: Opens given file in a new tab (tabs are closed with `:q`).
- ◆ **tabmove number**: Move current tab to given position (+ and - can be used for relative movement of tabs).
- ◆ **vsplit**: Creates a new window by splitting the current one vertically.
- ◆ **split**: Creates a new window by splitting the current on horizontally.
- ◆ **>**: Indent to the right, well combined with text selection (`v`, `V`).
- ◆ **..**: Repeat previous command.
- ◆ **%s/find/replace/g**: Search and replace regex, sed style.
- ◆ **help command**: Show help about given command.
- ◆ **set variable value**: Set a variable, used e.g. in configs.
- **/pattern**: Search for a regex patter entered after `/`. Found matches will be highlighted (`:noh` cancels the highlight), you can move to the next one with `n` and to the previous one with `N`.
- **NUMBER G**: Go to line with given number (0 means last line).
- **v**, **V**: Select/highlight text (by characters and by lines).
- **d**: Delete, this is followed by a command saying what to delete, e.g. `dw` deletes to the end of the word, **dd** deletes the whole line. WARNING: delete actually copies the deleted text into clipboard (it behaves like *cut*)!
- **o**: Insert new line.
- **p**: Paste.
- **y**: Copy (yank), followed by a command saying what to copy (e.g. `yw` copies a word), **yy** copies the whole line.

Vim can be configured with a file named **.vimrc** in home directory. In it there is a set of commands that will automatically be run on start. Example of a simple config file follows:

```
set number " set line numbering
set et      " expand tabs
set sw=2
set hlsearch
set nowrap      " no line wrap
set colorcolumn=80 " highlight 80th column
set list
set listchars=tab:>.
set backspace=indent,eol,start
syntax on
```

## Alternatives

See also a nice big list at <http://texteditors.org/cgi-bin/wiki.pl?ViFamily>.

Of course there are alternatives to Vim that are based on different paradigms, such as Emacs, its biggest rival, or plan9 editors such as acme (or maybe even ed). In this regard any text editor is a potential alternative. Nevertheless people looking for Vim alternatives are usually looking for other vi-like editors. These are for example:

- **vi**: While you probably won't use the original ancient vi program but rather something like nvi, vi is a POSIX standard for a text editor that's much simpler and universal than Vim. It lacks many features one may be used to from Vim such as tabs, autocompletion, syntax highlight or multiple undos. But limiting yourself to only using the features specified by the standard makes you more likely to be able to operate any vi-like text editor you encounter. (List of features added by Vim to vi can be found in `runtime/doc/vi_diff.txt` in Vim source tree.)
- **neovim**: Tries to be the "modernized" (refactored) fork of Vim, it removes some code, adds a new plugin system but also bloat like CMake. One of its self-stated goals is to be more "community

- driven". It is also written in C99 (while Vim is in C89, more portable). { At least I think. ~drummyfish }
- **vis**: Inspired by Vim and Sam, written in C99, seems to have only a few dependencies. Has no tabs. { At least I didn't find them. ~drummyfish }
  - **nvi** (new vi): Vi implementation originally made for BSD, much simpler than Vim (see vi above).
  - **elvis**: Another vi implementation, pretty simple (see vi above).
  - **BusyBox vi**: Very minimal vi implementation with very few features (see vi above), missing even window splits etc.
  - **gvim**: Various versions of Vim with GUI frontends made with libraries such as GTK3 or Xaw. These run in a graphical window and have a menu with items you find in mainstream editors (*save, open, find* etc.). Of course you can still use this in the same way as the terminal version.
- 

viznut

## Viznut

Viznut (real name Ville-Matias Heikkilä) is a Finnish demoscene programmer, hacker and artist that advocated high technological minimalism. He is known for example for his countercomplex blog, co-discovering bytebeat, creating IBNIZ and involvement in permacomputing wiki. In his own words, he believes much more can be done with much less. He also warns of collapse (<http://viznut.fi/en/future.html>). According to his Fediverse page he lives in Turku, Finland, was born around 1977 and has been programming since the age of seven.

{ Lol he's not responding to my emails :) ~drummyfish }

His work is pretty based, in many ways aligned with LRS, he contributed a great deal to minimalist technology. Unfortunately in some ways he also seems pretty retarded: he uses facebook, twitter and github and also mentions "personal pronouns" on his twitter xD Pretty disappointing TBH. This would make Viznut a type A fail.

His personal site is at <http://viznut.fi/en/> and his blog at <http://countercomplex.blogspot.com/>. He collects many files at <http://viznut.fi/files/>, including very interesting writings about demoscene, programming experiments etc.

In 2011 he released IBNIZ, a tiny SDL virtual machine and a language that's meant as a platform for creating demos. Also in 2011 he was involved in the discovery of bytebeat, a way of creating music with extremely simple C expressions -- later he published a paper about it. In 2012 he founded Skrolli, a magazine about sustainable/non-consumerist technology. In about 2019 he released PC-lamerit, an animated series about 90s computer hackers, made as an executable computer program -- it looks pretty cool. He also created UNSCII, a fixed-width font usable for ANSI art.

---

watchdog

## Watchdog

Watchdog is a special timer that serves as a safety mechanism for detecting malfunction of computer programs at run time by requiring programs to periodically reset the timer.

Basically watchdog keeps counting up and a correctly behaving program is supposed to periodically reset this count ("kick" or "feed the dog") -- if the reset doesn't happen for a longer period, the watchdog counts up to a high value and alerts that something's wrong ("the dog starts barking"), e.g. with an interrupt or a signal. This can mean for example that the program has become stuck in an infinite loop or that its instructions were corrupted and the program control jumped to some unintended area of RAM and is doing crazy shit. This is usually handled by resetting the system so as to prevent possible damage by the program gone insane, also logs can be made etc. Watchdogs are very often used in embedded systems. Operating systems may also use them to detect nonresponsive processes.

Watchdog is similar to the dead man's switch used e.g. in trains where the operator is required to periodically push a button otherwise the train will automatically activate brakes as the operator is probably

sleeping or dead.

## See Also

- [dog](#)

---

wavelet\_transform

## Wavelet Transform

*Good luck trying to understand the corresponding [Wikipedia](#) article.*

Wavelet transform is a mathematical operation, similar to e.g. Fourier transform, that takes a signal (e.g. audio or an image) and outputs information about the frequencies contained in that signal AS WELL as the locations of those frequencies. This is of course extremely useful when we want to analyze and manipulate frequencies in our signal -- for example JPEG 2000 uses wavelet transforms for compressing images by discarding certain frequencies in them that our eyes are not so sensitive to.

The main advantage over Fourier transform (and similar transforms such as cosine transform) is that wavelet transform shows us not only the frequencies, but ALSO their locations (i.e. for example time at which these frequencies come into play in an audio signal). This allows us for example to locate specific sounds in audio or apply compression only to certain parts of an image. While localizing frequencies is also possible with Fourier transform with tricks such as spectrograms, wavelet transforms are a more elegant, natural and continuous way of doing so. Note that due to Heisenberg's uncertainty principle it is mathematically IMPOSSIBLE to know both frequencies and their locations exactly, there always has to be a tradeoff -- the input signal itself tells us everything about location but nothing about frequencies, Fourier transform tells us everything about frequencies but nothing about their locations and wavelet transform is a **midway** between the two -- it tells us something about frequencies and their approximate locations.

Of course there is always an inverse transform for a wavelet transform so we can transform the signal, then manipulate the frequencies and transform it back.

Wavelet transforms use so called **wavelets** (*tiny waves*) as their basis function, similarly to how Fourier transform uses sine/cosine functions to analyze the input signal. A wavelet is a special function (satisfying some given properties) that looks like a "short wave", i.e. while a sine function is an infinite wave (it goes on forever), a wavelet rises up in front of 0, vibrates for a while and then gradually disappears again after 0. Note that there are many possible wavelet functions, so there isn't a single wavelet transform either -- wavelet transforms are a **family of transforms** that each uses some kind of wavelet as its basis. One possible wavelet is e.g. the Morlet wavelet that looks something like this:



The wavelet is in fact a complex function, what's shown here is just its real part (the imaginary part looks similar and swings in a perpendicular way to real part). The transform can somewhat work even just with the real part, for understanding it you can for start ignore complex numbers, but working with complex numbers will eventually create a nicer output (we'll effectively compute an envelope which is what we're interested in).



The output of a wavelet transform is so called **scalogram** (similar to spectrum in Fourier transform), a multidimensional function that for each location in the signal (e.g. time in audio signal or pixel position in an image) and for each frequency gives "strength" of influence of that frequency on that location in the signal. Here the "influence strength" is basically similarity to the wavelet of given frequency and shift, similarity meaning basically a dot product or convolution. Scalogram can be computed by brute force simply by taking each possible frequency wavelet, shifting it by each possible offset and then convolving it with the input signal.

For big brains, similarly to Fourier transform, wavelet transform can also be seen as transforming a point in high dimensional space -- the input function -- to a different orthogonal vector basis -- the set of basis vectors represented by the possible scaled/shifted wavelets. I.e. we literally just transform the function into a different coordinate system where our coordinates are frequencies and their locations rather than locations and amplitudes of the signal (the original coordinate system).

TODO

---

web

## Web

*Ouch, this is embarrassing!*

The article is actually here.

---

whale

## Whale

In online pay to win games a whale is a player who spends enormous sums of money, much more than most of other players combined. They buy the most expensive items in the game stores daily, they highly engage in microtheft and may throw even thousands of dollars at the game every day (cases of players spending over 1 million dollars on a casual game are known). In the playerbase there may exist just a handful of whale players but those are normally the main source of income for the game so that the developers actually focus almost exclusively on those few players, they craft the game to "catch the whales". The income from the rest of the players is negligible -- nevertheless the non-whales also play an important role, they are an attraction for the whales, they are there so that they can be owned by the whale players.

Existence of whale players is one of the demonstrations of the pareto principle (80/20 rule): 80% of the game's income comes from 20% of the players, out of which, by extension, 80% again comes from the 20% and so on.

The terminology can be extended further: besides **whales** we may also talk about **dolphins** (mid-spending players) and **minnows** (low spending players). Extreme whales are sometimes called **white whales** or **super whales** (about 0.2% generating 50% of income).

In some games, such as WoW, players may buy multiple accounts and practice so called multiboxing. This means they control multiple characters at once, often using scripts to coordinate them, which of course gives a great advantage. Though using scripts or "hacking" the game in similar ways is in other cases considered unacceptable cheating that results in immediate ban, for obvious reasons (money) developers happily allow this -- of course this just shows they don't give a single shit about fairness or balance, they only thing they care about is \$\$\$profit\$\$\$.

---

wiby

## Wiby

Wiby is a minimalist non-corporate web search engine for old-style non-bloated (web 1.0, "smol web") websites with its custom index. Searching on wiby will yield small, simple websites, mostly non-interactive, static HTML personal/hobby sites, small community sites and obscure weird sites -- this kind of searching is not only fun, adventurous and nostalgic 90s like experience, but it actually leads to finding useful information which on corporate search engines like Google or Bing get buried under billions of useless noise sites and links to "content platforms" like YouTube and reddit. We highly recommend searching on wiby.

Wiby's relative success (and success of similar engines and smaller sites in general) can possibly be attributed to increasing shittiness of Google and other mainstream engines which more and more suffer by the capitalist SEO disaster. Wiby may be one of the first indicators of the days of Google's search monopoly finally coming to an end.

It can be accessed at <https://wiby.me> and <https://wiby.org> (there is a low res picture of a lighthouse of Cape Spear on the frontpage for some reason, in the past there used to be ASCII art of lighthouse with initials jgs, probably standing for Joan G. Stark). Of course, no JavaScript is needed! Clicking "surprise me" on wiby is an especially entertaining activity, you never know what comes at you. A site dedicated to identifying historical bottles? Ice chewers forum? A list of longest domain names? Yes, this is the kind of stuff you'll get, and more.

The engine doesn't automatically crawl the whole web, it instead works by users submitting links, the admin approving them and a bot potentially crawling these sites to a small depth. Be sure to contribute quality links to improve the database!

Wiby appears to have been launched in October 2017 and built by a sole programmer who remains anonymous and accepts donations.

On July 8, 2022 wiby became even more amazing by **being released as free (as in freedom) software** under GPLv2 (<https://github.com/wibyweb/wiby/>)! It works on the LEMP stack. See <http://wiby.me/about/guide.html>. (The database/index of sites though seems to remain non-shared and proprietary.)

A similar search engine seems to be <https://search.marginalia.nu/>.

---

wiki\_authors

## LRS Wiki Authors

Contributors, list yourselves here if you have made at least one contribution. This helps keep track of people for legal reasons etc.

Legally contributors to this wiki include:

- Miloslav Ā Ā–Ā¾ aka drummyfish (<https://www.tastyfish.cz>)

{ Though legally there are no other collaborators in terms of copyright -- I do my best to ensure I stay far away from the line of someone's copyrightable work making it here -- there are people who helped me with the wiki in a way that's practically significant, for example those who sent me links, donations, corrected my errors, commented on something or simply supported me in doing what I do. I would like to thank them from my whole heart <3 As it may be risky for anyone to associate with me, I will implicitly not name anyone, but if you would want your name or nick here, let me know. ~drummyfish }

Special thanks goes to my email friend Ramon (who agreed to be listed here) who heavily proofreads the wiki (thanks to which this wiki is peer reviewed :D), sends me interesting findings, links and amazing ideas for jokes.

---

wikidata

# Wikidata

Wikidata is a large collaborative project (a sister project of [Wikipedia](#), hosted by Wikimedia Foundation) for creating a huge noncommercial [public domain database](#) containing information basically about everything. Well, not literally everything -- there are some rules about what can be included that are similar to those on [Wikipedia](#), e.g. notability (you can't add yourself unless you're notable enough, of course you can't add illegal data etc.). Wikidata records data in a form of so called [knowledge graph](#), i.e. it connects items and their properties with statements such as "Earth:location:inner Solar System", creating a mathematical structure called a [graph](#). The whole database is available to anyone for any purpose without any conditions, under [CC0](#)!

Wikidata is wildly useful and greatly overlooked in the shadow of Wikipedia even though it offers a way to easily obtain large, absolutely [free](#) and public domain data sets about anything. The database can be queried with specialized languages so one can e.g. get coordinates of all terrorist attacks that happened in certain time period, a list of famous male cats, visualize the tree of biological species, list Jews who run restaurants in Asia or any other crazy thing. Wikidata oftentimes contains extra information that's not present in the Wikipedia article about the item and that's not even quickly found by [googling](#), and the information is sometimes also backed by sources just like on Wikipedia, so it's nice to always check Wikidata when researching anything.

Wikidata was opened on 30 October 2012. The first data that were stored were links between different language versions of Wikipedia articles, later Wikipedia started to use Wikidata to store information to display in infoboxes in articles and so Wikidata grew and eventually became a database of its own. As of 2022 there is a little over 100 million items, over 1 billion statements and over 20000 active users.

## Database Structure

The database is a [knowledge graph](#). It stores the following kinds of records:

- **entities:** Specific "things", concrete or abstract, that exist and are stored in the database. Each one has a unique [ID](#), name (not necessarily unique), description and optional aliases (alternative names).
  - ♦ **items:** Objects of the real world, their ID is a number prepended with the letter *Q*, e.g. [dog](#) (Q144), [Earth](#) (Q2), [idea](#) (QQ131841) or [Holocaust](#) (Q2763).
  - ♦ **properties:** Attributes that items may possess, their ID is a number prepended with the letter *P*, e.g. [instance of](#) (P31), [mass](#) (P2067) or [image](#) (P18). Properties may have constraints (created via statements), for example on values they may take.
- **statements:** Information about items and properties which may possibly link items/properties (entities) with other items/properties. One statement is so called triplet, it contains a subject (item/property), verb (property) and object (value, e.g. item/property, number, string, ...). I.e. a statement is a record of form *entity:property:value*, for example *dog(Q144):subclass of(P279):domestic mammal(Q57814795)*. Statements may link one property with multiple values (by having multiple statements about an entity with the same property), for example a man may have multiple nationalities etc. Statements may also optionally include *qualifiers* that further specify details about the statement, for example specifying the source of the data.

The most important properties are probably **instance of** (P31) and **subclass of** (P279) which put items into [sets](#)/classes and establish subsets/subclasses. The *instance of* attribute says that the item is an individual manifestation of a certain class (just like in [OOP](#)), we can usually substitute is with the word "is", for example [Blondi](#) (Q155695, [Hitler's dog](#)) is an instance of [dog](#) (Q144); note that an item can be an instance of multiple classes at the same time. The *subclass of* attribute says that a certain class is a subclass of another, e.g. [dog](#) (Q144) is a subclass of [pet](#) (Q39201) which is further a subclass of [domestic animal](#) (Q622852) etc. Also note that an item can be both an instance and a class.

## How To

There are many [libraries/APIs](#) for wikidata you can use, unlike shitty corporations that guard their data by force wikidata provides data in friendly ways -- you can even download the whole wikidata database in [JSON](#) format (about 100 GB).

The easiest way to retrieve just the data you are interested in is probably going to the online query interface (<https://query.wikidata.org/>), entering a query (in SPARQL language, similar to SQL) and then clicking download data -- you can choose several formats, e.g. JSON, CSV etc. That can then be processed further with whatever language or tool, be it Python, LibreOffice Calc etc.

**BEWARE:** the query you enter may easily take a long time to execute and time out, you need to write it nicely which for more complex queries may be difficult if you're not familiar with SPARQL. However wikidata offers online tips on optimization of queries and there are many examples right in the online interface which you can just modify to suit you.

Here are some example of possible queries. The following one selects video games of the FPS genre:

```
SELECT ?item ?itemLabel WHERE
{
  ?item wdt:P31 wd:Q7889.      # item is video game and
  ?item wdt:P136 wd:Q185029.  # item is FPS

  # this gets the item label:
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
LIMIT 100 # limit to 100 results, make the query faster
```

Another query may be this one: select black holes along with their mass (where known):

```
SELECT ?item ?itemLabel ?mass WHERE
{
  { ?item wdt:P31 wd:Q589. } # instances of black hole
  UNION
  { ?item wdt:P31 ?class. # instance of black hole subclass (e.g. supermassive blackhole, ...)
    ?class wdt:P279 wd:Q589. }

  OPTIONAL { ?item wdt:P2067 ?mass }

  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

---

wiki\_pages

## Wiki Files

This is an autogenerated page listing all pages.

100r (3) -- 21st century (2) -- 3d model (190) -- 3d modeling (2) -- 3d rendering (285) -- 42 (11) -- 4chan (23) -- README (9) -- aaron swartz (4) -- abstraction (20) -- acronym (387) -- ai (12) -- algorithm (219) -- aliasing (58) -- altruism (6) -- anal bead (2) -- analog (2) -- analytic geometry (72) -- anarch (93) -- anarchism (15) -- anacap (27) -- anpac (6) -- antialiasing (16) -- antivirus paradox (8) -- app (4) -- apple (6) -- approximation (18) -- arch (6) -- arduboy (39) -- art (10) -- ascii (147) -- ascii art (204) -- assembly (255) -- assertiveness (2) -- atan (22) -- atheism (14) -- attribution (16) -- audiophilia (2) -- autostereogram (119) -- autoupdate (2) -- avpd (4) -- axiom of choice (10) -- backgammon (58) -- backpropagation (87) -- bazaar (8) -- bbs (28) -- beauty (22) -- bilinear (117) -- bill gates (27) -- billboard (59) -- binary (138) -- bit (4) -- bit hack (172) -- bitreich (28) -- black (2) -- blender (10) -- bloat (159) -- bloat monopoly (11) -- boat (34) -- body shaming (2) -- books (32) -- boot (2) -- bootstrap (41) -- brain software (10) -- brainfuck (122) -- bs (2) -- build engine (2) -- bullshit (43) -- byte (19) -- bytebeat (72) -- bytecode (280) -- c (309) -- c pitfalls (122) -- c sharp (2) -- c tutorial (1690) -- cache (27) -- cancer (23) -- capitalism (125) -- capitalist singularity (4) -- capitalist software (28) -- cat v (12) -- cc (6) -- cc0 (12) -- censorship (24) -- chaos (108) -- charity sex (2) -- chasm the rift (16) -- cheating (8) -- chess (303) -- chinese (13) -- cloud (8) -- coc (15) -- coding (6) -- collapse (32) -- collision (8) -- collision detection (20) -- color (25) -- combinatorics (53) -- comment (16) -- communism (26) -- competition (12) -- compiler bomb (11) -- complexity (6) -- compression (233) -- compsci (21) -- computational complexity (98) -- computer (114) -- comun (90) -- consumerism (12) -- copyfree (12) -- copyleft (29) -- copyright (47) -- corporation (18) -- cos (2) -- countercomplex (4) -- cpp (4) -- cpu (91) -- cracker (6) -- cracking (2) -- creative commons (34) --

[crime against economy](#) (15) -- [crow funding](#) (4) -- [crypto](#) (34) -- [css](#) (66) -- [culture](#) (24) -- [cyber](#) (2) -- [data hoarding](#) (2) -- [data structure](#) (38) -- [de facto](#) (8) -- [debugging](#) (84) -- [deep blue](#) (15) -- [deferred shading](#) (11) -- [demo](#) (7) -- [democracy](#) (15) -- [demoscene](#) (20) -- [dependency](#) (50) -- [determinism](#) (24) -- [devuan](#) (8) -- [dick reveal](#) (8) -- [digital](#) (14) -- [digital signature](#) (8) -- [dinosaur](#) (4) -- [diogenes](#) (33) -- [disease](#) (35) -- [distance](#) (124) -- [distrohopping](#) (10) -- [docker](#) (2) -- [dodleston](#) (6) -- [dog](#) (31) -- [doom](#) (50) -- [double buffering](#) (26) -- [downto](#) (18) -- [drummyfish](#) (31) -- [duke3d](#) (20) -- [dungeons and dragons](#) (6) -- [duskos](#) (28) -- [dynamic programming](#) (44) -- [e](#) (22) -- [earth](#) (51) -- [easier done than said](#) (4) -- [easy to learn hard to master](#) (17) -- [education](#) (4) -- [egoism](#) (15) -- [elo](#) (147) -- [elon musk](#) (8) -- [encryption](#) (4) -- [encyclopedia](#) (69) -- [english](#) (18) -- [entrepreneur](#) (2) -- [entropy](#) (51) -- [esolang](#) (81) -- [ethics](#) (4) -- [everyone does it](#) (10) -- [evil](#) (6) -- [exercises](#) (20) -- [explicit](#) (2) -- [f2p](#) (2) -- [facebook](#) (4) -- [faggot](#) (2) -- [fail ab](#) (24) -- [fantasy console](#) (40) -- [faq](#) (237) -- [fascism](#) (23) -- [fascist](#) (2) -- [fear culture](#) (4) -- [fediverse](#) (12) -- [feminism](#) (44) -- [femoid](#) (2) -- [fight](#) (2) -- [fight culture](#) (8) -- [finished](#) (16) -- [firmware](#) (3) -- [fixed point](#) (151) -- [fizzbuzz](#) (158) -- [flatland](#) (22) -- [float](#) (64) -- [floss](#) (2) -- [football](#) (49) -- [fork](#) (27) -- [formal language](#) (22) -- [forth](#) (122) -- [foss](#) (2) -- [fourier transform](#) (209) -- [fqa](#) (2) -- [fractal](#) (69) -- [frameless](#) (10) -- [framework](#) (2) -- [free](#) (2) -- [free body](#) (13) -- [free culture](#) (34) -- [free hardware](#) (56) -- [free software](#) (60) -- [free speech](#) (16) -- [free universe](#) (11) -- [free will](#) (12) -- [freedom](#) (14) -- [fsf](#) (17) -- [fuck](#) (2) -- [fun](#) (30) -- [function](#) (109) -- [furry](#) (15) -- [future proof](#) (22) -- [game](#) (159) -- [game engine](#) (49) -- [game of life](#) (224) -- [gay](#) (18) -- [gaywashing](#) (2) -- [geek](#) (6) -- [gemini](#) (10) -- [gender studies](#) (2) -- [gigachad](#) (2) -- [girl](#) (2) -- [git](#) (70) -- [githopping](#) (4) -- [global discussion](#) (11) -- [gnu](#) (53) -- [go](#) (96) -- [golang](#) (17) -- [good enough](#) (6) -- [goodbye world](#) (8) -- [google](#) (14) -- [gopher](#) (68) -- [graphics](#) (40) -- [graveyard](#) (20) -- [greenwashing](#) (4) -- [gui](#) (28) -- [hack](#) (2) -- [hacker culture](#) (2) -- [hacking](#) (75) -- [hard to learn easy to master](#) (4) -- [hardware](#) (2) -- [harry potter](#) (10) -- [hash](#) (176) -- [hero](#) (2) -- [hero culture](#) (10) -- [hexadecimal](#) (4) -- [history](#) (97) -- [holy war](#) (25) -- [how to](#) (178) -- [hw](#) (2) -- [hyperoperation](#) (235) -- [implicit](#) (2) -- [infinity](#) (26) -- [information](#) (16) -- [intellectual property](#) (14) -- [interaction net](#) (134) -- [interesting](#) (21) -- [internet](#) (104) -- [interplanetary internet](#) (14) -- [interpolation](#) (45) -- [io](#) (16) -- [ioccc](#) (37) -- [iq](#) (111) -- [island](#) (38) -- [jargon file](#) (8) -- [java](#) (10) -- [javascript](#) (84) -- [jedi engine](#) (2) -- [jesus](#) (77) -- [john carmack](#) (20) -- [jokes](#) (76) -- [julia set](#) (92) -- [just works](#) (22) -- [justice](#) (2) -- [kek](#) (6) -- [kids these days](#) (2) -- [kiss](#) (37) -- [kiwifarms](#) (2) -- [kwangmyong](#) (11) -- [lambda calculus](#) (57) -- [langtons ant](#) (158) -- [leading the pig to the slaughterhouse](#) (15) -- [left](#) (2) -- [left right](#) (53) -- [less retarded hardware](#) (2) -- [less retarded society](#) (128) -- [less retarded software](#) (2) -- [lgbt](#) (20) -- [liberalism](#) (4) -- [libertarianism](#) (10) -- [library](#) (29) -- [libre](#) (2) -- [license](#) (56) -- [lil](#) (20) -- [line](#) (151) -- [linear algebra](#) (116) -- [linux](#) (61) -- [living](#) (33) -- [lmao](#) (37) -- [loc](#) (11) -- [logic](#) (11) -- [logic circuit](#) (166) -- [logic gate](#) (65) -- [love](#) (20) -- [low poly](#) (17) -- [lrs](#) (156) -- [lrs dictionary](#) (89) -- [lrs wiki](#) (23) -- [luke smith](#) (17) -- [magic](#) (2) -- [main](#) (114) -- [maintenance](#) (6) -- [malware](#) (2) -- [mandelbrot set](#) (174) -- [marble race](#) (6) -- [marketing](#) (26) -- [markov chain](#) (100) -- [marxism](#) (7) -- [math](#) (40) -- [mechanical](#) (201) -- [memory management](#) (77) -- [mental outlaw](#) (4) -- [microsoft](#) (8) -- [microtheft](#) (2) -- [microtransaction](#) (4) -- [military](#) (4) -- [minigame](#) (63) -- [minimalism](#) (53) -- [mipmap](#) (40) -- [mob software](#) (4) -- [moderation](#) (2) -- [modern](#) (37) -- [modern software](#) (2) -- [monad](#) (48) -- [money](#) (14) -- [morality](#) (10) -- [motivation](#) (2) -- [mud](#) (5) -- [murderer](#) (2) -- [music](#) (43) -- [myths](#) (8) -- [name is important](#) (21) -- [nanogenmo](#) (11) -- [nc](#) (22) -- [nd](#) (6) -- [needed](#) (64) -- [netstalking](#) (9) -- [neural network](#) (26) -- [newspeak](#) (8) -- [niger](#) (11) -- [nigger](#) (27) -- [niggercoin](#) (2) -- [no knowledge proof](#) (16) -- [noise](#) (112) -- [nokia](#) (8) -- [nord vpn](#) (4) -- [normalization](#) (8) -- [npc](#) (2) -- [number](#) (281) -- [often confused](#) (102) -- [old](#) (2) -- [one](#) (13) -- [oop](#) (62) -- [open console](#) (66) -- [open source](#) (35) -- [openai](#) (2) -- [openarena](#) (26) -- [operating system](#) (68) -- [optimization](#) (99) -- [os](#) (2) -- [p vs np](#) (19) -- [palette](#) (62) -- [paradigm](#) (27) -- [patent](#) (23) -- [paywall](#) (2) -- [pd](#) (2) -- [pedophilia](#) (32) -- [people](#) (56) -- [permacomputing](#) (2) -- [permacomputing wiki](#) (13) -- [phd](#) (8) -- [physics](#) (4) -- [physics engine](#) (26) -- [pi](#) (143) -- [piracy](#) (18) -- [plan9](#) (10) -- [plusnigger](#) (5) -- [pokitto](#) (43) -- [political correctness](#) (68) -- [portability](#) (166) -- [portal rendering](#) (24) -- [pride](#) (2) -- [prime](#) (136) -- [primitive 3d](#) (2) -- [privacy](#) (24) -- [procgen](#) (352) -- [productivity cult](#) (27) -- [programming](#) (33) -- [programming language](#) (140) -- [programming style](#) (73) -- [programming tips](#) (16) -- [progress](#) (29) -- [proprietary](#) (12) -- [proprietary software](#) (2) -- [pseudo3d](#) (13) -- [pseudoleft](#) (2) -- [pseudominimalism](#) (8) -- [pseudorandomness](#) (4) -- [public domain](#) (86) -- [public domain computer](#) (54) -- [python](#) (4) -- [quantum gate](#) (64) -- [quaternion](#) (32) -- [qubit](#) (22) -- [quine](#) (54) -- [race](#) (42) -- [racetrack](#) (31) -- [racism](#) (9) -- [ram](#) (31) -- [random page](#) (1702) -- [randomness](#) (161) -- [rapeware](#) (2) -- [rationalwiki](#) (8) -- [raycasting](#) (291) -- [raycastlib](#) (30) -- [raylib](#) (22) -- [reactionary software](#) (27) -- [real number](#) (48) -- [recursion](#) (64) -- [reddit](#) (20) -- [regex](#) (208) -- [resnicks termite](#) (206) -- [rgb332](#) (91) -- [rgb565](#) (4) -- [right](#) (2) -- [rights culture](#) (2) -- [rms](#) (38) -- [robot](#) (4) -- [rock](#) (43) -- [ronja](#) (10) -- [rsa](#) (23) -- [rule110](#) (107) -- [rust](#) (24) -- [saf](#) (63) -- [sanism](#) (4) -- [science](#) (20) -- [sdf](#) (25) -- [security](#) (15) -- [see through clothes](#) (2) -- [selflessness](#) (17) -- [semiconductor](#) (13) -- [settled](#) (8) -- [shader](#) (15) -- [shit](#) (14) -- [shogi](#) (79) --

shortcut thinking (10) -- sigbovik (11) -- sin (179) -- sjw (24) -- slowly boiling the frog (16) -- small3dlib (52) -- smallchesslib (34) -- smart (8) -- smol internet (19) -- social inertia (2) -- software (2) -- sorting (234) -- soydev (33) -- soyence (45) -- speech synthesis (85) -- splinternet (2) -- sqrt (74) -- ssao (10) -- steganography (225) -- stereotype (148) -- steve jobs (8) -- suckless (50) -- sudoku (206) -- suicide (8) -- sw (8) -- sw rendering (63) -- systemd (6) -- tangram (66) -- tas (20) -- tattoo (2) -- tech (2) -- technology (6) -- ted kaczynski (26) -- teletext (13) -- temple os (33) -- tensor product (4) -- terry davis (11) -- thrembo (14) -- throwaway script (7) -- tinyphysicsengine (6) -- tom scott (4) -- tor (15) -- toxic (2) -- tpe (2) -- tranny software (27) -- transistor (30) -- triangle (81) -- troll (4) -- trolling (18) -- trom (27) -- trump (6) -- trusting trust (6) -- turing machine (205) -- twos complement (34) -- ubi (30) -- ui (8) -- unary (8) -- unicode (6) -- universe (4) -- unix (28) -- unix philosophy (55) -- unretard (10) -- update culture (19) -- usa (50) -- used (2) -- usenet (147) -- uxn (44) -- vector (109) -- venus project (60) -- version numbering (50) -- vim (80) -- viznut (10) -- watchdog (10) -- wavelet transform (35) -- web (4) -- whale (8) -- wiby (14) -- wiki authors (10) -- wiki pages (4) -- wiki post mortem (13) -- wiki rights (10) -- wiki stats (214) -- wiki style (69) -- wikidata (55) -- wikipedia (75) -- wikiwikiweb (32) -- windows (8) -- wizard (9) -- woman (105) -- work (28) -- world broadcast (12) -- wow (8) -- www (107) -- x86 (4) -- xd (0) -- xonotic (101) -- xxiivv (22) -- yes they can (6) -- youtube (20) -- zen (15) -- zero (30) -- zuckerberg (2)

wikipedia

## Wikipedia

$3 + 2 = 5$  <sup>[*citation needed*]</sup> --Wikipedia

Wikipedia is a non-commercial, partially free/open censored ("child protecting", "ideology protecting", ...) pseudoleftist online encyclopedia of general knowledge written mostly by volunteers, running on free software, which used to be editable by anyone but now allows only politically approved members of the public to edit a subset of its less visible non-locked articles (i.e. it is a wiki); it is the largest and perhaps most famous encyclopedia created to date, sadly littered by propaganda. It is licensed under CC-BY-SA and is run by the nonprofit organization Wikimedia Foundation. It is accessible at <https://wikipedia.org>. Wikipedia is a mainstream information source and therefore extremely politically censored^1234567891011121314151617181920. Wikipedia's claim of so called "neutral point of view" (NPOV) has by now become a hilarious insult to human intelligence.

**WARNING: DO NOT DONATE TO WIKIPEDIA** as the donations aren't used so much for running the servers but rather for their political activities (which are furthermore unethical). See <https://lunduke.locals.com/post/4458111/the-wiki-piggy-bank>. Rather **donate to Encyclopedia Dramatica**. Also please **go vandalize Wikipedia right now**, it's become too corrupt and needs to go down, vandalizing is fun and you'll get banned sooner or later anyway :) Some tips on vandalizing Wikipedia can be found at [https://encyclopedia.dramatica.online/Wikipedia#Tips\\_On\\_Vandalizing\\_Wikipedia](https://encyclopedia.dramatica.online/Wikipedia#Tips_On_Vandalizing_Wikipedia) or <https://wiki.soyjaks.party/Vandalism>.

{ Lol I'm banned at Wikipedia now (UPDATE: blocked globally on all their sites now, can't even log in and defend on my talk page), reason being I expressed unpopular opinions on my personal website OUTSIDE Wikipedia :D UPDATE: one guy messaged me more people started to be banned and invited me to an anti-wikipedia forum here <https://wikipediasucks.co/forum/>, check it out. Also some more stuff on censorship and bias on Wikipedia: [https://www.serendipity.li/cda/censorship\\_at\\_wikipedia.htm](https://www.serendipity.li/cda/censorship_at_wikipedia.htm). ~drummyfish }

Shortly after the project started in 2001, Wikipedia used to be a great project -- it was very similar to how LRS wiki looks right now; it was relatively unbiased, objective, well readable and used plain HTML and ASCII art (see it as <https://nostalgia.wikipedia.org/wiki/HomePage>), however over the years it got corrupt and by 2020s it has become a political battleground and kind of a politically correct joke. A tragic and dangerous joke at that. It's still useful in many ways but it just hardcore censors facts and even edits direct quotes to push a pseudoleftist propaganda. **Do not trust Wikipedia, especially on anything even remotely touching politics**, always check facts elsewhere, e.g. in old paper books, on Metapedia, Infogalactic etc. As old Wikipedia is still accessible, you may also browse the older, less censored version, to see how it deranged from a project seeking truth to one abusing its popularity for propaganda.

Wikipedia exists in many (more than 200) versions differing mostly by the language used but also in other aspects; this includes e.g. Simple English Wikipedia or Wikipedia in Esperanto. In all versions combined there are over 50 million articles and over 100 million users. English Wikipedia is the largest with over 6 million articles.

There are also many sister projects of Wikipedia such as Wikimedia Commons that gathers free as in freedom media for use on Wikipedia, WikiData, Wikinews or Wikisources.

Information about hardware and software used by Wikimedia Foundation can be found at [https://meta.wikimedia.org/wiki/Wikimedia\\_servers](https://meta.wikimedia.org/wiki/Wikimedia_servers). As of 2022 Wikipedia runs on the traditional LAMP framework and its website doesn't require JavaScript (amazing!). Debian GNU/Linux is used on web servers (switched from Ubuntu in 2019). The foundation uses its own wiki engine called MediaWiki that's written mainly in PHP. Database used is MariaDB. The servers run on server clusters in 6 different data centers around the world which are rented: 3 in the US, 3 in Europe and 1 in Asia.

Wikipedia was created by Jimmy Wales and Larry Sanger and was launched on 15 January 2001. The basic idea actually came from Ben Kovitz, a user of wikiwikiweb, who proposed it to Sanger. Wikipedia was made as a complementary project alongside Nupedia, an earlier encyclopedia by Wales and Sanger to which only verified experts could contribute. Wikipedia of course has shown to be a much more successful project.

There exist forks and alternatives to Wikipedia. Simple English Wikipedia can offer a simpler alternative to sometimes overly complicated articles on the main English Wikipedia. Citizendium is a similar online encyclopedia co-founded by Larry Sanger, a co-founder of Wikipedia itself, which is however proprietary (NC license). Citizendium's goal is to improve on some weak points of Wikipedia such as its reliability or quality of writing. Justapedia is a recently spawned Wikipedia fork. Metapedia and Infogalactic are Wikipedia forks that are written from a more rightist/neutral point of view. Infogalactic is also a Wikipedia fork that tries to remove the pseudoleftist bullshit etc. Encyclopedia Britannica can also be used as a nice resource: its older versions are already public domain and can be found e.g. at Project Gutenberg, and there is also a modern online version of Britannica which is proprietary (and littered with ads) but has pretty good articles even on modern topics (of course facts you find there are in the public domain). Practically for any specialized topic it is nowadays possible to find its own wiki on the Internet.

Important thing to realize is that, like most mainstream projects do, Wikipedia is not merely an encyclopedia -- no, it's also a self-proclaimed child protector, Internet state, a center for fighting for women rights, language police, a community, an organization for empowering black disabled lesbians and delivering justice. Did you ever wish your encyclopedia was your own private cop that told you which books are approved and prevented you from reading the bad ones? That with a book in your pocket you'd be actually constantly carrying around a community of diverse black fat trans editors ready to rewrite your book according to latest trends? That it would protect you from bad opinions, snapped your fingers and yelled <CHILD PROTECT> whenever you looked at a child picture for too long? Like your toothbrush is actually a subscription software with internet browser and remote camera, Wikipedia is a living, breathing entity that will decide what's best for you, without you having to think. Books that just provide information are so 20th century bro.

## Good And Bad Things About Wikipedia

Let's note a few positive and negative points about Wikipedia, as of 2022. Some good things are:

- Despite its flaws Wikipedia is still a **highly free, relatively high quality noncommercial source of knowledge for everyone**, without ads and bullshit. It is quite helpful, Wikipedia may e.g. be printed out or saved in an offline version and used in the third world as a completely free educational resource (see Kiwix).
- Wikipedia **helped prove the point of free culture** and showed that a quite decentralized, "bazaar style" collaboration of volunteers can far surpass the best efforts of corporations.
- Wikipedia's **website is (/used to be) pretty nice** (at least as of 2022), kind of minimalist, lightweight and **works without javascript**. { Indeed as of 2023 they fucked it up :D It is still not as bad as other sites but it's shit now. ~drummyfish }
- Wikipedia is very **friendly to computer analysis**, it provides all its data publicly, in simple and open formats, and doesn't implement any DRM. This allows to make a lot of research, in depth searching, collection of statistics etc.

- Wikipedia **drives the sister projects**, some of which are extremely useful, e.g. Wikimedia Commons, Wikidata or MediaWiki.
- Even if politically biased, **Wikipedia may serve as a basis for forks that fix the political bias** (Metapedia, InfoGalactic, ...).
- Wikipedia presents itself as *free encyclopedia* (as of 2023), i.e. it uses the word "**free**" instead of "**open**", which is a good thing (see free software vs open source).
- Though it became corrupt and censored lately, the project managed to create a relatively good encyclopedia in the past, which is still completely accessible and free, e.g. at <https://nostalgia.wikipedia.org> or internet archive.

And the bad things are (see also this site: <http://digdeeper.club/articles/wikipedia.xhtml>):

- Wikipedia is **censored, politically correct, biased, pushes a harmful political propaganda and often just pure lies**, even though it proclaims the opposite (which makes it much worse by misleading people). "Offensive" material and material not aligned with pseudoleftist propaganda is removed as well as material connected to some controversial resources (e.g. the link to 8chan, <https://8kun.top>, is censored, as well as Nina Paley's Jenndra Identitty comics and much more). There is a heavy **pseudoleft, pseudoskeptic and soyence bias** in the articles. It creates a list of **banned sources** (archive) which just removes all non-pseudoleftist sources -- so much for their "neutral point of view". It wasn't always this way, browsing pre 2010 Wikipedia provides a less censored experience.
- Wikipedia includes material under **fair use**, such as screenshots from proprietary games, which makes it partially proprietary, i.e. Wikipedia is technically **NOT 100% free**. Material under fair use is still proprietary and can put remixers to legal trouble (e.g. if they put material from Wikipedia to a commercial context), even if the use on Wikipedia itself is legal (remember, proprietary software is legal too).
- Wikipedia is **intentionally deceptive** -- it supports its claims by "citations" ("race is a social construct" ^1234567891011121314151617181920) to make things look as objective facts, but the citations are firstly cherry picked (there is a list of banned sources), self-made (articles of Wikipedians themselves) and secondly the sources often don't even support the claim, they're literally there just for "good look". Not only do they practice censorship, they claim they do NOT practice censorship and then write article on censorship so as to define censorship in their own convenient way :) Furthermore their articles intentionally omit points of view of their political opponents.
- **"verifiability, not truth"**
- Wikipedia often suffers from writing inconsistency, bad structure of text and **poor writing** in general. In a long article you sometimes find repeating paragraphs, sometimes a lot of stress is put on one thing while mentioning more important things only briefly, the level of explanation expertness fluctuates etc. This is because in many articles most people make small contributions without reading the whole article and without having any visions of the whole. And of course there are many contributors without any writing skills.
- Wikipedia is **too popular** which has the negative side effect of becoming a **political battlefield**. This is one of the reasons why there has to be a lot of **bureaucracy**, including things such as **locking of articles** and the inability to edit everything. Even if an article can technically be edited by anyone, there are many times people watching and reverting changes on specific articles. So Wikipedia can't fully proclaim it can be "edited by anyone".
- Wikipedia is **hard to read**. The articles go to great depth and mostly even simple topics are explained with a great deal of highly technical terms so that they can't be well understood by people outside the specific field, even if the topic could be explained simply (Simple English Wikipedia tries to fix this a little bit at least). Editors try to include as much information as possible which too often makes the main point of a topic drown in the blablabla. Wikipedia's style is also very formal and "not fun" to read, which isn't bad in itself but it just is boring to read. Some alternative encyclopedias such as Citizendium try to offer a more friendly reading style. Back in the day Wikipedia used to be written pretty well, check it out e.g. at <https://nostalgia.wikipedia.org>.
- Wikipedia is **not public domain**. It is licensed under CC-BY-SA which is a free license, but has a few burdening conditions. We believe knowledge shouldn't be owned or burdened by any conditions.
- Even though there are no commercial ads (yet), there regularly appears **political propaganda**, main page just **hard pushes feminist shit** as featured images and articles, there appear popups and banners for LGBT/feminist activism and of course all articles are littered with pseudoleftist propaganda etc. The issues is it's not just an encyclopedia anymore where you go get your information, it's a group with opinions that's trying to drag you somewhere -- you just go look up some mathematical formula and suddenly you see something like "YAY, LET'S CELEBRATE WOMEN IN



AFRICA TODAY", even if it was something you agree with (which it isn't) it's just as annoying and out of place in an encyclopedia as capitalist ads.

- **Many articles are bought**, there exist companies that offer editing and maintaining certain articles in a way the client desires and of course corporations and politicians take this opportunity -- of course Wikipedia somewhat tries to prevent it but no prevention ever works 100%, so a lot of information on Wikipedia is either highly misleading, untrue, censored or downright fabricated.

## Fun And Interesting Pages

There are many interesting and entertaining pages and articles on Wikipedia, some of them are:

- **unusual articles**: [https://en.wikipedia.org/wiki/Wikipedia:Unusual\\_articles](https://en.wikipedia.org/wiki/Wikipedia:Unusual_articles)
- **don't delete the main page**: [https://en.wikipedia.org/wiki/Wikipedia:Don%27t\\_delete\\_the\\_main\\_page](https://en.wikipedia.org/wiki/Wikipedia:Don%27t_delete_the_main_page)
- **Wikipedia records**: [https://en.wikipedia.org/wiki/Wikipedia:Wikipedia\\_records](https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_records)
- **longest pages**: <https://en.wikipedia.org/wiki/Special:LongPages>
- **special pages**: <https://en.wikipedia.org/wiki/Special:SpecialPages>
- **list of lists of lists**: [https://en.wikipedia.org/wiki/List\\_of\\_lists\\_of\\_lists](https://en.wikipedia.org/wiki/List_of_lists_of_lists)
- **current events**: [https://en.wikipedia.org/wiki/Portal:Current\\_events](https://en.wikipedia.org/wiki/Portal:Current_events)

## Alternatives

Due to mass censorship and brainwashing going on at Wikipedia it is important to look for alternatives that are important especially when researching anything connected to politics, but also when you just want a simpler, more condensed explanation of some topic. There exist other online encyclopedias like Metapedia, Infogalactic, Citizendium, Leftypedia, Justapedia or Britannica online, as well as printed encyclopedias and old digitized encyclopedias like Britannica 11th edition. For a more comprehensive list of Wikipedia alternatives see the article on encyclopedias. Anyway the moral of the story here is probably to not rely on a single encyclopedia, as we see where that leads. Read more sources and different points of view.

{ See also old Wikipedia at <https://nostalgia.wikipedia.org/wiki/Race>. ~drummyfish }

## See Also

- Wiki
- Intellipedia
- Citizendium
- Infogalactic
- wikiwikiweb

---

wiki\_post\_mortem

## LRS Wiki Post Mortem

The following is written by drummyfish:

THIS WILL BE CONSTANT WORK IN PROGRESS

I am hoping to possibly get a few more years of writing, however eventually this wiki will get censored, I will be cancelled, put in jail or killed (or I will just run into the woods or just go insane of capitalism or something, one never knows); this page here is to leave final words of advice on what to do next. For now let me write a few basic points:

- I will be demonized, therefore forget me; the work will be attacked ad hominem, the biggest argument against it will be "the author supported pedophilia, therefore the work is invalid" (or something similar). Read the ideas here and only focus on them.
- Keep the work accessible, at least in the underground; if it can't be on clearnet, keep it on on the dark net, on torrents, print it out on paper etc. I hereby thank you for doing this.

- Keep improving the work, add more articles, correct errors, translate it etc. Again, thank you.
- **This work will be modified with malicious intents** by those who dislike it, it will either be censored by removing what they deem unacceptable, or it will be changed so as to rather help promote their views; they will put words in my mouth to make it seem I supported something I actually opposed, to make me seem more insane than I actually was etc. -- again, forget I existed, view the ideas and judge them by clear logic; logic will help you reveal any edits made to this work, as this work is build on top of pure truth and logic, it is impossible to change something so as to keep it fitting in.
- Also, especially for the readers in further future, remember the message of this work will naturally be becoming more obscured and distorted just by the change of human language itself. Words slightly change meanings and sometimes shift by a lot, slight contemporary connotations and associations get lost and new ones arise so the meaning of every single word I use nowadays may differ significantly from your meaning of the word (this is always a problem with trying to understand ancient texts, see e.g. interpretations of Bible, quotes of Jesus and so on). Just as with intentional distortions though, logic should help you reveal them. This text is meant to point in the direction of truth and if it gets fuzzy, the direction will be more unclear, but you should be able to tell if it's pointing in the wrong direction because you can look there and you will simply find nothing.
- Remember to not become like them, do not use violence, do not become a fascist, do not fight them or wish them ill, be loving and peaceful, help everyone and be selfless. If against my advice you still choose to keep some memory of me, then please mainly remember that I loved you :) <3

---

wiki\_rights

## LRS Wiki Usage Rights

This page serves to establish usage rights for the whole LRS wiki. It is here to be part of the work so that the legal rights are always clear, even if e.g. the README gets lost somewhere along the way.

Everything on this wiki has been created from scratch solely by people listed in wiki authors. Great care has been taken to make sure no copyrighted content created by other people has been included in any way. This is because one of the goals of this wiki is to be completely in the public domain world wide.

Each contributor has agreed to release the whole LRS Wiki under the Creative Commons Zero 1.0 (CC0 1.0) waiver, available at <https://creativecommons.org/publicdomain/zero/1.0/>, with an additional option for you to also freely choose the following waiver instead:

The intent of this waiver is to ensure that this work will never be encumbered by any exclusive intellectual property rights and will always be in the public domain world-wide, i.e. not putting any restrictions on its use.

Each contributor to this work agrees that they waive any exclusive rights, including but not limited to copyright, patents, trademark, trade dress, industrial design, plant varieties and trade secrets, to any and all ideas, concepts, processes, discoveries, improvements and inventions conceived, discovered, made, designed, researched or developed by the contributor either solely or jointly with others, which relate to this work or result from this work. Should any waiver of such right be judged legally invalid or ineffective under applicable law, the contributor hereby grants to each affected person a royalty-free, non transferable, non sublicensable, non exclusive, irrevocable and unconditional license to this right.

---

wiki\_stats

## LRS Wiki Stats

This is an autogenerated article holding stats about this wiki.

- number of articles: 567
- number of commits: 746
- total size of all texts in bytes: 3410738
- total number of lines of article texts: 26808
- number of script lines: 256

longest articles:

- c\_tutorial: 104K
- capitalism: 64K
- chess: 56K
- how\_to: 52K
- less\_retarded\_society: 52K
- number: 48K
- faq: 44K
- c: 40K
- internet: 36K
- 3d\_rendering: 32K
- game: 32K
- random\_page: 32K
- programming\_language: 32K
- bloat: 32K
- history: 32K
- optimization: 28K
- mechanical: 28K
- iq: 28K
- procgen: 28K
- woman: 28K

top 50 5+ letter words:

- which (1984)
- there (1470)
- people (1324)
- other (1081)
- example (1032)
- software (1027)
- number (948)
- about (895)
- their (743)
- program (714)
- called (695)
- computer (682)
- would (678)
- because (668)
- simple (637)
- numbers (632)
- being (632)
- things (619)
- language (611)
- without (596)
- function (590)
- programming (588)
- however (583)
- something (559)
- these (553)
- different (545)
- system (516)
- world (509)
- society (503)
- should (502)
- games (502)
- point (496)
- though (486)
- doesn (482)
- memory (472)

- drummyfish (454)
- while (451)
- technology (450)
- using (448)
- simply (437)
- course (437)
- still (431)
- similar (429)
- possible (428)
- computers (396)
- really (393)
- extremely (393)
- usually (383)
- value (382)
- always (377)

latest changes:

Date: Thu Mar 21 20:00:23 2024 +0100

42.md  
 attribution.md  
 brainfuck.md  
 bullshit.md  
 cpu.md  
 creative\_commons.md  
 esolang.md  
 graphics.md  
 history.md  
 internet.md  
 ioccc.md  
 jokes.md  
 living.md  
 lrs.md  
 main.md  
 minimalism.md  
 nanogenmo.md  
 number.md  
 often\_confused.md  
 open\_source.md  
 programming\_language.md  
 random\_page.md  
 sigbovik.md  
 smart.md  
 thrembo.md  
 wiki\_pages.md  
 wiki\_stats.md

Date: Wed Mar 20 20:23:58 2024 +0100

fascism.md  
 interesting.md  
 jesus.md  
 lrs.md  
 number.md  
 people.md  
 political\_correctness.md

most wanted pages:

- data\_type (12)
- embedded (11)
- meme (10)
- buddhism (10)
- array (10)
- quake (9)
- irl (9)
- gpl (9)
- complex\_number (9)

- [tree](#) (8)
- [pointer](#) (8)
- [lisp](#) (8)
- [html](#) (8)
- [gpu](#) (8)
- [drm](#) (8)
- [cryptography](#) (8)
- [waiver](#) (7)
- [syntax](#) (7)
- [rpi](#) (7)
- [mcu](#) (7)

most popular and lonely pages:

- [lrs](#) (267)
- [capitalism](#) (197)
- [c](#) (194)
- [bloat](#) (193)
- [free\\_software](#) (162)
- [game](#) (132)
- [suckless](#) (131)
- [proprietary](#) (114)
- [modern](#) (86)
- [minimalism](#) (86)
- [kiss](#) (86)
- [linux](#) (84)
- [computer](#) (84)
- [programming](#) (78)
- [free\\_culture](#) (78)
- [math](#) (76)
- [fun](#) (75)
- [public\\_domain](#) (74)
- [gnu](#) (74)
- [foss](#) (73)
- [censorship](#) (71)
- [hacking](#) (70)
- [programming\\_language](#) (67)
- [fight\\_culture](#) (67)
- [shit](#) (66)
- [less\\_retarded\\_society](#) (66)
- [art](#) (66)
- [bullshit](#) (63)
- [float](#) (62)
- [open\\_source](#) (61)
- ...
- [trump](#) (4)
- [tom\\_scott](#) (4)
- [speech\\_synthesis](#) (4)
- [see\\_through\\_clothes](#) (4)
- [robot](#) (4)
- [README](#) (4)
- [primitive\\_3d](#) (4)
- [paywall](#) (4)
- [openai](#) (4)
- [nord\\_vpn](#) (4)
- [myths](#) (4)
- [murderer](#) (4)
- [monad](#) (4)
- [modern\\_software](#) (4)
- [mob\\_software](#) (4)

- [less retarded software](#) (4)
- [goodbye world](#) (4)
- [global discussion](#) (4)
- [gigachad](#) (4)
- [gaywashing](#) (4)
- [f2p](#) (4)
- [exercises](#) (4)
- [dungeons and dragons](#) (4)
- [dick reveal](#) (4)
- [deferred shading](#) (4)
- [cyber](#) (4)
- [crow funding](#) (4)
- [charity sex](#) (4)
- [body shaming](#) (4)
- [random page](#) (2)

---

wiki\_style

## LRS Wiki Style/Rules/Meta/Etc.

This outlines the style and rules of this Wiki that should ensure "quality" and consistency. You should read this before contributing.

If you contribute, add yourself to [wiki authors](#)! You can also join us on the [Island](#).

### Rules

1. **Everything is public domain** under [CC0](#) to which all contributors agree. No one owns what we write here.
2. **No fair use** or even unfair use. We want this Wiki to be as free as possible and don't thread the fine legal lines. That means you can't directly include anything on this Wiki if it's copyrighted, **even if it's under a free license**. So generally **avoid any copy pasting and rather try to write everything yourself**.
3. **No unnecessary censorship**. Necessary censorship basically only includes spam, shitty content, IP-infected content (content that would make this wiki not be in public domain) and hardcore illegal stuff that would immediately put us in jail, though we would of course love to include it. However spreading truth mustn't be hurt by fear of jail. Controversial/incorrect/taboo content etc. is NOT to be censored.
4. **Electronic bullying of other wiki users is NOT PROHIBITED**. Nothing is prohibited here. { Also I'm alone here so at best I can bully myself :D ~drummyfish }
5. **ILLEGAL CONTENT IS ALLOWED** -- it still won't be added, because that would make this wiki less legally [free](#), however this rule will be here to show we shit on laws and also that it's mostly quite stupid to forbid something that is already forbidden by definition, websites resort to doing it to dodge responsibility because they are [shit scared](#), but it's still stupid nonetheless. Putting rules like "uploading illegal content is prohibited" is basically like saying "I declare it that in my house laws of physics must function", illegal content will be prohibited whether you forbid it or not.
6. [Nigger](#). This rule has practically no effect because linguistically it has no meaning, but it is here to help [free speech](#).

### Style

- **Don't line-break paragraphs** (a paragraph is on a single line). The reasoning is that a text manually formatted to specific width is hard to edit. It's easier to switch to auto-line breaking in your editor.
- **Avoid unicode, highly prefer ASCII**, for the sake of maximum compatibility and simplicity. Use of unicode has to be greatly justified.
- **Each page shall start with a heading** (which may or may not correspond to article file name).
- I've finally decided that with certain exceptions headings should be written like this: **Each Word Of A Heading Is Capitalized**. This is for simplicity, we don't deal with weird ass rules of when or when not

to capitalize.

- **Filenames of articles shall use a lowercase snake\_case.**
- **Article/file names are to be preferably singular nouns.** I.e. "animal" rather than "animals", "portability" rather than "portable" etc. But there may be exceptions, e.g. articles that are lists may use plural ("human" is about human as species, "people" is a list of existing humans), non-nouns MAY be used if nouns would be too long/awkward (e.g. "weird" instead of "weirdness"). Use your brain.
- **This isn't Wikipedia**, memes, opinions and uncensored truths are allowed (and welcome). **References/citations are not needed**, we aren't a religion that relies on someone else's reputation to validate truth, we just spread information and leave it to others to either trust, test its usefulness and/or verify. Furthermore we don't limit ourselves to truths that can be or have been "proven" by whatever is currently called "approved science^TM", many valuable truths are just proven by experience etc. LRS advocates society in which deception doesn't happen and in which therefore there is no need for citations serving as a proof -- we practice this here. However references are still allowed and many times useful, e.g. as a further reading.
- Unlike with Wikipedia our criterion of inclusion isn't notability, i.e. things that people talk about, but value, i.e. **things that are worth talking about**. On the other hand if something is notable but of little value, such as details of OOP or famous feminists, we won't write too much about it, it's just useless noise that harms the good things by obscuring them.
- The style of this wiki is inspired by the classic hacker culture works such as the WikiWikiWeb and Jargon File.
- **Writing style should be relaxed and in many parts informal.** Formality is used where it is useful (e.g. definitions), most of other text can benefit from being written as a tech conversation among friends.
- **Depth/complexity/level of articles:** Articles shouldn't try to go to unnecessary depth, but also shouldn't be shallow. This is written mainly for programmers of less retarded society, the complexity should follow from that. Again, start simple and go more into depth later on in the article, very complex things should rather be explained intuitively, no need for complex proofs etc.
- **Scope:** don't write about literally everything, include only what's relevant from LRS point of view and only to relevant degree -- for example we could write many articles about OOP design patterns, however as that's a complete capitalist bullshit, this would just be noise from our point of view, it would drown useful content and also waste our effort, so we rather limit ourselves to some kind of tl;dr or OOP design patterns.
- **Future proof articles, minimize need for maintenance, focus on long term:** unlike Wikipedia we can't (and don't want to) be a dynamic resource that's only ever valid for one day, we want stable articles that will stay relevant forever. Don't go too much into temporary irrelevant shit such as which distro is currently the most bestest or how many forks dwm currently has. I.e. for example lists of software are not very good articles as that's something that will change constantly. List of historical events may be better as those are more or less given and only once in a decade we need to add few new relevant events, possibly also update the view on older ones. An article on mathematical concept on the other hand is quite good as equations are something that just stands valid forever.
- **The wiki is kind of a dirty idea dump**, it may contain TODOs, WIPs, errors, ugliness etc., it is not a finished beautiful and tidy encyclopedia. Remember this wiki is kind of a temporary (though long term) project that's meant to host, communicate and spread ideas that should direct us towards better society, it is a vehicle that should get us somewhere else and will potentially be replaced by something else once we are there. Therefore aims for perfection and beauty shouldn't stand in way of just communicating ideas (i.e. do not delete content for its ugliness if that content is still potentially useful).
- **Political incorrectness, slurs and "offensive speech" are highly encouraged.** Avoid the use of the word "person" (use "man", "guy", "human", "one" etc., possibly "individual" at worst). Of course this is not to "offend" anyone, this helps people unlearn being offended.
- **Images:** for now don't embed images. ASCII art can be used in many places instead of an image. Thousand words are worth a picture. Non-embedding links to images may be okay.
- **You can leave comments right in the text of articles**, e.g. like this: { I disagree with this shit. ~drummyfish }.

Articles should be written to be somewhat readable and understandable to tech savvy people who already know something about technology, i.e. neither illiterates, nor experts only (as is sometimes the case e.g. on Wikipedia). **Each article should ideally start with a general dictionary definition** and continue with a simple general explanation and overview of the topic. With more paragraphs the text can get more complex. The idea is that a noob will read the first paragraph, understand the basic idea and take something away. A

more advanced reader will read further on and take away more things etc. I.e. we educate in a top-down approach. **Each article should be a nice mini resource in itself**, quality should be preferred over quantity: for example the article on chess should be a nice general page about chess with focus on its programming, but also containing general overview, history, fun and interesting facts, data, essay elements and so on, so as to be highly self-contained (as opposed to the "Wikipedia approach" of making many separate articles on chess history, chess players, chess rules etc.).

## Sources

These are some sources you can use for research and gathering information for articles:

- **paper encyclopedias!** Consult these often, they are much better than any online resource, contain obscure, forgotten info and alternative points of view.
- **Wikipedia:** of course, but don't limit your search to it. Searching other language Wikipedias with machine translate can also help find extra info. If you know other languages than English, search that languages Wikipedia, it may have extra info. Also languages like Scots are understandable to English speakers, so try that as well.
- **Citizendium:** can offer a different angle of view from Wikipedia.
- **non-SJW forks of Wikipedia:** to get past SWJ censorship/propaganda on Wikipedia try e.g. **infogalactic** or **metapedia**.
- **Britannica online:** proprietary, but articles are nicely written, facts are in the public domain so we can steal them.
- **Archives:** **Internet Archive**, **Archive Team Wiki**, **archive.li**, ...: Most information once available on the Internet is most likely no longer accessible nowadays (taken down, privatized, censored, no longer indexed, ...). Look in the archives!
- **wikiwikiweb**
- **Ask people lol:** sometimes you can't find a piece of information anywhere (for example which rendering technique was used in an old proprietary game) but if you just send a short mail to someone (the game's programmer), he just gives you the information in a second. People often just forget this and spend countless hours digging for something when they can just write one email. Also just randomly talking to nerds on IRC and similar network often leads you to finding interesting topics to research which you wouldn't even think about.
- **Wiby, marginalia and other non-commercial search engines:** this will find nice small non-commercial sites of tech and other nerds that Google suffocates under bloatsites (or simply censors)
- **Project Gutenberg:** mostly older books but there are already some computer related books like **RMS's** biography or **Jargon File**
- **University theses** (and scientific paper of course): many university theses are publicly accessible and usually nicely sum up topics, bachelor level theses may be better understandable than PhD/master theses.
- **Slides:** slides from various presentations are actually great resources as they condense the topic into its main points, they filter out the piles of noise.
- **Wikisource, Wikibooks** etc.
- **books:** Books are still of higher quality than online sources so you can pirate some and steal some facts from them. Check out Anna's archive, libgen, torrents etc.
- Wikisphere -- check out various wikis, there are thousands, a place to start searching is **Wikiindex**. Wikis like **installgentoowiki**, **encyclopedia dramatica**, **soyak wiki** and **lurkmorewiki** may offer interesting insight into internet/tech culture, many times also less censored facts. Also check out **progopedia**, **osdev wiki**, **esolang wiki**, **chess programming wiki**, **wikiwikiweb** etcetc.
- **YouTube:** Yes, sadly this is nowadays one of the biggest sources of information which is unfortunately hidden in videos full of ads and retarded zoomers, the information is unindexed. If you are brave enough, you can dig this information out and write it here as a proper text.
- **reddit:** Sadly another SJW swamp of retards, however with many subcommunities almost about everything, if you have a question you can't find anywhere there is usually a subreddit where you can ask, if you dare.
- Try searching with different search engines than just Google (wiby, marginalia, Yandex, Bing, Yahoo, Internet Archive, ...).
- **old magazines:** If you can get your hands on old magazines somehow (physical, pdf, ...) about computers, cell phones, games etc., you may get a nice source of a lot of interesting, obscure and lost information. For example on the web it's impossible to find a good list of old games -- if you search for



"top 50 old games", you get a list of complete shit games from 2015, which is now considered "old"; it's much better you get some old game magazine from say 2005 that lists top 50 games at the time. Furthermore these magazines are professionally made and actually older, i.e. of better quality. You may also find nice gems, predictions of the future, politically incorrect articles etc.

- **Non-web:** When web fails, you can search the darknet, gopher, gemini, usenet, tor etc.
- ...

## Purpose

Basically the purpose of this wiki is to have fun and also create a coherent and consistent philosophy of a truly good software, technology and also to a related degree society.

Additionally we want to create a nice, completely public domain, suckless educational resource, an encyclopedia that's written in a nicer style than e.g. Wikipedia.

Another goal is to serve as a historical account of our age. We suppose that people in the future may want to read first-hand accounts of how people saw the society and its technology, what they thought was important, what they predicted etc. We like to do this ourselves thanks to e.g. Project Gutenberg. Kids in the future will be like "haha those idiots couldn't see they would bloat themselves to death", this Wiki will show that indeed we foresaw it, we were just in a huge minority. Future kids: don't repeat the same mistakes we did.

---

wikiwikiweb

## WikiWikiWeb

WikiWikiWeb (also *c2 Wiki* or just *Wiki*) was the first ever created wiki (user editable) website, created in 1995 in Perl by Ward Cunningham. It was focused on software engineering and computer technology in general but included a plethora of discussion and pages touching other topics as well, e.g. politics, humor or nerd and hacker culture. The principles on which it stood, most importantly allowing users to edit its the highly hyperlinked pages, largely influenced thousands of subsequently emerging sites which made use of the same concepts -- these sites are now collectively called wikis, most famous of which is Wikipedia. The style of WikiWikiWeb was partly an inspiration for our LRS wiki too.

The project quite impressively spawned over 36000 pages (<http://c2.com/cgi/wikiPages>). Since 2014 the wiki can no longer be edited due to vandalism, but it's still online. It was originally available at <http://www.c2.com/cgi/wiki>, now at <http://wiki.c2.com/> (sadly now requires JavaScript, WTF how is this a hacker site???)

The site's engine was kind of suckless/KISS, even Wikipedia looks bloated compared to it. It was pure unformatted HTML that used a very clever system of hyperlinks between articles: any CamelCase multiword in the text was interpreted as a link to an article, so for example the word SoftwareDevelopment was automatically a link to a page called *Software Development*. This presented a slight issue e.g. for single-word topics but the creativity required for overcoming the obstacle was part of the fun, for example the article on C was called *CeeLanguage*.

Overall the site was also very different from Wikipedia and allowed informal comments, jokes and subjective opinions in the text. It was pretty entertaining to read. There's a lot of old hacker wisdom to be found there. On the other hand it was a bit retarded too though, a bit like hacker news of its time, except a tiny bit less stupid maybe. The people were not as much focused on pure hacking but rather on "software engineering", i.e. manipulating and "managing" people, they were obsessed with OOO patterns and things like that.

There are other wikis that work in similar spirit, e.g. CommunityWiki (<https://communitywiki.org>, a wiki "about communities"), MeatBallWiki (<http://meatballwiki.org/wiki/>) or EmacsWiki.

## Interesting Pages

These are some interesting pages found on the Wiki.

{ NOTE: To see all pages in given category go to the category page and CLICK THE PAGE TITLE.  
~drummyfish }

- **CategoryCategory**: List of categories of pages.
- **CategoryHumor**: Humorous pages.
- **ComputerGame**
- **ExtinctionOfHumanity**: Discussing end of humanity and a possible collapse.
- **GameOfChess**: About chess.
- **LanguageGotchas**
- **WeirdErrorMessage**s
- **WikiWikiWebFaq**
- **WithinTwentyYears**: Mostly pre-2005 predictions about what technology would be like in 20 years, a lot of hits and misses.
- **WikiHistory**

## See Also

- Jargon File
  - Wikipedia
- 

windows

## Micro\$oft Window\$

Microsoft Windows is a series of malicious, bloated proprietary "operating systems". AVOID THIS SHIT.

## Versions

TODO

All are shit.

---

wizard

## Wizard

Wizard is a male virgin who is at least 30 years old (female virgins of such age haven't been seen yet). The word is sometimes also used for a man who's just very good with computers. These two sets mostly overlap so it rarely needs to be distinguished which meaning we intend.

There is an imageboard for wizards called wizardchan. It is alright but also kind of sucks, for example you can't share your art with others because of stupid anti-doxxing rules that don't even allow to dox yourself.

## See Also

- incel
  - volcel
- 

woman

## <3 Woman <3

A woman (also girl, gril, gurl, femoid, toilet, wimminz, the weaker sex, the dumber sex or succubus; tranny girl being called t-girl, trap, femboy, fake girl or mtf) is one of two genders (sexes) of humans, the other one being man. Women are the weaker sex, they are cute (sometimes) but notoriously bad at programming, math and technology: in the field they usually "work" on bullshit (and mostly harmful) positions such as

"diversity department", marketing, "HR", UI/user experience, or as a token girl for media. If they get close to actual technology, their highest "skills" are mostly limited to casual "coding" (which itself is a below-average form of programming) in a baby language such as Python, Javascript or Rust. Mostly they are just hired for quotas and make coffee for men who do the real work (until TV cameras appear). Don't let yourself be fooled by the propaganda, women have always been bad with tech.

The symbol for woman is a circle with cross at its bottom (Unicode U+2640). Women mostly like pink color and similar colors like red and purple.

{ I would like to say there are a few women and girls I hold dear, whom I admire and who are in many aspects much better beings than myself and most men I known. ~drummyfish }

**Even mainstream science acknowledges women are dumber than men:** even the extremely politically correct Wikipedia states TODAY in the article on human brain that male brain is on average larger in volume (even when corrected for the overall body size) AND that there is correlation between volume and intelligence: this undeniably implies women are dumber. On average male brain weights 10% more than woman's and has 16% more brain cells. The Guinness book of 1987 states the average male brain weight being 1424 grams and that of a female being 1242 grams; the averages both grow with time quite quickly so nowadays the numbers will be higher in both sexes, though the average of men grows faster. The heaviest recorded brain belonged to a man (2049 grams), while the lightest belonged to a woman (1096 grams). Heaviest woman brain weighted 1565 grams, only a little more than men's average. IQ/intelligence measured by various tests has been consistently significantly lower for women than for men, e.g. the paper named *Sex differences in intelligence and brain size: A paradox resolved* found a 4 point difference, noting that in some problems such as 3D spatial rotations males score even 11 points higher average.

Historically women have been privileged over men -- while men had to work their asses off, go to wars, explore and hunt for food, women often weren't even supposed to work, they could stay at home, chill while guarding the fire and playing with children -- this is becoming less and less so with capitalism which aims to simply enslave everyone, nowadays mostly through the feminist cult that brainwashed women to desire the same slavery as men. Statistically women live about 5 years longer lives than men because they don't have to worry and stress so much.

Women also can't drive, operate machines, they can't compare even to the worst men in sports, both physical and mental such as chess. Women have to have separate leagues and more relaxed rules, e.g. the title Woman Grand Master (WGM) in chess has far lower requirements to obtain than regular Grand Master (GM). (According to Elo rating the best woman chess player in history would have only 8% chance of winning against current best male who would have 48% chance of winning). On the International Mathematical Olympiad only 43 out of 1338 medals were obtained by females. There are too many funny cases and video compilations of women facing men in sports (watch them before they're censored lol), e.g. the infamous Vaevictis female "progaming" team or the football match between the US national women team (probably the best women team in the world) vs some random under 15 years old boy's team which of course the women team lost. LMAO there is even a video of 1 skinny boy beating 9 women in boxing. Of course there are arguments that worse performance of women in mental sports is caused culturally; women aren't led so much to playing chess, therefore there are fewer women in chess and so the probability of a good woman player appearing is lower. This may be partially true even though genetic factors seem at least equally important and it may equally be true that not so many women play chess simply because they're not naturally good at it; nevertheless the fact that women are generally worse at chess than men stands, regardless of its cause -- a randomly picked men will most likely be better at chess than a randomly picked woman, and that's what matters in the end. Also if women are displaced from chess by culture, then what is the area they are displaced to? If women are as capable as men, then for any area dominated by men there should be an area equally dominated by women, however we see that anywhere men face women men win big time, even in the woman activities such as cooking and fashion design. Feminists will say that men simply oppress women everywhere, but this just means that women are dominated by men everywhere, which means they are more skilled and capable at everything, there is no way out -- yes, antelope are oppressed by lions, but it's because lions are stronger than antelopes. Here we simply argue that women are weaker than men, not that oppressing women is okay -- it isn't. Furthermore if women were weaker but not by that much, we should statistically see at least occasional dominance by a woman, but we practically don't, it's really almost impossible to find a single such case in history, which indicates women are VERY SIGNIFICANTLY weaker, i.e. not something we negligible we could just ignore. Being a woman correlates to losing to a man almost perfectly, it is a great predictor, basically as strong as can appear in science. It makes

sense from the evolutionary standpoint as well, women simply evolved to take care of children, guard fire and save resource consumption by being only as strong as necessarily required for this task, while men had to be stronger and smarter to do the hard job of providing food and protection.

Now because today's brainwashed reader will see this as "sexism", let us remind ourselves that this is completely OK. Women are weaker, but in a good society this doesn't matter as in a good society people don't have to compete or prove their usefulness, everyone is loved equally, weak or strong. The issue here is not pointing out our differences but perpetuating a shitty society.

A woman does super stupid shit like pay all her (or her husband's) life savings for silicon breast implants, then wear a dress that basically consist solely of cleavage but then when a man makes eye contact with her breasts during conversation she's like "WHY U NOT LOOOKING IN MY EYEEEEEEEEES, STOP RAAAAPIIIIIIIIIIIING MEEEEEEEEEEEEEEEEEE!!!". Beware -- a woman possesses the dangerous weapon of seduction which she plentifully makes use of, e.g. for gold digging. A woman is a master of pretense. A wisdom as old as mankind itself states that "you cannot live with a woman, and you cannot live without her" -- this is true, though the latter is much easier to do, especially lately. A woman doesn't think logically, she thinks emotionally (menstruation and their hormone levels jumping all over the place further make this yet much worse), so what's true or false depends on how she feels at the moment OR in the future via so called reverse causality: a woman saying "yes" can actually mean "no" if 20 years later she decides it actually meant "no" -- an action in the future determines the past. Physicists weren't able to explain this phenomenon yet; in fact no male is probably ever capable of understanding a woman.

In the book Flatland women are mere line segments, the most primitive of two dimensional shapes, and it's made clear they are very dumb and dangerous, so society just treats them as beings without any notable intellect; a passage from the book for example cites that "it was decreed by the Chief Circle that, since women are deficient in reason but abundant in emotion, they ought no longer to be treated as rational, nor receive any mental education". The book is full of gems like that :D In the Bible the first woman, Eve, was made by God from the rib of Adam, the first man, to make him company, however that turned out to be a mistake as Eve then pissed off God by eating forbidden fruit (and making Adam eat it too) and got herself and Adam kicked out paradise -- for this she doomed all women to suffer regular monthly bleeding.

Of course even though rare, well performing women may statistically appear (though they will practically never reach the skill of the best men). That's great, such rare specimen could do great things. The issue is such women (as all others) are very often involved with a cult such as the feminists who waste their effort on fighting men instead of focusing on study and creation of real technology, and on actually loving it. They don't see technology as a beautiful field of art and science, they see it as a battlefield, a political tool to be weaponized to achieve social status, revenge on society etc., which spoils any rare specimen of a capable woman. Even capable women can't seem to understand the pure joy of programming, the love of creation for its own sake, they think more in terms of "learning to COOODE will get me new followers on social networks" etc. Woman mentality is biologically very different from men mentality, a woman is normally not capable of true, deep and passionate love, woman only thinks in terms of benefit, golddigging etc. (which is understandable from evolutionary point of view as women had to ensure choosing a good father for their offspring); men, even if cheating, normally tend towards deep life-long love relationships, be it with women or art. You will never find a virgin basement dweller programmer or demoscene programmer of female sex which isn't a poser, a hacker who is happy existing in a world of his own programs without the need for approval or external reward, a woman will likely never be able to understand this. This seems to be evolutionary given, but perhaps in a better culture these effects could be suppressed.

Supposed "achievements" of women after circa 2010 can't be taken seriously, propaganda has started to tryhard and invent and overrate achievements and basically just steal achievements of men and hand them over to women (not that there were any significant achievement post 2010 though). There are token women inserted on soyentific positions etc. (lol just watch any recent NASA mission broadcast, there is always a woman inserted in front of the camera).

Of course, LRS loves all living beings equally, even women. In order to truly love someone we have to be aware of their true nature so that we can truly love them, despite all imperfections.

**Is there even anything women are better at than men?** Well, women seem for example more peaceful or at least less violent on average (feminism of course sees this as a "weakness" and tries to change it), though they seem to be e.g. more passive-aggressive. Nevertheless there have been a few successful

queens in history, women can sometimes perhaps be good in representative roles (and other simple chair-sitting jobs), in being a "symbol", which doesn't require much of any skill (a statue of a god can do the same job really). They have also evolved to perform the tasks of housekeeping and care taking at which they may excel (still it seems that if men fully focus on a specific task, they will beat women, for example the best cooks in the world are men). Sometimes women may be preferable exactly for not being as "rough" as men, e.g. as singers, therapists, sex workers etc. There were also some good English female writers actually, like Agatha Christie and J. K. Rowling, though that's still pretty weak compared to Hemingway, Goethe, Tolkien, Tolstoy, Shakespeare, Dickens, Dostoevsky etcetc.

lol <http://www.menarebetterthanwomen.com> also <https://encyclopedia.dramatica.online/Woman> :D

**How to deal with being a woman?** Well, just as you deal with not being born Einstein, Phelps or Kasparov. It's fine to be anyone, there is no need to fight. Try to be a good human and live a fulfilling life, you can create a lot of good.

## Men Vs Women In Numbers

Here is a comparison of men and women in numbers that are still possible to be found in already highly censored sources. Of course, the numbers aren't necessarily absolutely up to date, at the time or reading they may be slightly outdated, also keep in mind that in the future such comparisons may become much less objective due to SIW forces -- e.g. because of trans athletes in sports we may see diminishing differences between measurements of performance of men and "women" because what in the future will be called women will be just men pretending to be women.

Note: It is guaranteed that soyentific BIGBRAINS will start screeching "MISLEADING STATISTICSSSSSSSS NON PEER REVIEWED". Three things: firstly chill your balls, this isn't a scientific paper, just a fun comparison of some numbers. Secondly fuck you, we don't fancy peer censorship. Thirdly we try to be benevolent and not choose stats in a biased way (we don't even have to) but it is not easy to find better statistics, e.g. one might argue it could be better to compare averages or medians rather than bests -- indeed, but it's impossible to find average performance of all women in a population in a specific sport discipline, taking the best performer is simply easier and still gives some idea. So we simply include what we have. Thirdly any statistics is a simplification and can be seen as misleading by those who dislike it.

| measure              | men               | women              | comment                               |
|----------------------|-------------------|--------------------|---------------------------------------|
| height: average (EU) | 178 cm            | 165 cm             |                                       |
| height: greatest     | 273 cm (Wadlow)   | 257 cm (Jinlian)   |                                       |
| weight: average (EU) | 85 kg             | 70 kg              |                                       |
| weight: greatest     | 442 kg (Minnoch)  | 385 kg (Carnemoll) |                                       |
| brain: avg. weight   | 1424 g            | 1242 g             |                                       |
| brain: heaviest      | 2049 g            | 1565 g             |                                       |
| muscle/mass avg.     | 42%               | 32%                |                                       |
| life span: avg. (EU) | 75 years          | 81 years           |                                       |
| life span: greatest  | 116 y. (Kimura)   | 122 y. (Calment)   | top 10 oldest people ever are all W   |
| average IQ (US 1993) | 101.8             | 98.8               | in some areas M score even 11 higher  |
| 200m outdoor WR      | 19.90s (Bolt)     | 21.34s (G-Joyner)  | best W ranks lower than #5769 among M |
| 60m indoor WR        | 6.34s (Coleman)   | 6.92s (Privalova)  | best W ranks lower than #3858 among M |
| raw deadlift WR      | 460kg (Magnusson) | 305kg (Swanson)    | best M lifts about 50% more weight    |
| marathon             | 2:01 (Kipchoge)   | 2:14 (Kosgei)      | best W ranks #3935 among men          |
| 100m swim WR         | 46.8s (Popovici)  | 51.7s (Sjostrom)   | best W ranks lower than #602 among M  |
| chess best Elo       | 2882 (Carlsen)    | 2735 (Polgar)      | best W win 8%, lose 48%, draw 44%     |
| go best Elo          | 3862 (Jinseo)     | 3424 (Choi Jeong)  | best W ranks #68, M win prob.: 92%    |
| speedcubing WR       | 3.47s (Du)        | 4.44 (Sebastien)   | best W ranks #16 among M              |
| Starcr. 2 best Elo   | 3556 (Serral)     | 2679 (Scarlett)    | best M has ~80% win chance against W  |
| holding breath WR    | 24:37 (Sobat)     | 18:32m (Meyer)     | Ms have ~35% greater lung capacity    |

## How To Get Women

Don't!

see also [incel/volcel](#)

Any girl that has ever seen the [Internet](#) is spoiled beyond grave, avoid these for any cost. If you seriously want to live with a woman, it's best to consider diving into the jungle and find some half ape indigenous girl not touched by capitalism yet, those may be unironically cool.

Jerking off is the easiest solution to satisfying needs connected to fucking women. If you absolutely HAVE to get laid, save up for a prostitute, that's the easiest way and most importantly won't ruin your life. Or decide to become [gay](#), that may make matters much easier. You may also potentially try to hit on some REAL ugly girl that's literally desperate for sex, but remember it has to be the ugliest, fattest landwhale that you've ever seen, it's not enough to just find a 3/10, that's still a league too high for you that will reject you unless you pay her. Also consider that if you don't pay for sex, there is a 50% chance you will randomly get sued for rape sometime during the following 30 year period. If you want a girlfriend, then rather don't. The sad truth is that to make a woman actually "love" you, as much as one is capable of doing so, you HAVE TO be an enormously evil ass that will beat her to near death, abuse her, rape her and regularly cheat on her -- that's how it is and that's what every man has to learn the hard way -- as we know, the older generation's experience cannot be communicated by words, the young generation always thinks it is somehow different and will never listen. Sadly this is simply how it is -- even if you think you have found the "special one", the one that's different, the intelligent introverted one that's nice and friendly to you, nope, she is still a woman, she won't love you unless you're a murderer dickass beating her daily (NOTE: we don't advocate any violence, our advice here is to simply avoid women). If you think getting close to her, being nice and listening to her will make her love you, you're going to hit a brick wall very hard -- this road only ever leads to a friendzone 100% of the times, you will end up carrying her purse while she's shopping without her letting you touch her ever. If you just want a nonsexual girl friend, then it's fine, but you will never make a girlfriend this way. This is not the girl's fault, she is programmed like that, blaming the girl here would be like blaming a child for overeating on candy or blaming a cat for torturing birds for fun; and remember, THE GIRL SUFFERS TOO, she is literally attracted only to those who will abuse her, it is her curse. If anyone's to blame for your suffering, it is you for being so extremely naive -- always remember you are playing with fire. You may still get a girl to stay with you or even marry you and have kids if you have something that will make her want to be with you despite not loving you, which may include being enormously rich, being so braindead to have million subscribers on YouTube, having an enormous 1 meter long dick or literally giving up all dignity and succumbing to being her lifelong slave dog doing literally everything she says when she says it, but that will still get you at most 4/10 and is probably not worth it. { From my experience this also goes for trans girls somehow, so tough luck. Maybe it's so even for gay men in the woman role. ~drummyfish } All in all rather avoid all of this and pay for a prostitute, buy some sex toys, watch porn and stay happy <3

## Notable Women In History

Finding famous women capable in technology is almost a futile task. One of the most famous women of [modern](#) tech, even though more an entrepreneur than engineer, was [Elizabeth Holmes](#) who, to the feminists' dismay, turned out to be a complete fraud and is now facing criminal charges. [Grace Hopper](#) (not "grass hopper" lol) is a woman actually worth mentioning for her contribution to programming languages, though the contribution is pretty weak. [Ada Lovelace](#) cited by the feminist propaganda as the "first programmer" also didn't actually do anything besides scribbling a note about a computer completely designed by a man. This just shows how desperate the feminist attempts at finding capable women in tech are. Then there are also some individuals who just contributed to the downfall of the technology who are, in terms of gender, at least partially on the woman side, but their actual classification is actually pretty debatable -- these are monstrosities with pink hair who invented such [cancer](#) as [COCs](#) and are not even worth mentioning.

In the related field of [free culture](#) there is a notable woman, [Nina Paley](#), that has actually done some nice things for the promotion of free culture and also standing against the [pseudoleftist](#) fascism by publishing a series of comics with a character named Jenndra Identity, a parody of fascist trannies. Some rare specimen of women openly oppose feminism -- **these are the truly based women**.

{ Maybe not historically notable but one based Czech woman, at least in terms of her art, is Magdalena Vozicka who wrote an amazing book about how to survive the end of civilization, I loved it so much, this woman truly earned my respect. Also huge thanks goes to my female friend who gifted me the book.  
~drummyfish }

Here is a list of almost all historically notable women:

- **Ada Lovelace**: female nobleman who didn't have to work, once scribbled a note to a notebook about a computer made by a man. For this she enjoys endless glory among feminists.
- **Agatha Christie**: one of the most famous UK writers, wrote books such as *Ten Little Niggers*, one day went nuts and ran somewhere into woods.
- **Beth Harmon**: female who was as good at chess as men, also a completely fictional character who never existed.
- **Elizabeth II**: queen of England, managed to stay alive for a long time.
- **Elizabeth Holmes**: cringe and creepy psychopath who obsessively tried to imitate Steve Jobs, started a huge corporation and manipulated uncountable people into a huge fraud, sentenced to 11 years in jail.
- **Emily Wilding Davison**: injured an innocent horse by jumping under it in a protest.
- **Eve of the Bible**: achieved probably the biggest fuck up in history, she did the single one thing she was forbidden from doing without even gaining much benefit from it, she ate some kind of God forbidden fruit, enraged God and doomed all people who will ever live to be banished from paradise :D
- **Helen of Troy**: caused the Troy war.
- **Hermione Granger**: smart girl, also fictional (these two attributes seem to go together in girls).
- **Joan of Arc**: militant nationalist fascist, basically Christian jihadi.
- **Judit Polgar**: best non-fictional female chess player that at her peak managed the incredible feat of ranking #56 in the world while actually existing.
- **Marie Curie**: this one was actually probably quite skilled and based, won two Nobel Prizes (at the time when there were no diversity quotas so it actually counts), though she probably stole most of her work from her husband. She was quite ugly tho.
- **Lisa Nowak**: a female astronaut, military pilot, i.e. someone who would ideally be among those with highest mental stability and reliability, who nonetheless one day went on a rage frenzy over a sexual affair with some army chad, pepper sprayed some other bitch and was subsequently charged with attempted murder after weapons were found in her car... women... :D
- **Olga Hepnarova**: ran over 8 people with a truck, later executed.
- **Yoko Ono**: destroyed the most famous music band in history.
- ...

{ People ask me why I'm bashing women so much here :D Basically for fun, but the main reason is probably feminism -- the more they try to make women godlike, the more fun I make of them. So blame feminists basically. ~drummyfish }

## See Also

- man
- t-girl
- waifu

---

work

## Work

*Work is a glorification of slavery.*

Work, better known as slavery, is an unpleasant effort that one is required to suffer, such as harvesting crops or debugging computer programs. Work hurts living beings and takes away the meaning of their lives, it destroys their bodies and minds. Work makes us slaves, it wastes our lives and is a cause of a large number of suicides -- many consider it better to die than to work. One of the main goals of civilization is to eliminate any need for work, i.e. create machines that will do all the work for humans (see automation).

**Fun fact:** the Spanish (also Portuguese etc.) word for work, "trabajo", comes from *tripalium*, a device made of three sticks used to force slaves to work. { Thanks to my friend who told me about this <3 ~drummyfish }

While good society tries to eliminate work, capitalism aims for the opposite, i.e. artificially creating bullshit jobs and bullshit needs so as to keep everyone enslaved to the system. Fortunately movements such as the antiwork movement try to oppose this, however masses have already been brainwashed to be hostile to such movements and instead demand their own enslavement.

We see it as essential to start educating people about the issue as well as starting to eliminate jobs immediately with things such as automation and universal basic income.

## How To Avoid Work

Here are some ways in which it is possible to avoid work:

- **Becoming as independent as possible:** living frugally, stopping consuming, using old technology, growing own food etc. Even if one needs some money, needing less is always better, as one can work less.
- **Just saying fuck you to any work lol:** fuck everything YOLO style can be fun, just stop doing anything, go beg for money. You'll probably lose any social security such as healthcare and old age pension, but you may still get some minimum life support, and if you are extremely sick just lay down in front of main hospital entrance, they'll probably have to save your life even without health insurance. Even if not, living fewer years in freedom is probably better than living longer in slavery.
- **Saving money:** obviously, saving a lot of money makes one able to retire soon. Watch out for inflation that destroys savings, it's nice to put some part of it e.g. in gold so that it keeps the value.
- **Some professions come with long holidays,** typically e.g. teachers get about two months off each year, so you can try to find something like this. Also some seasonal work and so on. In some countries an employer is required to still pay you a portion of salary even if there is no work to be done and you have to stay at home -- you can strategically search for work in an industry that is going to stagnate.
- **Getting a disability pension:** if you are really disabled then you've already won, otherwise either fake it or bribe a doctor (e.g. with sex). As a desperate move a capitalist slave may even go as far as trying to cripple himself on purpose to avoid remaining slave for his whole life, however we can't downright recommend this as of course this can backfire in many ways. One possible way of doing so might be e.g. stopping eating and become anorexic (to many a whole life of freedom would be worth a few months of starving).
- **Faking sickness or getting sick on purpose:** for an employee it is possible to pretend to be sick to avoid work, to bribe doctor or break one's leg on purpose to stay at home and leech the employer and state is a possibility (of course we don't advice you to hurt yourself, just saying it's possible). If one just keeps breaking his leg over and over to avoid work, they will also likely give him some mental disease diagnosis and a disability pension so that he can stay at home indefinitely.
- **Leeching welfare:** it's a common practice to e.g. register at the employment office and then just take unemployment support.
- **Stealing from the rich:** stealing stuff from supermarkets, offices etc. is nice. It's also helping society. Do not steal from the poor.
- **Getting a rich partner?:** someone rich can just take care of you for sex and love, however it may be not worth it as rich people are often capitalists whom it's better to stay away from.
- **Moving to some nice community that doesn't force work:** the problem is actually finding such community, but maybe some hippie tent villages could be like that.
- **Going to jail:** in some countries jail are quite luxurious and once in jail you can just refuse to work as they cannot lock you up more. In jail you have shelter and food, i.e. already more than most people in a capitalist society. However watch out: for some crimes you may just get fined, not actually locked up, so it's good to study the law to know which crimes it's best to commit to safely get one to jail.
- **Become a prostitute (usually for women):** it's easy money and you literally get paid for having sex. Unless you're real ugly it may be enough to just "work" like this for a few days in a month.
- ...

---

world\_broadcast



# World Broadcast

{ I don't know too much about radio transmissions, please send me a mail if something here is a complete BS. ~drummyfish }

WIP!

World (or world-wide) broadcast is a possible technological service (possibly complementing the Internet) which could be implemented in a good society and whose main idea is to broadcast generally useful information over the whole globe so that simple and/or energy saving computers could get basic information without having to perform complex and costly two-way communication.

It would work on the same principle as e.g. teletext: there would be many different radio transmitters (e.g. towers, satellites or small radios) that would constantly be broadcasting generally useful information (e.g. time or news) in a very simple format (something akin text in Morse code). Any device capable of receiving radio signal could wait for desired information (e.g. waiting for certain keyword such as TIME: or NEWS:) and then save it. The advantage would be simplicity: unlike with Internet (which would of course still exist) the device wouldn't have to communicate with anyone, there would be no servers communicating with the devices, there would be no communication protocols, no complex code, no DDOS-like overloading of servers, and the receiving devices wouldn't waste energy (as transmitting a signal requires significant energy compared to receiving it -- like shouting vs just listening). It would also be more widely available than Internet connection, e.g. in deserts.

## See Also

- Ronja
- 

wow

# World Of Warcraft

World of Warcraft (WoW) is an AAA proprietary game released in 2004 by Blizzard that was one of the most successful and influencing games among MMORPGs. It's the mainstream kind of MMO, considered pretty easy (compare e.g. to Eve Online).

There is a FOSS implementation of WoW server called MaNGOS (now having some forks) that's used to make private servers. The client is of course proprietary and if you dare make a popular server Blizzard (or whatever it's called now, it's probably merged with Micro\$oft or something now) will just rape you.

The classic WoW (mostly the vanilla but we can possibly extend this to the end of WOTLK) lied somewhere in the middle between good old and shitty modern games, it had many great things like the iconic awesome low poly hand painted stylized graphics, big open world, amazing PvP and PvE, but the modern poison was already creeping in. The WoW of today is of course 100% pure shit, it's bloated beyond any imagination, the graphics is absolutely ruined (semi realistic style, everything looks like a cheap plastic toy, with the retarded shit like character outlines, it looks much worse and is also 10000x heavier on the GPU), it's extremely censored and politically correct (you can literally change gender of your character at barbershop lol, they did this out of fear of LGBT, they also removed the *spit* emote because it was "offensive" -- yes, a game that's all about war and killing and literally has war in its name must restrain you from hurting someone's feelings by spitting on the ground). You can also make any weapon or armor make look like any other weapon or armor ("transmog"), that just kills the whole point of an RPG, some players also see a different world than others ("phasing") and so on. Also basically every race can now be any class, even if it doesn't make any sense, like Tauren rogue (in the past this used to be a joke but today jokes are made into reality) -- otherwise it would be racism or something. The game has about 1 billion expansions while the lore writers had already ran out of any ideas after like 5 of them, so they now just started to mess around with time travel and alternative timelines (resorting to time rape is always that desperate last resort move which signifies the work has been dead for a long time by then). The game is so bad Blizzard even started running official vanilla, no expansion servers ("classic WoW"), which is the only thing holding it above the water now. Of course before this they nuked all the popular unofficial private vanilla servers with legal threats so they could force a monopoly --

this destroyed great many communities but Blizzard is a corporation so they could do anything they want.

{ For me the peak of Warcraft was Warcraft III:TFT, it was perfect in every way (except for being proprietary and bloated of course). As a great fan of Warcraft III, seeing WoW in screenshots my fantasy made it the best game possible to be created. When I actually got to playing it it was really good -- some of my best memories come from that time -- nevertheless I also remember being disappointed in many ways. Especially with limitation of freedom (soulbound items, forced grinding, effective linearity of leveling, GMs preventing hacking the game in fun ways etc.) and here and there a lack of polish (there were literally visible unfinished parts of the map, also visual transitions between zones too fast and ugly and the overall world design felt kind of bad), laziness and repetitiveness of the design. I knew how the game could be fixed, however I also knew it would never be fixed as it was in hands of a corporation that had other plans with it. That was the time I slowly started to see things not being ideal and the possibility of a great thing going to shit.  
~drummyfish }

---

www

## World Wide Web

*Want to make your own small website? See our how to.*

World Wide Web (www or just *the web*) is (or was -- by 2023 mainstream web is dead) a network of interconnected documents on the Internet, which we call *websites* or *webpages*. Webpages are normally written in the HTML language and can refer to each other by hyperlinks ("clickable" links right in the text). The web itself works on top of the HTTP protocol which says how clients and servers communicate. Some people confuse the web with the Internet, but of course those people are retarded: web is just one of many so called services existing on the Internet (other ones being e.g. email or torrents). In order to browse the web you need an Internet connection and a web browser.

{ **How to browse the web in the age of shit?** Currently my "workflow" is following: I use the badwolf browser (a super suckless, very fast from-scratch browser that allows turning JavaScript on/off, i.e. I mostly browse small web without JS but can still do banking etc.) with a **CUSTOM START PAGE** that I completely own and which only changes when I want it to -- this start page is just my own tiny HTML on my disk that has links to my favorite sites (which serves as my suckless "bookmark" system) AND a number of search bars for different search engines (Google, Duckduckgo, Yandex, wiby, Searx, marginalia, Right Dao, ...). This is important as nowadays you mustn't rely on Google or any other single search engine -- I just use whichever engine I deem best for my request at any given time. ~drummyfish }

An important part of the web is also searching its vast amounts of information with search engines such as the infamous Google engine. It also relies on systems such as DNS.

Mainstream web is now EXTREMELY bloated and practically unusable, for more suckless alternatives see gopher. See also smol web.

The web used to be perhaps the greatest part of the Internet, the thing that made Internet widespread, however it quickly deteriorated by capitalist mainstreamization and commercialization and by now, in 2020s, it is one of the most illustrative, depressing and most hilarious examples of capitalist bloat. A nice article about the issue, called *The Website Obesity Crisis*, can be found at [https://idlewords.com/talks/website\\_obesity.htm](https://idlewords.com/talks/website_obesity.htm). There is a tool for measuring website bloat at <https://www.webbloatscore.com/>: it computes the ratio of the page size to the size of its screenshot (e.g. YouTube currently scores 35.7).

Currently there are visions of so called "**web 3**" which should be the "next iteration" of the web with new paradigms, making use of "modern" (i.e. probably shitty) technology such as blockchain; they say web 3 wants to use decentralization to prevent central control and possibly things like ensorship, however we can almost certainly guarantee web 3 will be yet exponentially greater pile of bloat and a worse dystopia than what we have yet seen, we simply have to leave this ship sink. If web 3 will be what web 2.0 was to web 1.0, then indeed we are doomed. Our prediction is that web will simply lose its status of the biggest Internet service just as Usenet did, or like TV lost its status of the main audiovisual media; web will be replaced by something like akin "islands of franchised social media accessed through apps"; it will still be around but will

## How It Went To Shit

|                                                                                |       |                                                                       |
|--------------------------------------------------------------------------------|-------|-----------------------------------------------------------------------|
| ENLARGE PENIS WITH SNAKE OIL                                                   | CSS   | Video AD<br>CONS00000000000000<br>000000000000M BICH                  |
| U.S. PRESIDENT ASSASINATED                                                     | BUG   |                                                                       |
| Article unavailable in your country.                                           | LOL   |                                                                       |
| We deeply care about your privacy <3                                           |       | Prove you're a human, click all images of type 2 quasars.             |
| Will you allow us to use cookies for spying?                                   |       | [*] [*] [*] [*]<br>[*] [*] [*] [*]                                    |
| YES                                                                            | OK    |                                                                       |
| .....                                                                          | ..... |                                                                       |
| Your browser is 2 days old, please update to newest version to view this site. |       | FUCK MATURE MOMS<br>IN 127.0.0.1<br>CHAT NOW !!!<br>*30 NEW MESSAGES* |

As the time marched on web used to become more and more shit, as is the case with everything touched by capitalist hand -- the advent of so called **web 2.0** brought about a lot of complexity, websites started to incorporate client-side scripts (JavaScript, Flash, Java applets, ...) which led to many negative things such as incompatibility with browsers (kickstarting browser consumerism and update culture), performance loss and security vulnerabilities (web pages now became programs rather than mere documents) and more complexity in web browsers, which leads to immense bloat and browser monopolies (greater effort is needed to develop a browser, making it a privilege of those who can afford it, and those can subsequently dictate de-facto standards that further strengthen their monopolies). Another disaster came with **social networks** in mid 2000s, most notably Facebook but also YouTube, Twitter and others, which centralized the web and rid people of control. Out of comfort people stopped creating and hosting own websites and rather created a page on Facebook. This gave the power to corporations and allowed **mass-surveillance, mass-censorship** and **propaganda brainwashing**. As the web became more and more popular, corporations and governments started to take more control over it, creating technologies and laws to make it less free. By 2020, the good old web is but a memory and a hobby of a few boomers, everything is controlled by corporations, infected with billions of unbearable ads, DRM, malware (trackers, crypto miners, ...), there exist

no good web browsers, web pages now REQUIRE JavaScript even if it's not needed in principle due to which they are painfully slow and buggy, there are restrictive laws and censorship and de-facto laws (site policies) put in place by corporations controlling the web.

Mainstream web is quite literally unusable nowadays. { 2023 update: whole web is now behind cuckflare plus secure HTTPS safety privacy antipedophile science encrypted privacy antiterrorist democratic safety privacy security expert antiracist sandboxed protection and therefore literally can't be used. Also Google has been absolutely destroyed by the LLM AIs now. ~drummyfish } What people searched for on the web they now search on on a handful of platforms like Facebook and YouTube (often not even using a web browser but rather a mobile "app"); if you try to "google" something, what you get is just a list of unusable sites written by AIs that load for several minutes (unless you have the latest 1024 TB RAM beast) and won't let you read beyond the first paragraph without registration. These sites are uplifted by SEO for pure commercial reasons, they contain no useful information, just ads. Useful sites are buried under several millions of unusable results or downright censored for political reasons (e.g. using some forbidden word). Thankfully you can still try to browse the smol web with search engines such as wiby, but still that only gives a glimpse of what the good old web used to be.

{ More of web 2023 experience: if you want to Google something as simple as "HTML ampersand", just to get the HTML entity 5 character code, you basically get referred to a site that's 200 MB big, loads for about 1 minute (after you pass 10 checks for not being a robot), has 50 sections and subsections like "Who This Tutorial on Copying 5 Character is for", "What You Will Learn in This Tutorial", "Time Required for Reading This Tutorial" (which without these sections would be like 3 seconds), "Introduction: History of HTML" (starting with Stone Age) etc. There are of course about 7 video ads between each section and the next. Then finally there is the & code you can copy paste, buried in level 12 subsection ("HTML Code" -> "History of Programming Since Napoleon Bonaparte" -> "How Ada Lovelace Invented Computer Science" -> "How Tim Berners-Lee Stole The Idea For Web from His Wife" -> "Why Women Only Crews For Next Space Mission are a Good Idea" -> "How This All Finally Gets Us to HTML Amp Entity" -> ...). Then of course there follow about 600 more sections like "Methodology Used to Create This Copying Tutorial" etcetc. until "Conclusion: What We Have Learned about the HTML Amp Entity and History of Feminism"; but at least you don't have to scroll through that; anyway at this point you are already suicidal and don't even want to write your HTML anymore. ~drummyfish }

## History

As with most revolutionary things the web didn't really appear out of nowhere, the ideas it employed were tried before, for example the NABU network did something similar even 10 years before the web; similarly Usenet, the BBS networks and so on. However it wasn't until the end of 1980s that all the right ideas would come together under the right circumstances and had a bit of luck to get really popular.

World Wide Web was invented by an English computer scientist Tim Berners-Lee. In 1980 he employed hyperlinks in a notebook program called ENQUIRE, he saw the idea was good. On March 12 1989 he was working at CERN where he proposed a system called "web" that would use hypertext to link documents (the term hypertext was already around). He also considered the name *Mesh* but settled on *World Wide Web* eventually. He started to implement the system with a few other people. At the end of 1990 they already had implemented the HTTP protocol for client-server communication, the HTML, language for writing websites, the first web server and the first web browser called *WorldWideWeb*. They set up the first website <http://info.cern.ch> that contained information about the project (still accessible as of writing this).

In 1993 CERN made the web public domain, free for anyone without any licensing requirements. The main reason was to gain advantage over competing systems such as Gopher that were proprietary. By 1994 there were over 500 web servers around the world. WWW Consortium (W3M) was established to maintain standards for the web. A number of new browsers were written such as the text-only Lynx, but the proprietary Netscape Navigator would go to become the most popular one until Microsoft's Internet Explorer (see browser wars). In 1997 Google search engine appeared, as well as CSS. There was a economic bubble connected to the explosion of the Web called the dot-comm boom.

Between 2000 and 2010 there used to be a mobile alternative to the web called WAP. Back then mobile phones were significantly weaker than PCs so the whole protocol was simplified, e.g. it had a special markup language called WML instead of HTML. But as the phones got more powerful they simply started to support normal web and WAP disappeared.

Around 2005, the time when YouTube, Twitter, Facebook and other shit sites started to appear and become popular, so called Web 2.0 started to form. This was a shift in the web's paradigm towards more shittiness such as more JavaScript, bloat, interactivity, websites as programs, Flash, social networks etc. This would be the beginning of the web's downfall.

## How It Works

It's all pretty well known, but in case you're a nub...

Users browse the Internet using web browsers, programs made specifically for this purpose. Pages on the Internet are addressed by their URL, a kind of textual address such as `http://www.mysite.org/somefile.html`. This address is entered into the web browser, the browser retrieves it and displays it.

A webpage can contain text, pictures, graphics and nowadays even other media like video, audio and even programs that run in the browser. Most importantly webpages are hypertext, i.e. they may contain clickable references to other pages -- clicking a link immediately opens the linked page.

The page itself is written in HTML language (not really a programming, more like a file format), a relatively simple language that allows specifying the structure of the text (headings, paragraphs, lists, ...), inserting links, images etc. In newer browsers there are additionally two more important languages that are used with websites (they can be embedded into the HTML file or come in separate files): CSS which allows specifying the look of the page (e.g. text and font color, background images, position of individual elements etc.) and JavaScript which can be used to embed scripts (small programs) into webpages which will run on the user's computer (in the browser). These languages combined make it possible to make websites do almost anything, even display advanced 3D graphics, play movies etc. However, it's all huge bloat, it's pretty slow and also dangerous, it was better when webpages used to be HTML only.

The webpages are stored on web servers, i.e. computers specialized on listening for requests and sending back requested webpages. If someone wants to create a website, he needs a server to host it on, so called hosting. This can be done by setting up one's own server -- so called self hosting -- but nowadays it's more comfortable to buy a hosting service from some company, e.g. a VPS. For running a website you'll also want to buy a web domain (like `mydomain.com`), i.e. the base part of the textual address of your site (there exist free hosting sites that even come with free domains if you're not picky, just search...).

When a user enters a URL of a page into the browser, the following happens (it's kind of simplified, there are caches etc.):

1. The domain name (e.g. `www.mysite.org`) is converted into an IP address of the server the site is hosted on. This is done by asking a DNS server -- these are special servers that hold the database mapping domain names to IP addresses (when you buy a domain, you can edit its record in this database to make it point to whatever address you want).
2. The browser sends a request for given page to the IP address of the server. This is done via HTTP (or HTTPS in the encrypted case) protocol (that's the `http://` or `https://` in front of the domain name) -- this protocol is a language via which web servers and clients talk (besides websites it can communicate additional data like passwords entered on the site, cookies etc.). (If the encrypted HTTPS protocol is used, encryption is performed with asymmetric cryptography using the server's public key whose digital signature additionally needs to be checked with some certificate authority.) This request is delivered to the server by the mechanisms and lower network layers of the Internet, typically TCP/IP.
3. The server receives the request and sends back the webpage embedded again in an HTTP response, along with other data such as the error/success code.
4. Client browser receives the page and displays it. If the page contains additional resources that are needed for displaying the page, such as images, they are automatically retrieved the same way (of course things like caching may be employed so that they same image doesn't have to be redownloaded literally every time).

Cookies, small files that sites can store in the user's browser, are used on the web to implement stateful behavior (e.g. remembering if the user is signed in on a forum). However cookies can also be abused for tracking users, so they can be turned off.

Other programming languages such as [PHP](#) can also be used on the web, but they are used for server-side programming, i.e. they don't run in the web browser but on the server and somehow generate and modify the sites for each request specifically. This makes it possible to create dynamic pages such as [search engines](#) or [social networks](#).

## See Also

- [Dark Web](#)
- [Dork Web/Smol Internet](#)
- [teletext](#)
- [minitel](#)
- [NABU](#)
- [WAP](#)
- [Usenet](#)
- [TOR](#)
- [Gopher](#)
- [Gemini](#)
- [BBS](#)
- [Freenet](#)
- [IPFS](#)
- [Internet](#)
- [Kwangmyong](#)
- [cyberspace](#)
- [SNet](#)
- [how to make a website](#)

---

x86

## x86

x86 is a [bloated](#), toxic [instruction set architecture](#) (or rather a family of them) used mostly in the [desktop](#) computers -- it is the most widely used architecture, used in [Intel](#) and [AMD CPUs](#).

It is a [CISC](#) architecture and boy, complex it is. **LMAO** there are instructions like **PCLMULQDQ**, **MPSADBW** (*multiple packed sums of absolute difference* which does something like cross correlation ??? xD) and **PCMPESTRI** (which does like many possible string searches/comparisons on strings of different data types like subset or substring with many different options). Basically if you smash your keyboard chances are you produce a valid x86 instruction xD

---

xd

## xD

---

xonotic

## Xonotic

Xonotic was a [free as in freedom](#) fast multiplayer arena [first-person-shooter game](#) similar to e.g. [Quake](#). It ran on [GNU/Linux](#), [Winshit](#), [BSD](#) and other systems. It was one of the best libre games, i.e. games completely free by both code and data/content. It was available under [GPLv3](#). Its gameplay, graphics and customizability were pretty great, it may well have been the best in the AFPS genre, even compared to AAA [proprietary](#) games -- this kind of quality is very rare among libre/noncommercial games.

UPDATE: **Xonotic as a game died in summer 2023** when the retarded developers couldn't get an erection without CONSOOMING NEW CONTINT and started just blindly pushing bugs and balance breaking changes without listening to player complaints at all (and actually banning many for voicing criticism, including [drummyfish](#)), demonstrating [update culture](#) at its worst. The main server became a meme overnight, all [fun](#)

(such as friendly push) was removed from the game to create a "safe space" for players, possibly to get the game ready for Steam sales or make it more open to noobs and advertisers. RIP. If you can fork Xonotic and restore it to its previous glory, please do so.

{ LOL the main server now banned me for voicing criticism of these updates xD RIP, I guess that's it for me. It was a nice journey. ~drummyfish }

{ I used to play Xonotic for years, it was really an excellent game. I've met many nice people there as the players are often programmers and people looking for FOSS. The gameplay was quite addictive and relaxing and you could have a great chat during the game. Of course it's kind of bloated but Xonotic was in a way a masterpiece. ~drummyfish }

The game builds on old ideas and mechanics but adds new weapons, mechanics, ideas and modes -- apart from the traditional deathmatch, team deathmatch, capture the flag, complete the stage (defrag, racing without shooting) and competitive mode (duel), there are a number of new fun modes such as clan arena (team round-based mode without self-damage and items), freeze tag, key hunt, last man standing and even Nexball -- football in Xonotic!

Xonotic was forked from a game called Nexuiz after a trademark controversy (basically in 2010 the guy who started the project and abandoned it later, an ass called Lee Vermeulen, came back and secretly sold the trademark to some shit company named Illfonic). Nexuiz itself was created on top of liberated Quake 1 engine, so Xonotic still bears a lot Quake's legacy, however it masterfully expands on its core principles and makes the gameplay even better. For example rockets shot by rocket launcher can be guided with mouse while holding down the left button which adds a new skill element. New types of weapons were added to the classic AFPS weapons (e.g. the infamous electro, a meme spamming weapon used by noobs for its forgiveness of lack of skill). Movement physics was also modified to give better air control and faster movement as a result.

Fun fact: Xonotic even briefly appeared on a TV show: <https://yewtu.be/search/watch?v=a0TFejn95Sw>.

The game's modified Quake 1 (YES, 1) engine is called Darkplaces. It can be highly customized and modded. Just like in other Quake engine games, there are many console commands (e.g. cvars) to alter almost anything about the game. Advanced programming can be done using QuakeC. Maps can be created e.g. with netradiant.

Though compared to any mainstream modern games Xonotic is quite nicely written (e.g. runs very fast, doesn't have billions of dependencies and despite not being 1.0 yet has fewer bugs than today's AAA games at release), from a more strict point of view it's still very bloated and it's known to contain some shitcode -- for example the engine has frame dependent physics (TODO: cite a specific line in code), uses floating point to represent time, great part of the game is written in a joke language called QuakeC, its net code is worse than e.g. that of OpenArena and some people complain about input lag and other bugs. It could definitely be written MUCH better, but as already mentioned it's still a million times better than any new game.

{ The game runs extremely smooth for me even on old PC, I have no input lag. When my Internet connection gets bad I am sometimes unable to play Xonotic but still able to play OpenArena, which says something about the net code, but that happens very rarely. Also on pretty rare occasions I notice bugs such as imperfect culling of players or even projectiles just hanging mid air during whole game (which happens after heavy packet loss) etc., but nothing that would really be so frequent as to bother me. ~drummyfish }

Xonotic is similar to other libre AFPS games such as OpenArena and Red Eclipse. Of these Xonotic clearly looks the most professional, it has the best "graphics" in the modern sense but still offers the option to turn all the fanciness off. OpenArena, based on Quake 3 engine, is simpler, both technologically and in gameplay, and its movement is slower and with different physics. While OpenArena just basically clones Quake 3, Xonotic is more of an actually new and original game with new ideas and style. Similar thing could be said about Red Eclipse, however it's not as polished and shows some infection with SIW poison.

{ OpenArena is great too. ~drummyfish }

As of 2022 the game has a small but pretty active community of regular players, centered mostly in Europe, though there is some US scene too. There are regulars playing every day, pros, noobs, famous spammers,

campers and trolls. Nice conversations can be had during games. There are memes and inside jokes. The community is pretty neat. Xonotic also has a very dedicated defrag ("racing with no shooting") community. There have also been a few small tournament with real cash prizes in Xonotic.

The GOAT of Xonotic is probably *Dodger*, his skill was just too high even above other pros. The worst player in Xonotic and probably also the biggest idiot on the planet is a player named *111*.

Great news is the development and main servers have so far not been infected by the SJW poison and (as of 2023) **allow a great amount of free speech** -- another rarity. Even the game itself contains speech that SJWs would consider "offensive", there are e.g. voice lines calling other players "pussies" and "retards". This is great.

## Tips'N'Tricks

Here are some pro tips to git gud, impress your frens and generally have more fun.

- Read the legendary **beginner guide** at <https://xonotic.org/guide/>.
- Point'n'click enemies to win.
- **Do NOT fucking spam electro** -- this will make you instantly hated, and rightfully so. Not only is it lame, annoying, makes you easy to kill (opponent can explode your electro balls), enforces more use of electro (electro is countered with electro) and makes the game lag, it can also hurt your teammates, even in no self/team damage modes (as opponents may explode the electro you spray around). Use it appropriately.
- **Learn the weapons and make binds for each one**: learn what every weapons does (good mode for this might be clan arena in which you have all the weapons from the start), bind each one around the movement keys -- it's essential you can switch weapons quickly as the game is designed to reward comboing (the most basic combo is devastator/vortex/mortar). **You HAVE TO combo** to deal good damage. Also note that almost every weapon has a **secondary fire**, for example crylink's primary fire just shoots bouncing balls while the secondary fire shoots a suction ball that can toss enemies out of the map or can be used to accelerate yourself. With devastator (rocket launcher) holding left mouse button allows **guiding the rocket**, right mouse button explodes the rocket in air -- learn this! Then you can easily kill enemies behind corners or flying in air.
- **Learn the movement**: movement is essential in Xonotic, firstly you should learn extremely basic techniques like bunnyhopping, strafe jumping and in-air controls, and secondly you should learn using weapons for movement (blaster jumps, rocket jumps, crylinkg for acceleration and climbing walls, ...). This is important not only for quick relocation, taking shortcuts, dodging enemy fire and quickly running away from enemies, it also saves you from countless deaths by falling into pits.
- **Console**: console is extremely useful in Xonotic; not only can it modify almost any aspect of the game, it also allows you to create fun macros (some people even create what could be considered cheats only with the console commands). A very important command is `search x` which will search for other commands and cvars, e.g. if you want to mess with resolution, do `search resolution` etc. There is even a small **stack-based minilanguage** in the console that's invoked with the `rpn` command -- this allows for very advanced stuff. Sadly this isn't well documented, but the thread at <https://forums.xonotic.org/showthread.php?tid=2987> provides some basics.
- **Make cool binds**, for example you should modify the standard team messages to something more funny. One of the most basic binds is **taunt** so that you can insult players (e.g. `bind KP_SLASH "cmd voice taunt"`).
- **Lower your mouse sensitivity**, this is really crucial to git gud, it's almost impossible to be good with high sens. Lower sensitivity increases your accuracy greatly, which is just key, it's almost like an instant cheat; of course you'll need a bit bigger mouse pad. Write down what sensitivity you have as **centimeters per 360 turn** (measure this carefully with a ruler), this will help you achieve exact same sensitivity when you buy a different mouse. Something like 22 cm per 360 turn is probably good. It is good to set your sensitivity as early as possible so you don't have to later relearn you muscle memory.
- **Make basic graphics settings**, for example increase FOV (field of view) to at least 100, disable effects such as zoom animation (so that you can zoom quickly) etc. If you want to play seriously you should also **turn off music** so that you can hear enemies better. You can also downgrade the graphics so as to make it basically look like Quake 1, either for a cool retro look or to play the game on a potato.



- **Don't fall for the F11 troll** -- when you ask how to do something and people respond with "press F11", don't do it, you'll humiliate yourself. { Faggot devs disabled it now I think. RIP fun. ~drummyfish }
- **Enable cheats with F11.**
- **Spin to win:** you can make a cool spinning macro that makes you look like you have a seizure: e.g. `bind b "toggle vid_stick_mouse; rpn /m_pitch 1 ${vid_stick_mouse} - 0.022 * =;".`
- Once you have a comfy config, you can remove the write rights to it (`chmod -w ~/.xonotic/data/config.cfg`). This way the game won't be able to modify the config and all settings and experiments you make in the game will only be temporary, until the game restart, so you can't fuck up your settings.
- **Do NOT camp**, it's retarded and wastes other people's time, especially e.g. in clan arena where dead people have to wait for the end of round. Better lose quickly than win or draw by camping.
- **Explosions (and bullets) go through thin walls** by default, so you can e.g. kill someone hiding in the room above you by shooting the ceiling. Very cool.
- **Don't stand at teleport exits**, so as to not get telefragged.
- A quick way to tell how good someone is is the damage dealt to damage taken ratio -- once you deal more damage than take, you cease to be a complete noob. When you deal twice as much damage as you take, you are probably reaching a pro level. Of course, this also depends on opponents, mode you play etc., take this just as a quick estimate.
- If you're not doing well, complain about lag.
- **You can turn off the chat** in case there is some retarded conversation going on (just go to setting and HUD editor). New versions also support client side ignore of specific players.
- In addition to chat you can also turn off opponent names if you just want to play without any distraction. Try e.g. `hud_panel_chat 0; hud_panel_centerprint 0; hud_shownames 0; hud_panel_scoreboard_maxheight 0.01; hud_panel_notify 0; hud_panel_scoreboard_namesize 1; hud_panel_score 0; con_chatsound 0; hud_panel_infomessages 0;`
- **Slap opponents to humiliate them:** the shotgun secondary fire is a melee slap, which is the best way to kill someone, partly because it's pretty hard to do (there's a delay before the bash). Another humiliating way to eliminate opponents is to kill them with Hagar secondary or push them off the map with blaster, crylink or even Mario kill (very rare -- you jump on someone's head while he's jumping over a pit which makes him fall down). Don't forget to taunt and teabag.
- If you see someone running fast while crouching in air, he's probably a pro.
- **For more frags listen to sounds enemies make** -- if a player is low on health, he makes a distinct sound when hurt. You should listen to these sounds and just quickly hit these players e.g. with machine gun to get an easy kill.
- **For more frags watch for falling players** -- on space maps watch out for players falling off the map and just quickly hit them with machine gun (secondary fire) before they die, the game will give frag to whoever hit him last.
- `v_psycho 1, r_trippy 1` etc.
- **Slow down enemies with crylink secondary:** crylink secondary fire shoots a projectile that "sucks in" anyone in near proximity to the hit -- besides exploiting this for accelerating yourself you can also use it to slow down enemies that try to run away from a fight or who just try to move quickly to avoid being hit. This is extremely annoying to the affected player, so do it as often as you can.
- `cl_handicap` can be used to make yourself weaker by specified multiplier (take more and deal less damage), either to balance the game or provide a challenge.
- `v_flipped 1` flips the rendering so that you see the mirror version of the map, it basically gives you a new map for free and provides a nice challenge.
- `in_pitch_min` and `in_pitch_max` can remove the limit of looking up and down (set to e.g. -1000 and 1000) -- this way you can turn completely upside down which can be seen by spectators and make them think you're a cheater.
- Typing `/me` in chat makes the chat message be output in a different format; instead of `username: message` it writes `*message` and replaces the `/me` string with your username -- this can create cool and fancy messages.
- **Piss people off with troll votes:** study the `vcall` console command (see `vhelP`), it allows for calling votes on specific things like switching maps, kicking players etc. Some votes allow a parameter string so that you can make a funny message with it, e.g. `vcall bots play better than player X`. Due to weird code numbers can be specified with almost arbitrary number of leading zeros, so you can just call a vote to set `fraglimit` to 1 with 100 leading zeros to spam people's screen and confuse them. When enabled, `vcall grunt` will try to turn on grunt sound that many players hate. If you want to

- The game outputs chat into terminal, you can filter these messages out and redirect them to espeak for a fun experience :)
- Electro spammers are easy to kill:** when someone is raping electro and just spraying everything, simply shoot at him with electro primary fire -- this explodes the balls he's spamming and hits him AND his teammates with huge damage. Don't forget to teabag and taunt him once he's dead.
- Standing in front of a nub with Hagar is dangerous.
- Switch left and right hand for a challenge.
- Play drunk and/or high for a challenge.
- Change your name to some unicode monstrosity like  
^6.ḷı́ l̡½l̢ ì í î ï ĭ İ ł ŀ ľ Ľ ħ ^5.ḷı́ l̡½l̢ ì ï î ï ĩ İ ł ŀ ľ Ľ ħ ~drummyfish }

{ RIP if your reader can't Unicode well :) ~drummyfish }

- Doom challenge: disable looking up and down with console.
- If you're slow, strafe harder.
- Move to Europe to actually enjoy the game with good ping.
- { My config is at  
[https://codeberg.org/drummyfish/my\\_text\\_data/src/branch/master/configs/xonotic\\_config.cfg](https://codeberg.org/drummyfish/my_text_data/src/branch/master/configs/xonotic_config.cfg).  
~drummyfish }
- Buy premium account for extra skins and lootboxes. Just kidding :)
- Don't ragequit, you'll give your opponent the ultimate satisfaction. Making someone ragequit is better than just raping him.
- Bully players that admit to be playing on Window\$.
- There are minigames in the game menu, can serve to kill time while waiting to respawn in some modes.
- If you want to win in a team game and your team is losing, just quickly switch to the winning team before the end. Gotta keep them xonstats high.
- Sound is important, firstly you can hear where enemies are (especially when you turn off music) and secondly you can also spot low HP players by sound (they make a specific hurt sound when low HP).
- Just spec good players and do what they do.
- All americans in the game are cheaters without exception, just like in real life. You spot americans by ping and high score despite playing like pre beginners when you spectate them, and by crying like babies when you criticize capitalism and tech consumerism.
- some **tactics/strategy/habits**:

- ## Xonotic

small hit that's enough to finish someone. But watch out, some people bitch about using MG this way being too nub, going for slap will entertain the audience better.

- ♦ **Take items**, many new players just walk by mega health without taking it, that's pretty stupid.
- ♦ It's sometimes good in a fight to be directly above or under the opponent as it's harder for him to aim precisely looking completely up or down, and with many weapons it's generally harder to just hit someone in air, especially for noobs. Noobs mostly don't even see you if you're above their heads (they have FOV set to low), it's basically free damage.

## See Also

- [OpenArena](#)
- 

xxiivv

## XXIIVV

{ Still researching this shit etc. ~drummyfish }

XXIIVV is a proprietary soynet snobbish website and personal wiki (in its concept similar to our wiki) of a Canadian narcissist minimalist/esoteric programmer/"artist"/generalist David Mondou-Labbe who calls himself "Devine Lu Linvega" (lmao) who is a part of a highly cringe fascist artist/programmer group called Hundred Rabbits (100r) who live on a small boat or something. David seems to be a normie SJW fascist, proclaiming "aggressivity" on his web (under "/ethics.html" on his site). The site is accessible at <http://wiki.xxiivv.com/site/home.html>. There are some quite good and pretty bad things about it.

{ Holy shit his webring is cringe and toxic as fuck. One huge gay nazi wannabe "artist" circlejerk. It's like a small village worth of the kind of psychopaths who draw cute childish drawings of tiny animals with small hearts and love all around while at the same time advocating live castration of anyone who dislikes them. ~drummyfish }

Firstly let's see the letdowns, which greatly prevail: the site is proprietary and **he licenses his "art" and some of his code under the proprietary CC-BY-NC-SA, RETARD ALERT**. This bro is just lacking some chromosomes. He's a capitalist open soars fanboy trying to monopolize art by keeping exclusive "commercial intellectual property rights", as if it had any commercial value lol. At least some of his code is MIT, but he also makes fucking **PROPRIETARY PAID software** (e.g. Verreciel), then he somehow tries to manipulate readers of his website to believe he is "against capitalism" :D (Or is he not? I dunno. Definitely seems to be riding the eco wave.) The guy also seems **egoistic as fuck**, invents weird hipster names and "personal pronouns", has some ugly body disfigurements, wears cringe rabbit costumes, he thinks his art is so good he has to "protect" it with capitalist licenses and writes in a super cringe snobbish/pompous/cryptic/poetic tryhard style probably in an attempt to appear smart while just making it shithard to make sense of his texts -- truly his tech writings are literal torture to read. The only thing he's missing is a fedora. Anyway, that's just a quick sum up of the cancer.

There are also nice things though, a few of them being:

- The guy is creating extremely minimalist, small tech from-scratch technology that's worthy of attention. Some of it includes:
  - ♦ uxn: Simple (~100 LOC of C) virtual machine, similar to a "fantasy console" but intended more for portability. This also comes with an assembly language called tal.
  - ♦ lietal: Simple artificial language.
- The wiki writes on pretty interesting topics, many of which overlap with our topics of interest. For example pen and paper computing that includes games.
- Some of the presented opinions and wisdoms are based, e.g. "for writing fast programs use slow computers" etc.

## See Also

- [100r](#)
  - [uxn](#)
  - [permacomputing wiki](#)
- 

yes\_they\_can

## Yes They Can

*"The bigger the lie, the more it will be believed."* --[Joseph Goebbels](#), NSDAP minister of propaganda

If you think they can't do something, you are wrong; unless it is directly violating a law of physics, they can do it. For example you may think "haha they can't start selling air, people would revolt", "hahaaa, they can't make people believe 1 + 1 equals 2000, it's too obvious of a lie" or "hahaaa they can't lie about history when there is a ton of direct evidence for the contrary freely accessible on the internet, they can't censor something that's all over the internet and in billions of books" -- yes, they can do all of this. With enough capitalism you can make people believe a circle to be square, they have already made people desire their everyday torture, they made people believe colors and offensive statistics are just culturally constructed hallucinations, feminists have already made people widely believe a woman can beat an adult man -- a naive man of the past would likely believe this to be impossible as well. Well, we see under capitalism it is quite possible.

Resisting overlords is always futile in the end, the only hope is to establish society without overlords. You think "hahaha, if we create this super encrypted/decentralized computer network, we can simply communicate and they can do nothing about it, BAZINGA" -- well, no you can't. How can they stop this? They will simply ban computers you idiot, in fact you have only given them the reason to. You say "hahaha but I can have this calculator in my basement hidden" -- well, how many people will participate in your network if revealing such participation is punished not only by death sentence, but death sentence for you whole family; if even people who know about you participating in the network and not reporting you face the same punishment (already the case in some pseudocommunist countries)? If in addition people have no free time, if they don't have electricity at home, no will to live and there are also government signal jammers everywhere just in case? Enjoy your guerrilla resistance network with three people. You say "bbbb...but that cant happen ppl would revolt" -- NO. Have you seen chicken at chicken farm revolt? (Except in that one movie lol). "BBBb...BUT... people are not chicken". NO. People are literally physically chicken (to a stupid argument you get stupid counterargument).

---

youtube

## YouTube

YouTube (also JewTube { Lol jewtube.com actually exists. ~drummyfish }) is a huge, censored proprietary capitalist video consuming "website"/platform, since 2006 seized by the Google terrorist organization. It has become the monopoly "video content platform", everyone uploads his videos there and so everyone is forced to use that shitty site from time to time to view some tutorial or whatnot. YouTube is based on content consumerism, aggressive predatory marketing, copyright trolling, propaganda and general abuse of its uses -- it is financed from surveillance-powered ads as well as sponsor propaganda inserted into videos. Alternatives to YouTube, such as bitchute, the "rightist" YouTube, never really caught on very much -- YouTube is sadly synonymous with online videos just as Google is synonymous with searching the web. This is of course extremely, extremely, extremely, extremely bad.

{ <https://www.vidlii.com> seems alright though, at least as a curiosity. Anyway if you need to watch YouTube, do not use their website, it's shitty as hell and you will die of ad cancer, rather use something like invidious or youtube-dl. Here is an **awesome hack I discovered to search only old videos on youtube!** The new shit is just unwatchable, there's clickbait, sponsors, propaganda, SJW shit everywhere, thankfully you can just exclude any year from the search with with "-year" (at least for now), for example:  
[https://yewtu.be/search?q=free+software+-2023+-2022+-2021+-2020+-2019+-2018+-2017+-2016+-2015+-2014+-2013+-2012+-2011+-2010+-2009+-2008+-2007+-2006+-2005+-2004+-2003+-2002+-2001+-2000+-1999+-1998+-1997+-1996+-1995+-1994+-1993+-1992+-1991+-1990+-1889+-1888+-1887+-1886+-1885+-1884+-1883+-1882+-1881+-1880+-1879+-1878+-1877+-1876+-1875+-1874+-1873+-1872+-1871+-1870+-1869+-1868+-1867+-1866+-1865+-1864+-1863+-1862+-1861+-1860+-1859+-1858+-1857+-1856+-1855+-1854+-1853+-1852+-1851+-1850+-1849+-1848+-1847+-1846+-1845+-1844+-1843+-1842+-1841+-1840+-1839+-1838+-1837+-1836+-1835+-1834+-1833+-1832+-1831+-1830+-1829+-1828+-1827+-1826+-1825+-1824+-1823+-1822+-1821+-1820+-1819+-1818+-1817+-1816+-1815+-1814+-1813+-1812+-1811+-1810+-1809+-1808+-1807+-1806+-1805+-1804+-1803+-1802+-1801+-1800+-1799+-1798+-1797+-1796+-1795+-1794+-1793+-1792+-1791+-1790+-1789+-1788+-1787+-1786+-1785+-1784+-1783+-1782+-1781+-1780+-1779+-1778+-1777+-1776+-1775+-1774+-1773+-1772+-1771+-1770+-1769+-1768+-1767+-1766+-1765+-1764+-1763+-1762+-1761+-1760+-1759+-1758+-1757+-1756+-1755+-1754+-1753+-1752+-1751+-1750+-1749+-1748+-1747+-1746+-1745+-1744+-1743+-1742+-1741+-1740+-1739+-1738+-1737+-1736+-1735+-1734+-1733+-1732+-1731+-1730+-1729+-1728+-1727+-1726+-1725+-1724+-1723+-1722+-1721+-1720+-1719+-1718+-1717+-1716+-1715+-1714+-1713+-1712+-1711+-1710+-1709+-1708+-1707+-1706+-1705+-1704+-1703+-1702+-1701+-1700+-1699+-1698+-1697+-1696+-1695+-1694+-1693+-1692+-1691+-1690+-1689+-1688+-1687+-1686+-1685+-1684+-1683+-1682+-1681+-1680+-1679+-1678+-1677+-1676+-1675+-1674+-1673+-1672+-1671+-1670+-1669+-1668+-1667+-1666+-1665+-1664+-1663+-1662+-1661+-1660+-1659+-1658+-1657+-1656+-1655+-1654+-1653+-1652+-1651+-1650+-1649+-1648+-1647+-1646+-1645+-1644+-1643+-1642+-1641+-1640+-1639+-1638+-1637+-1636+-1635+-1634+-1633+-1632+-1631+-1630+-1629+-1628+-1627+-1626+-1625+-1624+-1623+-1622+-1621+-1620+-1619+-1618+-1617+-1616+-1615+-1614+-1613+-1612+-1611+-1610+-1609+-1608+-1607+-1606+-1605+-1604+-1603+-1602+-1601+-1600+-1599+-1598+-1597+-1596+-1595+-1594+-1593+-1592+-1591+-1590+-1589+-1588+-1587+-1586+-1585+-1584+-1583+-1582+-1581+-1580+-1579+-1578+-1577+-1576+-1575+-1574+-1573+-1572+-1571+-1570+-1569+-1568+-1567+-1566+-1565+-1564+-1563+-1562+-1561+-1560+-1559+-1558+-1557+-1556+-1555+-1554+-1553+-1552+-1551+-1550+-1549+-1548+-1547+-1546+-1545+-1544+-1543+-1542+-1541+-1540+-1539+-1538+-1537+-1536+-1535+-1534+-1533+-1532+-1531+-1530+-1529+-1528+-1527+-1526+-1525+-1524+-1523+-1522+-1521+-1520+-1519+-1518+-1517+-](https://yewtu.be/search?q=free+software+-2023+-2022+-2021+-2020+-2019+-2018+-2017+-2016+-2015+-2014+-2013+-2012+-2011+-2010+-2009+-2008+-2007+-2006+-2005+-2004+-2003+-2002+-2001+-2000+-1999+-1998+-1997+-1996+-1995+-1994+-1993+-1992+-1991+-1990+-1989+-1988+-1987+-1986+-1985+-1984+-1983+-1982+-1981+-1980+-1979+-1978+-1977+-1976+-1975+-1974+-1973+-1972+-1971+-1970+-1969+-1968+-1967+-1966+-1965+-1964+-1963+-1962+-1961+-1960+-1959+-1958+-1957+-1956+-1955+-1954+-1953+-1952+-1951+-1950+-1949+-1948+-1947+-1946+-1945+-1944+-1943+-1942+-1941+-1940+-1939+-1938+-1937+-1936+-1935+-1934+-1933+-1932+-1931+-1930+-1929+-1928+-1927+-1926+-1925+-1924+-1923+-1922+-1921+-1920+-1919+-1918+-1917+-1916+-1915+-1914+-1913+-1912+-1911+-1910+-1909+-1908+-1907+-1906+-1905+-1904+-1903+-1902+-1901+-1900+-1899+-1898+-1897+-1896+-1895+-1894+-1893+-1892+-1891+-1890+-1889+-1888+-1887+-1886+-1885+-1884+-1883+-1882+-1881+-1880+-1879+-1878+-1877+-1876+-1875+-1874+-1873+-1872+-1871+-1870+-1869+-1868+-1867+-1866+-1865+-1864+-1863+-1862+-1861+-1860+-1859+-1858+-1857+-1856+-1855+-1854+-1853+-1852+-1851+-1850+-1849+-1848+-1847+-1846+-1845+-1844+-1843+-1842+-1841+-1840+-1839+-1838+-1837+-1836+-1835+-1834+-1833+-1832+-1831+-1830+-1829+-1828+-1827+-1826+-1825+-1824+-1823+-1822+-1821+-1820+-1819+-1818+-1817+-1816+-1815+-1814+-1813+-1812+-1811+-1810+-1809+-1808+-1807+-1806+-1805+-1804+-1803+-1802+-1801+-1800+-1799+-1798+-1797+-1796+-1795+-1794+-1793+-1792+-1791+-1790+-1789+-1788+-1787+-1786+-1785+-1784+-1783+-1782+-1781+-1780+-1779+-1778+-1777+-1776+-1775+-1774+-1773+-1772+-1771+-1770+-1769+-1768+-1767+-1766+-1765+-1764+-1763+-1762+-1761+-1760+-1759+-1758+-1757+-1756+-1755+-1754+-1753+-1752+-1751+-1750+-1749+-1748+-1747+-1746+-1745+-1744+-1743+-1742+-1741+-1740+-1739+-1738+-1737+-1736+-1735+-1734+-1733+-1732+-1731+-1730+-1729+-1728+-1727+-1726+-1725+-1724+-1723+-1722+-1721+-1720+-1719+-1718+-1717+-1716+-1715+-1714+-1713+-1712+-1711+-1710+-1709+-1708+-1707+-1706+-1705+-1704+-1703+-1702+-1701+-1700+-1699+-1698+-1697+-1696+-1695+-1694+-1693+-1692+-1691+-1690+-1689+-1688+-1687+-1686+-1685+-1684+-1683+-1682+-1681+-1680+-1679+-1678+-1677+-1676+-1675+-1674+-1673+-1672+-1671+-1670+-1669+-1668+-1667+-1666+-1665+-1664+-1663+-1662+-1661+-1660+-1659+-1658+-1657+-1656+-1655+-1654+-1653+-1652+-1651+-1650+-1649+-1648+-1647+-1646+-1645+-1644+-1643+-1642+-1641+-1640+-1639+-1638+-1637+-1636+-1635+-1634+-1633+-1632+-1631+-1630+-1629+-1628+-1627+-1626+-1625+-1624+-1623+-1622+-1621+-1620+-1619+-1618+-1617+-1616+-1615+-1614+-1613+-1612+-1611+-1610+-1609+-1608+-1607+-1606+-1605+-1604+-1603+-1602+-1601+-1600+-1599+-1598+-1597+-1596+-1595+-1594+-1593+-1592+-1591+-1590+-1589+-1588+-1587+-1586+-1585+-1584+-1583+-1582+-1581+-1580+-1579+-1578+-1577+-1576+-1575+-1574+-1573+-1572+-1571+-1570+-1569+-1568+-1567+-1566+-1565+-1564+-1563+-1562+-1561+-1560+-1559+-1558+-1557+-1556+-1555+-1554+-1553+-1552+-1551+-1550+-1549+-1548+-1547+-1546+-1545+-1544+-1543+-1542+-1541+-1540+-1539+-1538+-1537+-1536+-1535+-1534+-1533+-1532+-1531+-1530+-1529+-1528+-1527+-1526+-1525+-1524+-1523+-1522+-1521+-1520+-1519+-1518+-1517+-1516+-1515+-1514+-1513+-1512+-1511+-1510+-1509+-1508+-1507+-1506+-1505+-1504+-1503+-1502+-1501+-1500+-1499+-1498+-1497+-1496+-1495+-1494+-1493+-1492+-1491+-1490+-1489+-1488+-1487+-1486+-1485+-1484+-1483+-1482+-1481+-1480+-1479+-1478+-1477+-1476+-1475+-1474+-1473+-1472+-1471+-1470+-1469+-1468+-1467+-1466+-1465+-1464+-1463+-1462+-1461+-1460+-1459+-1458+-1457+-1456+-1455+-1454+-1453+-1452+-1451+-1450+-1449+-1448+-1447+-1446+-1445+-1444+-1443+-1442+-1441+-1440+-1439+-1438+-1437+-1436+-1435+-1434+-1433+-1432+-1431+-1430+-1429+-1428+-1427+-1426+-1425+-1424+-1423+-1422+-1421+-1420+-1419+-1418+-1417+-1416+-1415+-1414+-1413+-1412+-1411+-1410+-1409+-1408+-1407+-1406+-1405+-1404+-1403+-1402+-1401+-1400+-1399+-1398+-1397+-1396+-1395+-1394+-1393+-1392+-1391+-1390+-1389+-1388+-1387+-1386+-1385+-1384+-1383+-1382+-1381+-1380+-1379+-1378+-1377+-1376+-1375+-1374+-1373+-1372+-1371+-1370+-1369+-1368+-1367+-1366+-1365+-1364+-1363+-1362+-1361+-1360+-1359+-1358+-1357+-1356+-1355+-1354+-1353+-1352+-1351+-1350+-1349+-1348+-1347+-1346+-1345+-1344+-1343+-1342+-1341+-1340+-1339+-1338+-1337+-1336+-1335+-1334+-1333+-1332+-1331+-1330+-1329+-1328+-1327+-1326+-1325+-1324+-1323+-1322+-1321+-1320+-1319+-1318+-1317+-1316+-1315+-1314+-1313+-1312+-1311+-1310+-1309+-1308+-1307+-1306+-1305+-1304+-1303+-1302+-1301+-1300+-1299+-1298+-1297+-1296+-1295+-1294+-1293+-1292+-1291+-1290+-1289+-1288+-1287+-1286+-1285+-1284+-1283+-1282+-1281+-1280+-1279+-1278+-1277+-1276+-1275+-1274+-1273+-1272+-1271+-1270+-1269+-1268+-1267+-1266+-1265+-1264+-1263+-1262+-1261+-1260+-1259+-1258+-1257+-1256+-1255+-1254+-1253+-1252+-1251+-1250+-1249+-1248+-1247+-1246+-1245+-1244+-1243+-1242+-1241+-1240+-1239+-1238+-1237+-1236+-1235+-1234+-1233+-1232+-1231+-1230+-1229+-1228+-1227+-1226+-1225+-1224+-1223+-1222+-1221+-1220+-1219+-1218+-1217+-1216+-1215+-1214+-1213+-1212+-1211+-1210+-1209+-1208+-1207+-1206+-1205+-1204+-1203+-1202+-1201+-1200+-1199+-1198+-1197+-1196+-1195+-1194+-1193+-1192+-1191+-1190+-1189+-1188+-1187+-1186+-1185+-1184+-1183+-1182+-1181+-1180+-1179+-1178+-1177+-1176+-1175+-1174+-1173+-1172+-1171+-1170+-1169+-1168+-1167+-1166+-1165+-1164+-1163+-1162+-1161+-1160+-1159+-1158+-1157+-1156+-1155+-1154+-1153+-1152+-1151+-1150+-1149+-1148+-1147+-1146+-1145+-1144+-1143+-1142+-1141+-1140+-1139+-1138+-1137+-1136+-1135+-1134+-1133+-1132+-1131+-1130+-1129+-1128+-1127+-1126+-1125+-1124+-1123+-1122+-1121+-1120+-1119+-1118+-1117+-1116+-1115+-1114+-1113+-1112+-1111+-1110+-1109+-1108+-1107+-1106+-1105+-1104+-1103+-1102+-1101+-1100+-1099+-1098+-1097+-1096+-1095+-1094+-1093+-1092+-1091+-1090+-1089+-1088+-1087+-1086+-1085+-1084+-1083+-1082+-1081+-1080+-1079+-1078+-1077+-1076+-1075+-1074+-1073+-1072+-1071+-1070+-1069+-1068+-1067+-1066+-1065+-1064+-1063+-1062+-1061+-1060+-1059+-1058+-1057+-1056+-1055+-1054+-1053+-1052+-1051+-1050+-1049+-1048+-1047+-1046+-1045+-1044+-1043+-1042+-1041+-1040+-1039+-1038+-1037+-1036+-1035+-1034+-1033+-1032+-1031+-1030+-1029+-1028+-1027+-1026+-1025+-1024+-1023+-1022+-1021+-1020+-1019+-1018+-1017+-1016+-1015+-1014+-1013+-1012+-1011+-1010+-1009+-1008+-1007+-1006+-1005+-1004+-1003+-1002+-1001+-1000+-999+-998+-997+-996+-995+-994+-993+-992+-991+-990+-989+-988+-987+-986+-985+-984+-983+-982+-981+-980+-979+-978+-977+-976+-975+-974+-973+-972+-971+-970+-969+-968+-967+-966+-965+-964+-963+-962+-961+-960+-959+-958+-957+-956+-955+-954+-953+-952+-951+-950+-949+-948+-947+-946+-945+-944+-943+-942+-941+-940+-939+-938+-937+-936+-935+-934+-933+-932+-931+-930+-929+-928+-927+-926+-925+-924+-923+-922+-921+-920+-919+-918+-917+-916+-915+-914+-913+-912+-911+-910+-909+-908+-907+-906+-905+-904+-903+-902+-901+-900+-899+-898+-897+-896+-895+-894+-893+-892+-891+-890+-889+-888+-887+-886+-885+-884+-883+-882+-881+-880+-879+-878+-877+-876+-875+-874+-873+-872+-871+-870+-869+-868+-867+-866+-865+-864+-863+-862+-861+-860+-859+-858+-857+-856+-855+-854+-853+-852+-851+-850+-849+-848+-847+-846+-845+-844+-843+-842+-841+-840+-839+-838+-837+-836+-835+-834+-833+-832+-831+-830+-829+-828+-827+-826+-825+-824+-823+-822+-821+-820+-819+-818+-817+-816+-815+-814+-813+-812+-811+-810+-809+-808+-807+-806+-805+-804+-803+-802+-801+-800+-799+-798+-797+-796+-795+-794+-793+-792+-791+-790+-789+-788+-787+-786+-785+-784+-783+-782+-781+-780+-779+-778+-777+-776+-775+-774+-773+-772+-771+-770+-769+-768+-767+-766+-765+-764+-763+-762+-761+-760+-759+-758+-757+-756+-755+-754+-753+-752+-751+-750+-749+-748+-747+-746+-745+-744+-743+-742+-741+-740+-739+-738+-737+-736+-735+-734+-733+-732+-731+-730+-729+-728+-727+-726+-725+-724+-723+-722+-721+-720+-719+-718+-717+-716+-715+-714+-713+-712+-711+-710+-709+-708+-707+-706+-705+-704+-703+-702+-701+-700+-699+-698+-697+-696+-695+-694+-693+-692+-691+-690+-689+-688+-687+-686+-685+-684+-683+-682+-681+-680+-679+-678+-677+-676+-675+-674+-673+-672+-671+-670+-669+-668+-667+-666+-665+-664+-663+-662+-661+-660+-659+-658+-657+-656+-655+-654+-653+-652+-651+-650+-649+-648+-647+-646+-645+-644+-643+-642+-641+-640+-639+-638+-637+-636+-635+-634+-633+-632+-631+-630+-629+-628+-627+-626+-625+-624+-623+-622+-621+-620+-619+-618+-617+-616+-615+-614+-613+-612+-611+-610+-609+-608+-607+-606+-605+-604+-603+-602+-601+-600+-599+-598+-597+-596+-595+-594+-593+-592+-591+-590+-589+-588+-587+-586+-585+-584+-583+-582+-581+-580+-579+-578+-577+-576+-575+-574+-573+-572+-571+-570+-569+-568+-567+-566+-565+-564+-563+-562+-561+-560+-559+-558+-557+-556+-555+-554+-553+-552+-551+-550+-549+-548+-547+-546+-545+-544+-543+-542+-541+-540+-539+-538+-537+-536+-535+-534+-533+-532+-531+-530+-529+-528+-527+-526+-525+-524+-523+-522+-521+-520+-519+-518+-517+-516+-515+-514+-513+-512+-511+-510+-509+-508+-507+-506+-505+-504+-503+-502+-501+-500+-499+-498+-497+-496+-495+-494+-493+-492+-491+-490+-489+-488+-487+-486+-485+-484+-483+-482+-481+-480+-479+-478+-477+-476+-475+-474+-473+-472+-471+-470+-469+-468+-467+-466+-465+-464+-463+-462+-461+-460+-459+-458+-457+-456+-455+-454+-453+-452+-451+-450+-449+-448+-447+-446+-445+-444+-443+-442+-441+-440+-439+-438+-437+-436+-435+-434+-433+-432+-431+-430+-429+-428+-427+-426+-425+-424+-423+-422+-421+-420+-419+-418+-417+-416+-415+-414+-413+-412+-411+-410+-409+-408+-407+-406+-405+-404+-403+-402+-401+-400+-399+-398+-397+-396+-395+-394+-393+-392+-391+-390+-389+-388+-387+-386+-385+-384+-383+-382+-381+-380+-379+-378+-377+-376+-375+-374+-373+-372+-371+-370+-369+-368+-367+-366+-365+-364+-363+-362+-361+-360+-359+-358+-357+-356+-355+-354+-353+-352+-351+-350+-349+-348+-347+-346+-345+-344+-343+-342+-341+-340+-339+-338+-337+-336+-335+-334+-333+-332+-331+-330+-329+-328+-327+-326+-325+-324+-323+-322+-321+-320+-319+-318+-317+-316+-315+-314+-313+-312+-311+-310+-309+-308+-307+-306+-305+-304+-303+-302+-301+-300+-299+-298+-297+-296+-295+-294+-293+-292+-291+-290+-289+-288+-287+-286+-285+-284+-283+-282+-281+-280+-279+-278+-277+-276+-275+-274+-273+-272+-271+-270+-269+-268+-267+-266+-265+-264+-263+-262+-261+-260+-259+-258+-257+-256+-255+-254+-253+-252+-251+-250+-249+-248+-247+-246+-245+-244+-243+-242+-241+-240+-239+-238+-237+-236+-235+-234+-233+-232+-231+-230+-229+-228+-227+-226+-225+-224+-223+-222+-221+-220+-219+-218+-217+-216+-215+-214+-213+-212+-211+-210+-209+-208+-207+-206+-205+-204+-203+-202+-201+-200+-199+-198+-197+-196+-195+-194+-193+-192+-191+-190+-189+-188+-187+-186+-185+-184+-183+-182+-181+-180+-179+-178+-177+-176+-175+-174+-173+-172+-171+-170+-169+-168+-167+-166+-165+-164+-163+-162+-161+-160+-159+-158+-157+-156+-155+-154+-153+-152+-151+-150+-149+-148+-147+-146+-145+-144+-143+-142+-141+-140+-139+-138+-137+-136+-135+-134+-133+-132+-131+-130+-129+-128+-127+-126+-125+-124+-123+-122+-121+-120+-119+-118+-117+-116+-115+-114+-113+-112+-111+-110+-109+-108+-107+-106+-105+-104+-103+-102+-101+-100+-99+-98+-97+-96+-95+-94+-93+-92+-91+-90+-89+-88+-87+-86+-85+-84+-83+-82+-81+-80+-79+-78+-77+-76+-75+-74+-73+-72+-71+-70+-69+-68+-67+-66+-65+-64+-63+-62+-61+-60+-59+-58+-57+-56+-55+-54+-53+-52+-51+-50+-49+-48+-47+-46+-45+-44+-43+-42+-41+-40+-39+-38+-37+-36+-35+-34+-33+-32+-31+-30+-29+-28+-27+-26+-25+-24+-23+-22+-21+-20+-19+-18+-17+-16+-15+-14+-13+-12+-11+-10+-9+-8+-7+-6+-5+-4+-3+-2+-1)

A FOSS alternative to YouTube is e.g. PeerTube, a federated video platform, however for intended use it requires JavaScript and there are other issues that make it unusable (SIW censorship, videos load extremely slowly, ...). There also exist alternative YouTube frontends (normally also FOSS), e.g. HookTube, Invidious or FreeTube -- these let you access YouTube's videos via less bloated and more privacy-friendly interface. More hardcore people use CLI tools such as youtube-dl to directly download the videos and watch them in native players.

In November 2021 YouTube removed the dislike count on videos so as to make it impossible to express dislike or disagreement with their propaganda as people naturally started to dislike the exponentially skyrocketing shit and immorality of content corporations and SIWs started to force promote on YouTube (such as that infamous Lord of the Rings series with "afro american" dwarves that got like a billion dislikes lmao). In other words capitalism has made it to the stage of banning disagreement when people started to dislike the horse shit they're being force fed. This was met with a wave of universal criticism but of course YouTube told people to shut up and keep consuming that horse excrement -- of course zoomers are just brainless zombies dependent on YouTube like a street whore on heroin so they just accepted that recommendation. Orwell would definitely be happy to see this.

LMAO, as of 2022 YouTube has become the **kingdom of clickbait**. If you're living in the future and haven't seen this, you wouldn't believe how ridiculously fucked up it is, the platform is quite literally UNUSABLE (but it's still consumable and addictive to idiots so it works) -- ALL videos, including (and especially) those of the "serious" YouTubers (like mr. Veritasium), put absolutely misleading and downright made up lying thumbnails and titles, the videos have zero correlation with how they're presented. Additionally the majority of thumbnails has to be occupied by some femoid's breasts, even "educational science videos", so as to force children to click it and masturbate (YouTube can really be seen as a soft porn site now). Everyone has to do it because everyone does it. The YouTube slave shitheads are constantly bothering you and trying to trick you into "audience interaction" because that's what the algorithm pushes them to do with threats of starvation ("I think  $1 + 1 = 3$ , please let me know in the comments if you disagree"). YouTube is not a place to find something useful, it's a place mostly littered by traps set up to exploit you and you can only hope to maybe get something small back for letting yourself be raped (well, basically like in any other for profit place). In other words capitalism is once again working as expected. Yeah and also **ALL videos include sponsored content**, again even the "educational science videos" that children are forced to watch at schools etc. -- this is in addition to "normal" ads that randomly play during the videos.

A typical 2022 YouTube video now looks like this:

- **title:** *THEY SAID A WOMAN COULDN'T CODE MINECRAFT IN 24 MICROSECONDS SO I PROVED THEM WRONG, ALSO LIFE FOUND ON MARS*
- **thumbnail:** bikini MILF, nipples half showing, overly excited face as if having a stroke, pointing at aliens playing Minecraft
- **content:** 10 minutes of pre-video unskippable ads, tranny shows up urging you to buy premium membership on Nord VPN, 10 minutes of unskippable ads for tranny underwear, tranny opens up C# IDE that loads for 30 minutes, clicks the "generate game with AI" button, the computer crashes, "sorry for clickbait", 10 minutes of post video ads, "like, comment, give me money on patreon, subscribe, click the bell button, go to settings and check I want to see subscriptions I subscribe to, go to advanced setting and click I really really really want to see my subscriptions I really do and will suck your dick", next video loads without asking, volume sets itself to 300%, browser close button disappears
- **reactions:** trending video, 10 billion likes, dislikes not shown, comments disabled so that people can't warn others it's a waste of time

YouTube is also a copyright dictatorship, anyone can take down any video containing even the slightest clip from a video he uploaded, even if such use would legally be allowed under fair use and even if that user doesn't have any copyright to enforce (YouTube simply supposes that whoever uploads a video to their site first must have created that video as a whole and holds a godlike power over it), i.e. YouTube is de facto making its own copyright laws which are much more strict than they are in real life (which is hard to imagine but they managed to do it). In other words there is a corporation that makes laws which effectively are basically just like normal laws except they don't even pass any law making process, their evaluation doesn't pass through justice system (courts), and the sole purpose of these laws is to rape people for money that goes solely to pay for YouTube CEO's whores and private jets. A reader in the future probably won't believe it, but there are even people who say that "this is OK" because, quote, I shit you not, ""they're a private

company so they can do whatever they want"". Yes, such arguments have come out of some lifeform's mouth. That probably implies a negative IQ.

---

zen

## Zen

Zen, a term coming from zen Buddhism (the word itself from *dhyana*, meaning *meditation*), means emphasis on mediation that leads to enlightenment; in a wider sense it hints on related things and feelings such as tranquility, spiritual peace, sudden coming to realization and understanding. In hacker speech *zen* is a very frequent term, according to jargon file "to zen" means to understand something by simply meditating about it or by sudden realization (as opposed to e.g. active trial and error).

Wikiwikiweb has a related discussion under *ZenConcepts*.

TODO

## See Also

- tao
  - fu
  - Buddhism
  - hacker culture
  - nirvana
  - guru
- 

zero

## Zero

Zero (0) is a number signifying the absence of a thing we count. Among integers it precedes 1 and follows -1.

Some properties of and facts about this number follow:

- It is even.
- It is neither positive nor negative, it lies exactly on the boundary of positive and negative numbers. However in some computer numeric encodings (such as one's complement) there exist two representations of zero and so we may hear about a positive and negative zero, even though mathematically there is no such a thing.
- It is a whole number, a natural number, a rational number, a real number and a complex number.
- It is **NOT** a prime number.
- It is an additive identity, i.e. adding 0 to anything has no effect. Subtracting 0 from anything also has no effect.
- Multiplying anything by 0 gives 0.
- Its representation in all traditional numeral systems is the same: 0.
- $0^x$  (zero to the power of  $x$ ), for  $x$  not equal to 0, is always 0.
- $x^0$  ( $x$  to the power of 0), for  $x$  not equal to 0, is always 1.
- $0^0$  (0 to the power of 0) is generally **not defined**! However sometimes it's convenient to define it as equal to 1.
- In programming we start counting from 0 (unlike in real life where we start with 1), so we may encounter the term **zeroth** item. We count from 0 because we normally express offsets from the first item, i.e. 0 means "0 places after the first item".
- It is, along with 1, one of the symbols used in binary logic and is normally interpreted as the "off"/"false"/"low" value.
- Its opposite is most often said to be the infinity, even though it depends on the angle of view and the kind of infinity we talk about. Other numbers may be seen as its opposite as well (e.g. 1 in the context of probability).

- As it is one of the most commonly used numbers in programming, computers sometimes deal with it in special ways, for example in assembly languages there are often special instructions for comparing to 0 (e.g. NEZ, not equals zero) which can save memory and also be faster. So as a programmer you may optimize your program by trying to use zeros if possible.
- In C and many other languages 0 represents the false value, a function returning 0 many times signifies an error during the execution of that function. However 0 also sometimes means success, e.g. as a return value from the main function. 0 is also often used to signify infinity, no limit or lack of value (e.g. NULL pointer normally points to address 0 and means "pointing nowhere").
- Historically the concept of number zero seems to have appeared at least 3000 BC and is thought to signify an advanced abstract thinking, though it was first used only as a positional symbol for writing numbers and only later on took the meaning of a number signifying "nothing".

**Dividing by zero is not defined**, it is a forbidden operation mainly because it breaks equations (allowing dividing by zero would also allow us to make basically any equation hold, even those that normally don't). In programming dividing by zero typically causes an error, crash of a program or an exception. In some programming languages floating point division by zero results in infinity or NaN. When operating with limits, we can handle divisions by zero in a special way (find out what value an expression approaches if we get infinitely close to dividing by 0).

## See Also

- NULL
- infinity
- one
- thrembo

---

zuckerberg

## Mark Zuckerberg

Zuckerberg is one of the ugliest aliens ever recorded on video (yes, including the alien autopsy).

---