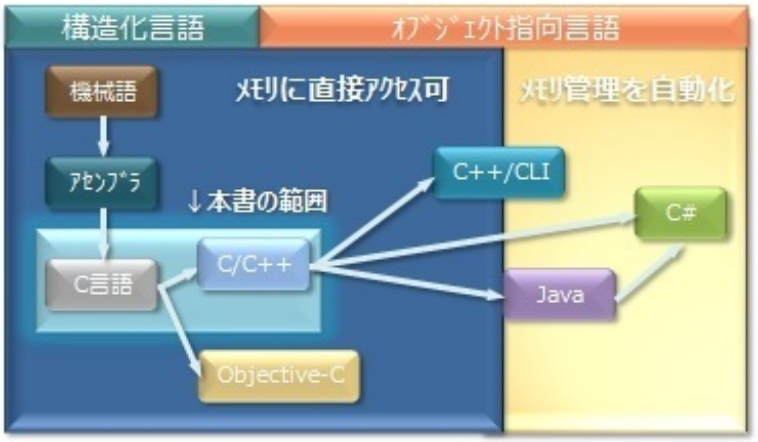
ハム太と作るC++ライブラリ0

-構文編-

haseham

今日における**C/C++**言語



「**C**言語」は、ﾒﾓﾘを直接操作する言語であり、
「機械語」 (ｱｾﾝﾌﾞﾗ含む) に次いで、ｺﾝﾋﾟｭｰﾀｰに近い処理を書くため、
ﾒﾓﾘ効率がよく、実行速度も極めて高速です。

しかし、「**Java**」などの簡略化された言語と比べると、
やはり「書きにくい言語」といえます。

C言語が開発された当初からすれば、
ﾊﾟｿｺﾝの性能は飛躍的に向上しました。

これにともなって、ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ言語も、
「実行速度」や「ﾒﾓﾘ効率」を突き詰めて書くよりも、
「書きやすさ」、つまり、「開発効率」が優先されるようになり、
最近では、「**Java**」や「**C#**」などで開発されることが多くなりました。

ﾌﾟﾛｸﾞﾗﾏｰが**C**言語で書く箇所は、格段に少なくなりました。

しかし、工業用ﾛﾎﾞｯﾄなどの組み込みは
たいてい**C**言語やｱｾﾝﾌﾞﾗで書かれていますし、
「ｹﾞｰﾑﾌﾟﾛｸﾞﾗﾐﾝｸﾞ」などでも、実行速度が重視されるため、
ﾈｲﾃｨﾌﾞな**C/C++**言語による開発は、いまなお主流となっています。

また、「**C**」は、ｵｰﾌﾟﾝ系のｼｽﾃﾑ開発で主流となっている「**Java**」や
「**C#**」、「**JavaScript**」など、ほとんどの言語の先祖にあたり、
共通する点も多いため、その仕組みを知っておくことは、
将来的にも、たいへん意義のあることであると思います。

「**iPhone**ｱﾌﾟﾘ」の開発で使う「**Objective-C**」という言語も、
C言語にﾏｸﾛ関数をつけたもので、ほとんど**C**言語です。

本書では、従来の入門書や入門ｻｲﾄの内容に加えて、
より実践的なﾌﾟﾛｸﾞﾗﾐﾝｸﾞを行うにあたって欠けている点を補うと共に、
気の短い方にもﾑﾘなく読んでいただけるように

単語の種類ごとに色付けを行うなど、読みやすさを追求しています。

ｻﾝﾌﾟﾙｺｰﾄﾞは、ﾃｽﾄするようにしていますが、
絶対ということは言い切れませんので、
使用するにあっては、自己責任でお願いします。

本書の読み方

・画像が表示されない時は、
「再読み込み」をしてみて下さい。

・ﾋﾞｭｰｱの上の方に、もくじがあります。　(PDFにはありません)

・まだ執筆中ですので、
修正や、抜けている点があります。

・まずは、一通り、読んでいただいて、
最初は、わからない点もあると思いますが、
実際に、書いていれば、そのうち理解できます。

・また本書では、読者さんの読解力や計算力などを試したり、
きたえたりするといったことは目的としておらず、
そのような設問も、行っておりません。

・あくまでも、「**C/C++**」というｺﾝﾋﾟｭｰﾀ言語の書き方について、
基礎的な知識の理解を充実させるために、
その助けとなるような解説を行うことに主眼をおいています。

・したがって、その他の学習については、他書にゆずります。

・たまに更新を行っていますので、ﾀﾞｳﾝﾛｰﾄﾞされた方は、
以降も、定期的にﾀﾞｳﾝﾛｰﾄﾞし直すようにしてください。

・**WindowsAPI**などのﾗｲﾌﾞﾗﾘの使い方は、
続巻で取り扱う予定です。

- ・「**STL**」 ... おもによく使うｺﾝﾃﾅｸﾗｽをﾗｯﾋﾟﾝｸﾞ。
- ・標準ｺﾝﾄﾛｰﾙ、ｺﾓﾝｺﾝﾄﾛｰﾙ。(★執筆中。一部公開中)
- ・「**GDI**」 ... 描画系の関数いろいろ。(印刷くらいしか使い道ない？)
- ・「**Direct2D**」 ... ｸﾞﾗﾌｨｯｸﾃﾞﾊﾞｲｽに直接書き込む2D描画ﾗｲﾌﾞﾗﾘ。
- ・「**WIC**」 ... 「**PNG**」「**jpeg**」など、画像ﾌｧｲﾙを簡単に読み書きするﾗｲﾌﾞﾗﾘ。
- ・「**Direct3D 10.1＆11**」 ... 3D描画ﾗｲﾌﾞﾗﾘ。

・「**DirectInput**」「**XInput**」 ... ｺﾝﾄﾛｰﾗｰのﾗｲﾌﾞﾗﾘ。
・「**MME**」 ... 「**MIDI**」「**WAV**」「**MP3**」など、音声を扱うﾗｲﾌﾞﾗﾘ。
・「**XAudio2**」 ... 「**DirectSound**」の後継ﾗｲﾌﾞﾗﾘ。
・「**WASAPI**」 ... 「ｶｰﾈﾙﾐｷｻｰ」を使わないので速い。
・「**MSXML**」 ... 「**XML**」の読み書きでつかうﾗｲﾌﾞﾗﾘ。
・「**SSE**」「**AVX2**」 ... 並列処理用の組み込みﾗｲﾌﾞﾗﾘ。

・待ちきれないという方は、
ﾈｯﾄを利用するなどして、各自で勉強して下さい。

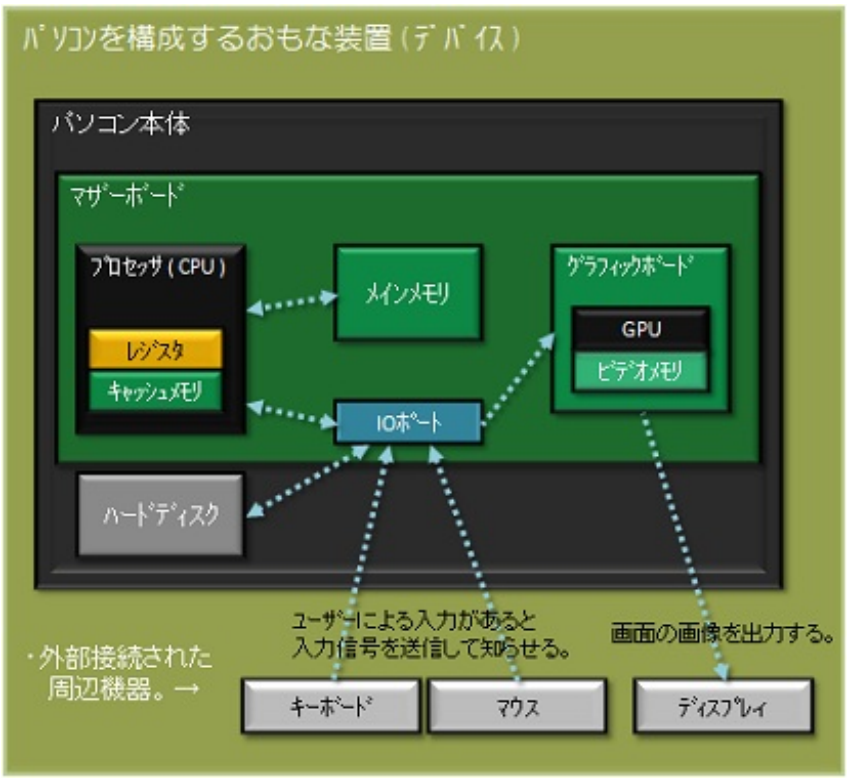
・新しい本がｱｯﾌﾟされたら、ﾌﾞﾛｸﾞでお知らせします。↓

http://haseham.wordpress.com/

http://p.booklog.jp/users/haseham

ｺﾝﾋﾟｭｰﾀｰの仕組み

ｺﾝﾋﾟｭｰﾀｰの仕組みをおさらいしておきましょう。



「**CPU**」(中央処理装置)は、計算を行うところです。

画面上の**1**ﾋﾟｸｾﾙの色が変化するのも、効果音の音程が変化するのも、
すべて**CPU**による計算によるものです。(足し算や引き算など)

「ﾒﾓﾘ」(記憶装置)は、ﾌﾟﾛｸﾞﾗﾑや、計算で使うﾃﾞｰﾀを、
「ﾊｰﾄﾞﾃﾞｨｽｸ」や「**CD-ROM**」から取り出して、
一時的に置いておく「作業台」とか、「配送ｾﾝﾀｰ」のようなものです。

私たちがこれから書いていく「ﾌﾟﾛｸﾞﾗﾑ」は、
「**CPU**」への「命令」を箇条書きにしたもので、
いわば「ｺﾝﾋﾟｭｰﾀｰへの作業手順書」ともいえるものです。

[CPU] ←(計算式ﾃﾞｰﾀ)― [ﾒﾓﾘ]

[CPU] ー(解答ﾃﾞｰﾀ)→ [ﾒﾓﾘ]

「ﾒﾓﾘ」は、比較的、高価な電子部品で、
「ﾊｰﾄﾞﾃﾞｨｽｸ」ほどの容量はないのですが、
磁気ﾃﾞｨｽｸから読み取る手間がかからないため、
CPUとﾃﾞｰﾀをやり取りする際のｱｸｾｽ速度が高速です。

CPUからすれば、CPUの内部にあるﾚｼﾞｽﾀやｷｬｯｼｭに
すべてのﾃﾞｰﾀを置いておく方が速いんですが、
画像のﾃﾞｰﾀや音声のﾃﾞｰﾀは、とても入りきらないので、
必要なﾃﾞｰﾀだけをﾒｲﾝﾒﾓﾘに読み込んでおいて、
その中でも特に、いま行っている計算に必要なﾃﾞｰﾀだけを
ﾚｼﾞｽﾀに転送して計算する仕組みをとっているのです。

ちなみに、ﾚｼﾞｽﾀは、ﾏｼﾝ語やｱｾﾝﾌﾞﾗで書くと、操作できるんですが、
ｷｬｯｼｭについては、よく使うﾃﾞｰﾀが自動的にｽﾄｯｸされるので、
ﾌﾟﾛｸﾞﾗﾏが操作することはできません。

ﾌﾟﾛｸﾞﾗﾑは、ﾃﾞｰﾀの移動に関する処理が多く、
倉庫管理や、配送などの仕事に通じるものがあります。

よく使うﾃﾞｰﾀだけを、厳選してﾒﾓﾘ上に用意しておくだとか、
よく使う計算結果は、ﾒﾓﾘ上に残しておくことで、
ﾃﾞｰﾀの移動回数や、計算回数を少なくすることができます。

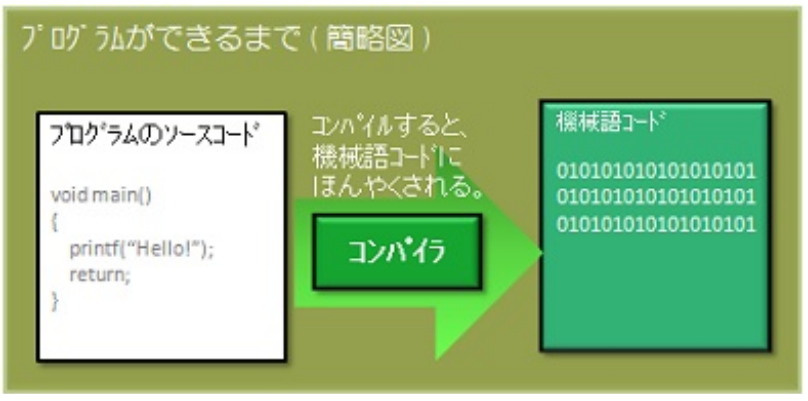
ﾃﾞｰﾀの転送が速く済めば、大量の計算を行ったとしても、
画面が固まることが少なくなり、快適な動作を実現することができます。

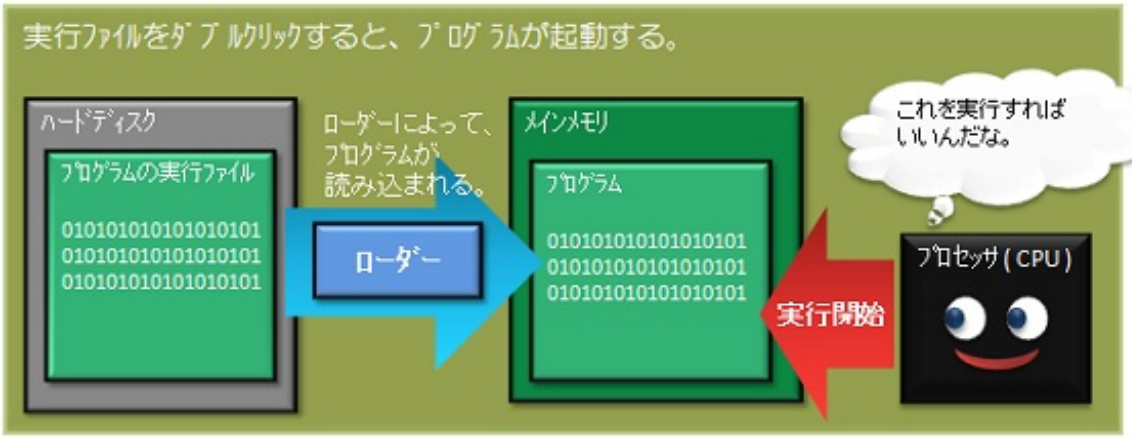
ｱﾌﾟﾘｹｰｼｮﾝｿﾌﾄのｱｲｺﾝをｸﾘｯｸすると、**OS**の「ﾛｰﾀﾞｰ」というﾌﾟﾛｸﾞﾗﾑによって
ｱﾌﾟﾘｹｰｼｮﾝの実行ﾌｧｲﾙがﾒﾓﾘ上に読み込まれ、ﾌﾟﾛｸﾞﾗﾑが実行されます。

「ﾌﾟﾛｸﾞﾗﾑ文」というと、**C**言語で書かれていることが多いのですが、
実際に、ﾌﾟﾛｸﾞﾗﾑが実行された時、CPUが読み取っているのは
ﾏｼﾝ語の命令番号です。（「ｵﾍﾟｺｰﾄﾞ」という）

しかし、ただ、数字を並べただけだと、書いてるうちに意味がわからなくなるので、
人間からみて、比較的読みやすい**C**言語でﾌﾟﾛｸﾞﾗﾑ文を書いておいて、
あらかたできたところで、それを「ｺﾝﾊﾟｲﾗ」というｿﾌﾄを使って、
ﾏｼﾝ語に置き換えるようにしているのです。(これを 「ｺﾝﾊﾟｲﾙ」 という)

この時に作られるのが実行ﾌｧｲﾙで、(実際には、ﾘﾝｶというｿﾌﾄによって作成される)
ｱﾌﾟﾘｹｰｼｮﾝｿﾌﾄのﾌｫﾙﾀﾞ内にある、「ﾎﾆｬﾗﾗ**.exe**」というﾌｧｲﾙがそれです。





ちなみに、ハードディスクについては、最近は「**SSD**」という記憶装置が普及しています。

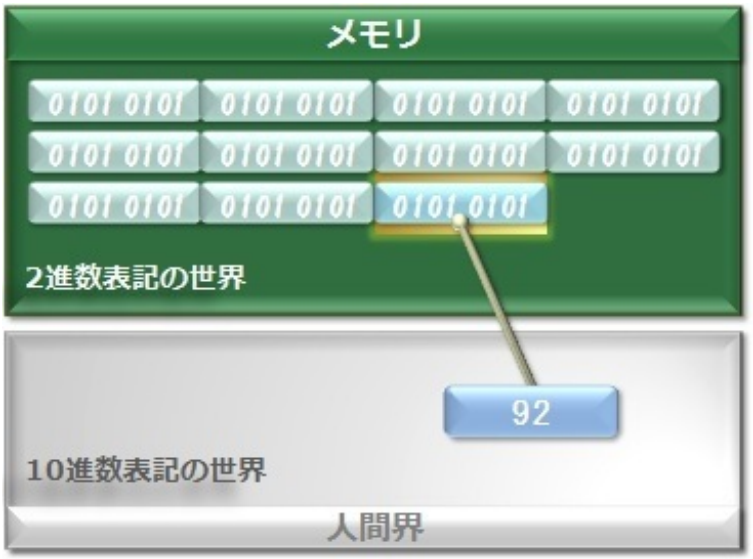
これは、フラッシュメモリの一種で、ハードディスクに比べると、
書き込み回数の制限があるものの、データの転送速度が格段に速いため、
メインメモリの少ない非力なシステム環境で絶大なパフォーマンス向上効果があり、
おもに、ノート**PC**やタブレット端末などでハードディスクの代わりに使用されています。

むかしの入門本と比較すると、この他にもいくつか変更点があって、
マルチコアプロセッサ(「**MPU**」という)による並列処理が一般化したことや、
グラフィックボードの中に、描画処理専用の**CPU**(「**GPU**」という)と
ビデオメモリ(「**VRAM**」という)があるんですが、
これを通常のCPUや、メインメモリの代わりに使う「**GPGPU**」という技術が、
ゲームソフトなどを中心に導入されています。

ディスプレイの解像度や、メインメモリの容量が大幅に向上したこと、
64bit化への移行については、後述します。

Cや、C++の言語仕様も、近年になって何度か拡張されていますが、
これはコンパイラや、それを含む開発環境のバージョンによって
対応にばらつきがあるため、本書では、さわり程度にとどめ、
基本的な仕様の理解に力点をおいて解説しています。

ﾃﾞｼﾞﾀﾙﾃﾞｰﾀは、**0**か**1**かの連続。



・「ｺﾝﾋﾟｭｰﾀｰ」というのは、「電子回路」 のことで、
その中では、 「電気信号」 が超高速で流れています。

・「ﾂｰ・ﾄﾝ・ﾂｰ・ﾄﾝ・ﾄﾝ」 とする 「ﾓｰﾙｽ信号」のように、
電気信号の有無を、連続して送信することで、
大量のﾃﾞｰﾀを伝えています。

・「ﾊｰﾄﾞﾃﾞｨｽｸ」や「**CD-ROM**」の円盤の表面にも、
この「**0**か**1**か」が、「磁気信号の有無」に置き換えられて
記録されています。

・「ﾓｰﾙｽ信号」の場合は、たとえば、
「・・ー」 なら 「**A**」という意味だ！ ...というように、
一定間隔の間に、どこでｷｰを押すのかで
対応する文字を申し合わせておいて、
暗号による通信を実現していました。

・ｺﾝﾋﾟｭｰﾀｰの内部のﾃﾞｰﾀ転送も、
これとよく似たことをしています。

・**Windows**をお持ちの方は、「電卓」を開いてみて下さい。

・ﾒﾆｭｰﾊﾞｰの「表示」をｸﾘｯｸして、「ﾌﾟﾛｸﾞﾗﾏ」を選んで下さい。

・そして、適当に数値を入力した後に、「2進」をｸﾘｯｸしてみて下さい。

・「0」 と 「1」 だけの表示に切り替わったはずです。

・これを「2進数表記」といいます。

・私たちがふだん使っている数字の表記法は、
「10進数表記」というもので、
1ｹﾀの数字が10になると、
1ｹﾀ繰り上がる仕組みになっています。

・「9」に「1」を足すと、1ｹﾀ繰り上がって、「10」になる。

・2進数表記の場合は、0か1しかないため、
1に1を足すと、1ｹﾀ繰り上がって、「10」と表記されます。

・が、これは10進数表記の「2」、
いわゆる2のことです。

2進数	10進数	
0	0	
1	1	
10	2	←ｺｺ で くり上がっている。
11	3	
100	4	
101	5	
:	:	

・ようするに、表記法 (書き方) が違うだけなのです。

ﾃﾞｼﾞﾀﾙﾃﾞｰﾀの単位、「ﾋﾞｯﾄ」と「ﾊﾞｲﾄ」

・ﾃﾞｼﾞﾀﾙﾃﾞｰﾀの最小単位は、「**1bit**」 (ﾋﾞｯﾄ)です。

・この「**1bit**」には、0～1までの
2通りの数値しか記憶できません。

・しかし、実際にあつかわれる数値は、
もっと大きなものばかりです。

・そこで登場するのが、「**1byte**」 (ﾊﾞｲﾄ) という単位です。

・「**1byte**」は、「**1bit**」を**8**つならべたもので、「**8bit**」のことです。

・ｺﾝﾋﾟｭｰﾀｰﾌﾟﾛｸﾞﾗﾑでは、ﾒﾓﾘ上に記憶されたﾋﾞｯﾄﾃﾞｰﾀを、
この「**1**ﾊﾞｲﾄ」単位で取り出して、
1つの数値として解釈して、操作します。

・それでは次に、この1byteのﾃﾞｰﾀを、
私たちがふだん使っている10進数表記の数値に
戻す方法について説明します。↓

1111 1111

・上記の値を、「**2**進数」ﾓｰﾄﾞで入力してから、
「**10**進数」ﾓｰﾄﾞに切り替えてみて下さい。

・「255」と表示されると思います。

・これが「**1byte**」に記憶できる最大値です。

・「0」を含めると、0～255までの
256通りの数値を、記憶できる、ということになります。

・この「ﾊﾞｲﾄ」という単位は、
ﾒﾓﾘ上の、**1**つの番地に、記録できる値です。

・そして、**CPU**は、というと、
4byteずつ計算するﾀｲﾌﾟが、現在の主流で、
CやC++のｺﾝﾊﾟｲﾗの標準の整数値が
4byte値なのは、そのためです。

・ちなみに、**64bit** ｼｽﾃﾑ (64bit版のWindowsなど) では、
CPUは、**8byte**ずつ計算するため、
標準の整数値も、**8byte**値になります。

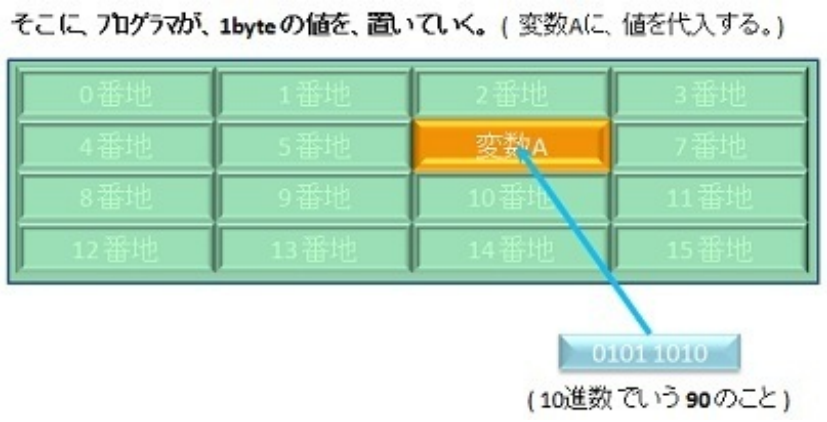
基本ﾃﾞｰﾀ型

・基数の話が出たついでに、
「変数」についても、少し説明しておきます。

・変数を宣言する時は、
変数名の前に必ず
「ﾃﾞｰﾀ型」を書き添えます。↓

```
int hensuu = 100;
```

・**C/C++**では、ﾒﾓﾘ上の数ﾊﾞｲﾄの領域に対して、
「変数名」という覚えやすい名前をつけて操作します。↓



・上の図では、**6**番地のところに、

「変数**A**」というお皿が、置かれています。

(どのアドレスに配置されるのかは、プログラムの実行時までわからず、
システム (WindowsなどのOS) が、空いている場所を探して決めてくれます。)

・ そして、このお皿の上に、数値を置いて行きます。

・お皿には、**2**倍ｻｲｽﾞのものや、**4**倍ｻｲｽﾞのものがあります。

・このお皿のｻｲｽﾞというのは、お皿を置く時に、
「ﾃﾞｰﾀ型」(DataType) というものを指定して、決めてあげます。↓

【 **C**の基本ﾃﾞｰﾀ型と そのﾊﾞｲﾄ数 】

データ型	サイズ	記録できる値の範囲
char	1byte	-128 ～ 127
unsigned char	1byte	0 ～ 255
short	2byte	-32768 ～ 32767
unsigned short	2byte	0 ～ 65535
int	4byte	-2147483648 ～ 2147483647
unsigned int	4byte	0 ～ 4,294,967,295
long long	8byte	-9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807
unsigned long long	8byte	0 ～ 18,446,744,073,709,551,615
float	4byte	3.4E + 38 (7 ｹﾀの数字)
double	8byte	1.7E + 308 (15ｹﾀの数字)
bool	1byte	0 ～ 1

・「**unsigned**」が付いているﾃﾞｰﾀ型は、**+-**符号が付かない整数で、
「負数」(0より小さい数) を 記録することができません。

・「**unsigned**」が付いていないﾃﾞｰﾀ型は、**+-**符号が付く整数で、
「負数」 を記録することができます。

・しかし、**+-**符号の記録に**1bit**使っているため、
記録できる絶対値が、半分になっています。

・ちなみに、「**signed**」が付いている場合は、
+-符号が付かない整数であるとみなされます。

・「**long**」は、ｺﾝﾊﾟｲﾗによってｻｲｽﾞが異なり、

Windows では **4byte**で、
「**int**」と同じｻｲｽﾞなんですが、
Unix系の**64bitOS**では、**8byte**となります。

・「float」と「double」は、「浮動小数点」で、
+-符号の付いた「小数」(1.0～0.0の小数ｹﾀを含む数値) を記録できます。

・「**long double**」については、
ｺﾝﾊﾟｲﾗによってｻｲｽﾞが異なり、
VC++では**8byte**、**BCC++**では**10byte**、
gccでは**12byte**となります。

・さいごの「bool」は「論理値」で、
「**true**」(1)か、「**false**」(0)かの
いずれかの値であるかしか、記録できません。

ﾃﾞｰﾀ型が short の変数には、2 byte の値を置くことができる。

・**C++**には、ｸﾗｽ型をはじめとして、
様々なﾃﾞｰﾀ型がありますが、
これらはすべて、
C言語の「基本ﾃﾞｰﾀ型」(ﾌﾟﾘﾐﾃｨﾌﾞ型) を
別の名前で再定義したり、
組み合わせたものです。

・**C++**は、**C**言語の構文をﾍﾞｰｽに、
追加で機能拡張したものですから、
ﾃﾞｰﾀ型も、**C**のものが、そのまま使用できます。

16進数表記

・これも、よく使う記法ですので、覚えておきましょう。

・「2進数表記」では、　「2」になると、1ｹﾀ繰り上がりました。
・「10進数表記」では、「10」になると、1ｹﾀ繰り上がりました。
・「16進数表記」では、「16」になると、1ｹﾀ繰り上がります。

・しかし、数字は、「0」～「9」までしかありません。

・「9」よりも上の数字は、どうやって表記するのでしょうか。

16進数	10進数	
9	9	
A	10	
B	11	
C	12	
D	13	
E	14	
F	15	
10	16	←ここで初めて1ｹﾀ繰り上がる。
11	17	
12	18	
:	:	
FF	255	

・2進数の時と同じように、「電卓」を開いて、
いろいろな数値を入力して、試してみて下さい。

・ﾌﾟﾛｸﾞﾗﾑで書く時は、「0x10」と書きます。

・「0x10」は、「10進数」でいうところの「16」です。

・この、1ｹﾀ繰り上がる数のことを、
「基数」といいます。↓

2進数表記 ... 2で繰り上がるので、基数は 2。
10進数表記 ... 10で繰り上がるので、基数は10。
16進数表記 ... 16で繰り上がるので、基数は16。

・また、各ｹﾀで繰り上がった時の数を
「ｹﾀの重み」といいます。↓

【2進数表記】

1ｹﾀ目の重みは、**1**。(0000 000**1**)
2ｹﾀ目の重みは、**2**。(0000 00**1**0)
3ｹﾀ目の重みは、**4**。(0000 0**1**00)
4ｹﾀ目の重みは、**8**。(0000 **1**000)

・この「ｹﾀの重み」を使うと、
計算機なしで**10**進数に戻すことができます。

・各ｹﾀに、それぞれのｹﾀの重みをかけて行き、
それらをすべて合算してみてください。

・「**16**進数表記」のおもな用途は、
ﾋﾞｯﾄ単位の操作です。

・詳しいことは、次の章で説明しますが、
ﾋﾞｯﾄ演算の「ﾏｽｸﾋﾞｯﾄ」として、よく使われています。

・ﾋﾞｯﾄ演算を使うと、

1byte値の中に含まれる数ﾋﾞｯﾄを
部分的に **0** にしたり、**1** にしたり
することができるのです。

32bitｼｽﾃﾑと、64bitｼｽﾃﾑへの移行。

・ｺﾝﾋﾟｭｰﾀｰﾌﾟﾛｸﾞﾗﾑでは、
数値を、4byte単位であつかうことが多いのですが、
これは、多くのﾊﾟｿｺﾝが、32bit、つまり、
4byte単位で計算を行うCPUと、
それを前提として動作するOSとで
構成されているからです。↓

・32bitｼｽﾃﾑは、
[32bit CPU] + [4GB未満のﾒﾓﾘ] +[32bit版 Windows]
(標準整数は4byte値。 ﾒﾓﾘｱﾄﾞﾚｽも4byte値。)

・64bitｼｽﾃﾑは、
[64bit CPU] +[4GB以上のﾒﾓﾘ] +[64bit版 Windows]
(標準整数は8byte値。 ﾒﾓﾘｱﾄﾞﾚｽも8byte値。)

・「32bitｼｽﾃﾑ」の時代は、Windowsでいうと、
「Windows95」の頃あたりからですから、
もうずいぶん長いこと続いていたんですが、
「Windows Vista」あたりから、
ﾒﾓﾘのｱﾄﾞﾚｽが、32bit値の限界に達したこともあって、
「64bit版のWindows」が登場しました。

・現在はちょうど過渡期にあたり、
「Windows7」搭載のﾉｰﾄﾊﾟｿｺﾝなどは、
すでに64bitｼｽﾃﾑなんですが、
32bitｼｽﾃﾑ用のﾌﾟﾛｸﾞﾗﾑが対応していないため、
「Windows7」などに搭載された「WOW64」という
ｴﾐｭﾚｰﾀｰを使って動作させています。

・「Vista」の頃は、未対応のｱﾌﾟﾘｹｰｼｮﾝが非常に多く、
周辺機器のﾃﾞﾊﾞｲｽﾄﾞﾗｲﾊﾞも、多くが未対応でした。

・**C/C++**でﾌﾟﾛｸﾞﾗﾐﾝｸﾞをするにあたって、
特に要注意なのが、この「ﾒﾓﾘｱﾄﾞﾚｽ」で、
従来の**4byte (32bit)** 値から
8byte (64bit) 値になったことです。

・「ﾒﾓﾘｱﾄﾞﾚｽ」というのは、
『ﾒﾓﾘ上の、どの位置にﾃﾞｰﾀを記憶したのか』 を示す
「ｱﾄﾞﾚｽ」(番地)のことです。

・いま使っているﾊﾟｿｺﾝが、どちらなのかを知るには、
「ｴｸｽﾌﾟﾛｰﾗ」を開いて、「ｺﾝﾋﾟｭｰﾀ」のｱｲｺﾝの上で
右ｸﾘｯｸして、「ﾌﾟﾛﾊﾟﾃｨ」をｸﾘｯｸします。

・ﾀﾞｲｱﾛｸﾞ画面に、「**x64**」とか「**64**ﾋﾞｯﾄｼｵﾍﾟﾚｰｼｮﾝｼｽﾃﾑ」
と表示されていたら、**64bit**ｼｽﾃﾑです。

・「**x86**」とか、「**32bit**ｵﾍﾟﾚｰｼｮﾝｼｽﾃﾑ」なら、
おそらく**32bit**ｼｽﾃﾑです。

・**64bit**ｼｽﾃﾑの場合は、
「**Program Files**」ﾌｫﾙﾀﾞが**2**つあって、
「**(x86)**」と書いてある方には、
32bitのｿﾌﾄｳｪｱがｲﾝｽﾄｰﾙされています。

・「**VisualStudio**」や「**Windows SDK**」のﾌｫﾙﾀﾞにも
2つありますので、間違えないようにして下さい。

・「**x86**」というのは、ﾏｲｸﾛﾌﾟﾛｾｯｻのｱｰｷﾃｸﾁｬ(構造)の種類で、
Intel社製の**32bit**ﾏｲｸﾛﾌﾟﾛｾｯｻは、おおむね、これに該当します。

・**32bit**より前のものも存在しますが、
386以降の**32bit**ﾌﾟﾛｾｯｻだけを区別するために
最近では、「**IA-32**」とかいわれるようになりました。

・**IA-32**では、32bit値でﾒﾓﾘｱﾄﾞﾚｽを記録するため、
理論的には、**4GB**までのﾒﾓﾘ空間に対応しています。

・しかし、**32bit**版の**Windows**などでは、
4GBのﾒﾓﾘを搭載したﾊﾟｿｺﾝであっても、
ｼｽﾃﾑが使用する領域などもあるため、
ｱﾌﾟﾘｹｰｼｮﾝﾌﾟﾛｸﾞﾗﾑは、**3GB**までしか使用できません。

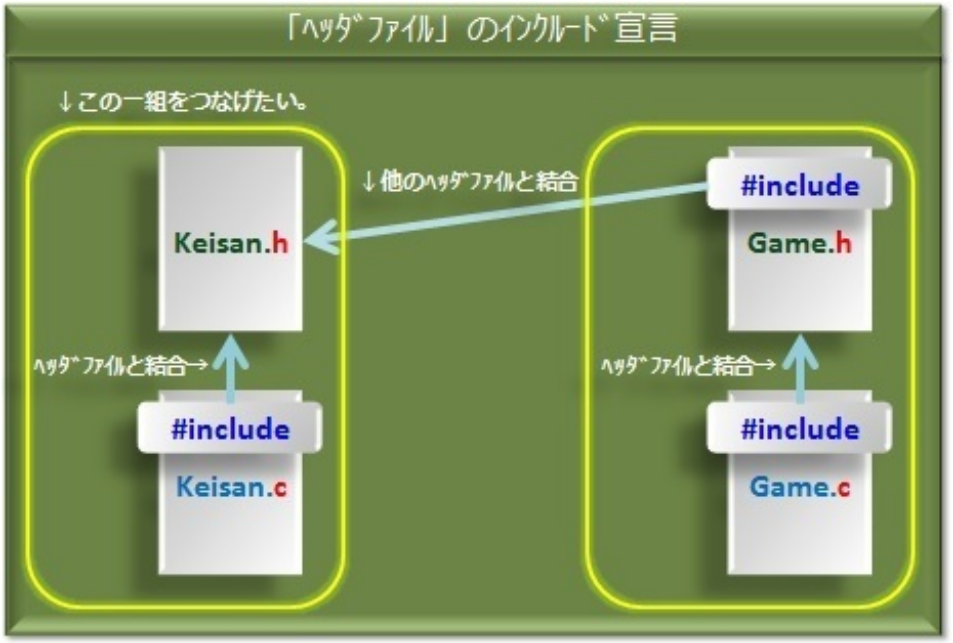
・また、ｱﾄﾞﾚｽ空間は、ｱﾌﾟﾘｹｰｼｮﾝﾌﾟﾛｸﾞﾗﾑごとに独立しており、
それぞれ**2GB**までしか使用できません。

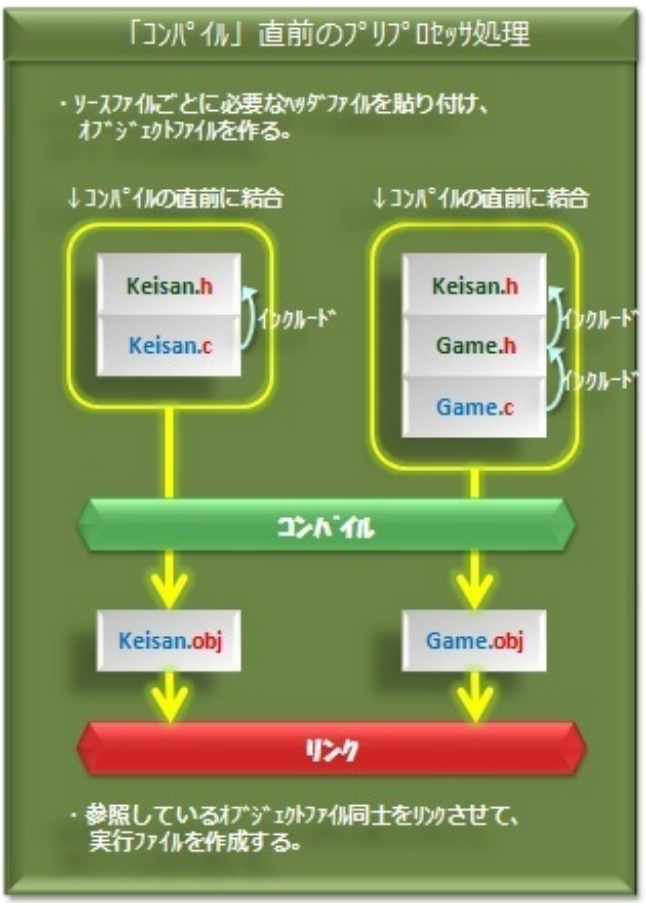
・ちなみに、**64bit**ﾌﾟﾛｾｯｻの主流である「**x64**」というのは、
AMD社の**AMD64**ｱｷｰﾃｸﾁｬがﾍﾞｰｽとなっていて、
これは現在、主流となっている**Intel**社のﾌﾟﾛｾｯｻでも、導入されています。

・**Intel**社は、当初、「**IA-64**」というのを用意していたんですが、
x86との互換性を切り捨ててしまったため、受けが良くなく、
結局、ﾗｲﾊﾞﾙ社の規格に乗る形になってしまったわけです。

・逆をいえば、この**x64**というのは、**x86**と互換性があって、
つまり、**x86**で動作するｿﾌﾄｳｪｱであれば、
x64上でも動作する、ということなんです。

「ｿｰｽﾌｧｲﾙ」と「ﾍｯﾀﾞﾌｧｲﾙ」





・さて、すでに述べたように、
「ﾌﾟﾛｸﾞﾗﾑ」というのは、数字を並べたものです。

・そしてその、一つ一つの数字が、
ﾏｼﾝ語の命令番号だったり、

計算に使う値だったりするわけです。

・しかし、**C**言語で書いたﾌﾟﾛｸﾞﾗﾑの文というのは、
これは「ｿｰｽｺｰﾄﾞ」(ﾌﾟﾛｸﾞﾗﾑを作るための情報源)というんですが、
ただのﾃｷｽﾄﾃﾞｰﾀ(文章)ですから、
ﾃｷｽﾄﾌｧｲﾙに保存します。

・これを「ｿｰｽﾌｧｲﾙ」といいます。

・「ﾃｷｽﾄﾌｧｲﾙ」(*.**txt**)のﾌｧｲﾙ拡張子を、
「**.c**」にしたものが「ｿｰｽﾌｧｲﾙ」です。(test.**c** など)

・**C/C++**の場合は、「**.cpp**」になります。(test.**cpp** など)

・ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞが長くなってくると、読みにくいので、
複数の「ｿｰｽﾌｧｲﾙ」に、分けて書きます。

・「ｿｰｽﾌｧｲﾙ」は、
ｺﾝﾊﾟｲﾗによって実行ﾌｧｲﾙが作成される時に、
別の「ｿｰｽﾌｧｲﾙ」に結合されますが、
その際に必要なのが、対応する「ﾍｯﾀﾞﾌｧｲﾙ」です。

・「ﾍｯﾀﾞﾌｧｲﾙ」も、ただの「ﾃｷｽﾄﾌｧｲﾙ」で、
ﾌｧｲﾙ名は、対応する「ｿｰｽﾌｧｲﾙ」と同じ名前を使い、
ﾌｧｲﾙ拡張子を「**.h**」にします。(text.**h** など)

・まあこのﾌｧｲﾙ名は、一致していなくてもいいんですが、
わかりにくいので、ふつうは、同じ名前にします。

・**C/C++**の場合は、「**.hpp**」も使えます。(test.**hpp** など)

・「ﾍｯﾀﾞﾌｧｲﾙ」には、次のようなことを書きます。↓

・「ﾌﾟﾛﾄﾀｲﾌﾟ宣言」というのは、
対応する「ｿｰｽﾌｧｲﾙ」上で定義された関数の、
処理内容 (中ｶｯｺの中) 以外の部分だけを書いたものです。↓

```
int Tasu( int a , int b );
int HiKu( int a , int b );
int Kakeru( int a , int b );
int Waru( int a , int b );
```

・「ｺﾝﾊﾟｲﾗ」は、ｺﾝﾊﾟｲﾙが開始されると、
まず、さいしょに、ｼﾝﾎﾞﾙ (単語) の意味を調べます。

・そして、それが「関数」の名前で、

さらに、呼び出し文だとわかったなら、
今度は、それに続くｶｯｺ内で、渡されている引数の値や、
「戻り値」を受け取っている変数の型が
適正か、不適正か、をﾁｪｯｸするために、
この「ﾌﾟﾛﾄﾀｲﾌﾟ宣言」を見にいきます。

・別のｿｰｽﾌｧｲﾙ上で定義された関数を呼び出す時に、
この「ﾍｯﾀﾞﾌｧｲﾙ」が必要なのは、そのためです。

・関数の呼び出しや、変数を参照している箇所は、
この情報に基づいて機械語に変換されます。

・ﾍｯﾀﾞ側の情報は、外部に公開するためのものですが、
関数の呼び出しや、変数を参照している箇所は、
同じｿｰｽﾌｧｲﾙ内にもある場合があるので、
やはり、ﾍｯﾀﾞﾌｧｲﾙのｲﾝｸﾙｰﾄﾞが必要なのです。

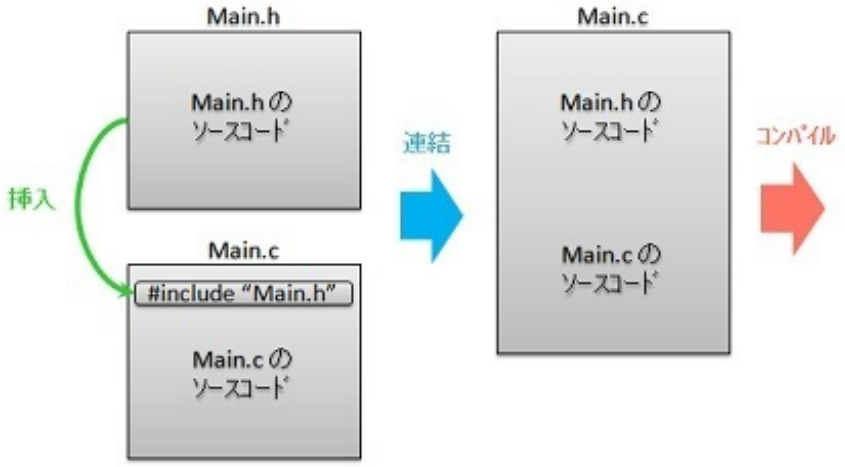
・ｺﾝﾊﾟｲﾗは、ｺﾝﾊﾟｲﾙを開始する直前に、
「ｲﾝｸﾙｰﾄﾞ宣言」を書いた所に、
指定されている「ﾍｯﾀﾞﾌｧｲﾙ」を貼り付けています。 ↓

```
#include "HeaderFileA.h"
```

・「ｲﾝｸﾙｰﾄﾞ宣言」は、
「ﾍｯﾀﾞﾌｧｲﾙ」の最初に、まとめて書きます。

・ｺﾝﾊﾟｲﾗは、ｺﾝﾊﾟｲﾙを行う直前に、
このｲﾝｸﾙｰﾄﾞ宣言が書かれている位置に、
指定されたﾍｯﾀﾞﾌｧｲﾙを書き込んでから、
ｺﾝﾊﾟｲﾙを始めます。

・上の例だと、**HeaderFileA.h** に書かれている内容が、
ｲﾝｸﾙｰﾄﾞ宣言の位置に、貼り付けられるのです。



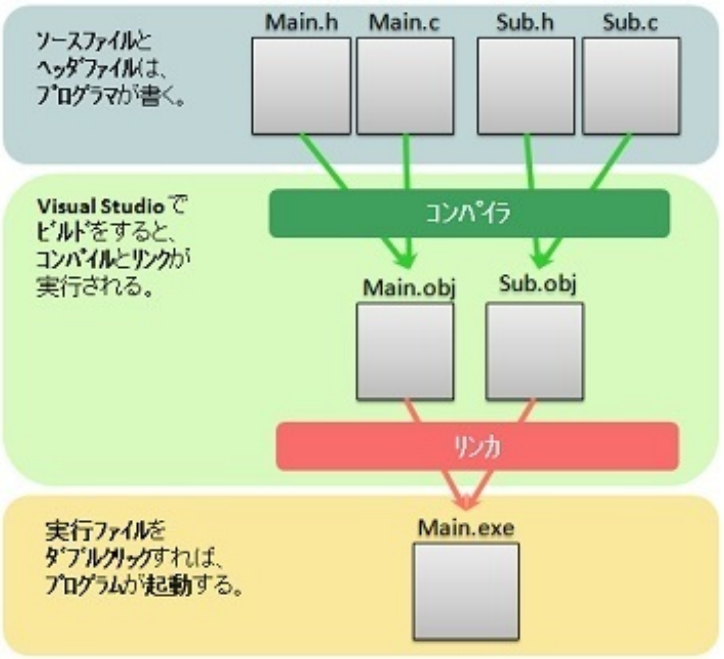
・こうして、他の「ﾍｯﾀﾞﾌｧｲﾙ」を結合させると、
その中で定義されている関数やﾃﾞｰﾀ型が有効になり、
使用することができるようになります。

・関数の処理部分 (「実装」という) は、
「ﾍｯﾀﾞﾌｧｲﾙ」側に書くこともでき、
その場合は、「ｿｰｽﾌｧｲﾙ」は不要です。

・詳細については、次章の
「ﾍｯﾀﾞﾌｧｲﾙの書き方」を参照して下さい。

・ｿｰｽﾌｧｲﾙ側は、こちら。↓

「ｺﾝﾊﾟｲﾗ」 と「ﾘﾝｶ」 (実行ﾌｧｲﾙを作るﾌﾟﾛｸﾞﾗﾑ)



・ｿｰｽﾌｧｲﾙというのは、ただのﾃｷｽﾄﾃﾞｰﾀでした。

・これをもとに、ﾌﾟﾛｸﾞﾗﾑを作成するには、
まず、「**C/C++ｺﾝﾊﾟｲﾗ**」というﾌﾟﾛｸﾞﾗﾑを使います。

・「**C/C++ｺﾝﾊﾟｲﾗ**」は、
ﾌﾟﾛｸﾞﾗﾏが書いたｿｰｽｺｰﾄﾞ を 解析します。

・「ｺﾝﾊﾟｲﾗ」では、
単語の読み取りと、構文ﾁｪｯｸが行われ、
「ｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙ」 (***.obj**) が作成されます。　(これを 「ｺﾝﾊﾟｲﾙ」 という)

・次に、「ﾘﾝｶ」では、その「ｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙ」 (***.obj**) 同士を結合させて、
「実行ﾌｧｲﾙ」 (***.exe**) を作ります。　(これを 「ﾘﾝｸ」 という)

・ｺﾝﾊﾟｲﾙ中のｴﾗｰは、おおむね構文ﾐｽが原因です。

・変数名が一文字ﾀｲﾌﾟﾐｽしているだけでも、
「定義されていない識別子です」
とかいうｴﾗｰが出ます。

・ﾘﾝｸ中のｴﾗｰは、わかりにくいものが多いんですが、
よく出るのが「外部ｼﾝﾎﾞﾙが未解決です」というｴﾗｰで、
ﾗｲﾌﾞﾗﾘﾌｧｲﾙか、ﾍｯﾀﾞﾌｧｲﾙのﾊﾟｽが間違っていて
ﾌｧｲﾙを開けない、だとか、
ﾍｯﾀﾞﾌｧｲﾙ側では関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言があるのに、
関数の実装(処理内容)が定義されていない、
などといったことが、そのおもな原因です。

・これも結局、よくにた話で、
ｿｰｽｺｰﾄﾞ上で使用されている名前が
定義されている箇所が見つからないため、
意味わかんねー使えねーということなんですね。

・「**Windows SDK**」や「**Visual Studio**」のフォルダの中には、
「**lib**」というフォルダがあり、ここに**lib**ファイルがまとめて置いてあります。

・ヘッダファイルは、「**inc**」とか「**include**」というフォルダにあります。

・これらのファイルの所在が、リンカから見えていないのです。

・**Visual Studio** を使っている場合は、メニューバーから、
[ツール]、[オプション]とクリックして、オプション画面を開いてください。

・次に、左側のツリービューから、[プロジェクトおよびソリューション]、
[**VC++**ディレクトリ] をクリックすると、フォルダへのパスが一覧表示されす。

・右上の「ディレクトリを表示するプロジェクト」で、「ライブラリファイル」を指定すると、
リンカから見えている **lib**ファイルへのパスが、一覧表示されます。

・**Visual Studio** のフォルダの中にあるものは、すでに登録されているはずですが、
Windows SDK など、後でダウンロードしたものは、ここで追加する必要があります。

・こうしておくと、ソース上で、**lib**ファイルのあるフォルダを含めたパスを、
指定する必要がなくなります。(つまり、ファイル名だけで通じる)

・インクルードファイルについても、同様に指定しておきます。

・また、**dll**ファイルについても、追加しておけますが、
うまくいかない場合は、カレントフォルダ内に置いてみてください。

(プログラムのカレントフォルダは、実行ファイルのあるフォルダで、
この中に置かれたファイルについては、ファイル名だけで通じる。)

無料版をｲﾝｽﾄｰﾙしてみる。

・「**Windows**」で、ｿﾌﾄｳｪｱを開発する場合は、
「**Visual Studio**」という開発ﾂｰﾙを使います。

・その他にも、いろいろありますが、
最も使いやすくて、安定している上に、
「**Communitiy 2013**」という無料版があるので、
これを使わない手はありません。　→ﾀﾞｳﾝﾛｰﾄﾞはこちら

・画面を少し、下にｽｸﾛｰﾙしていくと、
「**Community 2013 Update4**」と、書いてある所がありますので、
ここをｸﾘｯｸすると、小さい画面が開きます。

・その中の、左側にある、地球儀のｱｲｺﾝをｸﾘｯｸして下さい。

・見出しでいうと、
「**Microsoft Visual Studio Community 2013 with Update 4 - English**」
と、書いてあるところです。

・これは英語版なんですが、これをｲﾝｽﾄｰﾙした後で、
今度は、その右側にあるｱｲｺﾝをｸﾘｯｸして、
「**Microsoft Visual Studio 2013 Language Pack - 日本語**」
をｲﾝｽﾄｰﾙすると、画面上の表記を、日本語に変更することができます。

・この「**Visual Studio**」というのは、元々は、ﾌﾟﾛ向けの開発ｿﾌﾄで、
以前であれば、学生用でも、3万円くらいしたんですが、
無料版の「**Express Edition**」が配布されるようになりました。

・「**Community 2013**」は、ﾌﾘｰｳｪｱを開発している個人や、
ﾌﾟﾛｸﾞﾗﾑを学習中の学生を支援するために配布されているものです。

・ちなみに、**Window**ｱﾌﾟﾘｹｰｼｮﾝを開発するには、

「**Windows SDK**」が必要です。↓

http://msdn.microsoft.com/ja-jp/library/windows/desktop/ff381402(v=vs.85).aspx

・この中には、
ﾃﾞｽｸﾄｯﾌﾟｱﾌﾟﾘの開発で使用する「**Windows API**」や、
ｽﾄｱｱﾌﾟﾘの開発で使用する 「**Windows RT**」や、
ｹﾞｰﾑの開発でよく使う「**DirectX**」など、
基本的な関数ﾗｲﾌﾞﾗﾘが含まれています。

「ﾌﾟﾛｼﾞｪｸﾄ」を作成する。

・これからﾌﾟﾛｸﾞﾗﾑを書いていくんですが、
それには、まず「ﾌﾟﾛｼﾞｪｸﾄ」を作成します。

・「**VisualStudio**」を起動したら、
ﾒﾆｭｰの「ﾌｧｲﾙ」をｸﾘｯｸして、
「新しいﾌﾟﾛｼﾞｪｸﾄ」をｸﾘｯｸして下さい。

・すると、「新しいﾌﾟﾛｼﾞｪｸﾄ」というﾀﾞｲｱﾛｸﾞが開くので、
その左側の、開発言語のところで「**C/C++**」をｸﾘｯｸします。

・次に、右側で、「ｺﾝｿｰﾙｱﾌﾟﾘｹｰｼｮﾝ」をｸﾘｯｸします。

・「ｺﾝｿｰﾙｱﾌﾟﾘｹｰｼｮﾝ」では、「**DOS**画面」しか表示されませんが、
C言語の単元では、「**WindowsAPI**」は使いませんので、これにします。

・通常の、「ｳｨﾝﾄﾞｳ」によるｱﾌﾟﾘｹｰｼｮﾝを開発する場合は、
「**Windows**ﾌｫｰﾑｱﾌﾟﾘｹｰｼｮﾝ」をｸﾘｯｸします。

ﾌﾟﾛｸﾞﾗﾑを実行する。

・ﾒﾆｭｰの「ﾋﾞﾙﾄﾞ」をｸﾘｯｸして、
「ｿﾘｭｰｼｮﾝのﾋﾞﾙﾄﾞ」をｸﾘｯｸすると、
「ｿﾘｭｰｼｮﾝ」以下のすべての「ﾌﾟﾛｼﾞｪｸﾄ」が
「ﾋﾞﾙﾄﾞ」されます。

・「ﾋﾞﾙﾄﾞ」というのは、
「ｺﾝﾊﾟｲﾙ」や「ﾘﾝｸ」をまとめたもので、
これが完了すると、そのﾌﾟﾛｼﾞｪｸﾄのﾌｫﾙﾀﾞの中に
「実行ﾌｧｲﾙ」が作成されます。

・次に、ﾒﾆｭｰの少し下にある「開始」をｸﾘｯｸすると、
ﾌﾟﾛｸﾞﾗﾑが実行されます。

・ﾌﾟﾛｸﾞﾗﾑのﾋﾞﾙﾄﾞには**2**通りあって、
ちゃんと動くのかをﾃｽﾄしている間は、
「ﾃﾞﾊﾞｯｸ」ﾓｰﾄﾞでﾋﾞﾙﾄﾞします。

・「ﾃﾞﾊﾞｯｸﾋﾞﾙﾄﾞ」は、
「ﾘﾘｰｽﾋﾞﾙﾄﾞ」に比べて、時間がかかりますが、
実行中にｴﾗｰが起きた時に、
その原因を見つけることができます。

・ﾋﾞﾙﾄﾞ中にｴﾗｰが出た場合は、
「出力ｳｨﾝﾄﾞｳ」に、その原因が表示されます。

・これをｸﾘｯｸすると、入力ｶｰｿﾙが
原因と思われる行に移動します。

・実行中に、ﾌﾟﾛｸﾞﾗﾑを途中で止めるには、
止めたい行の左側に、「ﾌﾞﾚｲｸﾎﾟｲﾝﾄ」を付けておきます。

・ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞが表示されている「ﾃｷｽﾄｳｨﾝﾄﾞｳ」の
左側の灰色のところをｸﾘｯｸすると、
「●」が付いたと思いますが、これが「ﾌﾞﾚｲｸﾎﾟｲﾝﾄ」です。

・ﾌﾟﾛｸﾞﾗﾑが、「ﾌﾞﾚｲｸﾎﾟｲﾝﾄ」で止まっている時に、
「**F11**」ｷｰを押すと、**1**行だけ、処理を進めることができます。

・またこの時に、「**F12**」ｷｰを押すと、
次の「ﾌﾞﾚｲｸﾎﾟｲﾝﾄ」に進みます。

・実行時に、変数の値を確認するには、
変数名のところに、ﾏｳｽﾎﾟｲﾝﾀを当てるか、
調べたい箇所を選択します。

・常に値を表示させておきたい場合は、
選択されている状態で、右ｸﾘｯｸをして、
「ｳｫｯﾁ式の追加」をｸﾘｯｸします。

・「ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞ」のｴﾗｰが出たら、
自分が作ったﾍｯﾀﾞﾌｧｲﾙか、ｿｰｽﾌｧｲﾙ上で、
「**stdafx.h**」がｲﾝｸﾙｰﾄﾞされているか、確認して下さい。

・「**0x300**認識できない文字」とかいうｴﾗｰは、
全角ｽﾍﾟｰｽなどの全角文字が原因です。

（全角文字が使えるのは、文字列ﾘﾃﾗﾙとｺﾒﾝﾄのところだけです。）

・構造体など、ﾃﾞｰﾀ型の定義で、**1**ヶ所でも構文ﾐｽがあると、
それを使っているすべての箇所でｴﾗｰが出ます。

・大量のｴﾗｰが出力されるため、面食らいますが、
最初のｴﾗｰから、一つ一つ、潰していけば、
すぐにｺﾝﾊﾟｲﾙできるようになります。

・よくわからない時は、わかるｴﾗｰから潰していきます。

・ｴﾗｰ原因を修正したのに、ﾋﾞﾙﾄﾞできない時は、
「ｿﾘｭｰｼｮﾝのｸﾘｰﾝ」をするか、
「ｿﾘｭｰｼｮﾝのﾘﾋﾞﾙﾄﾞ」をしてみて下さい。

・ｴﾗｰの意味がわからない時は、
ｴﾗｰﾒｯｾｰｼﾞをｺﾋﾟｰして、ﾈｯﾄで検索してみて下さい。

・いろいろ聞いたけど、飲み込めない時は、
ﾉｰﾄを取ったりすると、全体像が見えてきます。

・さてそれでは、下記のｻﾝﾌﾟﾙｺｰﾄﾞをﾋﾞﾙﾄﾞして、実行してみてください。↓

```
/* ｻﾝﾌﾟﾙｺｰﾄﾞ */

#include <stdio.h>

void main( void )
{
   printf( "ﾌﾟﾛｸﾞﾗﾑが実行されました。¥n" );
}
```

・書き方や意味については、次節から説明して行きます。

C言語のﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞをみてみよう

・わりとｼﾝﾌﾟﾙなｺｰﾄﾞなんで、
ｶﾝのいい人なら、わかるかも。^^

```
/* ------------------------- */
/* 足し算の部分を、関数にしてみた。 */

int Tasu( int a , int b )
{
   int kotae = a + b; /*  足し算をして、その結果値を、変数「kotae」に代入している。 */

   return kotae; /* 結果値を、関数の呼び出し元である「main」関数に返す。*/
}

/* ------------------------- */
/* 最初に呼び出される関数「main」*/

int main( )
{
    /*  変数「banana」に、定数(固定)値「10」を代入する。 */
    int banana = 10;

    /* 変数「mikan」に、定数(固定)値「20」を代入する。   */
    int mikan = 20;
```

```
    /* 変数「kotae1」に、計算式「banana+mikan」の結果値「30」を、変数「kotae1」に代入する。*/
    int kotae1 = banana + mikan;

    /* 上の処理を関数にしたものを呼び出し、
       返された結果値「30」を、変数「kotae2」に代入している。 */
    int kotae2 = Tasu( banana, mikan );

    return 0; /* 最初に呼び出された関数「main」から戻るので、ﾌﾟﾛｸﾞﾗﾑも終了する。 */
}

/* ------------------------ */
```

・緑色の部分は、「ｺﾒﾝﾄ」欄です。

・「/*」 と 「*/」 で囲まれた範囲は、ｿｰｽｺｰﾄﾞとは見なされないため、
その行の説明を書いておきます。

・黄土色の部分は、「ﾃﾞｰﾀ型名」で、
変数に代入する数値の許容値や、種類などを決めています。

・ｵﾚﾝｼﾞ色は、「変数名」です。

・変数は、数値を入れて置くお皿のような存在です。

・定数値を代入したり、別の変数から、値を代入したりします。

(「定数」(ていすう)というのは、「10」などのように固定値のことで、
「変数」のように、値を入れ替えたりできません。)

・水色は「関数名」です。

・ﾌﾟﾛｸﾞﾗﾑの文は、この「関数」(かんすう)の中に書いていきます。

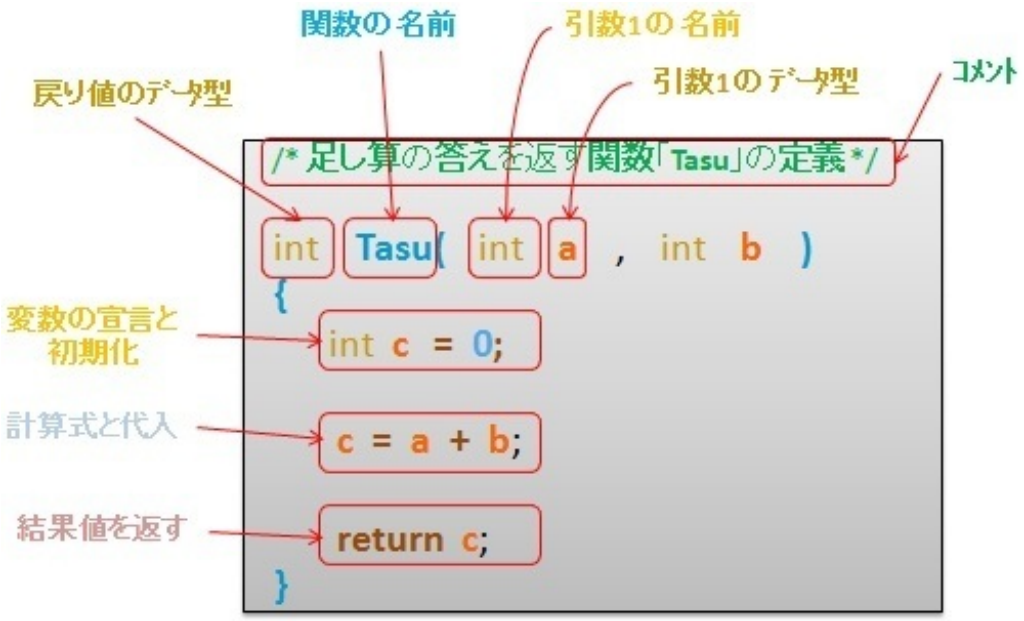
・ﾌﾟﾛｸﾞﾗﾑの行数が長くなって、画面に収まらない場合や、
何度も呼び出す箇所を切り分ける時には、

別の関数を作って、分けて書いていきます。

・ｶｯｺ内の変数は「引数」 (ひきすう) といいます。
関数を呼び出す際に、ここに変数名や定数や数式を指定すると、
その値が引数に代入され、関数内に渡すことができます。

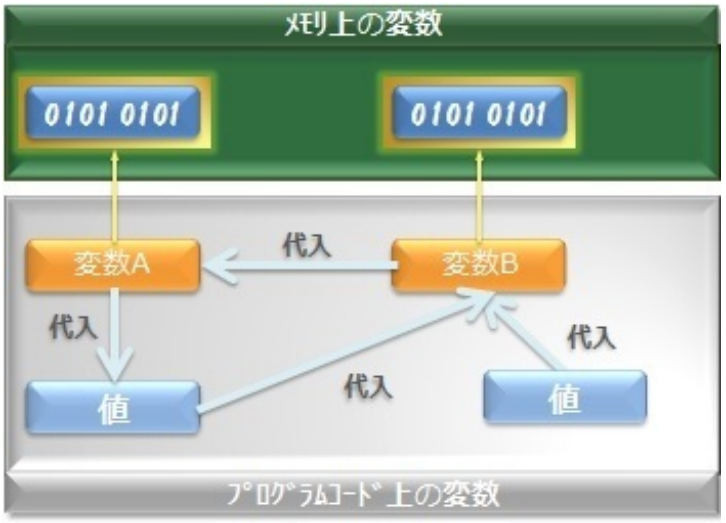
・関数名の左側のﾃﾞｰﾀ型名は、「戻り値」(もどりち)のもので、
このﾃﾞｰﾀ型の値を、呼び出し元の「**main**」関数へと
返すことができます。

・「**return kotae**」と書いてある箇所がそれで、
変数「**kotae**」の持つ値は、呼び出し元の
変数「**kotae2**」に代入されます。



・詳しいことは、あとで説明します。

変数



・「変数」は、
数値を入れて置くお皿
のような存在です。

・方程式でいうところの、
「**x**」とか「**y**」が、
この「変数」です。

・「変数」は、入れ物ですので、
別の値を代入すれば、
その名前が意味する値も変わります。

・方程式っぽく書くと、
x = **3** なら、
y = **x** + **1** という方程式の
解 (答え) である **y** は、 **4** になる。

・さっそく使ってみたいのですが、その前に、
その変数が、「どういう種類の変数なのか」ということを、
ｺﾝﾊﾟｲﾗ様に対して、「宣言」しておく必要があります。

・**C/C++**言語では、

「予約語」といわれる英単語が、あらかじめ登録されていて、
ﾌﾟﾛｸﾞﾗﾏが、それ以外の単語 (ｼﾝﾎﾞﾙ) を作って使用する場合は、
その単語が何を意味するのかを、「宣言」しておかないと
使えないことになっているからです。

・「**nedan**」という名前の変数を宣言してみます。↓

---

int **nedan**;

・ｵﾚﾝｼﾞ色は変数名です。

・黄土色の「**int**」は、ﾃﾞｰﾀ型「**int**」で、
この「**nedan**」という変数が、
『 **+-**符号付きの**32bit**整数値を代入できる変数である。 』
ということを、「宣言」しています。

・ﾃﾞｰﾀ型 (Data Type) というのは、
変数などのﾃﾞｰﾀの種類を表していて、
お皿でいうところの、容量だとか、形状です。

・ちなみに、右端の 「**;**」 (ｾﾐｺﾛﾝ) は、
1 ｽﾃｰﾄﾒﾝﾄの終わりを意味する記号です。

・**1**つの命令文のことを、
「**1** ｽﾃｰﾄﾒﾝﾄ」 (1文) といいます。

int **nedan1;** int **nedan2;** といったぐあいに、
複数のｽﾃｰﾄﾒﾝﾄを、**1**行に並べて書くこともできます。

また、同じﾃﾞｰﾀ型の変数宣言であれば、

```
int nedan1, nedan2;
```

と、ｶﾝﾏで区切って、
まとめて宣言することもできます。

・ところで、上記の変数には、
まだ一度も数値が代入されておらず、
まったく関係のない値が入っています。

・宣言をした時点で、
初期値を代入しておくことを、
「初期化」といいます。↓

```
int nedan = 1980;
```

・上記の例では、宣言をすると共に、
定数値「1980」を代入して、初期化しています。↑

・ﾌﾟﾛｸﾞﾗﾑが複雑になってくると、
代入するのを忘れることが多くなり、
ﾊﾞｸﾞの原因になります。

・初期化は、なるべく行うようにしましょう。

・特に初期化する必要がなさそうな場合でも、
0で初期化しておくようにしたほうが安全です。

・変数に対して、別の変数を代入した場合は、
その別の変数が持つ数値が代入されます。↓

```
int nedan1 = 1980;
 int nedan2 = nedan1; /* newdan2 に、1980 が代入される。 */
```

・数式を代入した場合は、その結果値が代入されます。↓

```
int nedan1 = 1980;
int nedan2 = ( nedan1 + 10 ); /* 1990 が代入される。 */
int nedan3 = ( nedan1 + 20 ); /* 2000 が代入される。 */
```

・上記では、計算式をｶｯｺで囲んでいますが、これは省略できます。

・ｶｯｺを付けると、ここからここまでが計算式であるということが、
ﾊﾟｯと見てわかりますから、付けただけです。

・計算式については、あとで詳しく説明しますが、
計算式から代入されてくる値というのは、
計算式の中の変数の値が、何なのかによって異なる、
ということが、理解できたと思います。

・また、複数の変数に、まとめて代入することもできます。↓

```
nedan1 = nedan2 = nedan3 = 1980; /* すべての変数に、1980が代入される。 */
```

・この場合は、順番に、値が代入されて行きます。

・次回からは、

「ﾃﾞｰﾀ型」について、

くわしく説明していきます。

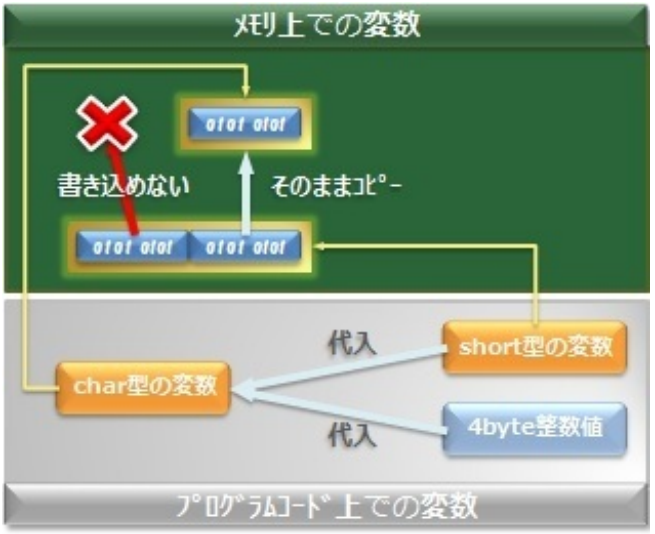
基本ﾃﾞｰﾀ型 (整数)

・変数には、それぞれｻｲｽﾞ (お皿の容量) が決まっていて、
それを決めているのが
宣言の際に書きそえるﾃﾞｰﾀ型でした。

・たとえば、**char**型の変数には、
-128から127までの数値を記憶できますが、
これは、使っているﾒﾓﾘ領域のｻｲｽﾞが
1byteしかないからです。

・このように、変数には、ﾃﾞｰﾀ型によって
それぞれ許容値が決まっていて、
それを超えた値を代入してしまうと、
「ｹﾀの切り捨て」 (※**byte**単位) が起きたり、
ｴﾗｰが表示されたりします。 ↓

```
char hensuu1 = 128;       /* この「128」は、char型の最大値「127」を超えている。 */
short hensuu2 = hensuu1; /* こちらは、「hensuu2」の許容値の方が大きいので、問題ない。
*/
```



・整数型のﾃﾞｰﾀ型は、↓

データ型	サイズ	説明
char	1byte	
short	2byte	
int	4byte	標準整数。
long	4byte	ｺﾝﾊﾟｲﾗによってｻｲｽﾞが異なる。
long long	8byte	VCでは、__int64 。

の、**5**つがあって、それぞれに、
負数が使える「**+-**符号あり整数型」(signed) と、
負数が使えない「**+-**符号なし整数型」(**un**signed) の、
2種類のﾃﾞｰﾀ型があります。

・整数ﾃﾞｰﾀ型名の前に、「 **unsigned** 」を付けると、
「**+-**符号なし整数型」とみなされます。↓

```
unsigned int hensu2 = -1; /* +-符号を記録しないため、負数は代入できない */
```

・「**+-**符号あり整数型」の範囲値が半分なのは、
最上位の**1**ﾋﾞｯﾄに、**+-**符号の有無を記録しているからです。

・整数の定数値は、**int** 型とみなされます。↓(**※32bit**ｼｽﾃﾑのみ)

```
int hensuu1 = 1000;
```

基本ﾃﾞｰﾀ型 (論理値)

・「**bool**」型の変数には、
「**true**」 (真) と 「**false**」 (偽) のうち、
いずれかの値しか、代入できません。 ↓

```
bool kekka1 = true; /* 真 */
bool kekka2 = false; /* 偽 */
```

・この真偽値というのは、実は、**1byte**の整数値で、
「**true**」 は1、「**false**」は0です。

・「**bool**」型の変数に、ｾﾞﾛ以外の値を代入した場合は、
trueとみなされます。 ↓

```
bool kekka1 = 0; /* false とみなされる。 */
bool kekka2 = 1; /* trueとみなされる。 */
bool kekka3 = 2; /* trueとみなされる。 */
```

・「**bool**」型は、関数の戻り値でよく使われています。

・関数内の処理が、成功した場合は、「**true**」が返され、
失敗した場合は、「**false**」が返されるように書きます。

基本ﾃﾞｰﾀ型 (浮動小数点)

・「浮動小数点型」は、
いわゆる小数値を記録する変数のﾃﾞｰﾀ型です。

・「浮動小数点型」として宣言された変数には、
小数点以上の値と、以下の値を記録できます。

「**float**」型　**4byte**
「**double**」型　**8byte**

・この2つの浮動小数点型は、
両方とも、「負数」を扱うことができます。

・「負数」というのはたとえば、「**-1**」などの
0よりも小さい数値のことです。

・また、**0**以上の数値を「正数」といいます。

・通常は、**float**型でも十分なんですが、
小数点以下の高い精度が求められる場合は、
ﾃﾞｰﾀｻｲｽﾞが大きく、有効ｹﾀ数の多い**double**型を使用します。

・整数値同士の計算であれば、
小数点以下は切り捨てなので、
たとえば、**1 ÷ 3** の答えは、**0** になります。

・**1**つのりんごを **3**つに割ったものを、
「**1**つのりんご」とはいいませんよね。

・**0.3**個は、**1**個ではないので、整数では「**0**個」なんです。

・これと同じで、整数型の変数に、

0.3333333333.... を代入しても、
これは1未満ですから、切り捨てられて、0 が代入されます。

・式に、浮動小数点値が含まれる場合は、
すべての項は、浮動小数点型に変換されますから、
1 ÷ 3.0 は、0.33333333333333331 となります。

・CPUは、指定されたﾃﾞｰﾀ型の有効ｹﾀ数の範囲内で、
可能な限り正確な値を算出しょうとします。

・しかし、こうした捉え様のない数値を計算して行くと、
わずかずつですが、誤差が蓄積されていきます。

・割り切れるのは、2の倍数で割った時だけで、
その場合には、計算による誤差は発生しません。

・下記のように、小数値を書いた場合は、
「double」型の値とみなされます。↓

```
double hensu1 = 0.333;
```

・小数値を、float型の値として評価させたい場合は、
数字の後に「f」を付けます。↓

```
double hensu1 = 0.333f; /* これで、float型の定数値とみなされる */
```

・計算速度は、「double」型の方が速いのですが、
ﾃﾞｰﾀｻｲｽﾞを抑える目的から、「float」型を使う場合があるようです。

・浮動小数点の計算は、整数型に比べて、非常に遅く、
特に、乗算と除算が遅いことで知られています。

・**Intel**系の**CPU**に限定してもよいのであれば、
SIMDの組み込み関数などで高速に計算することもできます。

(**AMD**系にも同様の組み込み関数があるようですが、
両方を視野に入れた場合、**CPU**の種類を判定して、
処理を切り替えることになります。)

・その他にも、整数 (固定小数点) で計算しておいて、
浮動小数点に戻す、といった方法もあります。 ↓

```
unsigned int fixed = 12345678; /* これを 12345.678 に見立てる。 */

unsigned int upper = fixed / 1000;  /* 12345 */
unsigned int under = fixed % 1000; /* 678 */
```

・浮動小数点型については、次のﾍﾟｰｼﾞと、
「型ｷｬｽﾄ」のところで、もう少し詳しく説明します。

浮動小数点の仕組み

・一般的なﾌﾟﾛｾｯｻでは、
　実数 (小数点以下を含む数値) を表現するために、
　「浮動小数点」 (※ 「IEEE754」準拠の形式)　を使用しています。

・たとえば、「固定小数点」のように、
　単純に、整数部と、小数部とに分けて記録した場合は、↓

整数部　　小数部
0000 . 0000

・片方の部分のｹﾀが大きいと、収まりきらないだとか、
　逆に、もう片方のｹﾀが小さいと、無駄になる、
　といったことが起きます。↓

　0.123123... (整数部は未使用だが、小数部が延々と続く。)
111111.0　(整数部のｹﾀが多いが、小数部が未使用。)

・「浮動小数点」では、小数点の位置をｽﾞﾗすことで、
　より多くの有効なｹﾀを記録することができ、
　こうしたﾑﾀﾞ を、少なくすることができます。

・ｹﾀ数が多いということは、切り捨てられるｹﾀが少なくなるため、
　より正確な、つまり精度の高い計算結果を求めることができます。

・さて、「浮動小数点」型のﾃﾞｰﾀは、
　上位ﾋﾞｯﾄから順に、符号部、指数部、仮数部という、
　連続する**3**つのﾌｨｰﾙﾄﾞからなります。↓

データ型		符号部	指数部	仮数部	合計
単精度	float	1bit	8bit	32bit	4byte
倍精度	double	1bit	11bit	52bit	8byte

・以前、ﾃﾞｰﾀ型の有効範囲を説明したﾍﾟｰｼﾞで、
「**-1.23E+03**」といった書き方がありましたが、
これは、「**-1.23×2**の**3**乗」を意味しています。

・最初の、「-」(ﾏｲﾅｽ) の部分は、いわゆる「符号」であり、
整数型のﾍﾟｰｼﾞで説明した「符号ﾋﾞｯﾄ」と同じで、
符号部 (最上位の1ﾋﾞｯﾄ) に記録されます。

・これは負数であれば1、正数であれば0になります。

・さて、浮動小数点では、数値を、**2**進数であつかいます。

・仮数部には、「仮数」を記録するのですが、
これは、ﾋﾞｯﾄが**1**になる最上位のｹﾀが、
最上位になるように、ｹﾀを上げ下げしたものです。

例） 00001111 → 11110000

・これを「正規化」といいます。

・たとえば、**10**進数表記の10.0 を正規化する場合は、
2進数表記だと、1010.0 ですから、
これを正規化すると、**3**ｹﾀ くり上がるので、1.01 になります。

・このとき、最上位ｹﾀは、必ず**1**になるため省略し、
それ以外の下のｹﾀが、仮数部に記録されます。

・そして、この時に、上げ下げさせたｹﾀ数が「指数」です。

・**1**ｹﾀ下げる場合は、指数は**1**、
1ｹﾀ上げる場合は、指数は**-1**となります。

・実際には、**float**の場合は、これに**+127**、
doubleの場合は、**+1023** したものを
指数部に記録します。

・これはなぜかというと、指数が負数になる場合、
指数の符号を記録する符号ﾋﾞｯﾄが無いため、
そのままでは記録できないからです。

・割り切れない数値となった場合は、
小数点以下の値が、延々と続くのですが、(循環小数)
仮数部には収まりきらず、下ｹﾀが切り捨てられるため、
計算していく中で、誤差が蓄積されていくのです。

・その他にも、「**NaN**」 (非数) という、特殊な数があります。

・ﾌﾟﾛｾｯｻの動作が未定義であるので、「未定義値」 ともいわれています。

・「**NaN**」は、非数の総称であって、同じであるとは限らないので、
比較しても、一致せず、比較結果は、常に **false** となります。

・一般的に、有用な数値 (使い道のある数字) は、**+-**の正規有限数で、
前述した方式で浮動小数点値に変換され、記録されます。

・しかし、それ以外の特殊な数を記録する際には、
少し特殊な形式で、記録されます。↓

値の種類	符号部	指数部	仮数部	説明
-0	1	0	0.000...	
+0	0	0	0.000...	
-非正規有限数	1	0	0.***...	小数部分は非ゼロ
+非正規有限数	0	0	0.***...	小数部分は非ゼロ
-正規有限数	1	1～254	1.***...	
+正規有限数	0	1～254	1.***...	
-無限	1	255	1.000...	
+無限	0	255	1.000...	
SNaN	-	255	1.0**...	符号bitは無視
QNaN	-	255	1.1**...	符号bitは無視

(※上図の中の指数部は、**float**の場合であり、
数値は、すべて10進数で表記しています。)

定数 (const)

・「定数」というのは、名前の通りで、
　定まった数のことであり、
　その値は、変数のように変更できたりはしません。

・「定数」は、宣言する時に、必ず初期化しておきます。↓

```
    /* 円周率は不変なので、定数として宣言し、変更できなくする。↓*/
  const double PI64  = 3.14159265358979323846264338327950288419716939937751;
```

・このように、ﾃﾞｰﾀ型の左横に、「**const**」と書き添えておけば、
　その変数は、「定数」とみなされ、値が変更できなくなります。

・「**const**」が無かった頃は、
　「**#define**」を使って、
　ｺﾝﾊﾟｲﾙ時に、数値を置換していました。↓

```
/* ﾏｸﾛ定数を、定義する。 */
#define PI64 3.14159265358979323846264338327950288419716939937751

/* ﾏｸﾛ定数を、無効にする。 */
#undef PI64
```

・この方法は、現在でも有効です。

・数値をそのまま書いている箇所は、
　ﾏｸﾛ定数にしておくと便利です。

・そうすれば、
あとで数値を修正することになった時に、
書き直しをする箇所が一箇所で済みますし、
数値の打ち間違いによるﾊﾞｸﾞも発生しません。

・ただし、#defineの定数を用いる場合には、
ﾃﾞｰﾀ型について注意が必要です。

宣言の種類	データ型
#define 定数名リテラル	リテラルのデータ型
const データ型　定数名;	指定されたデータ型

・それから、数値には、
次のような書き方があります。↓

```
a = 088;   /* 8進数表記 */
a = 0xFF; /* 16進数表記 */
a = 100l;  /* long値 */
a = 100ul;  /* unsigned long値 */
a = 1.5l;   /* long double値 */
a = 100u; /* unsigned int値 */
a = 1.5f;   /* float値 */
```

配列

・「配列」は、
同じﾃﾞｰﾀ型の変数を、
連続して並べたものです。

・たとえば、「**int**」型の変数を、
3つ並べる配列「**hairetsu1**」の宣言は、↓

```
int hairetsu1 [3];
```

・配列に含まれる個々の変数のことを、
「要素」と言います。

・「要素」に、値を代入するには、
下記のように、「要素番号」を指定します。↓

```
hairestu1[0] = 0;
hairestu1[1] = 1;
hairestu1[2] = 2;
```

・最初の要素番号は「**0**」番で、

最後の要素番号は 「 要素数 - 1 」番です。

・配列の初期化は、下記のように書きます。 ↓

```
int hairetsu1 [] = { 0, 1, 2 }; /* 要素数は省略できる。書いてもいい。 */
```

・すべての要素値を、「0」で初期化する場合は、
次のように、初期値を省略して書きます。 ↓

```
int hairetsu1 [] = { };
```

・それと、配列は、
別の配列に代入することはできません。

・値をまとめて移し替える場合は、
for文の中で、一つ一つ代入していくか、(後述)
memcpy関数などで、まとめてｺﾋﾟｰします。(後述)

多次元配列

・「**2**次元配列」は、
配列を、さらに並べたものです。↓

```
int tensuu2 [3][2]; /* 「2次元配列」の宣言。 */
```

・上記の、**2**次元配列「**tensuu2**」は、
ある田舎の学校の、ｸﾗｽごとの点数表です。

・この学校には、**3**学年まであり、
学年ごとに、**2**ｸﾗｽがあります。

・当初は、**1**学年分だけでよかったため、
要素数**2**つの**1**次配列でした。↓

```
int tensuu1 [2];
```

・この配列を、さらに配列にするわけですから、つまり、
tensuu2[**3**] の各要素には、 **tensuu1**[**2**] が入っている
というか、並んでいるわけです。↓

---

[0][0] [0][1]　[1][0] [1][1]　[2][0] [2][1]

・それぞれの色が、「1次配列」1つ分で、
各要素の左側の番号が「2次配列」の「要素番号」、
右側の番号が「 1 次配列」の「要素番号」です。

・話を戻して、2次元配列「tensuu2」 から、
「2年1組」の点数を代入するには、次のように書きます。 ↓

---

int a = tensuu2[1][0];

・要素番号は、0から始まっているため、-1 ｽﾞﾚています。

・つまり、最初の[1]が「2年」で、次の[0]が「1組」です。

・左側の要素番号が配列の番号で、
右側の要素番号が、その要素の番号です。

・さらに、全学年を含めた生徒ごとの点数表を作る場合は、
「3次元配列」となります。 ↓

---

int tensuu3[3][2][5]; /* 3学年2ｸﾗｽ、最大5人の生徒がいる、とする。 */

・最も生徒数が多いｸﾗｽは5名ですので、
要素数は、5にしてあります。 ↑

・この「3次元配列」に、
2年1組5番の生徒の点数を代入するには、 ↓

```
tensuu2[1][0][4] = 100;
```

・多次元配列を初期化する場合は、次のように書きます。↓

```
int tensuu2[][2] = { { 0, 1 } , { 0, 1 } , { 0, 1 } };

int tensuu3[][2][5] =
                    {
                      { {0,1,2,3,4}, {0,1,2,3,4} } ,
                      { {0,1,2,3,4}, {0,1,2,3,4} },
                      { {0,1,2,3,4}, {0,1,2,3,4} }
                    };
```

・宣言時の要素数は、最初の次元だけ、省略できます。↑

・中ｶｯｺ内の要素数が足りない場合は、
残りの要素は、「0」で初期化されます。

ポインタ

・「ﾎﾟｲﾝﾀ」は、
ﾒﾓﾘ上のｱﾄﾞﾚｽ(番地)を
入れておく専用のお皿です。

・「ﾒﾓﾘｱﾄﾞﾚｽ」は、**32bit**ｼｽﾃﾑでは、
4byteの整数値です。

・ちなみに、**64bit**ｼｽﾃﾑでは、
8byteの整数値です。

・まあですから、ｱﾄﾞﾚｽを代入できる変数、
変数の一種だと考えるとｲﾒｰｼﾞしやすいので、
本書では、「ﾎﾟｲﾝﾀ変数」とも書いていますが、
少し変わった性質を持っています。

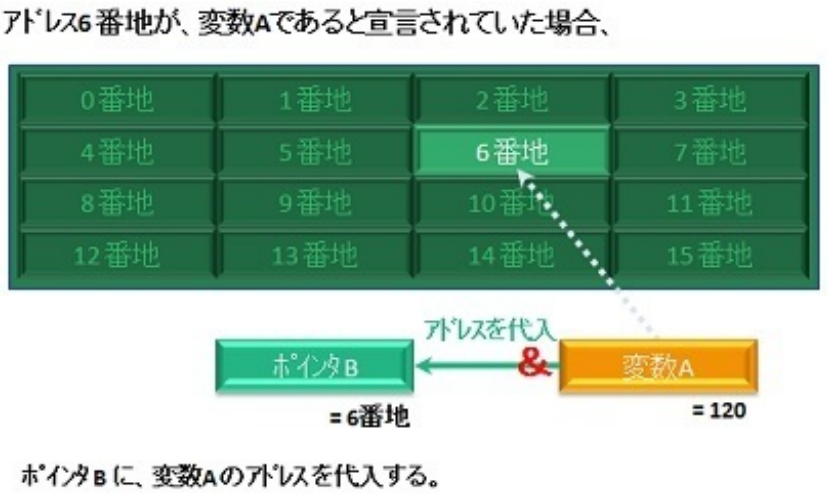
・さて、ﾎﾟｲﾝﾀに「ﾒﾓﾘｱﾄﾞﾚｽ」を代入するときには、
「魔法の記号」を使います。

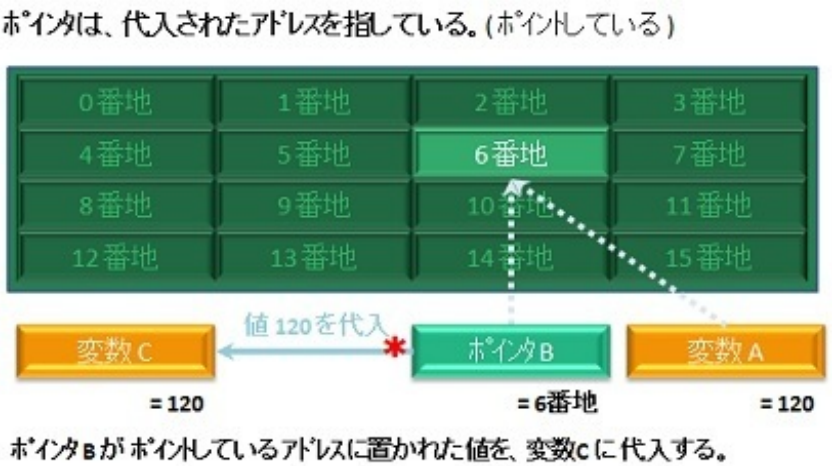
・下記の例では、変数「**hensu1**」のｱﾄﾞﾚｽを、
ﾎﾟｲﾝﾀ「**p_hensu1**」に代入しています。↓

```
int hensu1 = 100;        /* 変数「hensu1」を、定数値「100」で初期化する。 */
int* p_hensu1 =NULL;  /* 無効なｱﾄﾞﾚｽを意味する定数「NULL」で初期化する。 */
p_hensu1 = &hensu1;    /* 変数「hensu1」のｱﾄﾞﾚｽを代入する。 */
```

・このように、変数のﾒﾓﾘｱﾄﾞﾚｽを代入するには、
変数名の左側に、「**&**」を付けます。↑

・それと、もうひとつ注目してほしいのは、
ﾎﾟｲﾝﾀ変数の型名の右横についている「*」です。





・ﾒﾓﾘ領域のｱﾄﾞﾚｽは、4byteの整数値ですが、
int型ではありません。

・では、専用のﾃﾞｰﾀ型があるのかというと、
・「これがﾎﾟｲﾝﾀ型だ」というものは、明確には決まっておらず、
ﾎﾟｲﾝﾀを宣言する時に、ﾃﾞｰﾀ型名の右横に
「*」が付いたら、ﾎﾟｲﾝﾀとなります。↓

```
int* p_hensu1 =NULL;  /* 無効なｱﾄﾞﾚｽを意味する定数「NULL」で初期化する。 */
```

・ﾎﾟｲﾝﾀは、必ず 定数「NULL」で初期化します。

・これは、ｱﾄﾞﾚｽが代入されているか、いないかを、
NULLと比較して判定するためです。

・さいしょに、NULL で初期化をしていないと、
値が正常なｱﾄﾞﾚｽなのか、
無関係な値なのか、判別できないからです。

・しかし、このNULL定数の値は、
一般的には、「整数値のｾﾞﾛである」 と定義されていることが多いんですが、
ごくまれに、「-1 である」 と定義されている場合もあるようで、
0 と書いてはいけません。

・いずれにしても、このNULLというのは、整数値ですから、
整数値として使用されているのか、
ｱﾄﾞﾚｽ値として使用されているのかが、区別できず、
まぎらわしい、ということで、さいきんでは、
「nullptr」という定数が、使われるようになりました。

・これは整数値ではなく、
明確に無効なｱﾄﾞﾚｽを指している「ﾇﾙﾎﾟｲﾝﾀ」
であることを示す定数です。

・さて、次の例では、
先ほどのﾎﾟｲﾝﾀ「p_hensu1」から、

変数「hensu1」の値にｱｸｾｽし、
別の変数「hensu2」に代入しています。↓

```
int hensu2 = *p_hensu1; /* 変数「hensu1」の値が代入される。 */
*p_hensu1 = 200;        /* 変数「hensu1」に、200が代入される。 */
```

・ﾎﾟｲﾝﾀは、左側に「*」を付けると、
代入したｱﾄﾞﾚｽの位置にある変数に、
なりすますことができます。

・さて、次の例では、
配列「hairetsu1」の要素[0]のｱﾄﾞﾚｽに、+1することで、
次の要素[1]の値にｱｸｾｽしています。↓

```
int hairetsu1 [] = { 0, 1, 2 };
int* p_hairetsu1 = hairetsu1 + 1;
int a = *p_hairetsu1;
```

・配列「hairetsu1」の前には、「&」が付いていません。↑

・なぜこれで要素[0]のｱﾄﾞﾚｽがとれるのかといえば、
「配列」というのは、配列の先頭ｱﾄﾞﾚｽだからです。

・下記の2つのｽﾃｰﾄﾒﾝﾄは、どちらも同じことをしています。↓

```
int a = hairetsu1[1];
int b = *( hairetsu1 + 1 );
```

・ｶｯｺで囲んでいるのは、
「*hairetsu」だと、「要素[0]の値」という意味に
なってしまうからです。

・つまり、ｶｯｺ内の 「 hairetsu1 + 1 」 は、
要素[0]の位置 + 1 の ﾒﾓﾘｱﾄﾞﾚｽであり、
頭に「*」を付けることで、その位置の値となります。

・逆に、ﾎﾟｲﾝﾀを、配列として使うこともできます。 ↓

---

int a = p_hairetsu1[1];

ﾎﾟｲﾝﾀ・ﾎﾟｲﾝﾀ

・「ﾎﾟｲﾝﾀ」は、「変数のﾒﾓﾘｱﾄﾞﾚｽ」を
　入れておくための「専用のお皿」でした。

・ということは、「ﾎﾟｲﾝﾀのﾒﾓﾘｱﾄﾞﾚｽ」も
　代入できるのでしょうか。↓

```
int hensu1 = 100;
int* p1 = &hensu1;
int* p2 = &p1;
int hensu2 = *p2;
```

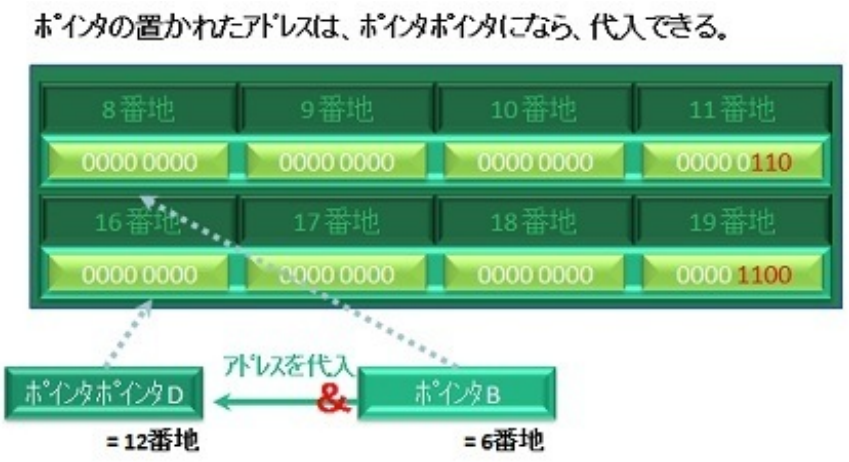
・おそらく、次のようなｴﾗｰが出て止まったはずです。↓

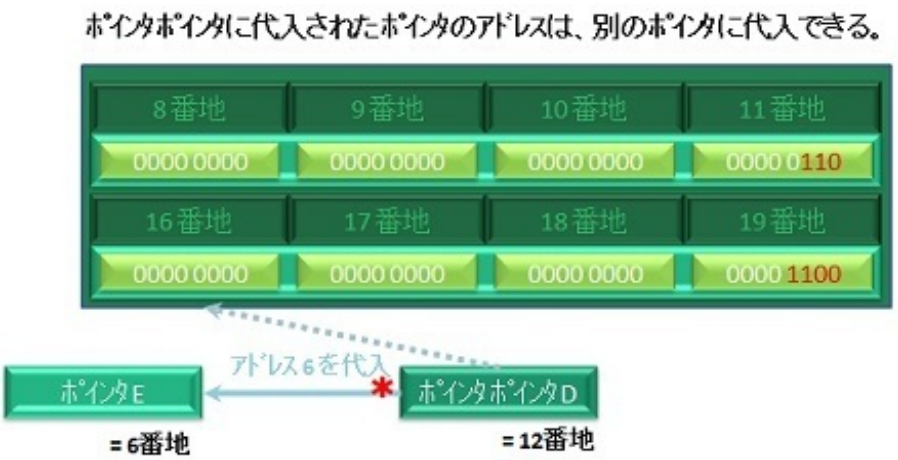
「'int **' から 'int *' に変換できません。」

・「ﾎﾟｲﾝﾀのﾒﾓﾘｱﾄﾞﾚｽ」を代入する時は、
「ﾎﾟｲﾝﾀ・ﾎﾟｲﾝﾀ」を使います。 ↓

```
int hensu1 = 100;
int* p1 = &hensu1;
int** pp1 = &p1;
int hensu2 = **pp1;
```

・ﾎﾟｲﾝﾀﾎﾟｲﾝﾀ は、ﾎﾟｲﾝﾀの番地 (ｱﾄﾞﾚｽ) を ﾎﾟｲﾝﾄするものです。





・ようするに、変数の頭に、＆が付いているときは、
「その変数のあるｱﾄﾞﾚｽ」 として評価され、
ﾎﾟｲﾝﾀの頭に、＊が付いているときは、
「そのﾎﾟｲﾝﾀが指す変数に配置された値」
として評価されるということです。

・これがどういう場合に役に立つのかは
そのうち述べることとして、
ここからは、少しﾏﾆｱｯｸな書き方について
触れておこうと思います。

・以前にも少し書きましたが、
「配列」は、「**const** ﾎﾟｲﾝﾀ」のように
使うことができました。

・ということは、「**2次元配列**」 も、
「**const** ﾎﾟｲﾝﾀ・ﾎﾟｲﾝﾀ」 と同じように
使うことができるのでしょうか。↓

```
int hairetsu1[2][2] = { { 10 , 20 } , { 100 , 200 } };

int** pp1 = hairetsu1;

int hensu1 = **pp1;
```

・おそらく、次のようなｴﾗｰが出て、止まったはずです。↓

「'int [2][2]' から 'int **' に変換できません。」

・「 (int**) hairetsu1 」と書けば、代入はできますが、
「 **pp1 」　と書いて、値を取り出そうとすると、
「ｱｸｾｽ違反ｴﾗｰ」が出て、止まります。

・しかし、次のような書き方はできます。↓

```
int* p1    = hairetsu1[0];
int  hensu1 = *p1;
```

```
int hensu1 = **hairetsu1;
```

```
int* p1    = (int*) hairetsu1;
int  hensu1 = *p1;
```

・いずれの場合でも、変数 「hensu1」 には、
2次元配列 「hairetsu1」の要素[0][0] の
値 「10」 が代入されます。

・2次元配列は、ﾎﾟｲﾝﾀ・ﾎﾟｲﾝﾀのように書くことができます。↓

```
int hensu1 = *(*hairetsu1 + 1);  /* 20が代入される */
  int hensu2 = hairetsu1[0][1];     /* 20が代入される */
```

```
int hensu1 = **(hairetsu1 + 1);  /* 100が代入される */
  int hensu2 = hairetsu1[1][0];     /* 100が代入される */
```

型ｷｬｽﾄ (整数)

・ﾃﾞｰﾀ型が異なる変数に、値を代入すると、
「型ｷｬｽﾄ」(値のﾃﾞｰﾀ型を変換する処理) が行われます。

(※ 正式な呼称は、「型変換」(**type cast**) で、これは俗称です。)

・受け取る側の変数の方が、ｻｲｽﾞが小さい場合は、
ﾃﾞｰﾀはそのままｺﾋﾟｰされるので、上位ﾊﾞｲﾄが削られます。 ↓

```
unsigned short hensu1 = 65295;
unsigned char  hensu2 =hensu1;
```

・ 最初の「**hensu1**」は、
65295 (0x**FF0F**) で初期化しています。

・そして、次の行で代入しているんですが、
受け取る側の変数「**hensu2**」は、
「**unsigned char**」型の値を持つ変数で、
その最大値は、**255** (0x**FF**) です。

・気になるのが代入後の値ですが、
最大値である **255** (0x**FF**)
... ではありません。

・答えは**15** (0x**0F**) です。

・つまり、下位**1**ﾊﾞｲﾄ目が、
そのままｺﾋﾟｰされたわけです。

・では次に、**+-**符号なしの値を、
+-符号ありの変数に代入するとどうなるでしょうか。 ↓

```
unsigned char hensu1  = 128;
char hensu2 = hensu1;
```

・1行目の 変数「hensu1」は、
「128」で初期化されています。

この値は、「char」型の最大値「127」を、
わずか 1 ｵｰﾊﾞｰした値です。 ↑

・気になる代入後の値ですが、
「char」型の最小値の -128 になります。

・これはﾅｾﾞなのかというと、
+-符号がある場合は、+なのか、-なのかを
最上位1ﾋﾞｯﾄで記録しているからです。

・「char」型の最大値は、127 ですが、
これは、2進数表記で書くと、 0111 1111で、
最上位1ﾋﾞｯﾄだけが 0 なのがわかります。

・負数(ﾏｲﾅｽ)になった時は、この0が1になります。

・代入値の128は、2進数表記で書くと、
1000 0000 です。

・これは+-符号付きの考え方では、
「符号がﾏｲﾅｽで、数字が最小値」 という意味になります。

・受け取り側のﾊﾞｲﾄｻｲｽﾞが同じか、または大きい場合は、
値は破壊されずに、そのまま代入されます。

・こうしたﾑﾘな代入をしていると、
ｺﾝﾊﾟｲﾙ時にｴﾗｰや警告が出ることがあります。

・ﾃﾞｰﾀ型が異なる変数に代入する場合は、
ｻｲｽﾞに、ゆとりがある場合でも、
型ｷｬｽﾄを行うように、次のように書いておく必要があります。↓

---

char **hensu2** = **(char)** **hensu1**;

・代入する値の左横に、受け取り側のﾃﾞｰﾀ型を
ｶｯｺで囲って書いています。

・これを 「明示的な型ｷｬｽﾄ」 といいます。

※これを書いたからといって、
上で紹介したような危険な型ｷｬｽﾄが
補正されるわけではありません。

・話が長くなりましたが、まとめると、↓

・大きいｻｲｽﾞの値を、小さいｻｲｽﾞの変数に
代入するのはやめよう。 (小 = 大 はﾀﾞﾒ!)

・大きいｻｲｽﾞの変数に代入する時でも、
明示的に型ｷｬｽﾄしょう。 (大 = (大) 小)

・ｻｲｽﾞが小さい変数への代入は、原則的にさけた方がいいんですが、
値が許容範囲であれば、上位ｹﾀが切り捨てられても
値が破壊されることはありません。

・さて、**unsigned** 値を引いていると、負数になるときがあります。

・もちろん、負数としては記録されず、**unsigned** 値ですので、
正数のおかしな数字になってしまい、このままでは使えません。

・そんなときは、**signed** 値にｷｬｽﾄすれば、負数に戻ってくれます。↓

```
unsigned int  v1 = 0U;
unsigned int  v2 = 2U;
int           v3 = v1 - v2; // v3 = (int) ( v1 - v2 ) と書いた方がよい。
```

・さて、上記の3行目ですが、式の評価値は、**unsigned** 値のままです。

・しかし、代入された値は、ｷｬｽﾄされていますので、**signed** 値となり、
「-2」となります。

型ｷｬｽﾄ (浮動小数点)

・浮動小数点型の変数に、
整数値を代入してみます。 ↓

```
float hensu1 = (float) 10; /* 「10.0」 が代入される。 */
```

・今度は逆に、整数型の変数に、
浮動小数点値を代入してみましょう。 ↓

```
unsigned int hensu1 = (unsigned int) 1.9999999f; /* 「1」 が代入される。 */
```

・小数点以下の 「0.999999...」 は 1未満 ですから、
切り捨てられて、「1」 が代入されます。

・ということは、
この小数点以上の値 「1」 を引いてしまえば、
小数点以下の値だけになるはずです。 ↓

```
float hensu1 = 1.9999999f;
float hensu2 = hensu1 - (int) hensu1;
```

・ここで 「型ｷｬｽﾄ」 を忘れてしまうと、値が壊れます。 ↓

```
char hensu1 =  (char)  -128;
```

```
char hensu2 = -128;
```

・変数「hensu1」の場合は、
型ｷｬｽﾄをしているので、-128が代入されます。↑

・しかし、変数「hensu2」の場合は、
下位1byteが、そのままｺﾋﾟｰされますので、
破壊された値が代入されます。

・「ftoi.asmが無い」とかいうｴﾗｰが出た場合は、
型ｷｬｽﾄの代わりに、下記の関数を使って下さい。↓

```
int ToInt32( float value_ )
{
    const int b[ 2 ] = { 0xBEFFFFFF, 0x3EFFFFFF };
    int i;

    __asm
    {
      fld    value_;
      mov    ecx, value_;
      shr    ecx, 29;
      and    ecx, 4;
      fadd   dword ptr [ b + ecx ]
      fistp  i;
    }

    return i;
}
```

・この関数を使った場合は、こうなります。↓

```
float hensu1 = 1.9999999f;
float hensu2 = hensu1 - ToInt32( hensu1 ); /* 整数部を引いて、小数部を抽出する。 */
```

・代入された値は、「0.9999999999....」 とはならず、
最終ｹﾀ付近で丸められて「0.99999905」 になったと思います。

・これがこの「folat」型の限界です。

・ちなみに、「math.h」の関数を使って、小数点以下を求める場合は、↓

```
float hensu2 = hensu1 - floorf( hensu1 );
```

・「floorf」関数は、
渡された浮動小数点値の
整数部を返します。

・ようするに、これは「切り捨て」で、
「切り上げ」で求める場合には、
「ceilf」関数を使います。

・「四捨五入」する場合は、↓

```
float hensu2 =  floorf( hensu1 + 0.5f );
```

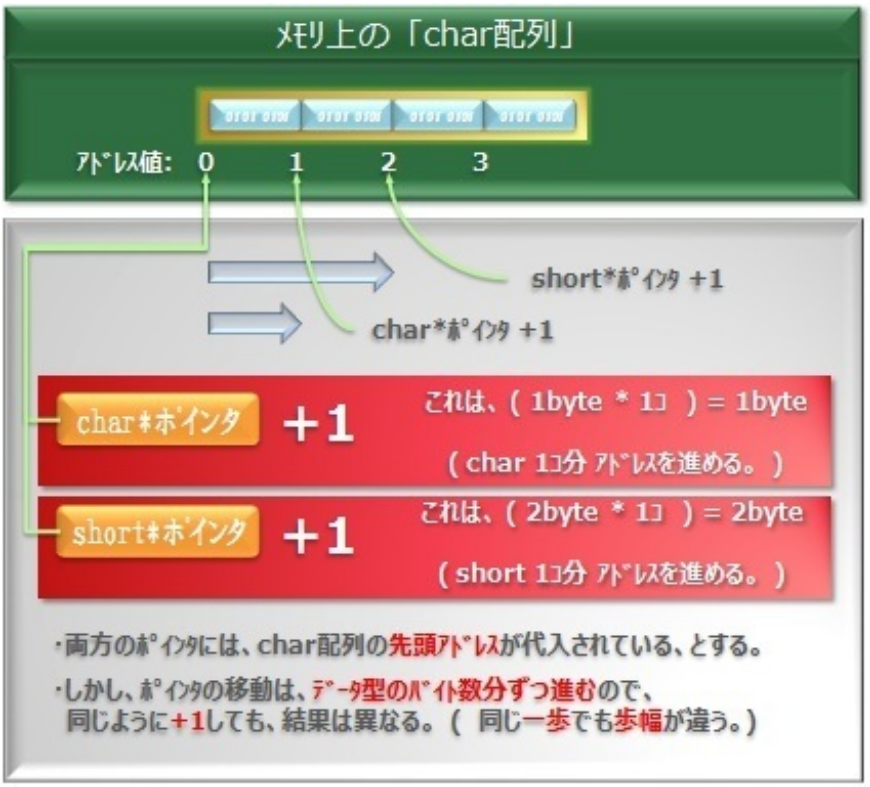
・「double」型の場合は、
関数名の、最後の「f」を省略します。

・それから、代入された時の切り捨てと、
計算式や切り捨て関数による切り捨てとは
方式が異なるため、結果値に違いがでます。

・浮動小数点型の値を比較する場合は、
どのくらいまで誤差を許すのか、
許容値を決めて比較します。↓

```
 /* 絶対値 (+-符号なしの値) の差が、許容値よりも小さいなら、 */
if ( abs( hensu1 - hensu2 ) < 0.000001 )
{
    /* ほぼ同じ値です。( 差は、「0.000001」よりも小さい。 ) */
}
```

型ｷｬｽﾄ (ﾎﾟｲﾝﾀ型)



・実際のところ、この「型ｷｬｽﾄ」(type cast) は、
ﾎﾟｲﾝﾀ型を変換するときに、よく使います。 ↓

```
unsigned int hairetsu1 [] = { 0x11223344, 0x55667788 }; /* 4byte値の配列を宣言する。 */

unsigned int* p1 =  hairestu; /* 配列のｱﾄﾞﾚｽを代入する。 ( 配列なので、 「&」 は不要。 ) */

unsigned char* p2 = (unsigned char*) p1; /* ﾎﾟｲﾝﾀ型を、 型ｷｬｽﾄして代入する。 */

unsigned char a = p1 + 3; /* ﾎﾟｲﾝﾀを、 3つ進める。 ( 1byteの3つ分だけ進む。 ) */

/* 変数「a」には、 「0x44」 が代入される。 ( 要素0の 下位1byte ) */
```

・最後の行の、変数「a」には、
「68」 (0x44) が代入されます。

・この数字は、配列「hairetsu1」の

下位1ﾊﾞｲﾄ目を、そのままｺﾋﾟｰしたものです。

・「unsigned int」型のﾎﾟｲﾝﾀ「p1」に+1すると、
ｱﾄﾞﾚｽは4ﾊﾞｲﾄ進みますが、
「unsigned char」型のﾎﾟｲﾝﾀ「p2」に+1した場合は、
1ﾊﾞｲﾄしか進みません。

・ﾎﾟｲﾝﾀで計算をすると、
ﾎﾟｲﾝﾀが指しているｱﾄﾞﾚｽが、
単純に、足したり、引いたりされる、わけではなく、
ﾃﾞｰﾀ型のｻｲｽﾞ分が、足したり、引いたり、されるのです。

・乗算した場合も、
ﾃﾞｰﾀ型のｻｲｽﾞ分 * 乗じた数 のｱﾄﾞﾚｽ になります。

・ﾒﾓﾘｱﾄﾞﾚｽは、4byteの整数値ですから、
「unsigned int」型の変数に代入することもできます。↓

```
/* ｱﾄﾞﾚｽを、整数値に型ｷｬｽﾄして代入する。 */
unsigned int hensuu1 = (unsigned int) p1;
```

・APIだとかの関数だと、「unsigned int」型の引数で
ｱﾄﾞﾚｽを渡せたりできたんですが、この方法は現在は
あまり推奨されていません。

・以前にも少し書きましたが、64bitｼｽﾃﾑでは、
標準整数とﾒﾓﾘｱﾄﾞﾚｽのｻｲｽﾞが、
従来の倍の64bit (8byte) に変更されているからです。

・ﾎﾟｲﾝﾀを演算する際に、気をつけていただきたいのは、
式を必ずｶｯｺで囲み、ｷｬｽﾄをするということです。

・そうしないと、ﾋｰﾌﾟﾒﾓﾘが破壊されてしまい、
これ以降の領域確保で失敗したり、
保護ｴﾗｰが出る危険性があります。

・そうなった場合には、ﾊﾞｯﾌｧｵｰﾊﾞｰﾗﾝが原因の可能性もありますが、
ｶｯｺとｷｬｽﾄが抜けていないかを確認してください。

・同じﾃﾞｰﾀ型への代入であっても、ﾎﾟｲﾝﾀ演算を代入する場合は、
ｷｬｽﾄをするようにして下さい。

ﾏﾙﾁﾊﾞｲﾄ文字 (char)

・C/C++の文字列は、「char」型の配列です。 ↓

```
char moji1 ='A';  /* 半角文字は、1ﾊﾞｲﾄ文字なので、「char」型1つで記憶できる。 */

char moji2 = 'あ';  /* この全角文字は、2ﾊﾞｲﾄのため、入りきらない。 */
```

```
char namae [] = "ハム太"; /* 文字列は、「char」型の配列変数に記憶する。 */

char*p_namae = "ハム太"; /* ﾎﾟｲﾝﾀ定数として宣言し、初期化することもできる。 */
```

・まず、1行目ですが、
　半角文字「A」で初期化しています。

・このように、1文字ずつ代入する時は、
　「'」(ｼﾝｸﾞﾙｸｫｰﾃｰｼｮﾝ)で囲みます。

・次に、2行目ですが、
　全角文字「あ」で初期化しています。

　しかし、この全角文字は、2ﾊﾞｲﾄであるため、
　切り捨てられてしまいます。

・「char」型で文字をあつかう場合は、
　各ﾊﾞｲﾄの値は、「S-JIS (cp932) ｺｰﾄﾞｾｯﾄ」の
　「文字ｺｰﾄﾞ」とみなされます。

・このｺｰﾄﾞｾｯﾄの文字ｺｰﾄﾞは、
　「ﾏﾙﾁﾊﾞｲﾄ文字」といわれるもので、
　使用するﾊﾞｲﾄ数が、文字によって異なります。

・3行目は、文字列定数 (文字列ﾘﾃﾗﾙ) を代入して
「char」配列を初期化していますが、
全角の1文字が、複数ﾊﾞｲﾄのため、
1要素につき1文字とは、なっていません。

・さいごの4行目では、
ﾎﾟｲﾝﾀ定数を、文字列定数で初期化しています。

・この場合、ﾎﾟｲﾝﾀ定数が持っているｱﾄﾞﾚｽは、
文字列の先頭ｱﾄﾞﾚｽであり、いいかえると、
1文字目の1ﾊﾞｲﾄ目のｱﾄﾞﾚｽとなります。

・誤って、途中のｱﾄﾞﾚｽを渡してしまうと、
半角文字とみなされ、表示が化けます。

・全角文字の最初の1ﾊﾞｲﾄのことを、
「ﾘｰﾄﾞﾊﾞｲﾄ」といいます。

・「cp932」のｺｰﾄﾞ表を見てわかるように、
0x80、0x90、0xE0、0xF0 のいずれかが
「ﾘｰﾄﾞﾊﾞｲﾄ」となります。

・「mbstring.h」には、「ﾘｰﾄﾞﾊﾞｲﾄ」を判定するための
「_mbsbtype」関数が用意されています。 ↓

```
bool is_lead = ( _mbsbtype( p_namae, i ) == 1 );
```

・この関数の戻り値は、次の4つうちのいずれかです。 ↓

定数名	値	意味
_MBC_SINGLE	0	ｼﾝｸﾞﾙﾊﾞｲﾄ文字。 (半角文字)
_MBC_LEAD	1	ﾏﾙﾁﾊﾞｲﾄ文字の先頭1ﾊﾞｲﾄ。 (cp 932では、0x81 ～ 0x9F or 0xE0 ～ 0xFC)
_MBC_TRAIL	2	ﾏﾙﾁﾊﾞｲﾄ文字の残りの数ﾊﾞｲﾄ中の 1ﾊﾞｲﾄ。
_MBC_ILLEGAL	-1	不正な文字ｺｰﾄﾞ。

・文字列を関数に渡す場合は、
終端に、「NULL文字」を入れておきます。↓

```
char* p_namae = "ハム太\0";
```

・ﾎﾟｲﾝﾀ定数を使用する場合は、
文字列を書き換えることができません。

・配列「namae」の要素には代入できますが、
ﾎﾟｲﾝﾀ定数「p_namae」だと、ｱｸｾｽ違反ｴﾗｰが出ます。↓

```
char* p_namae = "ハム太\0";
* ( p_namae + 2 ) = '\0';
```

・「p_namae」の宣言には、
「const」は付いていませんが、
文字列ﾘﾃﾗﾙで初期化した時点で
ﾎﾟｲﾝﾀ定数 となります。

・文字列を改行する時は、次のように書きます。↓

```
char moji1[] = "ABCD EFGH";
moji1[4] = '¥n';          /* 「改行ｺｰﾄﾞ」を代入する。 */
printf( "%s",moji1 );   /* 改行されて表示される。 */
```

・要素数**[4]**には、半角ｽﾍﾟｰｽがありますから、
それが改行ｺｰﾄﾞに置き換わるということは、

ABCD
EFGH

と表示されるということです。

・「**'¥n'**」は、半角文字が **2**文字 も入っていますが、
この組み合わせの場合は、
1文字分の文字ｺｰﾄﾞとみなされます。

・この改行ｺｰﾄﾞのように、制御文字は、
文字の手前に、「**¥**」記号を付けて入力します。

(文字化けして「＼」 (ﾊﾞｯｸｽﾗｯｼｭ) が表示されていますが、
ｺﾋﾟｰ･ﾍﾟｰｽﾄすると、「**¥**」 に戻ります。)

・これを「ｴｽｹｰﾌﾟｼｰｹﾝｽ」といいます。

【 よく使う ｴｽｹｰﾌﾟｼｰｹﾝｽ 】

「**¥r**」 ... 復帰ｺｰﾄﾞ。
「**¥n**」 ... 改行ｺｰﾄﾞ。

「¥t」 ... **Tab**文字。 (水平ﾀﾌﾞ)
「¥**0**」 ... 文字列の終端を示す「**NUL**文字」。 ("ハム太¥**0**")

・「改行ｺｰﾄﾞ」は、ﾌﾟﾗｯﾄﾌｫｰﾑによって異なります。 ↓

Windows ¥r¥n
UNIX ¥n
Mac ¥r

・「**printf**」関数だと、「¥**n**」でも改行されるんですが、
WindowsAPIの関数では、「¥**r**¥**n**」でないと改行されません。

・文字列定数 (ﾘﾃﾗﾙ) は、ｼﾝｸﾞﾙｸｫｰﾃｰｼｮﾝか、
ﾀﾞﾌﾞﾙｸｫｰﾃｰｼｮﾝで囲むことになっていますが、
これらの記号を、文字列の中で使う場合は、
次のようにｴｽｹｰﾌﾟします。 ↓

「¥**'**」 ... ｼﾝｸﾞﾙｸｫｰﾃｰｼｮﾝ (')
「¥**"**」 ... ﾀﾞﾌﾞﾙｸｫｰﾃｰｼｮﾝ (")
「¥¥」 ... YENﾏｰｸ (￥)
「¥**?**」 ... はてな記号 (**?**)

・どうみても半角**2**文字なんですが、
この組み合わせだけは、半角**1**文字として解釈されます。

・これを「ｴｽｹｰﾌﾟｼｰｹﾝｽ」といいます。

・ﾌｧｲﾙﾊﾟｽのｴｽｹｰﾌﾟを忘れていると、
ﾘﾝｸｴﾗｰになったり、ﾌｧｲﾙが開けなかったりします。

・その他にも、次のようなものがあります。↓

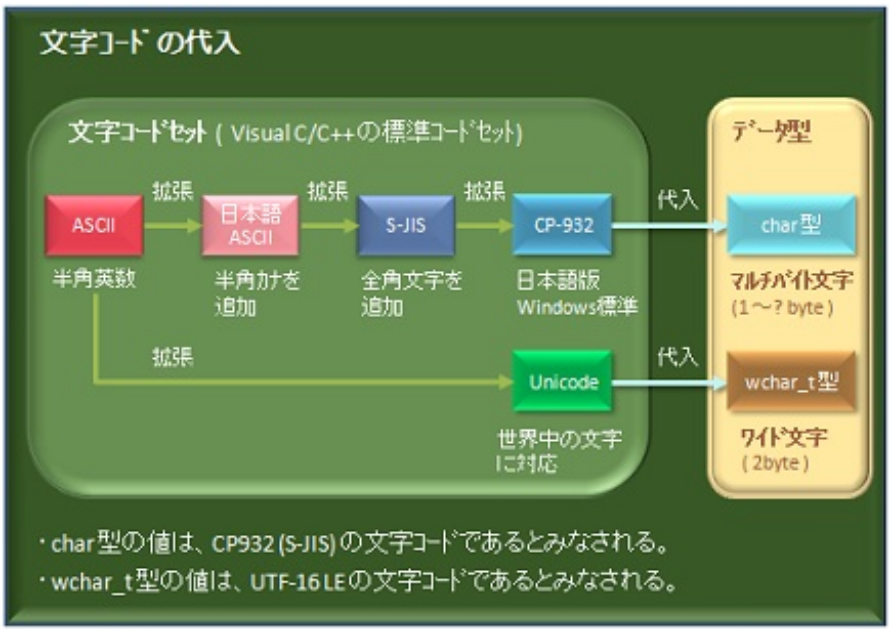
「**¥v**」　... 垂直Tab。
「**¥a**」　... ﾍﾞﾙ文字。　（ｱﾗｰﾄ）
「**¥b**」　... 1 文字分戻る。
「**¥f**」　... ﾍﾟｰｼﾞ送り。　（ｸﾘｱ）
「**¥N**」　... 8進定数。　（ **N**のところには、8進数の定数を書く。 ）
「**¥xN**」 ... 16進定数。（ **N**のところには、16進数の定数を書く。 ）

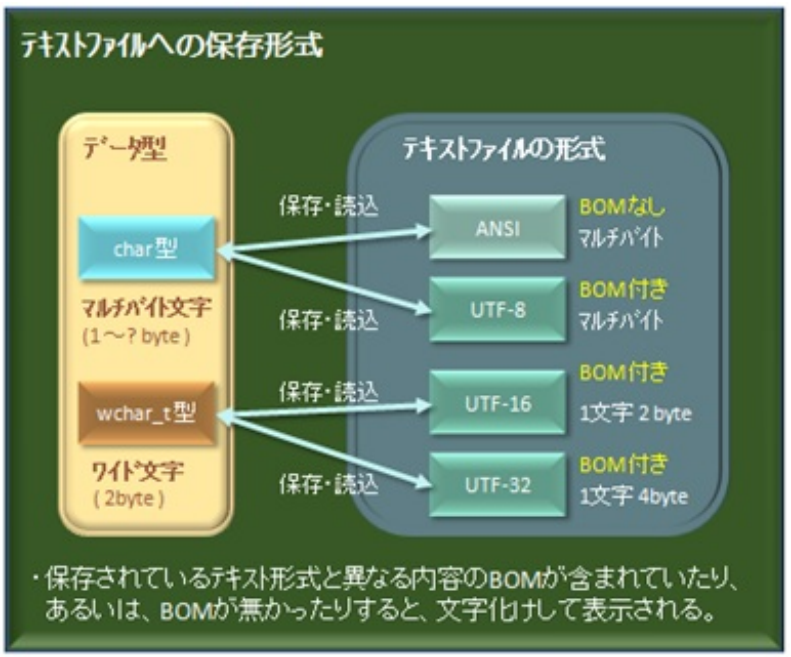
・まあ、めったに使いませんけどね。^^;

・「¥」を、「¥¥」と書かなかった時には、
意図しないｴｽｹｰﾌﾟがされてしまうことがありますので、
その点だけ、忘れないようにしていれば、
すべて覚えておく必要はありません。

・それと、
「￥」 (**YEN**ﾏｰｸ) が 「＼」 (ﾊﾞｯｸｽﾗｯｼｭ) に化けるのは、
日本語**ASCII**と本家**ASCII**との差異が原因で、
海外のｻｰﾊﾞｰだとかを使っている**Web**ｻｲﾄだと化けます。

文字ｺｰﾄﾞ





・「**.Net**」や「**Java**」で、文字列 をあつかう場合は、
「**string**」型という 基本ﾃﾞｰﾀ型 があります。

```
/* string型の文字列変数「namae」に、
        文字列「ハム太」を代入する。 */

string namae = "ハム太";
```

・ﾈｲﾃｨﾌﾞ C/C++で書く場合でも、
「STL」を使っていい場合なら、「std::string」というｸﾗｽ型がありますし、
「MFC」を使う場合は、「CString」というｸﾗｽ型を使えば、
同じような事ができます。

・しかし、WindowsAPI や、その他のCOMﾗｲﾌﾞﾗﾘなどでは、
これらのｸﾗｽ型は対応しておらず、直接渡すことはできません。

・これらの関数の引数には、「文字ｺｰﾄﾞ」の配列を渡します。↓

```
char multi_text [] = "あいうえお";       /* ﾏﾙﾁﾊﾞｲﾄ文字の配列。 */
wchar_t wide_text [] = L"あいうえお"; /* ﾜｲﾄﾞ文字の配列。 */
```

・よく使われる文字ｺｰﾄﾞの種類には、
大きく分けて、この2通りがあるんですが、
「S-JIS」で書く場合には、「char」型 (ﾏﾙﾁﾊﾞｲﾄ文字) を使い、
「Unicode」で書く場合は、「wchar_t」型 (ﾜｲﾄﾞ文字) を使います。

・あとで決めたい場合は、「TCHAR」型を使います。

・「TCHAR」型は、
「_UNICODE」定数が定義されている場合は
「wchar_t」型として解釈され、
そうでない場合は、「char」型として解釈されます。

```
/* TCHAR型の定義 */

#ifdef UNICODE
  typedef wchar_t TCHAR;
#elseif
  typedef char TCHAR;
#endif
```

・**C**の標準関数や、**WindowsAPI**の関数も、
　文字列を扱うものは、2通り定義されていて、

たとえば、「**MessageBox**」関数には、

「**ANSI**」(の、日本語拡張である「**S-JIS**」など、ﾏﾙﾁﾊﾞｲﾄ文字)
に対応した 「**MessageBoxA**」関数と、

「**Unicode**」(おおむねﾜｲﾄﾞ文字) に対応した
　「**MessageBoxW**」関数があり、

単に、「**MessageBox**」と書いた場合には、
「**TCHAR**」と同じで、「**UNICODE**」定数が定義されているか否かで、
どちらかに切り替わるようになっています。

・ちなみに、ﾏﾙﾁﾊﾞｲﾄ文字の引数は、
　関数内で、ﾜｲﾄﾞ文字に変換されていますので、
　ﾜｲﾄﾞ文字の関数の方が速いはずです。

・ｺﾝﾋﾟｭｰﾀｰの内部では、「文字」は、
「文字ｺｰﾄﾞ」という番号で、記録されています。

・「顔文字」などで有名な「**ASCII**ｺｰﾄﾞｾｯﾄ」は、
　英数字と基本的な記号しか扱わないため、
　1文字のｺｰﾄﾞを、**0**～**127**番までの**7bit**で記録しています。

　(**1**文字が**1**ﾊﾞｲﾄ固定であるので、「ｼﾝｸﾞﾙﾊﾞｲﾄ文字」という。
　　日本語版の**ascii**には、半角ｶﾅ文字が含まれていて、
　161～**223**番を使用するため、**8bit** (**1**ﾊﾞｲﾄ) をﾌﾙに使う)

・しかし、日本語環境では、ひらがな、カタカナに加えて、
膨大な数の漢字を使用するため、
1ﾊﾞｲﾄ (**0**～**255**) では到底足りません。

・そこで登場するのが「ﾏﾙﾁﾊﾞｲﾄ文字」で、
一般的には、「**SJIS**ｺｰﾄﾞｾｯﾄ」が使用されますが、

(日本語版の**Windows**では、「**cp932**ｺｰﾄﾞｾｯﾄ」が標準です。

つまり、特に指定しない場合は、「**char**」型の配列は、
「**cp932**の文字ｺｰﾄﾞの配列」として解釈される。)

これらのｺｰﾄﾞ体系のﾃﾞｰﾀでは、**ASCII**ｺｰﾄﾞに加えて、
2byte以上の文字が混在して記録されています。

・この「ﾏﾙﾁﾊﾞｲﾄ文字」は、全角文字が使用できる一方で、
1文字あたりのﾊﾞｲﾄ数が、文字によって異なるため、
何ﾊﾞｲﾄ目までが**1**文字目なのか、わかりにくい、
などの難点もあります。

・これに対して、ﾜｲﾄﾞ文字では、
1文字を、おおむね**2**ﾊﾞｲﾄで記録します。

(半角文字では、未使用の**1**ﾊﾞｲﾄが**0**で埋められます。)

・このﾜｲﾄﾞ文字は、おもに「**Unicode**」で使用されます。

・「**Unicode**」は、世界中の文字を
1つのｺｰﾄﾞｾｯﾄで記録できるようにするために
制定されたﾜｰﾙﾄﾞﾜｲﾄﾞなｺｰﾄﾞ体系で、正確には
1文字を**4**ﾊﾞｲﾄで記録している文字もあります。

・ただし、これらの文字はまだ、**API**側で未対応のため、
「**MessageBoxW**」関数に渡しても、文字化けしてしまいます。

(まあでもこれは、ﾌｯｰは絶対に使わないような
難解な漢字ばかりなので、無視しても無問題。)

・これに対応するためなのか、どうかわかりませんが、
1文字を**4**ﾊﾞｲﾄで記録しているﾌｧｲﾙﾌｫｰﾏｯﾄがあります。 ↓

文字ｺｰﾄﾞｾｯﾄ	説明
UTF-8	ﾏﾙﾁﾊﾞｲﾄ文字のUnicodeを記録している。
UTF-16	Unicode1文字を、一律2ﾊﾞｲﾄで記録している。
UTF-32	Unicode1文字を、一律4ﾊﾞｲﾄで記録している。←ｺﾚ

・ﾃｷｽﾄﾌｧｲﾙの場合は、
　先頭に、「**BOM**」 (**B**yte**O**rder**M**ark) が記録されていて、
　ﾃｷｽﾄｴﾃﾞｨﾀなどで読み込まれる際には、
　この「**BOM**」を手がかりにﾊﾞｲﾄの並びを読み取って、
　文字を表示しています。

・しかし、必ずも**BOM**が記録されているとは限らないため、
　その場合には、ｴﾃﾞｨﾀがどのｺｰﾄﾞｾｯﾄのﾃﾞｰﾀなのか判別できず、
文字化けして表示されてしまいます。

・**UNIX**や**Mac**では、「**UTF32BE**」が標準なんだそうで、
将来的には、「**UTF32**」で統一されるのかも知りません。

(「**BE**」は、「ﾋﾞｯｸｴﾝﾃﾞｨｱﾝ」の略。
Windowsは、「**LE**」(ﾘﾄﾙｴﾝﾃﾞｨｱﾝ)方式でﾌｧｲﾙに記録するので、
2ﾊﾞｲﾄ以上の値をﾌｧｲﾙに書き込んだり読み込む際には、
1ﾊﾞｲﾄ単位で逆順に並べなおす必要がある。)

・**Windows**では、「**S-JIS**」 (**cp932**) が標準であるため、
「**S-JIS**」のｺｰﾄﾞ表を眺めながら、
「**char**」配列を判定することが多かったのですが、
現在では、「**Unicode**」の使用が推奨されています。

・ちなみに、「**wchar_t**」型 (ﾜｲﾄﾞ文字) を使用する場合は、
「**Unicode**の配列である」と解釈されるようです。

・よく判定する「ｴｽｹｰﾌﾟｼｰｹﾝｽ」だとかは、
　半角文字ですので、どちらの場合でも、
　「日本語**ASCII**」となります。

文字列の長さと言語ﾛｹｰﾙ

・文字列をあつかう場合は、
　Cﾗﾝﾀｲﾑ関数を使用します。

・その前に、「言語ﾛｹｰﾙ」を設定しておきます。↓

```
#include <locale.h> /* 「setlocale」関数が定義されている。 */

 int main( )
{
   /* 「ﾛｹｰﾙ」には、日本語版Windowsの標準ｺｰﾄﾞｾｯﾄ「S-JIS ( cp932 )」を設定する。 */
   setlocale( LC_ALL , "Japanese_Japan.932" );
}
```

・「strlen」関数は、文字列のﾊﾞｲﾄ数を返します。↓

```
#include <string.h> /* 「strlen」関数が定義されている。 */

 int main( )
{
   char* p_namae="ハム太";
   unsigned int length = strlen( p_namae ); /* 結果は6ﾊﾞｲﾄ。 */
}
```

・strlen関数が返す値には、文字列の終端を示す「'¥0'」の分は、ｶｳﾝﾄされません。

(※ 「\」 (ﾊﾞｯｸｽﾗｯｼｭ) が表示されている場合は、文字化けしています。
　　これは半角の「￥」で、文字列の終端を示す「終端ｺｰﾄﾞ」は、正しくは「￥0」です。)

・これは、**wcslen**関数でも同様です。

・「ﾏﾙﾁﾊﾞｲﾄ文字」の文字数を取得するには、
「**_mbstrlen**」関数を使用します。 ↓

```
#include <stdlib.h> /* 「_mbstrlen」関数が定義されている。 */

 int main( )
{
   char* p_namae = "ハム太¥0";
   unsigned int length = _mbstrlen( p_namae ); /* 結果は3。 (※ﾛｹｰﾙ設定時) */
}
```

・上の例では、ﾛｹｰﾙ設定を省略していますが、
その場合は、文字列のﾊﾞｲﾄ数が返ります。

・それと、「**mbstring.h**」の、「**_mbslen**」関数ですが、
遅い上に、正当性ﾁｪｯｸが付いていないので、使いません。

・「ﾜｲﾄﾞ文字」 (Unicode) の場合は、
「**wcslen**」関数で文字数を取得します。 ↓

```
#include <wchar.h> /* 「wcslen」関数が定義されている。 */

 int main( )
{
   char* p_namae = L"ハム太¥0";
   unsigned int length = wcslen( p_namae ); /* 結果は3。 (※ﾛｹｰﾙ設定時) */
}
```

・「TCHAR」型の場合は、「_tcslen」関数を使用します。↓

```
#include <tchar.h> /* 「_tcslen」関数が定義されている。 */

 int main( )
{
   TCHAR* p_namae = TEXT( "ハム太¥0" );
   unsigned int length = _tcslen( p_namae );  /* 結果は3。 (※ﾛｹｰﾙ設定時) */
}
```

・Cﾗﾝﾀｲﾑ関数は、組み込み関数ですので、
実行速度は、自分で書くよりもかなり速いはずです。

標準入出力 (**printf_s scanf_s**)

・「**stdio.h**」 (標準入出力) の関数を使うと、
「**DOS**画面」に数値や文字列を表示 (出力) したり、
ﾕｰｻﾞｰに入力をしてもらった値を、受け取ったりできます。

・まずは、「出力」する場合の例から見て行きましょう。 ↓

```
#include <stdio.h> /* 「printf_s」「scanf_s」関数が定義されている。 */

  int main( )
  {
    unsigned int hensu1 = 100; /* 整数型の変数を宣言する。 */

    printf_s( "値は%dです。" , hensu1 ); /* 整数値を含む文字列を出力する。 */

    /* ｺｺで、「DOS画面」に、「値は100です。」と表示される。 */

    return 0; /* ｺｺで、ﾌﾟﾛｸﾞﾗﾑが終了するため、「DOS画面」が閉じてしまう。 */
  }
```

・「**VisualStudio**」で書かれている方は、
「ﾌﾟﾛｼﾞｪｸﾄ」を新規作成する時に、
「ｺﾝｿｰﾙｱﾌﾟﾘｹｰｼｮﾝ」を選択して下さい。

・「**Windows**ｱﾌﾟﾘｹｰｼｮﾝ」を選択した場合は、
「**DOS**画面」 (ｺﾝｿｰﾙ画面) が開きません。

・「**printf_s**」関数では、
引数**1**に 「書式文字列」 を書き、
引数**2**以降は、出力したい変数を、
「**,**」 (ｶﾝﾏ) で区切って、順に書いていきます。

・「書式文字列」というのは、
表示される文字列のことで、
ただし、変数の表示箇所を意味する「書式指定子」は
変数の値に置換されて表示されます。 ↓

```
int x = 1;  int y = 2; int z = 3; /* この3つの変数を表示させたい。 */

printf_s( "x = %d , y = %d , z = %d" , x , y , z ); /* 赤い所に、変数の値が置換される。 */

/* この場合は、「x = 1 , y = 2 , z = 3」 と、表示される。 */
```

・書式指定子「**%d**」は、
『 整数値の出力 』
を、指示する記号です。

・「書式指定子」には、この他にも、
次のようなものがあります。 ↓

【書式指定子】

「%**c**」 ... 1 文字 (**char**)
「%**s**」 ... 文字列 (**char***)
「%**d**」 ... 整数 (10進数)
「%**u**」 ... **+-**符号なし整数 (10進数)
「%**o**」 ... 整数 (8進数)
「%**x**」 ... 整数 (16進数)
「%**f**」 ... 浮動小数点 (**float**)
「%**e**」 ... 浮動小数点 (指数表示)　(**float**)
「%**g**」 ... 浮動小数点 (最適な形式) (**float**)
「%**ld**」 ... **long**整数 (10進数)
「%**lu**」 ... **+-**符号なし**long**整数 (10進数)

「%lo」 ... long整数 (8進)
「%lx」 ... long整数(16進)
「%lf」 ... 浮動小数点 (double)

・文字「%」を表示させる場合は、「%%」と書きます。

・表示するｹﾀ数を、指定することもできます。↓

```
/* %～f の間に、ｹﾀ数、または文字数 をはさむ。↓   */
printf_s( "■%8.3f■", 123.45678 );     /*  「■123.4567■」      */

/* 指数表記の場合は、↓ */
printf_s( "■%8.3e■", 1234.5678 );     /*  「■1.235e+003■」     */

/* 文字列の場合は、右寄せで表示される。↓ */
printf_s( "■%5s■",  "ABC" );          /*  「■  ABC■」          */

 /*  余白を切り詰める場合は、ｹﾀ数の前に「. 」(ﾋﾟﾘｵﾄﾞ) を付ける 。↓  */
printf_s( "■%.3s■",  "ABCDEFG" );    /*  「■ABC■」  */
```

・さらに、空白を、ｾﾞﾛ埋めするには、↓

```
/* ｹﾀ数の前に、「0」を付ける。↓ */
printf_s( "■%08.3f■", 123.45678 ); /*  「■0123.456■」  */
printf_s( "■%05d■", 1 );               /*  「■00001■」     */
```

・左寄せで表示するには、↓

```
/* ｹﾀ数の前に、「-」 (ﾏｲﾅｽ) を付ける。↓ */
printf_s( "■%-5s■", "ABC" ); /* 「■ABC  ■」 */
```

・+-符号を表示したい場合は、↓

```
/* ｹﾀ数の前に、「＋」 (ﾌﾟﾗｽ) を付ける。↓ */
printf_s( "■%+3d■" , 100 ); /* 「■+100■」 */
printf_s( "■%+3d■" , -100 ); /* 「■-100■」 */
```

・ﾕｰｻﾞｰに、「入力」をしてもらいたい時は、
「scanf_s」関数を使います。↓

```
char hensu1[256] = { }; /* これは、入力された文字列を、受け取る配列です。 */

scanf_s( "%c" hensu1 ); /* 入力待ちに入る。 */

/* ( ｷｰ入力が終わって、「Enter」ｷｰが押されるまで、処理が停止する。 ) */
```

・「DOS画面」を見ると、「ｶｰｿﾙ」 (白い線) が、点滅しています。

・ﾕｰｻﾞｰは、ここでｷｰﾎﾞｰﾄﾞを使って、何かを入力します。

・そして、「Enterｷｰ」が押されたら、ﾌﾟﾛｸﾞﾗﾑが再び流れ始めます。

・入力された値は、引数2でｱﾄﾞﾚｽを渡した配列に、格納されています。

・「ﾜｲﾄﾞ文字」の場合は、下記の関数を使用します。↓

```
unsigned int hensu1 = 100;
wprintf_s( L"値は%dです。" , hensu1 ); /* 「値は100です。」 と表示される。 */
```

```
wchar_t hensu1[256] = { };
wscanf_s( L"%c",  hensu1 ); /* 入力待ちに入る。 */
```

・それから、「tchar.h」には、
「TCHAR」型で文字列を渡すためのﾏｸﾛが定義されています。 ↓

```
unsigned int hensu1 = 100;
_tprintf_s( TEXT("値は%dです。") , hensu1 );
```

```
TCHAR hensu1[256] = { };
_tscanf_s( TEXT( "%c") , hensu1 );
```

・書式化された文字列は、
取得することもできます。 ↓

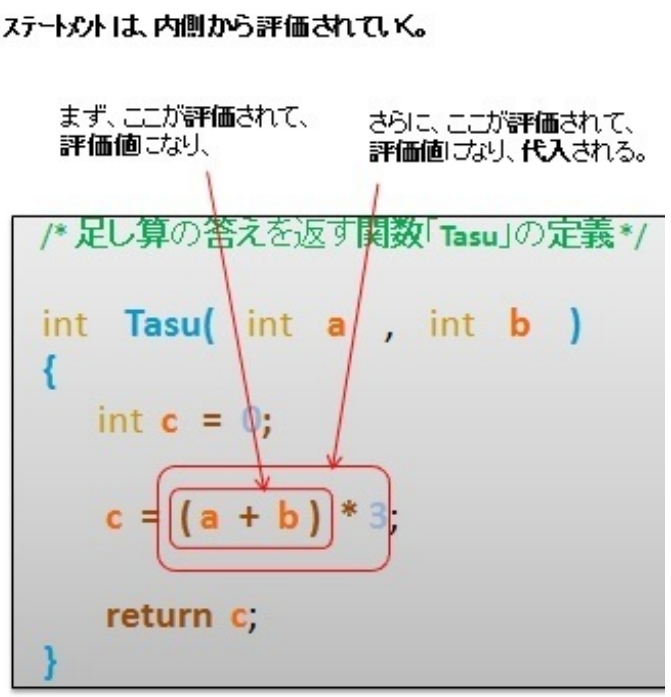
```
unsigned int hensu1 = 100;

char hairetsu1[ 256 ]; /* 書式化文字列を受け取る配列。

                          (※ｵｰﾊﾞｰしないように、要素数を設定しておく。) */

sprintf( hairetsu1, "%d",  hensu1 );

/* 戻り値は、書式化文字列の文字数。( ※NULL文字は、含まない。 )

   失敗した時には、EOF定数を返す。 */
```

/* ﾜｲﾄﾞ文字の場合は、「**w**sprintf」関数を呼ぶ。

戻り値は、書式化文字列の文字数。(※**NULL**文字は、含まない。)

失敗した時には、**WEOF**定数を返す。 */

数式と四則演算子



・ここでは、計算式の書き方を説明します。

・和・差・積・商を求める四則演算の例を見てみましょう。↓

```
int a = 10;
int b = 20;

int wa    = a + b;   /* 和 */
int sa    = a - b;   /* 差 */
int seki  = a * b;   /* 積 */
int syou  = a / b;   /* 商 */

 int jouyo  = a % b;   /* 剰余 ( 割り算のあまり ) */
```

・「=」の右側が計算式で、
左側の変数に、「答え」を代入しています。

・「+」とか「-」の記号は、「演算子」といいます。

・「*」 (ｱｽﾀﾘｽｸ) は掛け算 (乗算)、

「/」 (ｽﾗｯｼｭ) は割り算 (除算) です。

・割り算の 「あまり」 (剰余) を求める場合は、
「/」 の代わりに 「%」 (ﾊﾟｰｾﾝﾃｰｼﾞ) を使います。

・剰余演算子は、整数にしか使えません。

・「 a = a + b 」 は、「 a += b 」 と書くこともできます。

・「+=」は、「代入演算子」といいます。

```
int a = 10;
int b = 20;

a += b;  /* 加算代入 */
a -= b;  /* 減算代入 */
a *= b;  /* 乗算代入 */
a /= b;  /* 除算代入 */
 a %= b;  /* 剰余代入 */
```

・「 a += 1 」 は、「 ++a 」と書くこともできます。

・「++」 は 「ｲﾝｸﾘﾒﾝﾄ演算子」 といいます。

・「 a -= 1 」 は、「 --a 」と書くこともできます。

・「--」 は 「ﾃﾞｸﾘﾒﾝﾄ演算子」 といいます。

・計算式が複雑になってくると、
演算子の優先順位が重要になってきます。 ↓

```
    int kotae = a + b * c;
```

・この場合だと、まず最初に 「 b * c 」 が計算されて、
その答えが「a 」に加算されます。

・「 a + b 」 を、先に計算したい場合は、
「 (」 「) 」 (ｶｯｺ) でくくります。 ↓

```
int kotae = ( a + b ) * c;
```

・変数名が、持っている値として評価されるように、
数式も、その結果値として評価されます。

・たとえば、「 (10 + 20) 」 は、
「30」 として評価されます。

・「評価」は通常、
代入や比較をした時点で行われます。

・上で述べた「ｲﾝｸﾘﾒﾝﾄ」と「ﾃﾞｸﾘﾒﾝﾄ」には、
「前置」と「後置」という2通りの書き方があり、
「後置」では、この「評価」が重要な意味を持っています。

```
int b1 = ++a;
int b2 = a++;
```

・変数「b1」には、式 「a += 1 」の
結果値が代入されます。 (「前置ｲﾝｸﾘﾒﾝﾄ」という)

・変数「b2」には、変数 「a 」の値が代入され、(まず変数が評価される)
その後で、式 「 「a += 1 」が実行されます。 (「後置ｲﾝｸﾘﾒﾝﾄ」という)

・演算の優先順位については、こちらです。↓

---

1 a++ a--
2 ++a --a
3 a * b a / b a % b
4 a + b a - b
5 a >> b a << b
6 a > b a < b a >= b a <= b
7 a == b a != b
8 a & b
9 a ^ b (OR)
10 a ^ b (XOR)
11 a && b
12 a || b
13 a ? b : c
14 a = b a += b a -= b
15 a , b

比較式と比較演算子

・「**=**」 (ｲｺｰﾙ) という記号は、
C/C++では、「代入」を意味します。

・では、「等しい」 を表す 「=」 は、どう書くのでしょうか。

・**C/C++**では、値と値とを比較する場合は、
下記のように、「比較演算子」をもちいて比較式を書きます。 ↓

```
int hensu1 = 11;  /* この変数の値を比較する。 */

bool kekka1 = ( hensu1 == 10 ); /* 値が、 「10」と等しいなら、trueが代入される。 */
bool kekka2 = ( hensu1 != 10 );  /* 値が、 「10」と等しくないなら、trueが代入される。 */
bool kekka3 = ( hensu1 > 10 );   /* 値が、 「10」より大きいなら、trueが代入される。 */
bool kekka4 = ( hensu1 < 10 );   /* 値が、 「10」より小さいなら、trueが代入される。 */
bool kekka5 = ( hensu1 >= 10 ); /* 値が、 「10」以上なら、trueが代入される。 */
bool kekka6 = ( hensu1 <= 10 ); /* 値が、 「10」以下なら、trueが代入される。 */
```

・赤色の記号が「比較演算子」です。

・それぞれ、条件を満たしていれば、論理値 「true」 (真) が代入され、
そうでなければ、論理値 「false」 (偽) が代入されます。

・つまり、「比較式」は、ｺﾝﾊﾟｲﾙされる時に、
「bool型の値」として評価されます。

・「kekka5」と「kekka6」の場合は、以上と以下ですから、
両方の値が等しい場合も、「true」 (真) が代入されます。

論理演算子

・前回説明した「比較式」は、
「bool」型の値として評価されていました。

・今回説明する「論理演算子」は、
「bool」型の値を比較するもので、
「比較式」(bool値)と「比較式」(bool値)とを
比較するのに使います。↓

```
int tensu1 = 50; /* ﾊﾑ太くんの点数 */

/* 40点以上で、かつ、60点以下なら、居残りが確定する。
   (「&&」(AND) 演算では、すべてが「true」なら、「true」と評価される。) */
bool inokori = ( ( 40 <= tensu1 ) && ( count >= 60 ) );

/* 99点か、または、100点なら、表彰が確定する。
   (「||」(OR) 演算では、いずれかが「true」なら、「true」と評価される。) */
bool hyosyo = ( ( 99 == tensu1 ) || ( count == 100 ) );
```

・これも「論理演算」です。↓

```
/* 「!」(NOT) 演算では、評価値が逆になる。 */

bool zaigaku = ( tensu1 );  /* 値が、「0」以外なら、trueが代入される。 */
bool rakudai = ! ( tensu1 ); /* 値が、「0」なら、trueが代入される。 */
```

・「zaigaku 」は、何も付いてませんが、
「bool」型では、「0」(false)以外は、
「true」 として評価されます。

・「**rakudai**」の場合は、
頭左横に、「**!**」 (**NOT**演算子) が付いているため、
評価値が逆になります。

・つまり、評価値が「**true**」なら「**false**」になり、
「**false**」なら「**true**」になります。

ﾋﾞｯﾄ演算子


ﾊﾞｲﾄ値の呼び方
上位ﾊﾞｲﾄ
下位ﾊﾞｲﾄ
0101 0101
0101 0101
第1ﾊﾞｲﾄ
第2ﾊﾞｲﾄ


ﾋﾞｯﾄ演算
AND
a & b
・両方のﾋﾞｯﾄが1なら1、違うなら0。
a : 0 0 1 0 1 0 0 0
b : 0 0 0 0 1 0 0 0
結果 : 0 0 0 0 1 0 0 0
OR
a | b
・どちらかのﾋﾞｯﾄが1なら1、違うなら0。
a : 0 0 0 0 1 0 0 0
b : 0 0 0 0 0 0 0 0
結果 : 0 0 0 0 1 0 0 0
・上の例では、各2ｹﾀを演算しているが、実際には、すべてのｹﾀで演算される。


ﾋﾞｯﾄ演算
XOR
a ^ b
・両方のﾋﾞｯﾄが違うなら1、同じなら0。
a : 0 0 1 0 1 0 0 0
b : 0 0 1 0 0 0 0 0
結果 : 0 0 0 0 1 0 0 0
NOT
~ a
・ﾋﾞｯﾄが反転する。1なら0、0なら1。
a : 0 0 1 0 0 0 0 0
結果 : 0 0 0 0 1 0 0 0
・上の例では、各2ｹﾀを演算しているが、実際には、すべてのｹﾀで演算される。

(※ 符号付き整数値を右ｼﾌﾄした場合は、符号ﾋﾞｯﾄで埋められます。)

・ﾃﾞｰﾀ型の最小単位は**1**ﾊﾞｲﾄですが、
「ﾋﾞｯﾄ演算子」を使えば、
ﾋﾞｯﾄ単位で操作することができます。↓

```
unsigned char hensu1 = 0; /* 変数を初期化しておく。 */

hensu1 |= 0xF0;  /* 上位4ﾋﾞｯﾄをONにする。 */
bool kekka1 = (bool) ( hensu1 & 0xF0 ); /* 上位4ﾋﾞｯﾄはONなので、true。 */

hensu1 &= ~0xF0; /* 上位4ﾋﾞｯﾄOFFにする。 */
boolkekka2 = (bool) ( hensu1 & 0xF0 ); /* 上位4ﾋﾞｯﾄはOFFなので、false。 */
```

・ﾋﾞｯﾄの並びがわかりやすいように
2進数表記で書くと、
「0xF0」は、「**11110000**」です。

・「**11110000**」は、上位**4**ﾋﾞｯﾄが、
すべて「**1**」(**ON**ﾋﾞｯﾄ) になっています。

・「ﾋﾞｯﾄ演算」では、
この「16進数表記」で書いている値が重要で、
これを「ﾋﾞｯﾄﾏｽｸ」といいます。

・たとえば、7ﾋﾞｯﾄ目だけを切り替えたい場合は、
「電卓」を開いて、「2進数」ﾓｰﾄﾞで
「0100 0000」と入力します。

・そして、「10進数」ﾓｰﾄﾞか、または
「16進数」ﾓｰﾄﾞに切り替えると、
この「ﾋﾞｯﾄﾏｽｸ」が表示されます。

・ﾋﾞｯﾄの操作では、もう一つ
重要な手法があります。↓

```
unsigned char hensu1 = 1; /* この時点では、0000 0001 */

hensu1 = hensu1 << 1; /* 左に1ﾋﾞｯﾄ移動するので、1ｹﾀ繰り上がって、0000 0010 */
hensu1 = hensu1 >> 1; /* 右に1ﾋﾞｯﾄ移動するので、 1ｹﾀ繰り下がって、0000 0001 */
```

・これは「ﾋﾞｯﾄｼﾌﾄ」というﾋﾞｯﾄ演算で、
指定したｹﾀ数だけ、ﾋﾞｯﾄの並びをずらすことができます。

・ｺﾒﾝﾄ欄にある赤色の1ｹﾀは、
「ﾋﾞｯﾄｼﾌﾄ」によって追加されたものです。

・このように、元々の値にないｹﾀは、0で埋められます。

・「ﾋﾞｯﾄｼﾌﾄ」は、ｹﾀを上げ下げすることでもありますので、
乗算・除算の代わりに使うことができます。↓

```
unsigned char a1 <<= 1;  /* a1 *= 2   と書くのと同じです。 */
```

```
unsigned char a2 <<= 2; /* a2 *= 4   と書くのと同じです。 */
unsigned char a3 <<= 3; /* a3 *= 8   と書くのと同じです。 */
unsigned char a4 <<= 4; /* a4 *= 16   と書くのと同じです。 */

unsigned char a5 >>= 1; /* a5 /= 2   と書くのと同じです。 */
unsigned char a6 >>= 2; /* a6 /= 4   と書くのと同じです。 */
unsigned char a7 >>= 3; /* a7 /= 8   と書くのと同じです。 */
unsigned char a8 >>= 4; /* a8 /= 16   と書くのと同じです。 */
```

・乗算と除算は、加算や減算に比べて、
10倍以上の処理時間がかかります。

・ﾋﾞｯﾄｼﾌﾄの速度は、加算や減算と同程度ですから、
これは、非常に有効なﾃｸﾆｯｸだといえます。

・他の値で乗・除算する場合も、
+aするなどすれば、利用は可能です。

```
#define MUL3(a) (((a)<<1)+(a))
```

・ただし、符号付き整数値の場合は、
上位**1**ﾋﾞｯﾄが符号ﾋﾞｯﾄですから使えません。

・「ﾋﾞｯﾄﾏｽｸ」のおもな使い道は、
途中の数ﾋﾞｯﾄだけを取り出したり、
書き換えたりするような場合です。 ↓

```
unsigned char a1 = 0xF0;              /* 1111 0000 で初期化しておく。 */
unsigned char a2 = a1 >> 4;           /* 0000 1111 が代入される。 */
unsigned char a3 = ( a2 << 4 ) | 0x0F ;/* 1111 0000 と 0000 1111 を加算し、代入。 */
```

・「| 」(**OR**演算子) は、最初に少し出てきましたが、
「ﾋﾞｯﾄﾏｽｸ」を重ねる時に使います。

・上の例だと、**3**行目の時点で、変数「**a2**」の値は、
「0x**F0**」(1111 0000) であり、
下位**4**ﾋﾞｯﾄは、常にすべてｾﾞﾛで、
つまり、使用していないので、
「0x**0F**」(0000 1111) を、重ねることができます。

```
#define  ICHIRO      1    /* 0000 0001 */
#define JIRO         2    /* 0000 00**1**0 */
#define SABURO       4    /* 0000 0**1**00 */
#define SHIRO        8    /* 0000 **1**000 */
#define GORO        16    /* 000**1** 0000 */
#define ROKURO      32    /* 00**1**0 0000 */
#define NANARO      64    /* 0**1**00 0000 */
#define HACHIRO  128  /* **1**000  0000 */
```

・上記のように、「ﾋﾞｯﾄﾏｽｸ」は、
それぞれの使用するﾋﾞｯﾄが重ならないように定義します。↑

・この「ﾋﾞｯﾄﾏｽｸ」を使用すれば、
1ﾊﾞｲﾄ値の中に、**8**名分の出席状況が記録できます。↓

```
unsigned char hensuu1 = JIRO | TARO | ROKURO;
hensuu1 |= HACHIRO;
hensuu1 |=ICHIRO;
```

・組み込み系のｼｽﾃﾑでは、ﾒﾓﾘ容量にゆとりがなく、
CPUの演算速度も貧弱なことが多いため、
ﾋﾞｯﾄ演算は、今もって有効なﾃｸﾆｯｸといえます。

・ふつうの計算式と違って、
直感的にわかるものでもないので、
あまり深く考えない方がいいと思います。

・「ﾋﾞｯﾄﾌﾗｸﾞ」は、「電卓」で算出すればよく、
よく使う方法は今回説明したものなど限られていますから、
とにかくそれを使ってみて、やり方だけ覚えておけば、
そんなに困ることはありません。

ﾃﾞｰﾀ型の別名定義 (**typedef**)

・「**unsigned int**」と書くのが面倒なら、
別の名前を定義することができます。 ↓

```
typedef unsigned int UInt32; /* typedef 元のﾃﾞｰﾀ型 別名; */
```

・「**typedef**」の後にくるのは、元のﾃﾞｰﾀ型です。

・最後にくるのが、別名です。

・最後のｾﾐｺﾛﾝは、忘れないようにして下さい。

・次の2つは、いずれも同じﾃﾞｰﾀ型「**unsigned int**」の変数です。 ↓

```
unsigned int hensu1;
UInt32       hensu2;
```

構造体型の定義 (**struct**)

・「構造体型」というのは、特殊なﾃﾞｰﾀ型の一種です。

・構造体型で宣言された変数、つまり、構造体は、
複数の変数を、ひとまとめにしたもので、
その点では、「配列」とよく似ています。

・「配列」というのは、
同じﾃﾞｰﾀ型の変数を並べたものでした。↓

・すべての要素が同じｻｲｽﾞ (等間隔) だったため、
「要素番号」を指定すれば、それを手がかりに
その値が置かれているﾒﾓﾘｱﾄﾞﾚｽを特定でき、
要素の値に代入したり、
逆に、要素から値を取り出すことができました。

・これに対して、構造体は、
異なるﾃﾞｰﾀ型の変数などを並べたものです。↓

[1byte値] [2byte値] [1byte値] [4byte値] ....

・構造体では、各要素のｻｲｽﾞが、一定ではないため、
「要素番号」では、ﾒﾓﾘｱﾄﾞﾚｽを特定できません。

・そのため、構造体では、「要素番号」ではなく、
要素の「名前」(ﾒﾝﾊﾞ名) を指定してｱｸｾｽします。

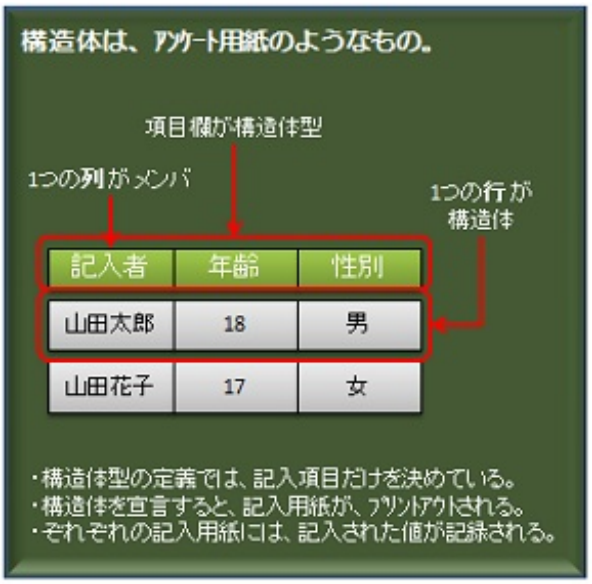
・このため、構造体を宣言するには、
まず、構造体型を定義しておく必要があります。↓

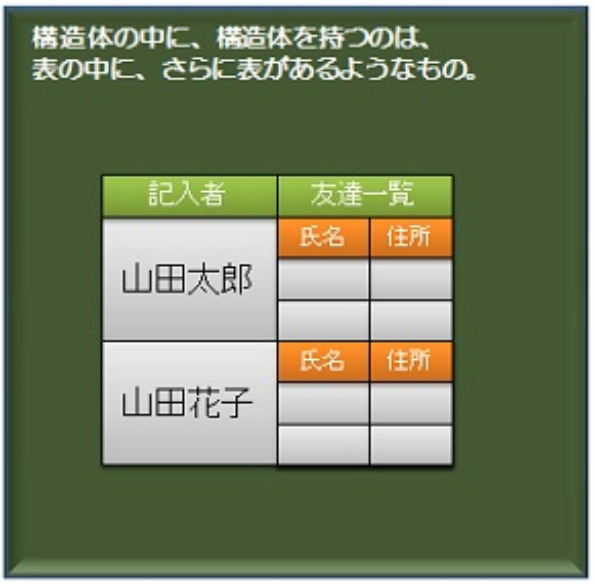
```
struct Meishi   /* ← 構造体型の名前  */
{
    char myouji[5];              /* ← ﾒﾝﾊﾞの定義 */
    char namae[5];               /*  ← ﾒﾝﾊﾞの定義 */
    unsigned int denwa_bangou;  /*  ← ﾒﾝﾊﾞの定義 */
};
```

・「**struct**」 の右となりが、構造体型の名前です。

・中ｶｯｺ 「 { } 」 の中には、
ﾒﾝﾊﾞを、並べて書いていきます。

・何となく、変数の宣言と似た書き方をしていますが、
これは「ﾃﾞｰﾀ型の定義」で、
あくまで、「こういうﾒﾝﾊﾞ-構成でいきますよ～」
という取り決めをしているだけです。





・「ｱﾝｹｰﾄ用紙」でいうところの、
「記入項目」を決めている段階です。

・実際に、ﾒﾓﾘ上に場所が確保されるのは、
構造体が宣言された時です。↓　（※正確には、ﾌﾟﾛｸﾞﾗﾑの実行時ですが）

```
/* 構造体の宣言。↓ */

Meishi meishi1;       /* この用紙は、初期化されていない。 */
Meishi meishi2 = { }; /* この用紙は、初期化されている。 */
```

・先ほどの、「ｱﾝｹｰﾄ用紙」の例えでいえば、
未記入の用紙が**2**枚、ﾌﾟﾘﾝﾄｱｳﾄされたことになります。

・最初の「**meishi1**」は、初期化されておらず、
各ﾒﾝﾊﾞには、関係のない値が入っています。

・次の、「**meishi2**」は、初期化をしているため、
各ﾒﾝﾊﾞは、「**0**」で初期化されています。

・以前、配列のところで説明したように、
初期値を省略した要素は、
すべて「**0**」で初期化されます。

・構造体の「ﾒﾝﾊﾞ」にｱｸｾｽするには、
「ﾒﾝﾊﾞ名」を指定します。↓

```
unsigned int hensu1 = meishi.denwa_bangou;  /* ﾒﾝﾊﾞの値を、取り出す。 */
 meishi.denwa_bangou = 12345678;              /* ﾒﾝﾊﾞに、値を代入する。 */
```

・少し見えにくいのですが、
構造体と、「ﾒﾝﾊﾞ名」の間には、
「. 」(ﾋﾟﾘｵﾄﾞ)が打たれています。

・「**VisualStudio**」などで書かれている方は、
ﾋﾟﾘｵﾄﾞを入力した時に、
「ﾒﾝﾊﾞ名」の一覧が表示されたはずです。

・さらに、もう一つ、書き方があります。↓

```
Meishi meishi1; /* 構造体を宣言する。 */
 meishi.denwa_bangou = 12345678; /* ﾒﾝﾊﾞに、値を代入してみる。 */
 Meishi* p_meishi = &meishi; /* 構造体のｱﾄﾞﾚｽを、ﾎﾟｲﾝﾀ変数に代入する。 */
 unsigned int hensu1 = p_meishi->denwa_bangou; /* ﾒﾝﾊﾞの値を、取り出す。 */
```

・ﾎﾟｲﾝﾀ「**p_meishi**」には、
構造体「meishi1」の
ﾒﾓﾘｱﾄﾞﾚｽが代入されています。(3行目です)

・ﾎﾟｲﾝﾀ名の右横に、「**->**」(ｱﾛｰ)を付けると、
先ほどと同じようにして
「ﾒﾝﾊﾞ名」を指定することができ、
ﾒﾝﾊﾞにｱｸｾｽすることができます。(4行目です)

・また、構造体は、

同じ構造体型で宣言された別の構造体に
代入することができます。↓

```
meishi2 = meishi1;
```

・配列では、これはできないのですが、
構造体の、ﾒﾝﾊﾞになっている配列では、
要素値は、すべてｺﾋﾟｰされます。

・配列の要素の時もそうでしたが、
構造体のﾒﾝﾊﾞもまた、
ﾒﾓﾘ上に、連続して並んでいます。

・ということは、そのｱﾄﾞﾚｽを指定して
ﾒﾝﾊﾞの値にｱｸｾｽすることができるはずです。↓

```
Meishi meishi1 = { };          /* 構造体を初期化して宣言する。 */
meishi1.namae[0] = 'A';       /* 2つ目のﾒﾝﾊﾞの、1文字目に、「A」を代入する。 */
char* p1 = (char*) &meishi1;  /* 構造体の先頭ｱﾄﾞﾚｽを、ﾎﾟｲﾝﾀに代入する。 */
char a = *( p1 + 5 );         /* ﾎﾟｲﾝﾀ演算で、5ﾊﾞｲﾄ進めてから、値を取り出す。 */
```

・4行目の変数「a」には、
ﾒﾝﾊﾞ「namae」の
1文字目の「A」が、代入されます。

・ということは、この構造体のｻｲｽﾞは、
14byteなのかな？ ...と思いきや、
sizeof で調べてみると、16byteなんですね。

・32bitのﾌﾟﾛｾｯｻ(CPU)では、

32bit、すなわち、**4byte**ずつ、ﾃﾞｰﾀを転送しています。

・このため、ｺﾝﾊﾟｲﾗは、ﾃﾞｰﾀの転送効率をよくするため、
ﾃﾞｰﾀを、**4byte**区切りに配置します。

(これを、「**4byte** 境界 (ｱﾗｲﾝﾒﾝﾄ) に揃える。」 という。)

・たとえば、次のような場合、↓

```
struct Kouzou
{
  char data1; // 1byte
  int  data2; // 4byte
};
```

この構造体が使用するﾒﾓﾘ領域のｻｲｽﾞは、合計**5byte**です。

この場合、ｺﾝﾊﾟｲﾗは、
使用するﾃﾞｰﾀ領域を、**4**の倍数にそろえるために、
1byteの領域しか使用していない**data1**に対して、
4byteの領域を割り当てます。

たとえば、
data1に **0xff** を代入し、
data2に、**0xffffffff** を代入した場合、
ﾒﾓﾘ上では、**ff 00 00 00 ff ff ff ff** と配置されます。

専門用語では、これを
「ｱﾗｲﾒﾝﾄ（境界）をそろえる」といいます。

「ｱﾗｲﾒﾝﾄ」というのは、そろえるﾃﾞｰﾀｻｲｽﾞのことで、
この場合だと、この構造体の中で一番大きな**4**ﾊﾞｲﾄが
ｱﾗｲﾒﾝﾄとなります。

それと、これも余談ですが、
C言語で構造体型を定義する場合は、次のような書き方をします。↓

```
typedef   struct _Meishi
{
    /*  中略 */
} Meisi;
```

・これはなぜかというと、
C言語では、構造体型の変数を宣言する時に、
次のように書かないといけないからです。↓

```
struct Meishi meisi1;
```

・**typedef** 演算子は、ﾃﾞｰﾀ型の別名を定義するもので、
この場合は、「**struct** _**Meishi**」というﾃﾞｰﾀ型に、
「**Meishi**」という別の名前をつけています。(詳細は後述)

・これによって、宣言時の「**struct** 」を省略しているのです。

・しかし、「**Visual Studio**」では、基本的に、
C/C++の構文規則が適用されていますから、
この書き方は、必要ありません。

不完全型の前方宣言

・構造体型の中に、
　同じ構造体型のﾒﾝﾊﾞを持たせたい時は、
　「不完全型の前方宣言」をしておきます。↓

```
struct ListItem; /* 構造体型 (不完全型) の前方宣言。*/

struct ListItem /* 構造体型の定義。*/
{
  int value;
  ListItem* p_prev; /* ﾒﾝﾊﾞで、自身の型を使っているため、前方宣言が必要。*/
  ListItem* p_next;
};
```

・「不完全型」というのは、
　『名前しか決まっていない
　不完全な型』のことです。

・上記の部分が、ｺﾝﾊﾟｲﾙされた時のことを
　考えてみていただきたいんですが、
　ｺﾝﾊﾟｲﾗが、「**ListItem**」型のﾒﾝﾊﾞを
　読み取っている時点では、
　この「**ListItem**」型は、
　まだ定義されていません。

・ｺﾝﾊﾟｲﾗは、「**ListItem**」という単語が
　何なのかを知りません。

・そこで、
　この単語は、少なくとも「構造体型」である、
　ということを、前もって知らせるために、
　「前方宣言」を行なっているのです。

・このように、ﾍｯﾀﾞﾌｧｲﾙでは、
前方宣言によって
定義をしていないﾃﾞｰﾀ型が使えますが、
それは、ﾎﾟｲﾝﾀだけです。

・たとえば、ﾒﾝﾊﾞの宣言で、
未定義の構造体型などを使うと、
ｺﾝﾊﾟｲﾙｴﾗｰ:C2079が出力されます。

共用体型の定義 (union)

・「共用体型」というのもまた特殊なﾃﾞｰﾀ型で、
　共用体型で宣言された共用体は、
　ﾃﾞｰﾀ型を、変更できます。

・変更できる、とはいっても、
　何型にでもなれるわけではありません。

・変更できるﾃﾞｰﾀ型を、
　あらかじめ定義しておいて、
　そのうちのいずれかになれるのです。 ↓

```
union UIntAny /* 共用体型の定義 */
{
    unsigned char     ui8;
    unsigned short   ui16;
    unsigned int      ui32;
    unsigned __int64  ui64;
};
```

・それでは、宣言して、使ってみましょう。 ↓

```
UIntAny kyoyo1; /* 共用体を宣言する。 */

kyoyo1.ui64 = 0x1122334455667788; /* 64bit (8byte) のﾒﾝﾊﾞに、値を代入してみる。 */
unsigned char    hensu1 = kyoyo1.ui8;   /*                  0x88 */
unsigned short   hensu2 = kyoyo1.ui16; /*                0x7788 */
unsigned int     hensu3 = kyoyo1.ui32; /*            0x55667788 */
unsigned __int64 hensu4 = kyoyo1.ui64; /* 0x1122334455667788 */
```

・変数「**hensu1**」には、
初期値の下位**1byte**が代入されています。

・変数「**hensu2**」には、
初期値の下位**2byte**が代入されています。

・変数「**hensu3**」には、
初期値の下位**4byte**が代入されています。

・共用体は、宣言された時点で
最も大きなｻｲｽﾞのﾃﾞｰﾀ型のｻｲｽﾞで
ﾒﾓﾘ上に置き場所を作ります。

・共用体は、
浮動小数点型 を 固定小数点型 に
変換する際などに使ったりします。

・**COM**では、あらゆる基本ﾃﾞｰﾀ型になれる
「**VARIANT**」型という共用体型が定義されていて、
これは「**VBA**」の変数のﾃﾞｰﾀ型です。

列挙体型の定義 (enum)

・「列挙体型」というのも特殊なﾃﾞｰﾀ型で、
　列挙体型で宣言された列挙体は、
　限定された値だけを代入する変数になります。

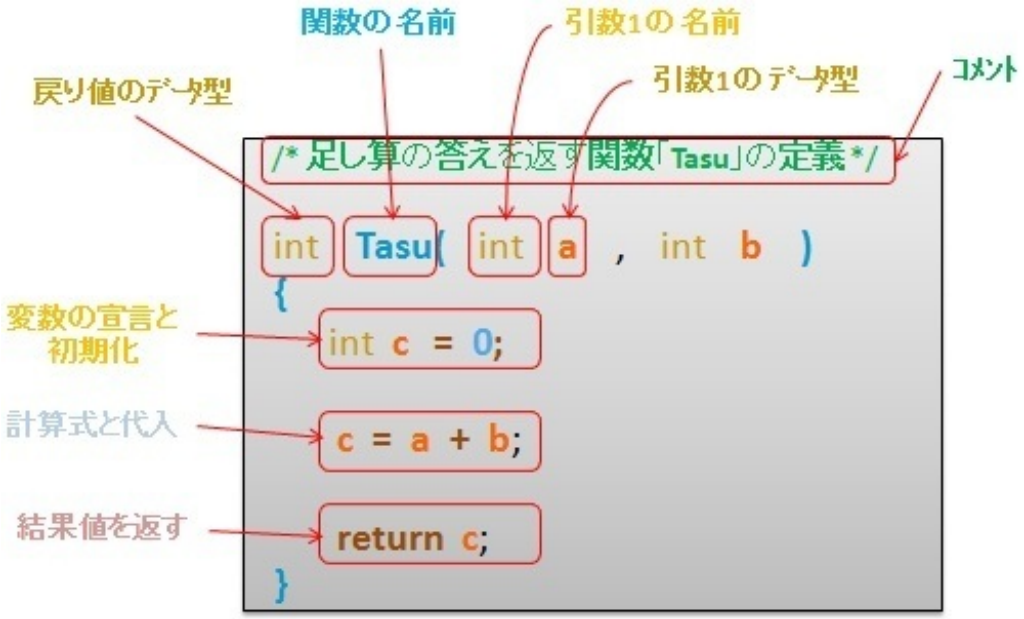
・代入できる値は、
　あらかじめ定義しておきます。 ↓

```
/* 列挙体型 「Shingoki」 の定義。 ↓ */

enum Shingouki
{
    Aka   = 0 ,
    Kiiro  = 1 ,
    Midori = 2
};
```

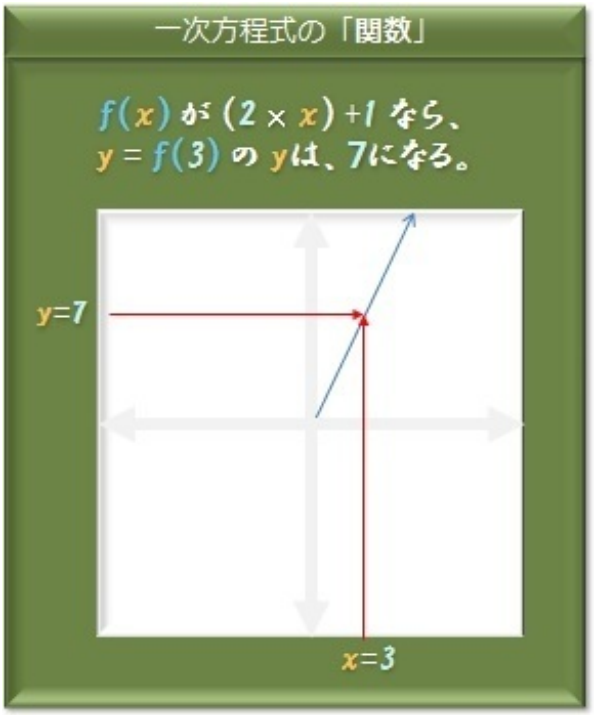
・変数は、おもに、
　状態を記憶しておくのに使用します。 ↓

```
    Singouki snigouki1 = Aka;
    Singouki snigouki2 = Midori;
```

関数



・「関数」というと、中学生より上の方なら、
おそらくご存知のはずなんですが、↓



・変数「x」の値が変化すると、
関数の結果である変数「y」の値が変化します。

・変数「x」に、1ずつ足していくと、yも、2ずつ増えていき、
ｸﾞﾗﾌで見ると、真っ直ぐな直線となります。

・ｺﾝﾋﾟｭｰﾀｰﾌﾟﾛｸﾞﾗﾑでは、
この「関数」を、たくさん書きます。

・というか、この、「関数」の中に、
命令文を書いて行きます。

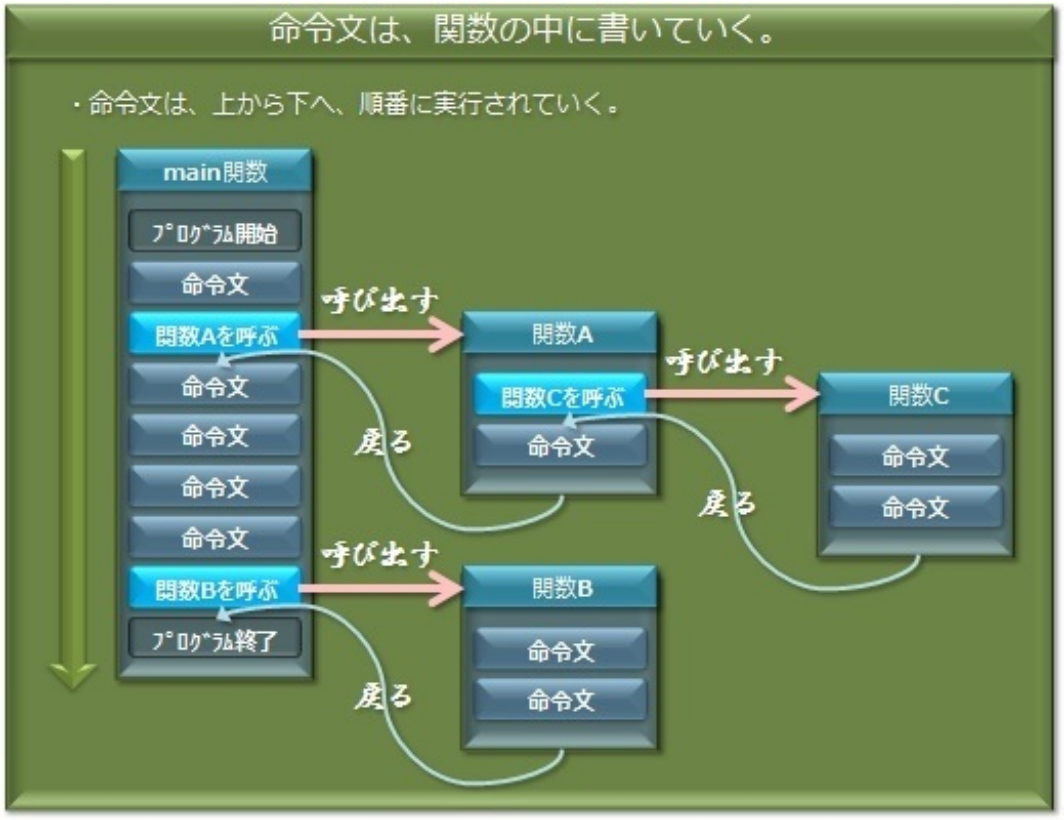
・それでは、この一次方程式
「 f(x) = (2 * x) + 1 」 を、
「C/C++」の「関数」にしてみましょう。↓

```
int f( int x )
{
    return ( ( 2 * x ) + 1 );
}
```

・次に、この関数「f」を、
ﾌﾟﾛｸﾞﾗﾑの最初に呼ばれる関数「main」の中から、
呼び出してみます。↓

```
int main( )
{
    int x = 3;

    int y = f( x );  /* 変数「y」には、結果値の「7」が代入される。 */

    return 0;   /* ﾌﾟﾛｸﾞﾗﾑの最初に呼ばれた「main」関数が
                                    終了したので、ﾌﾟﾛｸﾞﾗﾑが終了する。 */
}
```

・このように、関数 (の呼び出し文) を代入すると、
その関数内で「 return 」された戻り値が、代入されます。



・ﾕｰｻﾞｰが、「実行ﾌｧｲﾙ」 (*.exe) をｸﾘｯｸすると、
そのﾌﾟﾛｸﾞﾗﾑの中に書いてある
「main」という関数が、最初に呼ばれます。

・ﾌﾟﾛｸﾞﾗﾏは、この「main」関数の中で、
別の関数を呼ぶように
ﾌﾟﾛｸﾞﾗﾑを書いて行きます。

・そしてさらに、その関数の中から、
別の関数を呼ぶように書いていきます。

・それでは次に、
関数を「定義」する時の書き方について
もう少し、詳しく見てみましょう。↓

```
int KansuMei( int hikisu1 , int hikisu2 ) /* 左から順に、戻り値の型名、関数名、引数の宣言。 */
{
    int modorichi1 = hikisu1+ hikisu2; /* 引数で計算する */

    return modorichi1; /* 戻り値を返している */
}
```

・1行目のさいしょの「int」は、
「戻り値」のﾃﾞｰﾀ型です。

・「戻り値」を返さない時は、
「void」と書きます。

・その場合は、値は返せませんが、
処理の途中で 「 return; 」と書けば、
呼び出し元に戻ることができます。

(これを 「呼び出し元に、制御を返す」 という)

・で、その次が関数の名前で、
その後の、ｶｯｺ内には、
「引数」の宣言を書きます。

・「引数」が複数ある場合は、
ｶﾝﾏで区切ります。

・それから、これはあまり使わない言葉ですが、
関数の定義側で書いている引数のことは、「実引数」といい、
関数の呼び出し側で書いている引数は、「仮引数」といいます。

・「関数」というと、「計算式」というｲﾒｰｼﾞがありますが、
実際には、**Windowsの関数**を呼び出して
「ｳｨﾝﾄﾞｳのｻｲｽﾞを変更する」だとか、
「画像の位置を変更する」だとか、
「効果音の番号を指定して音を鳴らす」だとか、
簡単なものがほとんどです。↓

```
bool kekka = SetWindowSize( h_window, 640, 480 );
```

・極端な話、ﾌﾟﾛｸﾞﾗﾏが独自に定義する関数は、
既存の関数の呼び出しをまとめたもので、
ﾒﾓﾘ上のﾃﾞｰﾀも、結果値を記憶しておいたり、
それをまた別の関数に渡したりするくらいです。

・関数は、この先、数えきれないほど登場しますが、
それを逐一、丸暗記する必要は、まったくありません。

・必要になった時に、ﾘﾌｧﾚﾝｽﾏﾆｭｱﾙを読んだり、
ﾈｯﾄで検索するのが普通です。

・それから、実行速度が重視される場合には、
以下のことに留意して下さい。　↓

・関数内で使用する変数を、**3**つ以内に抑える。

・ｻｲｽﾞの小さい変数から宣言する。

・変数の宣言は、最初に書く。

・なるべく手短に書く。(ｲﾝﾗｲﾝ展開される)

・基本ﾃﾞｰﾀ型以外を、引数や戻り値に使う場合は、
ﾎﾟｲﾝﾀ型にしておき、ｱﾄﾞﾚｽを渡すようにする。

(構造体変数を渡したり、返したりすると、
すべてのﾒﾝﾊﾞの値がｺﾋﾟｰされてしまうため、
処理に時間がかかる。)

・一時的にしか使わない変数は、
「{ }」ﾌﾞﾛｯｸ内で宣言する。

関数の種類

・関数には、いくつかﾊﾟﾀｰﾝがあります。↓

```
/* 戻り値を返すもの。 */

int Tasu( const int a ,const int b )
{
    return ( a + b );
}
```

```
/* 戻り値のないもの。 */

void Hyouji( const char* p_namae )
{
     printf ( "私のﾏﾅｴは%sです。", p_namae );
}

/* 戻り値を返す必要がない場合は、
    戻り値のﾃﾞｰﾀ型のところに、void を書きます。 */
```

```
/* 渡された変数の値を変更するもの。   */

void Henkou( int* p_hensu )
{
     *p_hensu = 100;
}

/* 関数を呼び出すときに、変数を渡すと、
    変数の持つ値が渡されてしまう。

    変数の持つ値を変更する場合は、
    Henkou( &hensu1 );
```

```
    と書いて、変数のｱﾄﾞﾚｽを渡す必要がある。*/
```

```
/* 渡された配列の要素値を変更するもの。
    ( ※配列をそのまま渡すと、すべての値がｺﾋﾟｰされてしまう。 ) */

void Henkou(
                int hairetsu [] ,
                const int bangou ,
                const int atai
            )
           {
             hairetsu[ bangou ] = atai;
           }
```

```
/* 渡された構造体のﾒﾝﾊﾞ値を変更するもの。*/

void Henkou( Kouzoutai* p_kouzo )
{
     p_kouzo->denwa_bangou = 0120993906;
}
   /* ( ※構造体をそのまま渡すと、すべての値がｺﾋﾟｰされてしまい、
        大変な手間がかかるため、ｱﾄﾞﾚｽを渡している。
        ｱﾄﾞﾚｽであれば、構造体の持つ各ﾒﾝﾊﾞの値を、変更することもできる。)
```

```
/* 関数内で、ﾋｰﾌﾟ領域を割り当てる時は、ﾎﾟｲﾝﾀ・ﾎﾟｲﾝﾀを使う。  */

void Tsukuru( KouzouTai** pp_kouzo )
{
      *pp_kouzo = malloc( sizeof( KouzouTai ) );
}

/* 呼び出す時は、
    Kouzoutai* p_kouzo = NULL;
    Tsukuru( &p_kouzo ); */
```

・関数の中で宣言された変数のｱﾄﾞﾚｽは、
　その関数の外では使えませんので、
　「**return**」してはいけせん。

・引数の値を変更しない場合は、
　「**const**」を付けて、引数を定数にします。

```
/* 引数の数を、呼び出し時に、変更できる関数。 ( ※引数1は固定 ) */

#include <stdarg.h> /* 引数の値を取り出す関数が定義されている。 */

  int ZenbuTasu( int param_count_, ...) /* 引数2に、ﾋﾟﾘｵﾄﾞを3つ並べる。 */
  {
    va_list params; /* 引数の一覧を受け取る構造体。 */
     va_start( params, param_count_ ); /* 引数の一覧を取り出す。 */

    int total = 0; /* ｶｳﾝﾄ用の変数を、ﾘｾｯﾄしておく。 */

     for ( int pi = 0 ; pi < param_count_; pi++)
   {
         total += va_arg( params, int ); /* 引数の値を、取り出す。 */
           /* 呼び出すと、次の引数へ移る。 */
    }

    va_end( params ); /* 引数の一覧を解放する。 */

    return total; /* 合計値を返す。 */
  }
```

「関数ﾎﾟｲﾝﾀ」と「関数ﾃｰﾌﾞﾙ」

・「関数ﾎﾟｲﾝﾀ」は、
関数のｱﾄﾞﾚｽを入れておくための
専用のお皿です。

・代入できる関数は、
戻り値と引数の組み合わせが
関数ﾎﾟｲﾝﾀ型のものと同じものだけです。↓

```
/* 関数ﾎﾟｲﾝﾀ型「Keisan」の定義 */
typedef int (*Keisan) (int,int);

 /* 「Tasu」関数の定義 */
 int Tasu( int a , int b )
 {
     return ( a + b );
 }

 /* 「Hiku」関数の定義 */
 int Hiku( int a , int b )
 {
     return ( a - b );
 }

 /* 「main」関数の定義 */
 int main( )
 {

     Keisan p_keisan1 = Tasu;              /* 関数ﾎﾟｲﾝﾀを、関数名で初期化する。 */
     int kotae1 = (*p_keisan1)( 10, 20 ); /* これは、 Tasu( 10, 20 ); と同じ意味。 */

     /* 関数ﾎﾟｲﾝﾀを、配列に入れて使う場合は、↓ */
     Keisan table [] = { Tasu, Hiku };     /* 関数ﾎﾟｲﾝﾀの配列を、関数名で初期化する。 */
     int kotae2 = (*table[0])( 10, 20 ); /* これは、 Tasu( 10, 20 ); と同じ意味。 */
```

```
    int kotae3 = (*table[1])( 10, 20 );  /* これは、 Hiku( 10 , 20 );  と同じ意味。*/
}
```

・「**typedef**」のところでは、
関数ﾎﾟｲﾝﾀ型を定義していますが、
この場合は、「別名」は付けません。

・「**Keisan**」型は、
それ自体がﾎﾟｲﾝﾀ型ですから、
「Keisan*」とは書きません。

・「関数ﾎﾟｲﾝﾀ」に代入した関数を呼び出す時は、
「関数ﾎﾟｲﾝﾀ」の前に、「*」を付け、ｶｯｺで囲みます。

・「関数ﾎﾟｲﾝﾀﾃｰﾌﾞﾙ」(関数ﾎﾟｲﾝﾀの配列) を使うと、
「**switch**」文による分岐を無くすことができます。

・単純な「**switch**」文であれば、
ｺﾝﾊﾟｲﾙ時の「最適化」によって
「関数ﾎﾟｲﾝﾀﾃﾞｰﾌﾞﾙ」に置き換えられるので、
実行速度を計測しても、あまり差は見られません。

「ｸﾞﾛｰﾊﾞﾙ変数」と「ﾛｰｶﾙ変数」

・変数を宣言すると、ｼｽﾃﾑは、
ﾒﾓﾘ上に値の「置き場所」を確保します。

・最近は、**1TB** (ﾃﾗﾊﾞｲﾄ) のﾒﾓﾘを搭載したﾉｰﾄﾊﾟｿｺﾝが
ふつうに売られてたりしますが、
それでもﾒﾓﾘ容量には限りがあり、
そのままにしておくと、いずれ足りなくなります。

・物理ﾒﾓﾘが満杯になると、ｼｽﾃﾑは、
ﾊｰﾄﾞﾃﾞｨｽｸ上に「仮想ﾒﾓﾘ」を作り、
物理ﾒﾓﾘ上のﾃﾞｰﾀを、一時的に移すことで
空き領域を作ります。

・磁気ﾃﾞｨｽｸへのｱｸｾｽには時間がかかるため、
実行速度が低下して、画面が固まったりします。

・関数の中で宣言された変数は、「ﾛｰｶﾙ変数」といい、
関数定義の「 **{** 」「 **}** 」(中ｶｯｺ) の中でしか使えません。

・呼び出された側の関数内での処理が終了して、
呼び出し元の関数に戻った時点で、

関数内で宣言されていたﾛｰｶﾙ変数が使っていた
ﾒﾓﾘ上の「置き場所」も、解放されています。

・この、変数が変宣言されてから、解放されるまでの間を、
「変数の寿命」といいます。

```
int hensu1; /* 関数の外で宣言されているので、これは「ｸﾞﾛｰﾊﾞﾙ変数」 */

/* ---------------------------------------- */

int Tashizan( int a , int b )
{
   int hensu2 = a + b; /* これは関数の中で宣言されているので、「ﾛｰｶﾙ変数」。 */

    return hensu2;
}

/* ---------------------------------------- */

int main( )
{
   int hensu3 = Tashizan( 10, 20 );

   return 0;
}
```

・変数「hensu1」は、
関数の外で宣言されているので、
関数が終了しても使用できます。

・これを「ｸﾞﾛｰﾊﾞﾙ変数」といいます。

・「ｸﾞﾛｰﾊﾞﾙ変数」は、
引数として渡す必要がなく、

どの関数の中でも使えます。

・「ｸﾞﾛｰﾊﾞﾙ変数」が解放されるのは、
ﾌﾟﾛｸﾞﾗﾑの終了時です。

・下記のﾛｰｶﾙ変数「**hensu1**」は、
関数の中の、
さらに「**{**」「**}**」(中ｶｯｺ)で囲われた中で
宣言されています。↓

```
int main( )
{

  {
      int hensu1 = 10;
   }

   return 0;
}
```

・この変数「**hensu1**」は、
赤色の「**{**」「**}**」(中ｶｯｺ)で囲った中でしか
使用することができません。

・なぜかといえば、
このﾌﾞﾛｯｸから出ると、解放されてしまうからです。

「ｸﾞﾛｰﾊﾞﾙ変数」のextern宣言

「ｸﾞﾛｰﾊﾞﾙ変数」は、どの関数の中でも使えるのですが、
他のｿｰｽﾌｧｲﾙで使用するには、もう一工夫が必要です。↓

```
/* ﾍｯﾀﾞﾌｧｲﾙ 「 Test.h 」 */

extern int hensu1; /* ｸﾞﾛｰﾊﾞﾙ変数のextern宣言。 */
```

```
/* ｿｰｽﾌｧｲﾙ 「 Test.c 」 */

int hensu1 = 100; /* ｸﾞﾛｰﾊﾞﾙ変数の宣言と初期化 */
```

・たとえば、「 **Test2.c** 」 というｿｰｽﾌｧｲﾙの中で、
ｸﾞﾛｰﾊﾞﾙ変数「hensu1」を使用するには、
「**Test2.c**」か「**Test2.h**」の冒頭で、次のように書きます。↓

```
/* ﾍｯﾀﾞﾌｧｲﾙ 「 Test2.h 」 */

#include "Test.h" /* ﾍｯﾀﾞﾌｧｲﾙ「Test.h」 を ｲﾝｸﾙｰﾄﾞ宣言する。 */
```

・逆に、他のｿｰｽﾌｧｲﾙでは無効にしたい場合は、
「**static**宣言」をします。↓

```
static int hensu1; /* ｸﾞﾛｰﾊﾞﾙ変数のstatic宣言。 */
```

「ﾛｰｶﾙ変数」のstatic宣言

・関数内で宣言されたﾛｰｶﾙ変数は、
　呼び出しが終了すると、値がﾘｾｯﾄされます。

・ｶｳﾝﾀ変数など、値を残しておきたい場合は、
　「static宣言」をします。↓

```
void DaichanKazoeuta( )
{
   static hensu1 = 0; /* 最初の呼び出しの時だけ初期化される。 */

   ++hensu1; /* 前回の呼び出し時の値が残っているので、値は毎回増加していく。 */
}
```

・このように、ﾛｰｶﾙ変数に対するstatic宣言は、
　ｸﾞﾛｰﾊﾞﾙ変数のstatic宣言とは、まったく意味が異なります。

ﾋｰﾌﾟﾒﾓﾘ領域 (malloc free)

・変数が記憶される場所には、「ｽﾀｯｸ領域」と
「ﾋｰﾌﾟ領域」という区分があります。

・関数が呼ばれると、ｼｽﾃﾑは、
その関数の中で宣言されている「ﾛｰｶﾙ変数」の
「置き場」(領域) を、「ｽﾀｯｸ領域」上に確保します。

・そして、関数が終了して、呼び出し元に返ると、
この「置き場」は解放されます。

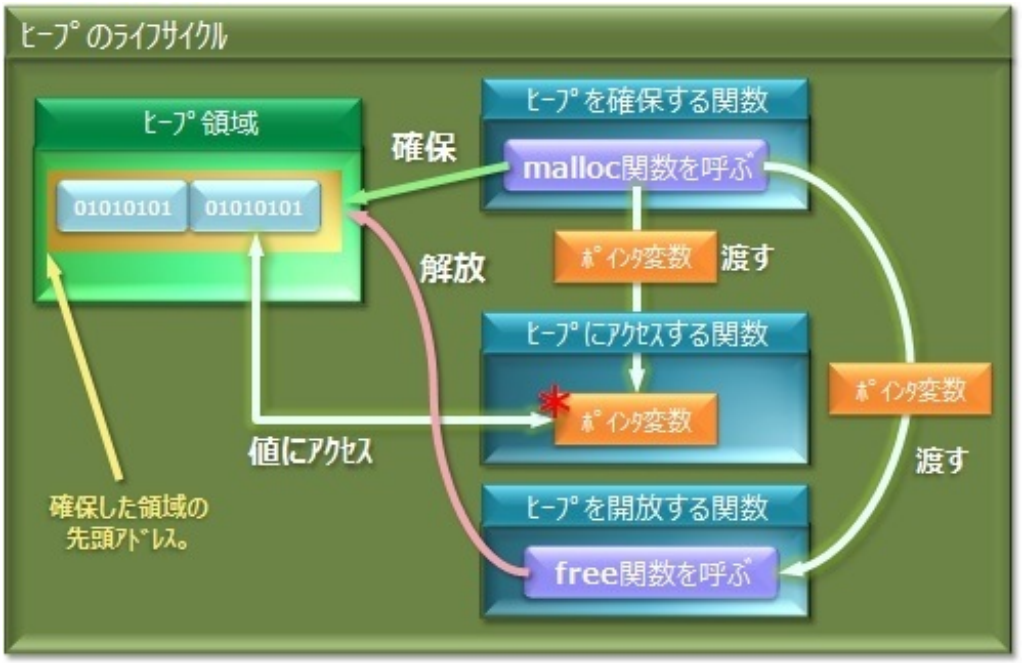
・解放されたくない場合は、関数の外で
「ｸﾞﾛｰﾊﾞﾙ変数」を宣言することもできました。

・しかし、大きめの配列を確保する場合は、
「ｸﾞﾛｰﾊﾞﾙ変数」だと、ﾌﾟﾛｸﾞﾗﾑが終わるまで
解放することができません。

・そういう場合は、「**malloc**」関数を使って、
「ﾋｰﾌﾟ領域」に「置き場」を確保します。↓

```
#include <stdlib.h>   /* malloc関数やfree関数が定義されている。 */

 int main( )
 {
    /* 要素数10個の配列を、ﾋｰﾌﾟ領域上に確保する。 */
    unsigned char* p1 = ( unsigned char* ) malloc( sizeof( unsigned char ) * 10 );
    *( p1 + 1 ) = 100;  /* 要素[1]に、「100」を代入する。 */
    free( p1 );         /* 領域を解放する。 */
```

```
    /* 構造体変数を、ﾋｰﾌﾟ領域上に確保する。 */
    KouzouTai* p2 = ( KouzouTai* ) malloc( sizeof( KouzouTai));
    p2->denwa = 993906; /* ﾒﾝﾊﾞ「denwa」に、「993906」を代入する。 */
    free( p2 );                 /* 領域を解放する。 */

    return 0;
}
```



・「**malloc**」関数は、引数で渡されたﾊﾞｲﾄ数の領域を、
「ﾋｰﾌﾟ領域」上に確保し、その先頭ｱﾄﾞﾚｽを返します。

・「**malloc**」関数の戻り値は「**void***」型ですので、
型ｷｬｽﾄをして、ﾎﾟｲﾝﾀ変数で受け取ります。

・「**void***」型のﾎﾟｲﾝﾀ変数には、
あらゆるﾃﾞｰﾀ型のｱﾄﾞﾚｽを代入できるのですが、
取り出す時に、型ｷｬｽﾄが必要です。

・「ﾋｰﾌﾟ領域」に確保した領域は、
「**free**」関数によって解放することができます。

・これを忘れると、領域が解放されずに残り続け、
ｼｽﾃﾑのﾊﾟﾌｫｰﾏﾝｽが低下し続け、
ｼｽﾃﾑがﾀﾞｳﾝすることがあります。

・これを「ﾒﾓﾘｰﾘｰｸ」といいいます。

・「sizeof」演算子は、渡したﾃﾞｰﾀ型の、
1個分のﾊﾞｲﾄ数を、戻り値として返しています。

・「ｽﾀｯｸ領域」は、比較的、小さな領域で、
実行中の関数のﾛｰｶﾙ変数を
記憶しておくのに使います。

・それに対して、「ﾋｰﾌﾟ領域」は、非常に広い領域で、
気軽に使うことができます。

・ところで、上記で確保された領域は、初期化されておらず、
無関係な値が入ったままになっています。

・領域全体を、0で初期化するには、
「memset」関数を使います。↓

```
memset( p1, 0, sizeof( unsigned char ) * 10 ); /* 指定ﾊﾞｲﾄ数の領域を初期化する。 */
/* 引数2が0なので、0で埋められる。 */
```

・引数1には、ﾛｰｶﾙの配列や、
構造体のｱﾄﾞﾚｽを渡すこともできます。

・引数2は、ﾊﾞｲﾄ数で、これはわかっている場合は、
「sizeof」演算子をわざわざ呼ぶ必要はありません。

・しかし、基本ﾃﾞｰﾀ型のﾊﾞｲﾄ数は、
ｺﾝﾊﾟｲﾗや、実行されるｼｽﾃﾑによって異なる場合があるため、
ﾃﾞｰﾀ型のｻｲｽﾞを求める時は、
「sizeof」演算子を使った方がいいと思います。

・「sizeof」演算子に関しては、注意点があって、
配列のｻｲｽﾞ(ﾊﾞｲﾄ数)を取得する場合なんですが、
これは、ﾛｰｶﾙで宣言された配列でないといけないということです。

・ﾋｰﾌﾟﾒﾓﾘ領域に確保した配列のｻｲｽﾞは、後で数えることができません。

・ということは、配列のｻｲｽﾞ (または要素数) は、どこか別の変数、
ｸﾞﾛｰﾊﾞﾙ変数にでもよけておかないといけないということです。

・領域の確保と同時に、初期化もしたい場合は、
「calloc」関数を使います。↓

```
unsigned char* p1 = ( unsigned char* ) calloc( 10, sizeof( unsigned char ) );
```

・領域のﾊﾞｲﾄ数を変更して、再確保する場合は、
「realloc」関数を使います。↓

```
realloc( p1, sizeof( unsigned char ) * 10 );
```

・それから、「ﾋｰﾌﾟ領域」の内容をｺﾋﾟｰするには、
「memcpy」関数を使います。↓

```
/* 「p_from」以降のﾊﾞｲﾄ配列を、
    「count」ﾊﾞｲﾄだけ、「p_to」以降にｺﾋﾟｰする。 */
memcpy( p_to, p_from, count );
```

・「memset」関数と「memcpy」関数は、

「**string.h**」で定義されています。

・ﾛｰｶﾙ変数は通常、
「ｽﾀｯｸ領域」上に確保されるため、
高速にｱｸｾｽできるのですが、
大きめの配列を、大量に確保したりすると、
満杯になることがあります。

・また、「ｽﾀｯｸ領域」上のﾛｰｶﾙ変数は、
関数が終了しないと解放されないため、
関数内から別の関数を呼び出すなど、
呼び出しのﾈｽﾄを深くしすぎるのも
小さなｼｽﾃﾑでは危険です。↓

---

main() → func1() → func2() → func3() .....

[mainの領域] [func1の領域] [func2の領域] [func3の領域]　...

・万が一、ｽﾀｯｸが満杯になると、ｼｽﾃﾑによって
ﾌﾟﾛｸﾞﾗﾑが強制終了させられてしまいます。

(これを 「ｽﾀｯｸｵｰﾊﾞｰﾌﾛｰ」 といいます。)

・したがって、実行速度を向上させるには、
次のことを心がけて下さい。↓

・ｶｳﾝﾀ用の変数など、小ぶりの変数はﾛｰｶﾙで宣言し、
大きめの配列は、「ﾋｰﾌﾟ領域」上に確保する。

(ｽﾀｯｸの容量は、ｼｽﾃﾑによって異なりますが、

ﾊﾟｿｺﾝの場合は、規定値が**1MB**だとかですから、
ﾛｰｶﾙで使うだけの小ぶりの配列なら、
ﾋｰﾌﾟに確保する必要はありません。
可変長の配列を確保する場合は、**_alloca**関数も使用できるが、
大きな配列をｽﾀｯｸ上に確保するのは、効率的ではない。)

・関数の呼び出し深度は、深くしないようにする。

(ふだんはそれほど意識しなくてもいいんですが、
延々と呼び出し続けるような処理は避けましょう。)

・一緒に使う値は、なるべく近くにまとめて配置する。

・ｸﾞﾛｰﾊﾞﾙ変数は使用しない。

【 ｽﾀｯｸ領域に可変長配列を確保する 】

・可変長のﾊﾞｯﾌｧ領域を確保する場合は、ﾋｰﾌﾟ領域上に確保する場合が多く、
そのように説明されていることが多いのですが、ｽﾀｯｸ上に確保することもできます。↓

```
/* #include <malloc.h> */

char* p_buf =(char*) _alloca( 100 ); /* 100byteの領域を確保する。*/
 /* この領域は、解放する必要はありません。関数が終了すると、自動的に解放されます。 */
```

```
/* #include <malloc.h> */

char* p_buf =(char*) _malloca( 100 ); /* 100byteの領域を確保する。*/
freea( p_buf  ); /* 明示的に解放する必要がある。 */
```

【 ﾊﾞｯﾌｧ・ｵｰﾊﾞｰﾗﾝ 】

・ﾋｰﾌﾟﾒﾓﾘに領域を確保する時に、
気をつけないといけないのが、
別の領域へのｱｸｾｽと、初期化のし忘れです。↓

```
char* p_nums = (char*) ::malloc( 4 ); /* 4byteの領域を確保する。 */

for ( int i = 0; i < 2; i++ ) /* 2回ﾙｰﾌﾟ。 */
{
   p_nums[i] = i; /* 要素番号0の値は0に、1の値は1に初期化される。 */
}

char num1 =p_nums[3]; /* ※要素番号3の値は、初期化されていない。 */
```

・このｺｰﾄﾞをｺﾝﾊﾟｲﾙして実行すると、
「0x00000005+を読み込み中にアクセス違反が発生しました。」
というような実行時ｴﾗｰが出て、ﾌﾟﾛｸﾞﾗﾑが停止したはずです。

・このｴﾗｰが出た場合は、原因箇所の特定が
非常に難しくなることがあります。

・というのも、「p_text」に書き込んでいる箇所だけが
原因とは限らないからです。

・ともあれ、さいしょに確認するのは、
ｶｳﾝﾄ数が配列の要素数を超えていないかです。

・非常に厄介なのが、初期化していないﾎﾟｲﾝﾀ配列などです。

・ﾎﾟｲﾝﾀ配列を確保した場合に、memset関数の呼び出しを怠ると、
ﾌﾟﾛｸﾞﾗﾑは、とんでもないｱﾄﾞﾚｽに書き込んだり、解放しょうとしたりします。

・こういう場合は、自分でも気付かずに書いていることが多いのですが、
他の人が作った関数にﾎﾟｲﾝﾀを渡す時には特に注意が必要です。

・**C標準関数**には、こうしたｴﾗｰを防ぐための、
　より安全な関数が定義されています。

・たとえば、「**memcpy_s**」では、ｺﾋﾟｰ先のﾊﾞｲﾄ数を指定します。

・また、実行時にｴﾗｰが出ても、ﾊﾝﾄﾞﾘﾝｸﾞされるようになっています。

【ﾋｰﾌﾟﾒﾓﾘに確保できる最大値】

・理論上は、「**SIZE_MAX**」定数 (size_t 型の最大値と同じ。) まで可能とされていますが、
　これは**4GB**であり、ﾌﾟﾛｸﾞﾗﾑが実行されるｺﾝﾋﾟｭｰﾀ (実行ｼｽﾃﾑ環境) の
　ﾒｲﾝﾒﾓﾘの容量が不足していると、確保に失敗します。

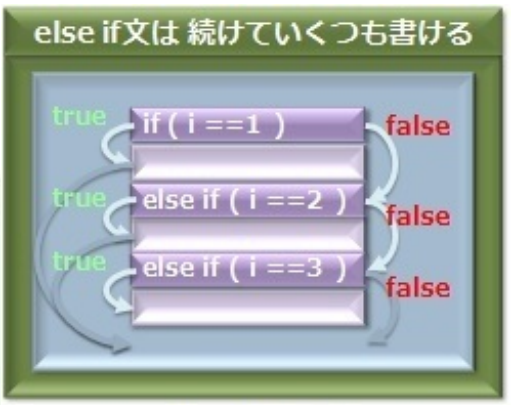
・一般的な**32bit**ｼｽﾃﾑでは、ひとつのｱﾌﾟﾘｹｰｼｮﾝﾌﾟﾛｸﾞﾗﾑで確保できる最大ｻｲｽﾞは、
　実際には、**2GB**までです。

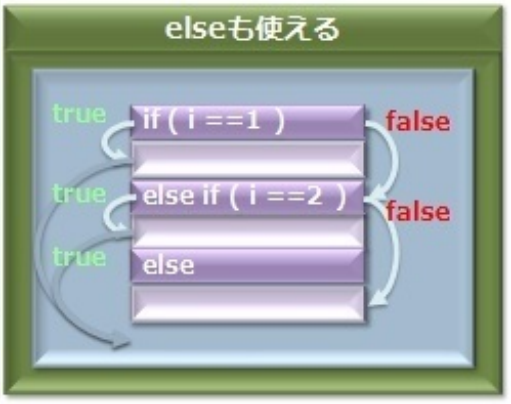
　1KB = 1024byte
　1MB = 1024KB = 1,048,576byte
　1GB = 1024MB = 1,073,741,824byte
　1TB = 1024GB = 1,099,511,627,776byte

　1KB = 2の10乗
　1MB = 2の20乗
　1GB = 2の30乗
　1TB = 2の40乗

・ちなみに、なぜ**32bit**ｼｽﾃﾑでは、理論上、**4GB**が限界なのかというと、
　それは、ﾎﾟｲﾝﾀが保持しているｱﾄﾞﾚｽ値が、**4byte**の整数値だからです。

条件分岐 (**if else switch**)





・ﾌﾟﾛｸﾞﾗﾑの処理は、
ﾕｰｻﾞｰの操作によって
幾重にも分岐していきます。↓

・「**A**ﾎﾞﾀﾝ」が押されたら、「ﾒﾆｭｰ画面」を開く。
・「**B**ﾎﾞﾀﾝ」が押されたら、音楽を再生する。

・これを、**C**言語で書くと、次のようになります。↓

```
/* ・「switch」文は、比較する値が、たくさんある場合に使う。

   ・ｶｯｺ内の値と、「case」の右横の数値が、一致するかを判定し、
         一致していたら、「：」 (ｺﾛﾝ) から「break;」までの処理を実行する。 */

 switch ( push_button ) /* この値と比較していく。 */
 {
   case BUTTON_A: /* 定数「BUTTON_A」なら、 */
```

```
        OpenMenu(); /* ﾒﾆｭｰを開く。 */
          break; /* 「switch」文の、中ｶｯｺの外に出る。 */

      case BUTTON_B: /* 定数「BUTTON_B」なら、 */
        PlayMusic(); /* 音楽を再生する。 */
          break; /* 「switch」文の、中ｶｯｺの外に出る。 */

      default: /* いずれの値にも合致しなければ、( 「default:」 は、必ず最後に書く。 ) */
          break; /* 「switch」文の、中ｶｯｺの外に出る。 */
  }

  /* 「break;」 すると、ここに来る。 */
```

・また、ﾌｧｲﾙの内容だとかも、一定とは限らず、
ﾌﾟﾛｸﾞﾗﾑは、その内容に応じて、
処理を切り替える必要があります。↓

```
int palette_count = p_bmp_file1->ReadPaletteCount();

if ( palette_count == 256 ) /* この条件に合致するなら、 */
{
    color_mode = PALETTE_COLOR; /* 中ｶｯｺ内の処理を行う。 */
}
else if ( palette_count == 0 ) /* 上記の条件に合致しなかった場合、この条件で判定する。 */
{
    color_mode = FULL_COLOR; /* 中ｶｯｺ内の処理を行う。 */
}
else /* 上記のいずれの条件にも合致しなかったら、 */
{
    /* このﾌﾟﾛｸﾞﾗﾑでは、対応していないので、処理を中断する。 */
    return false;
}
```

・また、ｼｽﾃﾑや機器が不調で、
関数が失敗した場合には、
それに対処するために、
別の処理を書いておいて、
切り替えないといけません。↓

```
bool kekka1 = PlayMisic( "C:\\Hamuta.mp3" ); /* 音声ﾌｧｲﾙを再生する。 */

if ( kekka1 == false ) exit(); /* 失敗したので、ﾌﾟﾛｸﾞﾗﾑを終了する。 */
else ShowMessage("再生に成功しました。" ); /* 成功したので、ﾒｯｾｰｼﾞを表示する。 */
```

・「switch」文の、ｶｯｺ内の値は、
変数でも、式でもいいんですが、
「case」の右横の値は、
定数でないといけません。↓

```
case 16: /* 「#define」で定義したﾏｸﾛ定数か、「const」宣言した定数でもOK。 */
```

・「break;」を書かなかった場合は、
すぐ下のcaseの処理へ流れて行ってしまい、
たとえ、そのcase値に合致していなくても、
処理が実行されてしまいます。↓

```
case OK:
    /* 「break;」していないので、このまま下へ流れる。↓ */

case NG:
    exit(); /* 「OK」の場合でも、ﾌﾟﾛｸﾞﾗﾑが終了してしまう。 */
```

・「if」文のｶｯｺの中は、条件式で、
比較式か、比較式同士の論理演算式を書きます。↓

```
if ( ( hensu1 > 0 ) && ( hensu2 > 0 ) )
else if ( hensu1 == 100 )
```

・「if」文の条件式の結果が「false」だった場合は、
すぐ下の「else if」文の条件式が判定されます。

・その結果がまた「false」なら、
次の「else if」が判定され、といった具合に、
下へ下へと下って行き、
いずれにも該当しなかった場合は、
「else」文の処理が実行されます。

・「else if」は、いくつ書いてもよく、
「else」もそうですが、省略できます。

・中ｶｯｺの中には、
さらに別の「switch」文や
「if」文を書くこともできます。↓

```
if ( h1 > 0 )
{
   if ( h2 > 0 )
   {
      /* 両方の条件に一致する場合は、この中ｶｯｺ内の処理が、実行される。 */
   }
}
else if ( ( h3 > 0 ) ) /* 上の最初の判定結果が「false」なら、ここの判定に移る。 */
```

```
{

}
```

・上記の処理の流れを要約すると、↓

・「h1」を判定する。→「true」なら、中の処理を実行する。(→「h2」を判定する。)
→「false」なら、「h3」を判定する。

・また、「if」文は、次のように書くこともできます。↓

```
hensu1 > 0 ? /* 「true」処理 */ : /* 「false」処理 */ ;
```

・これは、次の処理と同じです。↓

```
if ( hensu1 > 0 ) { /*「true」処理 */ }
else              { /*「false」処理 */ }
```

・これを、「三項演算子」といいます。

・「?」の左側が条件式で、
「?」から「：」(ｺﾛﾝ) までの1ｽﾃｰﾄﾒﾝﾄが、一致していた場合の処理で、
「：」(ｺﾛﾝ) の右側の1ｽﾃｰﾄﾒﾝﾄは、一致していなかった場合の処理です。

くり返し (for while do)

・指定した回数だけ、処理を繰り返したい時は、
「**for**」文を使います。 ↓

```
for ( int i = 0; i < item_count; i++ )
{
    /* 繰り返す処理を、ｺｺに書く。 */

    /* 処理が、1ｽﾃｰﾄﾒﾝﾄ (1つの命令文) しかない時は、中ｶｯｺは不要。 */
}
```

・まず最初に、ｶｳﾝﾀ変数「**i**」が、**0**で初期化されます。

(これは、初回に一度だけ、実行されます。)

・そして次に、ﾙｰﾌﾟ回数「**item_count**」との比較が行われ、
その結果が「**true**」なら、
中ｶｯｺ内の処理が実行されます。

・3つ目の、ｶｳﾝﾀ変数のｲﾝｸﾘﾒﾝﾄ (+1すること) は、
その後に行われます。

・ﾙｰﾌﾟが完了すると、ｶｳﾝﾀ変数の値は、
ﾙｰﾌﾟ回数と同じ値になっています。

・これはﾅｾﾞかといえば、
最終回の判定は、両方の値が等しくなるため、
(ｶｳﾝﾄ値 < 回数) が「**true**」 では無くなり、したがって、
その回の処理は実行されずに、ﾙｰﾌﾟを抜けます。

・わかりやすく書くと、次のようになります。 ↓

```
int i = 0;  /* ｶｳﾝﾄ用の変数を、初期化しておく。 */

for ( ; ; )
{
  if ( ( i < 10 ) == false ) break; /* 条件式の結果が「false」なら、終了する。 */

   /* くりかえし処理 */

  ++i; /* ｶｳﾝﾄ用の変数を、ｲﾝｸﾘﾒﾝﾄ (加算) する。 */
}
```

・途中でﾙｰﾌﾟを抜けたい時は、「**break**;」と書きます。

・今の回を中断して、次の回へ進みたい時は、
「**continue**;」と書きます。

・この**2**つは、これから説明する他のﾙｰﾌﾟ文でも使えます。

・回数以外の条件でくり返したい場合は、
「**while**」文を使います。↓

```
bool is_loop = true;
int count = 0;

while ( is_loop ) /* ｶｯｺ内の値がtrueの間は、中ｶｯｺ内の処理を繰り返す。 */
{
  /* 繰り返す処理を、このへんに書く。 */
```

```
    ++count; /* 一例としてこれは、ｶｳﾝﾀ変数を、ｲﾝｸﾘﾒﾝﾄ(加算)している。*/

    /* ｶｳﾝﾀ変数の値が10になったら、
       ﾌﾗｸﾞ変数を「false」にして、ﾙｰﾌﾟを抜ける。 */
    if ( count == 10 ) is_loop = false;

    /* 「 while ( count == 10 ) 」 と書く場合は、ｺｺで判定しなくてもいい。 */

}
```

・「while」文では、
まず最初に、条件式の判定が行われて、
その結果が「true」なら、
中ｶｯｺ内の処理が実行されます。

・そしてまた次の条件判定、というぐあいに、
結果が「false」になるまで、
この手順が繰り返されます。

・「do while」文だと、これが逆になり、
中ｶｯｺ内の処理を実行してから、

条件式の判定が行われます。↓

```
bool is_loop = true;
int count = 0;

do
{
  /* 繰り返す処理を、このへんに書く。 */

  ++count; /* 一例としてこれは、ｶｳﾝﾀ変数を、ｲﾝｸﾘﾒﾝﾄ(加算)している。*/

  /* ｶｳﾝﾀ変数の値が10になったら、
     ﾌﾗｸﾞ変数を「false」にして、ﾙｰﾌﾟを抜ける。 */
  if ( count == 10 ) is_loop = false;

}
while ( is_loop ); /* ｶｯｺ内の値がtrueの間は、中ｶｯｺ内の処理を繰り返す。 */
```

・「do while」文は、最初に必ず
　1回は、処理を実行します。

・それから、無限にﾙｰﾌﾟさせたい時は、次のように書きます。↓

```
bool end_flag = false;

for ( ; ; )
{
  /* くりかえし処理 */

  if (end_flag )break;
}
```

```
bool end_flag = false;

while ( true )
{
  /* くりかえし処理 */

  if ( end_flag ) break;
}
```

・通常は、中ｶｯｺ内で、条件式を判定して、
　一致したら、「break;」で脱出するように書きます。

・それから、「for」文には、
　さまざまな高速手法があります。↓

```
  int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; /* この配列の要素値を、すべて合算する。 */

  int *p_a = a; /* 配列の先頭ｱﾄﾞﾚｽを代入しておく。 */
  int*p_last = a+9; /* 最終要素のｱﾄﾞﾚｽを代入しておく。 */

  inttotal = 0;

  for (  p_a = a;  p_a !=p_last;  p_a++  )
        total+=*p_a;

        total += *p_a;

  /* p_last を a + 10 とした場合は、
     配列外のｱﾄﾞﾚｽにｱｸｾｽすることになります。

     そのため、最後の要素のｱﾄﾞﾚｽを終了ﾌﾗｸﾞとして使い、
     最終回は、for文の外で行なっています。 */
```

```
int item_count = 10;

for (int i = item_count; i; i--)
{
    printf("オラオラオラオラ!!!!! ( %d回目 ) \n",i + 1 );
}
```

・2つ目の、逆順ﾙｰﾌﾟでは、
ｶｳﾝﾄ用の変数「i」は、「10」からはじまります。

・毎回、処理が終わると、ﾃﾞｸﾘﾒﾝﾄ(-1すること)されて行きます。

・10回目の処理が終わると、
「i」は、「0」(つまり false)になるため、
処理が行われず、ﾙｰﾌﾟが終了します。

ﾗﾍﾞﾙ移動 (goto)

・ﾌﾟﾛｸﾞﾗﾑの処理を、何行か跳ばして
実行したい時があります。

そんな時は、「goto」文を使います。↓

```
void LoadBmp( )
{
        /* 関数が失敗したら、ﾗﾍﾞﾙ「SYUURYOU」までｼﾞｬﾝﾌﾟする。 */
        if ( OpenFile( ) == false ) goto SYUURYOU;

        /* ものすごく長い読み取り処理とかが、このﾍﾝにあるとする。 */

    SYUURYOU: /* このように、ﾗﾍﾞﾙ名の右横には、「：」 (ｺﾛﾝ) を付ける。 */

        /* このﾍﾝに、これまた長い終了処理があるとする。 */
}
```

・「goto」文は、非常に便利である反面、
それがｱﾀﾞとなって、処理の流れを
追いにくくなることがあります。

・使用はなるべく避けるべきですが、
どうしても必要な場合があります。↓

```
for (unsigned intyi = 0;yi <height;yi++)
{
    for ( unsigned int xi = 0; xi < width; xi++ )
    {
        /* ここで 「break;」 しても、
           2重 「for」 ﾌﾞﾛｯｸの外には出れない。 */
```

```
            if ( GetPixel( xi , yi ) == 0x0 ) goto SYUURYOU; /* これなら脱出できる。 */
        }
    }

    SYUURYOU: /* 脱出すると、ｺｺに跳んでくる。 */
```

・「**setjump.h**」には、別の関数へ
ｼﾞｬﾝﾌﾟできる関数が定義されています。↓

```
#include <setjmpex.h> /*  */

jmp_buf imakoko; /* 位置を記憶するｸﾞﾛｰﾊﾞﾙ変数。( ﾗﾍﾞﾙの代わりに使用する ) */

int main( )
{

    int result = _setjmp( imakoko ); /* 位置をｾｯﾄする。 */
     /* 初回の呼び出し時には0を返す。

        「longjmp」関数によって呼び出された場合は、
        その引数2で指定した番号が返る。*/

      /* 「longjmp」関数によって、ｾｯﾄ位置に戻った場合は、ｺｺにｼﾞｬﾝﾌﾟする。

          ( 一応、ｾｯﾄされた位置 ( _setjmp関数 )に戻っているらしく、
            「result」の番号は更新されているが、ﾌﾞﾚｲｸﾎﾟｲﾝﾄを付けても止まらない。 ) */

        longjmp( imakoko, 1 ); /* ｾｯﾄした位置までｼﾞｬﾝﾌﾟする。 */

         return 0;
}
```

・技術的には、関数の呼び出し時の各ﾚｼﾞｽﾀの値を
「**jmp_buf**」構造体に、すべて退避させておいて、
あとで戻すみたいなことをしているようです。

・「構造化ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ」以前の頃は、
とにかくﾌﾟﾛｸﾞﾗﾑを読み込むﾒﾓﾘ容量が少なくて、
また、**CPU**も極めて遅かったため、
前述の**goto**文と同様に、
ひんぱんに使われていたようですが、
やはり処理の流れが追いにくくなるので、
使わない方がいいと思います。

ﾏｸﾛ (#define)

・ﾌﾟﾛｸﾞﾗﾑを書いていると、
「こことここだけ、後で変更したいな～。」
と思う時があります。

・そんな時は、「ﾏｸﾛ」を使います。

・「ﾏｸﾛ」は、ｺﾝﾊﾟｲﾙの直前に実行され、
「ﾏｸﾛ名」が書かれた箇所は、
「定義された内容」に置換されます。

・「ﾏｸﾛ」の定義は、次のように書きます。↓

```
#define TESUURYOU  1980
```

・「#define」は、「ﾌﾟﾘﾌﾛｾｯｻ命令」ですので、
1ｽﾃｰﾄﾒﾝﾄの末尾に付ける「;」(ｾﾐｺﾛﾝ) は不要です。

・「#define」と「ﾏｸﾛ名」と「置換される内容」の間は、
半角ｽﾍﾟｰｽ1つ分は空けるようにして下さい。

・「ﾏｸﾛ」で置換できるのは、数値だけではありません。

・次の例では、計算式を、関数のように定義しています。↓

```
#define Tasu( a, b )  ( (a) + (b) )
```

・この「ﾏｸﾛ」を使って、

たとえば、
「 Tasu(hensu1, hensu2); 」
と書いた場合、

その箇所は、
「 ((hensu1) + (hensu2));」
に置換されます。

・「ﾏｸﾛ」では、使用できる変数は、
引数だけです。

・引数には、数値以外が渡されることもあるため、
ｶｯｺで囲っておきます。

・ﾃﾞｰﾀ型には制約はありませんが、
渡された値のﾃﾞｰﾀ型が違った場合は、
ｺﾝﾊﾟｲﾙｴﾗｰが出ることがあります。

・行を折り返して書く場合は、
行の最後に、「￥」記号を付けます。 ↓

```
#define Tasu( a, b )  ( (a) ¥
                      + (b) )
```

※上の例では、
「￥」が 「＼」 (ﾊﾞｯｸｽﾗｯｼｭ) に化けているかも知れませんが、 ↑
これは、「文字化け」しているだけですので、
ﾒﾓ帳などにｺﾋﾟｰ･ﾍﾟｰｽﾄすれば、元に戻ります。

※「￥」と改行の間に、ｽﾍﾟｰｽが入っていると、ｴﾗｰになります。

・複数のｽﾃｰﾄﾒﾝﾄを書きたい時は、

「, 」 (ｶﾝﾏ) で区切って書きます。 ↓

```
#define SWAP(a,b) ((a) ^= (b), (b) ^= (a), (a) ^= (b))
```

・printf関数のように、引数を 不定数、持たせることもできます。 ↓

```
#define PRINTF( format_, ... )( printf( format_, __VA_ARGS__ ) )
```

・「##」演算子を使ったﾏｸﾛでは、
変数名の一部を、引数で渡して使います。 ↓

```
#define HensuTasu( a, b ) ( (hensu##a) + (hensu##b) )

/* この場合の「引数」は、変数や値ではない。

    変数名「hensu」と、引数で渡された変数名を結合させて、
   新しい変数名を作っている。 */

void main( )
{
  int hensu1 = 10;
  int hensu2 = 20;

   int kotae = HensuTasu( 1, 2 ); /* hensu1 + hensu2 */
}
```

・「#」演算子を使ったﾏｸﾛでは、

文字列を渡すと、文字列ﾘﾃﾗﾙに置換されます。↓

```
#define ToLiteral( a ) ( (#a) )

void main( )
{
   char* p_ojiretsu = ToLiteral( あいうえお ); /* "あいうえお" という文字列ﾘﾃﾗﾙに置換される。 */
}
```

・それから、定義されたﾏｸﾛを
無効にしたい場合は、次のように書きます。↓

```
#undef HensuTasu
```

・定義されたﾏｸﾛの名前を、あとで変更することもできます。↓

```
#pragma push_macro( “HIKIZAN” )
#undef HIKIZAN
#define HIKU
#pragma pop_macro( “HIKU” )
```

・関数の場合は、ﾌﾟﾛｸﾞﾗﾑの実行時に呼ばれるため、
引数を渡すなどの「呼び出しｺｽﾄ」がかかりますが、
「ﾏｸﾛ」の場合は、ｺﾝﾊﾟｲﾙの直前に置換されるだけで済みます。

・定義済みのマクロについては、こちらをご覧ください。↓

→ 定義済みマクロ一覧 (msdn)

ｴﾗｰ処理 (abort exit)

・**C++**には、「例外」という便利な機能がありますが、
C言語だけでﾌﾟﾛｸﾞﾗﾑを書く場合は、
下記のような、伝統的な方法に頼らざるを得ません。↓

```
/* 「LoadTxtFile」関数が失敗して、falseを返したら、 */

if ( LoadTxtFile( "c:\\test.txt" ) == false )
{
    abort( ); /* ﾌﾟﾛｸﾞﾗﾑを終了して、ｴﾗｰﾒｯｾｰｼﾞを表示する。 */
}
```

・ﾌﾟﾛｸﾞﾗﾑを終了するだけなら、「**exit**」関数を呼びます。

・引数**1**には、異常終了なら「**0**」を、
正常終了なら「**1**」を指定します。

・両方とも、「**stdlib.h**」で定義されていますので、
使う場合は、ｲﾝｸﾙｰﾄﾞ宣言をしておいて下さい。

・それから、ｴﾗｰが起きた場所を、
ｺﾝｿｰﾙ (DOS画面) に表示させることもできます。↓

```
/* 「LoadTxtFile」関数が失敗して、falseを返したら、 */

if ( LoadTxtFile( "c:\\test.txt" ) == false )
{
   printf( "ｴﾗｰが発生しました。\n 場所は、 %s の %d 行目です。\n", __FILE__ , __LINE__ );
   printf( "最後にｺﾝﾊﾟｲﾙされたのは、 %s の、%s です。\n", __DATE__ , __TIME__ );
}
```

・「VisualStudio」を使っていれば、
使うことはありませんが、
その他の開発環境では、
ﾃﾞﾊﾞｯｸ中にひんぱんに使いますので、
その場合は、ﾏｸﾛにしておくと便利です。↓

```
#define OutputError printf( "ｴﾗｰが発生。\n 場所は、 %s の %d 行目。
\n", __FILE__ , __LINE__ );
```

・それから、「assert.h」には、
「assert」ﾏｸﾛが定義されていますが、
これは、渡した条件式の結果が「0」 (false) なら、
「abort」関数を呼び出しているだけです。

・わりとよく使うのが、「errno.h」です。↓

```
#include <stdio.h> /*  「printf」関数が定義されている。  */
#include <errno.h> /* ｸﾞﾛｰﾊﾞﾙ変数「errno」が宣言されている。  */

void main( )
{
  errno = 0; /* 前回のｴﾗｰが残ってると困るので、使う前には、ﾘｾｯﾄしておく。 */
  /* このへんでｴﾗｰが発生すると、その番号が、「errno」に代入される。 */
  perror( "ｴﾗｰが発生しました。" );
  /* 「perror」関数を呼び出すと、ｴﾗｰ番号に即したｴﾗｰﾒｯｾｰｼﾞが、
```

```
    引数1の文字列とともに、ｺﾝｿｰﾙ (DOS画面) 上に出力される。 */

    /* ｴﾗｰﾒｯｾｰｼﾞを表示するだけなら、これ↓でもOK。 */
    printf( "ｴﾗｰﾒｯｾｰｼﾞ：%s", strerror( errno ) );


}
```

・**C++**の「例外」も、これとよく似たことをやっていて、
ｴﾗｰが出そうな箇所を囲むように、ｴﾗｰ処理を書いておきます。

ﾜｲﾄﾞ文字列の加工 (基本編)

・「**wchar.h**」には、ﾜｲﾄﾞ文字列を加工するための
関数が定義されています。

・とはいえ、「**.Net**」や「**Java**」などと比べると、やはり使いにくいので、
近いものを自作しながら、使い方を説明して行きます。↓

```
/* 指定した文字数分だけｺﾋﾟｰする。 */

wchar_t* Copy(
                const wchar_t* p_src_,      /* ｺﾋﾟｰ元の文字列 */
                const size_t     src_length_ /* ｺﾋﾟｰ元の文字数 */
                 )
{
  /* ｺﾋﾟｰ先の配列を確保する。 ( ｺﾋﾟｰ元の文字数 + NULL文字 ) */
  wchar_t* p_result = ( wchar_t* ) ::malloc( sizeof( wchar_t ) * ( src_length_ + 1 ) );

   if ( ::wcsncpy_s( p_result,          /* ｺﾋﾟｰ先 */
                     src_length_ + 1, /* ｺﾋﾟｰ先の要素数。 ( ※文字数 + NULL文字 ) */
                      p_src_,           /* ｺﾋﾟｰ元 */
                      src_length_ )     /* この文字数分だけｺﾋﾟｰされる。*/
        == 0 )
   {
     /* 終端のNULL文字は、「wcsncpy_s」関数によって、自動的に付加される。 */
     return p_result;
   }
   else
   {
     ::free( p_result );
     return NULL;
   }
}
```

```
/* 2つの文字列を連結する。 */

wchar_t* Concat(
                const wchar_t* p_src1_,  /* ｺﾋﾟｰ元1 */
                const wchar_t* p_src2_   /* ｺﾋﾟｰ元2 */
                  )
{
  size_t src1_length = ::wcslen( p_src1_ ); /* ｺﾋﾟｰ元1の文字数を求める。 */
  size_t src2_length = ::wcslen( p_src2_ ); /* ｺﾋﾟｰ元2の文字数を求める。 */

  size_t dest_length = src1_length + src2_length + 1; /* ｺﾋﾟｰ先の要素数を求める。 */

  /* ｺﾋﾟｰ先の配列を確保する。 */

  wchar_t* p_result = ( wchar_t* ) ::malloc( sizeof( wchar_t ) * ( dest_length ) );

  /* ｺﾋﾟｰ元1をｺﾋﾟｰする。 */
  ::wcsncpy_s( p_result, dest_length, p_src1_, src1_length );

  /* その直後に、ｺﾋﾟｰ元2を連結する。 ( というか、ｺﾋﾟｰする。ｺﾋﾟｰよりも速い。 ) */
  ::wcsncat_s( p_result, dest_length, p_src2_, src2_length );

  return p_result;
}
```

```
/* 2つの文字列を比較する。 */

bool Compare(
               const wchar_t* p_text1_, /* 文字列1 */
               const wchar_t* p_text2_   /* 文字列2 */
                 )
{
   return ( ::wcscmp( p_text1_, p_text2_ ) == 0 ); /* 比較する。 */
}
```

```
/* 指定した文字列が、最初に出現する位置を検索する。 ( 見つからない場合は、falseが返る。 ) */
```

```
bool IndexOf(
                const wchar_t* p_src_text_, /* この文字列の中を検索する */
                const wchar_t* p_keyword_, /* 検索するｷｰﾜｰﾄﾞ文字列 */
                size_t*        p_result_     /* 最初に出現する位置 (1文字目の要素番号) */
                 )
{
   const wchar_t* p_find = ::wcsstr( p_src_text_, p_keyword_ ); /* 検索する。 */

   if ( p_find == NULL ) return false; /* 見つからなかったら、falseを返す。 */

    /* 位置を格納する。 (「wcsstr」関数は、ｱﾄﾞﾚｽを返すので、それ以降の文字数から求める。 )
*/
   *p_result_ = ::wcslen( p_src_text_ ) - ::wcslen( p_find );

   return true;
}
```

```
 /* 開始位置から終了位置までの文字数を返す。 */

size_t GetLength( const size_t start_index_, const size_t end_index_ )
{
      return end_index_ - start_index_ + 1;
}
```

```
/* 開始位置から終了位置までの文字列を返す。 */

wchar_t* GetRangeText(
                        const wchar_t* p_src_, /* この文字列から切り出す */
                        const size_t     start_index_,  /* 範囲の開始位置 */
                        const size_t     end_index_     /* 範囲の終了位置 */
                            )
{
   size_t dest_length = end_index_ - start_index_ + 1; /* ｺﾋﾟｰする文字数を求める。 */

   /* ｺﾋﾟｰ先の配列を確保する。 */
```

```
    wchar_t* p_dest = ( wchar_t* ) ::malloc( sizeof( wchar_t ) * ( dest_length + 1 ) );

    /* 範囲内の文字列を、ｺﾋﾟｰする。 */
    ::wcsncpy_s( p_dest, dest_length + 1, p_src_ + start_index_, dest_length );

    return p_dest;
}
```

・ﾜｲﾄﾞ文字列をｺﾋﾟｰする時は、
「**wcsncpy_s**」関数を使います。

・さらに、連結する時には、
「**wcsncat_s**」関数を使います。

・引数**1**は、ｺﾋﾟｰ先の配列で、
これは、直前に確保します。

・引数**2**は、ｺﾋﾟｰ先の配列の要素数で、
これは、ｺﾋﾟｰする文字数に、
終端の**NULL**文字分の**1**を、加算した数値です。

・引数**3**は、ｺﾋﾟｰ元の文字列で、
これは、**const** wchar_t* ですから、
文字列ﾘﾃﾗﾙでも構いません。

・引数**4**は、ｺﾋﾟｰしたい文字数で、
終端の**NULL**文字分は、含まれていません。

・それぞれの関数が完了すると、
ｺﾋﾟｰ先配列の終端には、
NULL文字が付加されていますが、
これは文字数としてはｶｳﾝﾄされません。

・「**wcslen**」関数が返す文字数には、
NULL文字分は、含まれていません。

・特に重要なのが、引数**2**と引数**4**で、
これが多すぎると、範囲外の領域にｺﾋﾟｰしてしまい、
「ﾋｰﾌﾟﾒﾓﾘが壊れました」というｴﾗｰが出ます。

ﾃｷｽﾄﾌｧｲﾙを読み込む。

```
#include <stdio.h>
#include <stdlib.h>

int main( )
{

    setlocale( LC_ALL , "Japanese_Japan.932" );

    /* ---------------------------- */
    /* ﾌｧｲﾙを開く。 ( ﾃｷｽﾄ読み取りﾓｰﾄﾞ ) */

    FILE* p_file = NULL;

    fopen_s( &p_file, "C:¥¥00_Doc¥¥ansi.txt", "r" ); /* 0が返れば成功。 */
    /* ﾜｲﾄﾞ文字の場合は、 _wfopen_s( &p_file, L"C:¥¥00_Doc¥¥ansi.txt", "r" ); */

    if ( p_file == NULL ) return 0;

    /* ---------------------------- */
    /* ﾌｧｲﾙｻｲｽﾞを求める。 ( ※ﾊﾞｲﾅﾘﾓｰﾄﾞでは無効 ) */

    fseek( p_file, 0, SEEK_END ); /* 終端に移動する。 ( ※引数2のｵﾌｾｯﾄ位置は0 ) */
    /* 0が返れば成功。 */

    long file_size = ftell( p_file ); /* 現在位置 ( 即ちﾌｧｲﾙｻｲｽﾞ ) を求める。 */
    /* 戻り値が-1でなければ成功。ftell関数は、fseek関数を呼んでない時は無効。 */

    fseek( p_file, 0, SEEK_SET ); /* 先頭に移動する。 ( ※引数2のｵﾌｾｯﾄ位置は0 ) */
    /* 指定位置に移動する場合は、引数2にｵﾌｾｯﾄ位置、引数3にSEEK_CURを渡す。 */

    /* ---------------------------- */
    /* ﾊﾞｲﾄﾃﾞｰﾀを読み取り、ﾊﾞｯﾌｧに格納する。 */
```

```
    int i = 0; /* ﾊﾞｯﾌｧ側の格納位置。 */

    unsigned char* p_buf =
        (unsigned char*) malloc( file_size ); /* ﾊﾞｯﾌｧを確保する。 */

    /* EndOfFileを返すまで、読み取りをくり返す。 */
    while ( ( p_buf[i] = (unsigned char) fgetc( p_file ) ) != EOF )
    {
    ++i; /* ﾊﾞｯﾌｧ側の格納位置を、1ﾊﾞｲﾄ進める。 */
    }

    p_buf[i] = 0; /* EOFの位置に、NULL文字を入れる。 */

    printf_s( "%s", p_buf ); /* 文字列を表示する。 */
    /* ﾜｲﾄﾞ文字の場合は、wprintf_s( L"%s", p_buf ); */

    /* ---------------------------- */

    fclose( p_file ); /* ﾌｧｲﾙを閉じる。 */

    free( p_buf ); /* ﾊﾞｯﾌｧを解放する。 */

    /* ---------------------------- */

    return 0;
}
```

・「fopen_s」関数は、引数1で指定したﾌｧｲﾙが
存在していない場合は、失敗します。 (戻り値は2)

・ﾃｷｽﾄﾌｧｲﾙは、「ﾒﾓ帳」で作って下さい。

・ﾌｧｲﾙを保存する時は、「ANSI」 (S-JIS) を選択して下さい。

・「wchar_t」型を使う場合は、「ﾜｲﾄﾞ文字」ですから、

「**Unicode**」 (UTF-16LE) を選択して下さい。

・「**C:¥¥ansi.txt**」だとか、ﾃﾞｨﾚｸﾄﾘのすぐ下に保存すると、
ｾｷｭﾘﾃｨｰ権限の関係で、ﾌｧｲﾙが開けないことがあります。

・ﾃｷｽﾄﾌｧｲﾙが、「ﾒﾓ帳」など、他のｿﾌﾄｳｪｱで、
すでに読み込まれている時も、ﾌｧｲﾙを開けません。

・ﾃｷｽﾄﾌｧｲﾙ内の改行ｺｰﾄﾞは、**Windows**では「¥r¥n」ですが、
ﾌｧｲﾙをﾃｷｽﾄﾓｰﾄﾞで開いた時は、「¥n」に変換されます。

・文字ｺｰﾄﾞを指定する場合は、「**fopen_s**」関数の引数2に、
「"r,ccs=<**UNICODE**>"」と書きます。

・「**UNICODE**」の箇所には、この他にも
「**UTF-8**」 (ﾏﾙﾁﾊﾞｲﾄ文字) または、
「**UTF-16LE**」 (ﾜｲﾄﾞ文字) を指定できます。

・ただし、ﾌｧｲﾙの中に、「**BOM**」が付加されていた場合は、
いずれを指定していても、無視されます。

・それから、**64bit**ｼｽﾃﾑでは、
ﾌｧｲﾙｻｲｽﾞの取得部分が少し異なります。 ↓

```
_fseeki64( p_file, 0, SEEK_END );
__int64 file_size = _ftelli64( p_file );
_fseeki64( p_file, 0, SEEK_SET );
```

・これは、**64bit**ｼｽﾃﾑの「**size_t**」型が
8byte(64bit)値となっているためです。

・**4GB**を超えるﾌｧｲﾙにも対応しています。

・その他にも、1行ずつ読み取るには、↓

```
char text1[ buf_size ];              /* ﾏﾙﾁﾊﾞｲﾄ文字列ﾊﾞｯﾌｧ */
fgets( text1, buf_size, p_file);   /* ﾏﾙﾁﾊﾞｲﾄ文字列を読み取る。 */
```

```
wchar_t text2[ buf_size ];          /* ﾜｲﾄﾞ文字列ﾊﾞｯﾌｧ */
fgetws( text2, buf_size, p_file ); /* ﾜｲﾄﾞ文字列を読み取る。 */
```

・「fgets」「fgetws」関数は、
改行ｺｰﾄﾞ (￥n) か、ﾌｧｲﾙの終端 (EOF) か、
または、「buf_size-1」になるまでﾌｧｲﾙを読み取り、
1つの文字列として、ﾊﾞｯﾌｧに格納します。

・ﾊﾞｯﾌｧに格納される文字列の終端には、
NULL文字 (￥0) が付加されています。

・引数2の値は、ﾊﾞｯﾌｧｻｲｽﾞですが、
これは多すぎても問題ありません。

・指定したﾊﾞｲﾄ数分をまとめて読み取る場合は、
「fread」関数を使います。↓

```
char text[10]; /* ﾏﾙﾁﾊﾞｲﾄ文字列ﾊﾞｯﾌｧ */
fread( (void*) text, 1, 10, p_file ); /* 1byte を 10つ、読み取る。 */
/* 途中に、「￥n」や「￥0」があっても、途切れない。 */
/* 「ﾜｲﾄﾞ文字」の場合は、引数2に、「2」を指定すると、引数3の文字数分が読み取られる。 */
```

・「ﾜｲﾄﾞ文字」を読み書きする時は、
Windowsは「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」でﾌｧｲﾙに書き込みますから、
上位**1**ﾊﾞｲﾄと下位**1**ﾊﾞｲﾄを、入れ替えます。 ↓

```
unsigned char b[2];
fread( (void*) b, 2, 1, p_file ); /* 2byte を 1つ、読み取る。 */
wchar_t moji = (wchar_t) ( ( b[1] << 8 ) | b[0]  ); /* ﾊﾞｹﾂﾘﾚｰの5倍速 */
```

・それから、指定した書式で読み取る場合は、 ↓

```
char text1[ 100 ];                                  /* ﾏﾙﾁﾊﾞｲﾄ文字列ﾊﾞｯﾌｧ */
fscanf( p_file , "私の名前は%dです。¥n", text1 ); /* 指定の書式で書き込む。 */
```

```
wchar_t text2[ 100 ];                                  /* ﾜｲﾄﾞ文字列ﾊﾞｯﾌｧ */
fwscanf( p_file , L"私の名前は%dです。¥n", text1 ); /* 指定の書式で書き込む。 */
```

ﾃｷｽﾄﾌｧｲﾙに書き込む。

```
#include <stdio.h>
#include <stdlib.h>

int main( )
{

   setlocale( LC_ALL , "Japanese_Japan.932" );

   /* ---------------------------- */
   /* ﾌｧｲﾙを開く。 ( ﾃｷｽﾄ書き込みﾓｰﾄﾞ ) */

    FILE* p_file = NULL;

    fopen_s( &p_file, "C:¥¥00_Doc¥¥ansi.txt", "w" ); /* 0が返れば成功。 */
     /* 読み込みも行いたい場合は、引数2で、 「"rw"」 を指定する。 */
     /* ﾜｲﾄﾞ文字の場合は、 _wfopen_s( &p_file, L"C:¥¥00_Doc¥¥ansi.txt", "w" ); */

    if ( p_file == NULL ) return 0;

    /* ---------------------------- */
    /* いろいろ書き込んでみる。 ( ﾏﾙﾁﾊﾞｲﾄ文字 ) */

    fputc( p_file, 'A' ); /* 文字を書き込む。 */

    char*p_text = "あいうえお¥nかきくけこ¥nさしすせそ¥0" ; /* 文字列ﾘﾃﾗﾙ。 */
    fputs(p_text, p_file ); /* 文字列を書き込む。 */

    int hensu1 = 100; /* 整数型の変数。 */
    fprintf( p_file , "変数の値は%dです。¥n", hensu1 ); /* 指定の書式で書き込む。 */

    /* ---------------------------- */
    /* いろいろ書き込んでみる。 ( ﾜｲﾄﾞ文字 ) */
```

```
    fputwc( p_file, L'A' ); /* 文字を書き込む。 */

    wchar_t* p_text = L"あいうえお¥nかきくけこ¥nさしすせそ¥0" ; /* 文字列ﾘﾃﾗﾙ。 */
    fputws( p_text, p_file ); /* 文字列を書き込む。 */

    int hensu1 = 100; /* 整数型の変数。 */
    fwprintf( p_file , L"変数の値は%dです。¥n", hensu1 ); /* 指定の書式で書き込む。 */

    /* ---------------------------- */

    fclose( p_file ); /* ﾌｧｲﾙを閉じる。 */

    free( p_buf ); /* ﾊﾞｯﾌｧを解放する。 */

    /* ---------------------------- */

    return 0;
}
```

・ﾃｷｽﾄﾓｰﾄﾞでﾌｧｲﾙを開いているので、
改行ｺｰﾄﾞ「**¥n**」は、書き込み時に、
「**¥r¥n**」に変換されます。

・指定したﾌｧｲﾙが存在していない場合は、
新規作成されます。

・ﾌｧｲﾙが存在している時は、上書きされます。

・ﾌｧｲﾙの終端から、続けて書き込みたい時は、
「**w**」の代わりに「**a+**」を指定します。

・ﾌｧｲﾙの終端を示す「**EOF**」は、自動的に書き込まれます。

・指定したﾊﾞｲﾄ数分を書き込む場合は、

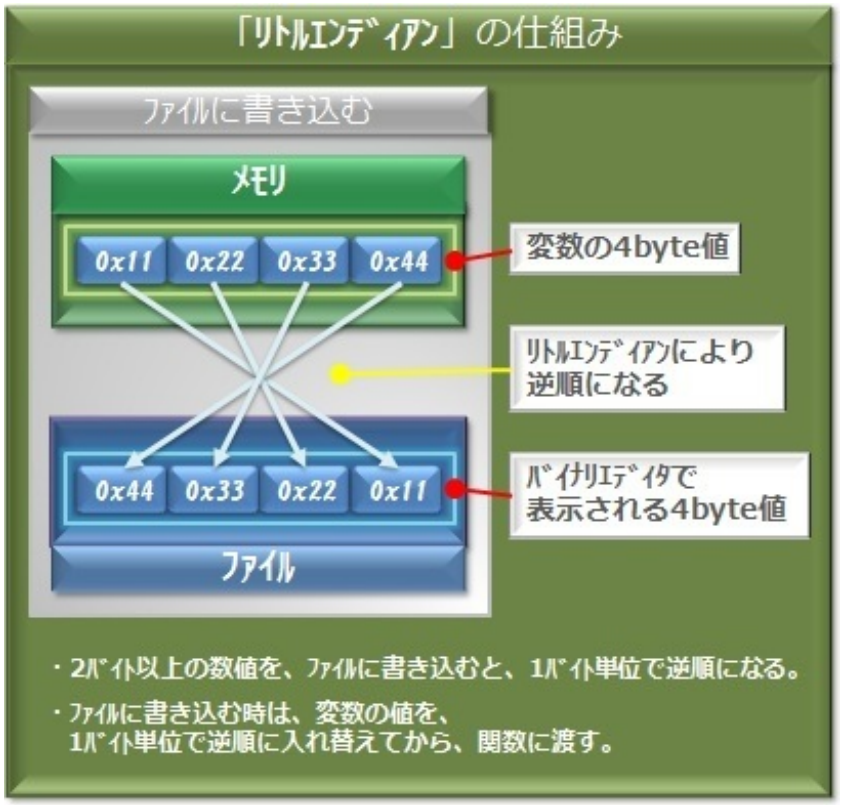
「fwrite」関数を使います。↓

```
char* p_text = "あいうえお¥nかきくけこ¥0";
fwrite( (void*) p_text, 1, 22, p_file );
/* 22ﾊﾞｲﾄ 書き込む。 ( 全角は2byte、制御文字は1byte ) */
 /* 「ﾜｲﾄﾞ文字」の場合は、引数2が「2」 (1文字のｻｲｽﾞ)、引数3が「12」 (文字数)。 */
 /* 途中に、「￥n」 や 「￥0」 があっても、途切れない。 */
```

・「ﾜｲﾄﾞ文字」を読み書きする時は、
Windowsは「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」でﾌｧｲﾙに書き込みますから、
上位**1**ﾊﾞｲﾄと、下位**1**ﾊﾞｲﾄを、入れ替えます。↓

```
wchar_t moji =L'あ';
unsigned char b[2];
b[0] = moji  &0xFF;
b[1] = ( moji >>8) & 0xFF;
fwrite( (void*) b, 2, 1, p_file ); /* 2byte を 1つ、書き込む。 */
```

「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」



・**Windows**では、「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」という方式で
ﾌｧｲﾙへの書き込みを行います。

・「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」方式のｼｽﾃﾑでは、
2byte以上の数値を、ﾌｧｲﾙに書き込むか、読み込むと、
1byte単位でﾊﾞｲﾄが逆順に並べ替えられます。

・**Mac**や**UINX**系は、「ﾋﾞｯｸｴﾝﾃﾞｨｱﾝ」という方式で、
並べ替えは行われず、従って、ﾌﾟﾛｸﾞﾗﾑ側でも
並べ替える必要はありません。

(**Java**で保存したﾌｧｲﾙも、「ﾋﾞｯｸｴﾝﾃﾞｨｱﾝ」です。)

・「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」で書き込んだﾌｧｲﾙは、
逆並びの箇所ができてしまうんですが、
これを「ﾋﾞｯｸｴﾝﾃﾞｨｱﾝ」形式のｼｽﾃﾑで読み取ると、
逆並びですから、値がおかしくなります。

・そういう場合に対応するには、
書き込む時と、読み込む時に、

並べ直しをします。

・「ﾌｧｲﾙ上の並びなんて、気にしない。」
という場合は、並べ直しは必要ありません。

(逆さまになったものを、また逆さまにするんで、元に戻る)

・しかし、同じ「ﾘﾄﾙｴﾝﾃﾞｨｱﾝ」のｼｽﾃﾑ間でも、
1ﾊﾞｲﾄずつ、読み書きしたりする場合がありますから、
そういう場合に対応するには、
読み込み時と、書き込み時の両方で並べ直しをして、
ﾌｧｲﾙ上の並びが、逆にならないようにします。

・並べ替えてから、書き込む場合は、次のように書きます。↓

```
unsigned short v = (unsigned short) ( (v_ & 0xFF) << 8 ) | ( (v_ >> 8) & 0xFF );
/* いろいろやっていますが、ﾊﾞｹﾂﾘﾚｰの5倍くらい速い。 */

fwrite( (void*) &v, 2, 1, p_file ); /* 2ﾊﾞｲﾄ値を書き込む。 */
```

```
unsigned int v = (unsigned int)
                 (
    ( (v_  & 0xFF)         << 24 )
  | ( ((v_ >> 8) & 0xFF)   << 16 )
  | ( ((v_ >> 16) & 0xFF)  << 8 )
  | ( (v_ >> 24) & 0xFF)
);

fwrite( (void*) &v, 4, 1, p_file ); /* 4ﾊﾞｲﾄ値を書き込む。 */
```

```
unsigned __int64 v = (unsigned __int64)
          (
    ( (v_ & 0xFF)          << 56 )
```

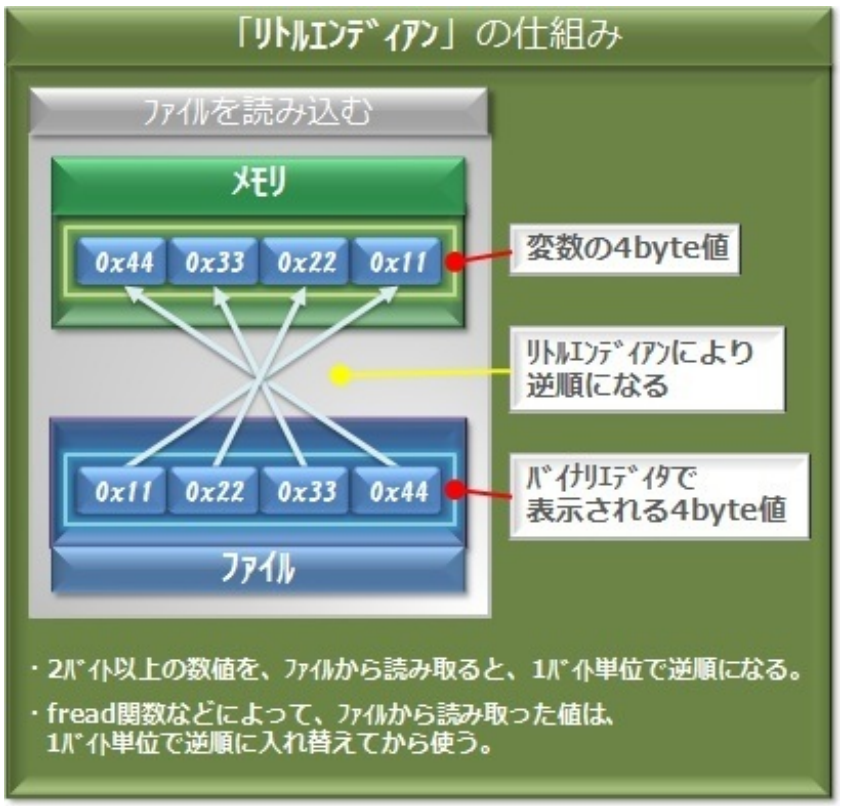
```
  | ( ( (v_ >> 8) & 0xFF)  << 48 )
  | ( ( (v_ >> 16) & 0xFF) << 40 )
  | ( ( (v_ >> 24) & 0xFF) << 32 )
  | ( ( (v_ >> 32) & 0xFF) << 24 )
  | ( ( (v_ >> 40) & 0xFF) << 16 )
  | ( ( (v_ >> 48) & 0xFF) << 8 )
  |   ( (v_ >> 56) & 0xFF)
);

   fwrite( (void*) &v, 8, 1, p_file ); /* 8ﾊﾞｲﾄ値を書き込む。 */
```

・そもそも、1ﾊﾞｲﾄずつ書き込めば、
　並べ替えは発生しません。↓

```
  unsigned int v = 0x11223344; // 元の4byte値。
  unsigned char b; // 1byteﾊﾞｹﾂ。

   b = ( v >> 24 ) & 0xFF; // 第1ﾊﾞｲﾄを取り出す。
   fwrite( (void*) &b, 1, 1, p_file ); /* 1ﾊﾞｲﾄ値を書き込む。 */

   b = ( v >> 16 ) & 0xFF; // 第2ﾊﾞｲﾄを取り出す。
   fwrite( (void*) &b, 1, 1, p_file ); /* 1ﾊﾞｲﾄ値を書き込む。 */

   b = ( v >> 8 ) & 0xFF; // 第3ﾊﾞｲﾄを取り出す。
   fwrite( (void*) &b, 1, 1, p_file ); /* 1ﾊﾞｲﾄ値を書き込む。 */

   b = ( v & 0xFF ); // 第4ﾊﾞｲﾄを取り出す。
   fwrite( (void*) &b, 1, 1, p_file ); /* 1ﾊﾞｲﾄ値を書き込む。 */
```

・ 配列の場合は、char型のﾎﾟｲﾝﾀ変数に、
　配列のｱﾄﾞﾚｽを入れておくと便利です。↓

```
int hairetsu1[] = { 0x11223344 ,0x55667788 }; // 元の4byte値の配列。
char* p = (char*) hairetsu1; // char型にｷｬｽﾄしておく。

for ( unsigned int i = 0; i < 2; i++ )
{
    fwrite( (void*) p, 1, 1, p_file ); /* 1ﾊﾞｲﾄ値を書き込む。 */
    ++p; // ﾎﾟｲﾝﾀを、一歩 (1byte) 進める。
}
```



・読み取った直後に並べ替える場合は、↓

```
unsigned char b[2];
fread( (void*) b, 2, 1, p_file ); /* 2ﾊﾞｲﾄ値を読み込む。 */

unsigned short v =  (unsigned short) ( ( b[1] << 8 ) | b[0] );
```

```
unsigned char b[4];
fread( (void*) b, 4, 1, p_file ); /* 4ﾊﾞｲﾄ値を読み込む。 */
```

```
    unsigned short v = (unsigned int) ( ( b[3] << 24 ) | ( b[2] << 16 ) | ( b[1] << 8 ) | b[0] );
```

```
    unsigned char b[8];
    fread( (void*) b, 8, 1, p_file ); /* 8ﾊﾞｲﾄ値を読み込む。 */

unsigned __int64 v = b[7];
      v <<= 8;

v |= b[6];
      v <<= 8;

v |= b[5];
      v <<= 8;

v |= b[4];
      v <<= 8;

v |= b[3];
      v <<= 8;

v |= b[2];
      v <<= 8;

v |= b[1];
      v <<= 8;

v |= b[0];
```

・どのﾀｲﾐﾝｸﾞでひっくり返っているのか、わかりますか？

・fwrite関数や、fread関数を呼び出しているところです。

ﾊﾞｲﾅﾘﾌｧｲﾙを読み込む。

・「ﾊﾞｲﾅﾘﾌｧｲﾙ」 (*.bin) は、ﾊﾞｲﾄﾃﾞｰﾀを、
そのままﾌｧｲﾙに書き込んだものです。

・ﾃｷｽﾄ以外のﾃﾞｰﾀを保存するのに使います。

・ﾌｧｲﾙを編集する時は、「ﾊﾞｲﾅﾘｴﾃﾞｨﾀ」を使います。↓

http://www.vector.co.jp/soft/win95/util/se079072.html

・ちなみに、「ﾃｷｽﾄﾌｧｲﾙ」を開いた場合は、
「文字ｺｰﾄﾞ」の配列が、「16進数表記」で表示されます。

・今回使用するｻﾝﾌﾟﾙﾌｧｲﾙには、わかりやすいように、
「00 11 22 33 44 55 66 77 88 99」と入力しておいて下さい。

・「文字ｺｰﾄﾞ」だとかいう区別はありませんが、
ﾏﾙﾁﾊﾞｲﾄ文字列を渡す方の関数を使っているため、
関数名の後に、「_Ansi」と、付けています。↓

```
#include <stdio.h> /* 標準入出力。 */
#include <stdlib.h> /* malloc free */

#include <io.h> /* 低水準入出力。 */
#include <fcntl.h>
#include <share.h>
#include <sys/types.h>
#include <sys/stat.h> /* 「_stat」関数が定義されている。 */

bool LoadBin_Ansi( const char* p_bin_path_ )
{
```

```
/* ---------------------- */
 /* ﾌｧｲﾙｻｲｽﾞを求める。（fseekによる方法が危険なため、fstatを使う。） */

 int h_file = 0; /* ﾌｧｲﾙのﾊﾝﾄﾞﾙ */

  _sopen_s( &h_file, p_bin_path_, _O_BINARY,
                _SH_DENYWR, _S_IREAD ); /* ﾌｧｲﾙを開く。 */

   struct _stat sb; /* ﾌｧｲﾙの情報を格納する構造体。 */
   /* 「_stat」という同名の関数があるため、
       「struct」を付けることで、構造体であることを明示している。 */

   _fstat( h_file,  &sb ); /* ﾌｧｲﾙの情報を取得する。 */
    /* 失敗したら、-1を返す。 */

   unsigned int file_size = sb.st_size; /* ﾌｧｲﾙｻｲｽﾞを取り出す。 */

    _close( h_file ); /* ﾌｧｲﾙを閉じる。 */

     /* ---------------------- */

     FILE* p_file = NULL; /* ﾌｧｲﾙﾎﾟｲﾝﾀ。 */

      /* ﾊﾞｲﾅﾘﾌｧｲﾙを開く。 */
      errno_t result = fopen_s( &p_file, p_bin_path_, "rb" );
      /* ﾊﾞｲﾅﾘﾌｧｲﾙの時は、引数2のｱｸｾｽ指定子の後ろに「b」を付ける。 */

       if ( p_file == NULL ) return false; /* 開けなかったら終了する。 */

       /* ---------------------- */

       int i = 0; /* ﾊﾞｯﾌｧの書き込み位置。 */

        unsigned char* p_buf =
        (unsigned char*) malloc( file_size ); /* ﾊﾞｯﾌｧを確保する。 */

         /* 「fread」関数の引数1は、読み取った値を格納するﾊﾞｯﾌｧへのﾎﾟｲﾝﾀ。
```

```
        引数2は、読み取るﾃﾞｰﾀの1つ分のﾊﾞｲﾄｻｲｽﾞ。
        引数3は、いくつ読み取るのか。短めのﾌｧｲﾙなら、ﾌｧｲﾙｻｲｽﾞでもいい。
        戻り値は、読み取ったﾊﾞｲﾄｻｲｽﾞ。 */

    while ( fread( p_buf + i, 1, 1, p_file ) > 0 ) /* 読み取りﾙｰﾌﾟ */
    {
        /* 読み取った1byteを、16進数で表示する。 */
        printf_s( "0x%02x ", (unsigned char) *( p_buf + i ) );

        ++i; /* 読み取った分だけ、書き込み位置を進める。 */
    }

    /* ---------------------- */

    fclose( p_file ); /* ﾊﾞｲﾅﾘﾌｧｲﾙを閉じる。 */

    free( p_buf ); /* ﾊﾞｯﾌｧを解放する。 */

    /* ---------------------- */

    return true;
}
```

・「_sopen_s」関数から「_close」関数の間で読み取る場合は、
「_read」関数を使います。 ↓

```
    unsigned char* p_buf =
      (unsigned char*) malloc( file_size ); /* ﾊﾞｯﾌｧを確保する。 */

    _read( h_file, (void*) p_buf, file_size );

    for ( int i = 0; i < file_size; i++ )
    {
        /* 読み取った1byteを、16進数で表示する。 */
```

```
    printf_s( "0x%02x ", (unsigned char) *( p_buf + i ) );

    }
```

・どちらの場合でも、結果は同じです。

・「**fopen_s**」でﾌｧｲﾙを開いた場合は、
読み書きしたﾃﾞｰﾀをﾊﾞｯﾌｧに貯めておいて、
まとめてﾌｧｲﾙに読み書きするため高速です。

・「**_sopen_s**」の場合は、
何回かに分けて読み書きする場合は低速ですが、
ﾘｱﾙﾀｲﾑに読み書きするため、ﾏﾙﾁｽﾚｯﾄﾞの場合に安全です。

・ﾊﾞｯﾌｧｻｲｽﾞを大きくすると、読み書き(ﾌｧｲﾙｱｸｾｽ)の回数が減るため、
読み込み処理を短縮できます。↓

```
size_t buf_size = 0x100000;                 /* 独自ﾊﾞｯﾌｧのｻｲｽﾞ。(1024KB) */
char* p_buf = (char*) malloc( 0x100000 );   /* 独自ﾊﾞｯﾌｧを確保する。 */
 setvbuf( p_file, p_buf, _IOFBF, 0x100000 ); /* 独自ﾊﾞｯﾌｧを設定する。 */

 /* 既定ﾊﾞｯﾌｧのｻｲｽﾞは、stdio.hのBUFSIZ定数が使用されている。 */
```

・ﾌｧｲﾙﾊﾟｽを、ﾜｲﾄﾞ文字列で渡したい時は、
「**_wsopen_s**」「**wfopen_s**」関数を使います。

ﾊﾞｲﾅﾘﾌｧｲﾙへの書き込み。

・低水準入出力による、ﾊﾞｲﾅﾘﾌｧｲﾙへの書き込み。 ↓

```
#include <io.h>
#include <fcntl.h>
#include <share.h>
#include <sys/types.h>
#include <sys/stat.h>

   bool SaveBin_Ansi( const char* p_bin_path_ )
   {
        /* ---------------------- */

        int h_file = 0; /* ﾌｧｲﾙのﾊﾝﾄﾞﾙ */

        errno_t result = _sopen_s( &h_file, p_bin_path_,
                                   _O_BINARY | _O_CREAT,
                                    _SH_DENYRD, _S_IWRITE ); /* ﾌｧｲﾙを開く。 */

         if ( result != 0 ) return false; /* 開けなかったら終了する。 */

          /* ---------------------- */

           unsigned char value = 0x00;

            for ( int i = 0; i < 10; i++ )
            {
                _write( h_file, (void*) &value, 1 );
                value += 0x11;
             }

              /* ---------------------- */
               _close( h_file ); /* ﾌｧｲﾙを閉じる。 */
```

```
        /* ---------------------- */

        return true;
}
```

・標準入出力による、ﾊﾞｲﾅﾘﾌｧｲﾙへの書き込み。 ↓

```
#include <stdio.h>

bool SaveBin_Ansi( const char* p_bin_path_ )
{
   /* ---------------------- */

   FILE* p_file = NULL; /* ﾌｧｲﾙﾎﾟｲﾝﾀ。 */

    errno_t result =
     fopen_s( &p_file, p_bin_path_, "wb" ); /* ﾊﾞｲﾅﾘﾌｧｲﾙを開く。 */

    if ( p_file == NULL ) return false; /* 開けなかったら終了する。 */

    /* ---------------------- */

     unsigned char value = 0x00;

     for ( int i = 0; i < 10; i++ )
     {
         fwrite( (void*) &value, 1, 1, p_file );
         value += 0x11;
      }

       /* ---------------------- */
       fclose( p_file ); /* ﾌｧｲﾙを閉じる。 */
       /* ---------------------- */
```

```
        return true;
}
```

・どちらの場合でも、
　指定したﾊﾞｲﾅﾘﾌｧｲﾙが新規作成され、
　「00 11 22 33 44 55 66 77 88 99」
　と書き込まれているはずです。

・ﾌｧｲﾙﾊﾟｽを、ﾜｲﾄﾞ文字列で渡したい時は、
　「**_wsopen_s**」「**wfopen_s**」関数を使います。

その他のﾌｧｲﾙ操作

```
/* ﾌｧｲﾙを削除する。 */
remove( "C:¥¥Test.test.txt" );    /* 「ﾏﾙﾁﾊﾞｲﾄ文字」版 */
_wremove( L"C:¥¥Test.test.txt" ); /* 「ﾜｲﾄﾞ文字」版 */
/* どちらの関数も、成功した場合は、「0」を返す。 */
```

```
/* ﾌｧｲﾙ名を変更する。 */
rename( "C:¥¥D¥¥old.txt", "C:¥¥D¥¥new.txt" );      /* 「ﾏﾙﾁﾊﾞｲﾄ文字」版 */
_wrename( L"C:¥¥D¥¥old.txt", L"C:¥¥D¥¥new.txt" ); /* 「ﾜｲﾄﾞ文字」版 */
/* どちらの関数も、成功した場合は、「0」を返す。 */
```

・この他にも、「**stdlib.h**」には、
ﾌｧｲﾙﾊﾟｽの最大文字数など、
便利なﾏｸﾛ定数が定義されています。↓

```
#define _MAX_DIR 256    /* ﾃﾞｨﾚｸﾄﾘ(ﾌｫﾙﾀﾞ)名の最大文字数。 */
#define _MAX_DRIVE 3    /* ﾊｰﾄﾞﾃﾞｨｽｸﾄﾞﾗｲﾌﾞ名の最大文字数。 */
#define _MAX_EXT 256    /* ﾌｧｲﾙ拡張子の最大文字数。 */
#define _MAX_FNAME 256 /* ﾌｧｲﾙ名の最大文字数。 */
#define _MAX_PATH 260   /* ﾌｧｲﾙﾊﾟｽの最大文字数 */
```

・これらのﾏｸﾛ定数は、
関数から、ﾌｧｲﾙﾊﾟｽなどを受け取る文字配列を
宣言する際に使用します。

ﾃﾞｨﾚｸﾄﾘの操作

・「direct.h」には、
「ﾃﾞｨﾚｸﾄﾘ」 (ﾌｫﾙﾀﾞ) を操作する関数が
定義されています。 ↓

```
/* ﾃﾞｨﾚｸﾄﾘを作成する。 */

int result = _mkdir( "c:¥¥Test" );

if ( result == -1 ) printf( "ﾃﾞｨﾚｸﾄﾘの作成に失敗しました。¥n" );
```

```
/* ﾃﾞｨﾚｸﾄﾘを削除する。

    ・ 削除できるのは、ｶﾗのﾃﾞｨﾚｸﾄﾘだけです。
    ・ﾙｰﾄﾃﾞｨﾚｸﾄﾘと、ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘも削除できません。 */

int result = _rmdir( "c:¥¥Test" );

if ( result == -1 ) printf( "ﾃﾞｨﾚｸﾄﾘの削除に失敗しました。¥n" );
```

```
/* ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘを変更する。 */

int result = _chdir( "c:¥¥Test" );

if ( result == -1 ) printf( "ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘの変更に失敗しました。¥n" );
```

```
/* ｶﾚﾝﾄﾄﾞﾗｲﾌﾞを変更する。 ( ﾊｰﾄﾞﾃﾞｨｽｸﾄﾞﾗｲﾌﾞ ) */

 /* cﾄﾞﾗｲﾌﾞ なら、1。 , d なら、 2。 eなら、3。 */
 int result = _chdrive( 1 );
```

```
if ( result == -1 ) printf( "ｶﾚﾝﾄﾄﾞﾗｲﾌﾞの変更に失敗しました。¥n" );
```

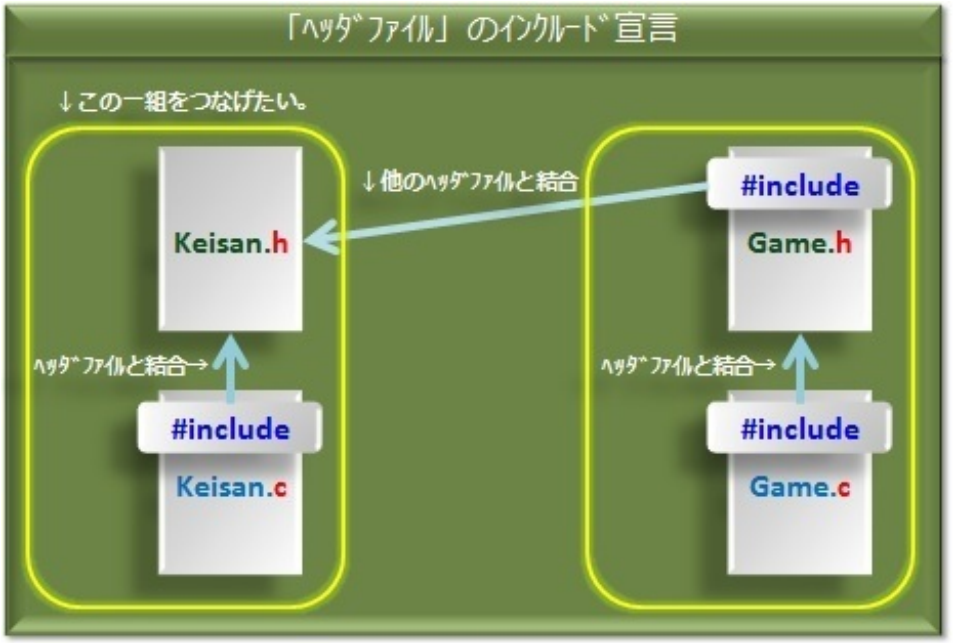
```
/* ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘ名を取得する。 */

 char cur_dir_path[ _MAX_PATH ];

 char* p_buf = _getcwd( cur_dir_path, _MAX_PATH );

if ( p_buf == NULL ) printf( "ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘ名の取得に失敗しました。¥n" );
```

```
/* 指定したﾄﾞﾗｲﾌﾞの、ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘ名を取得する。 */

 char cur_dir_path[ _MAX_PATH ];

 char* p_buf = _getdcwd( 1, cur_dir_path, _MAX_PATH );

if ( p_buf == NULL ) printf( "ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘ名の取得に失敗しました。¥n" );
```

```
/* ｶﾚﾝﾄﾄﾞﾗｲﾌﾞ (番号) を取得する。 */

int cur_drive_num = _getdrive();
```

```
/* 相対ﾊﾟｽから、絶対ﾊﾟｽを取得する。 */

 char full_path[ _MAX_PATH ];

 char* p_buf = _fullpath( full_path , ".¥test.txt", _MAX_PATH );

if ( p_buf == NULL ) printf( "絶対ﾊﾟｽの取得に失敗しました。¥n" );
```

ﾍｯﾀﾞﾌｧｲﾙの書き方



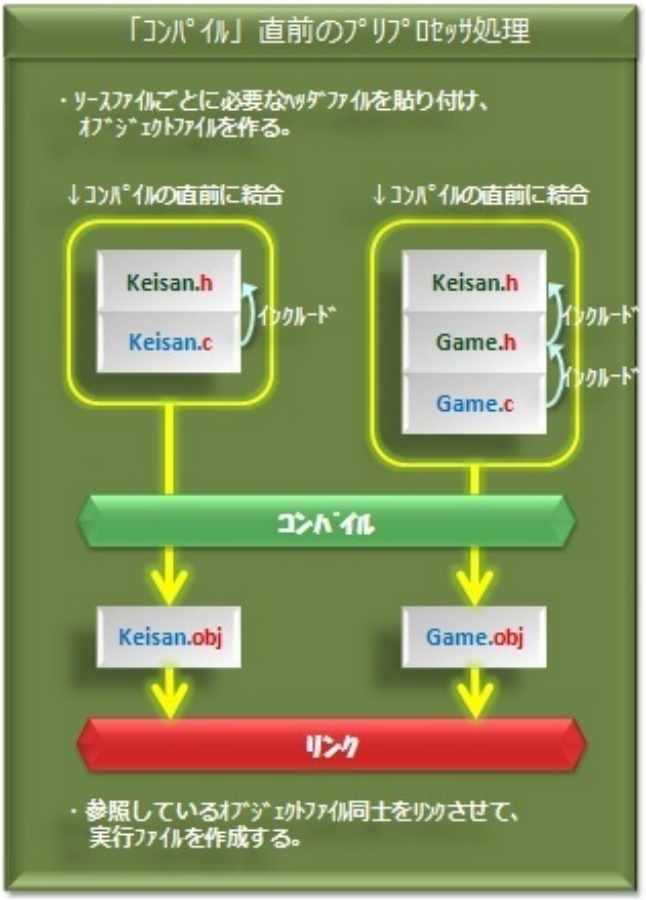
・「ﾍｯﾀﾞﾌｧｲﾙ」は、
「ｿｰｽﾌｧｲﾙ」のお品書きです。

・「ﾍｯﾀﾞﾌｧｲﾙ」ないし「ｿｰｽﾌｧｲﾙ」の冒頭で
「ｲﾝｸﾙｰﾄﾞ宣言」をすると、
その「ﾍｯﾀﾞﾌｧｲﾙ」上で定義されているﾃﾞｰﾀ型や
関数を使うことができるようになります。↓

```
/* ---------------------------------- */
/* ﾍｯﾀﾞﾌｧｲﾙ 「 Keisan.h 」 */

 /* ↓ 「関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言」。呼び出しに必要な情報だけを宣言する。 */
 int Tasu( int a , int b );
 int Hiku( int a , int b );

 /* その他にも、「#define」によるﾏｸﾛや定数の定義、
    構造体、列挙体、共用体のﾃﾞｰﾀ型の定義、
     「typedef」によるﾃﾞｰﾀ型の別名の定義も、ここに書く。 */

/* ---------------------------------- */
/* ｿｰｽﾌｧｲﾙ 「 Keisan.c 」 */

 #include "Keisan.h" /* ﾍｯﾀﾞﾌｧｲﾙ 「 Keisan.h 」 をｲﾝｸﾙｰﾄﾞ宣言する。 */

 int Tasu( int a , int b )
```

```
{
    return ( a + b );
}

int Hiku( int a , int b )
{
    return ( a - b );
}

/* ---------------------------------- */
/* ﾍｯﾀﾞﾌｧｲﾙ 「 Main.h 」 */

#include "Keisan.h" /* ﾍｯﾀﾞﾌｧｲﾙ 「 Keisan.h 」 をｲﾝｸﾙｰﾄﾞ宣言する。 */

/* ---------------------------------- */
/* ｿｰｽﾌｧｲﾙ 「 Main.c 」 */

#include "Main.h" /* ﾍｯﾀﾞﾌｧｲﾙ 「 Main.h 」 をｲﾝｸﾙｰﾄﾞ宣言する。 */

int main( )
{
   int wa = Tasu(10, 20 );
   int sa = Hiku( 10 , 20 );

   return 0;
}

/* ---------------------------------- */
```

・ｺﾝﾊﾟｲﾙを実行すると、「ﾍｯﾀﾞﾌｧｲﾙ」は、
「ｲﾝｸﾙｰﾄﾞ宣言」が書かれた箇所に書き込まれます。 ↓


「ｺﾝﾊﾟｲﾙ」直前のﾌﾟﾘﾌﾟﾛｾｯｻ処理
・ｿｰｽﾌｧｲﾙごとに必要なﾍｯﾀﾞﾌｧｲﾙを貼り付け、
ｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙを作る。
↓ｺﾝﾊﾟｲﾙの直前に結合
↓ｺﾝﾊﾟｲﾙの直前に結合
Keisan.h
Keisan.c
ｲﾝｸﾙｰﾄﾞ
Keisan.h
Game.h
Game.c
ｺﾝﾊﾟｲﾙ
Keisan.obj
Game.obj
ﾘﾝｸ
・参照しているｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙ同士をﾘﾝｸさせて、
実行ﾌｧｲﾙを作成する。

ｲﾝｸﾙｰﾄﾞｶﾞｰﾄﾞ

・ｿｰｽﾌｧｲﾙが増えてくると、
ｲﾝｸﾙｰﾄﾞによるｴﾗｰが出るようになります。

・まず、「重複ｲﾝｸﾙｰﾄﾞ」を防止するには、
ﾍｯﾀﾞﾌｧｲﾙの**1**行目に、次のように書きます。↓

```
#pragma once
```

・それから、有名なやり方で、
「ｲﾝｸﾙｰﾄﾞｶﾞｰﾄﾞ」というのがあります。↓

```
#ifndef _HAMUTA_H_
#define _HAMUTA_H_
    /* ここに処理を書く。 */
#endif
```

・「**_HAMUTA_H_**」の部分は何でもいいんですが、
他のﾍｯﾀﾞﾌｧｲﾙのものと、重複しないようにします。

・「**#ifndef**」は、
『右横の定数が、定義済みでなければ、
「**#endif**」までの部分をｺﾝﾊﾟｲﾙする。』
という意味です。

・ﾍｯﾀﾞﾌｧｲﾙは、ｺﾝﾊﾟｲﾙ時に、
別の複数のﾍｯﾀﾞﾌｧｲﾙやｿｰｽﾌｧｲﾙにｺﾋﾟｰされます。

・「重複ｲﾝｸﾙｰﾄﾞ」というのは、
この時、ｺﾝﾊﾟｲﾗが、

何度も同じﾍｯﾀﾞﾌｧｲﾙを読み取って、
何度も同じ変数や関数を
再定義、再宣言してしまう、というｴﾗｰです。

・ｺﾝﾊﾟｲﾗが、最初にﾍｯﾀﾞﾌｧｲﾙを読み取る際には、
「**_HAMUTA_H_**」は定義されていませんので、
「**#ifndef**」から「**#endif**」までが読み取られます。

・またこの時に、**2**行目の「**#define**」によって、
「**_HAMUTA_H_**」が定義されます。

・**2**回目以降からは、
「**_HAMUTA_H_**」は、すでに定義されているため、
「**#ifndef**」から「**#endif**」までの部分は、
読み取られずに無視されます。

・そしてもう一つのｴﾗｰは、
これは、**C/C++**でよく起こるｴﾗｰで、
原因箇所がわかりにくいんですが、
俗に「循環ｲﾝｸﾙｰﾄﾞ」（相互参照）といわれているもので、
これは、上記の**2**つの防止策では
防げません。

・このｴﾗｰを防止するには、
　次のことに留意して下さい。↓

・ｲﾝｸﾙｰﾄﾞ宣言は、
　可能な限り、ｿｰｽﾌｧｲﾙ側で行う。

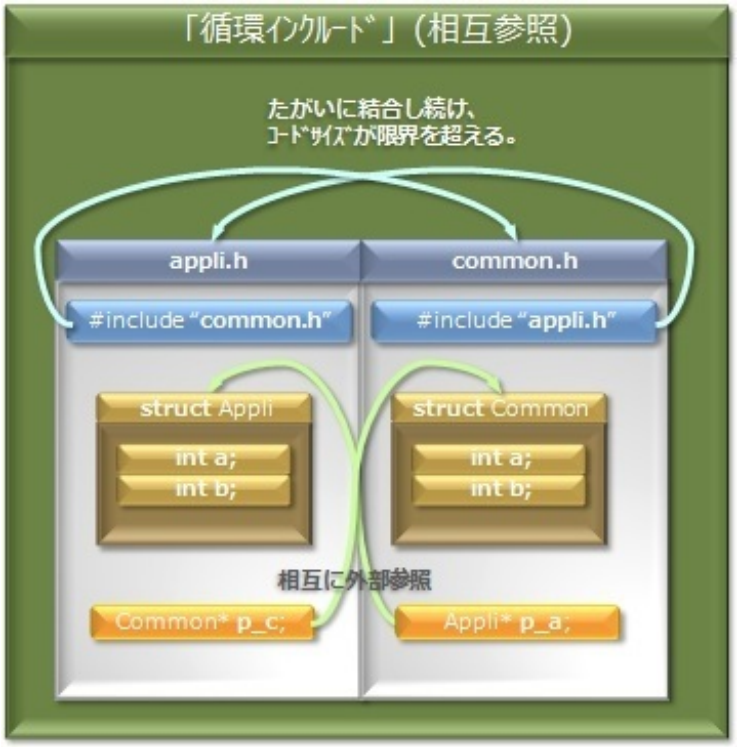
　他のﾍｯﾀﾞﾌｧｲﾙで定義されたﾃﾞｰﾀ型を
　ﾍｯﾀﾞﾌｧｲﾙ上で使っている場合は、
　前方宣言だけをして、ごまかす。

・ただし、派生ｸﾗｽを宣言する場合は、
　親ｸﾗｽのﾍｯﾀﾞﾌｧｲﾙは、
　ﾍｯﾀﾞﾌｧｲﾙ側でｲﾝｸﾙｰﾄﾞする。

・親ｸﾗｽが名前空間の中で宣言されている場合も、
　ﾍｯﾀﾞﾌｧｲﾙ側でｲﾝｸﾙｰﾄﾞする。

・ﾃﾝﾌﾟﾚｰﾄｸﾗｽのﾍｯﾀﾞﾌｧｲﾙも、
　ﾍｯﾀﾞﾌｧｲﾙ側でｲﾝｸﾙｰﾄﾞする。

・以上のことを守っていれば、
　まず遭遇することはないはずです。



・ちなみに、このｴﾗｰがよく起きるのは、
ﾍｯﾀﾞﾌｧｲﾙ同士が
相互にｲﾝｸﾙｰﾄﾞされているような場合です。

・ｸﾞﾛｰﾊﾞﾙ変数などを宣言したりするような
共通ﾍｯﾀﾞﾌｧｲﾙを使っている場合に、
共通ﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞしたﾍｯﾀﾞﾌｧｲﾙを、
共通ﾍｯﾀﾞﾌｧｲﾙ側からもｲﾝｸﾙｰﾄﾞしている。

・「ｲﾝｸﾙｰﾄﾞｶﾞｰﾄﾞ」もそうですが、
ｿｰｽﾌｧｲﾙ側でｲﾝｸﾙｰﾄﾞするようにすると、
不要なﾘﾝｸが無くなるため、ｺﾝﾊﾟｲﾙが早くなります。

ﾌﾟﾘﾌﾟﾛｾｯｻ命令

・「ﾌﾟﾘﾌﾟﾛｾｯｻ命令」は、ｿｰｽｺｰﾄﾞに対する置換処理で、
ｺﾝﾊﾟｲﾙの直前に実行されます。 ↓

```
#ifdef _DEBUG  /* ﾏｸﾛ定数「_DEBUG」 が 定義されていれば、true。 */
    /* trueなら、 この中の処理が、ｺﾝﾊﾟｲﾙされる。 */
#else
    /* falseなら、 この中の処理が、ｺﾝﾊﾟｲﾙされる。 */
#endif
```

```
#ifndef _DEBUG  /* ﾏｸﾛ定数「_DEBUG」 が 定義されていなければ、true。 */
    /* trueなら、 この中の処理が、ｺﾝﾊﾟｲﾙされる。 */
#else
    /* falseなら、 この中の処理が、ｺﾝﾊﾟｲﾙされる。 */
#endif
```

・最初の例だと、**true**ﾌﾞﾛｯｸの処理は、
ﾃﾞﾊﾞｯｸ時にだけｺﾝﾊﾟｲﾙされますから、
変数の値を表示したりするような処理を
この中に書いておきます。

・この処理は、ﾘﾘｰｽﾋﾞﾙﾄﾞをした時は、
ｺﾝﾊﾟｲﾙされません。

・「ﾌﾟﾘﾌﾟﾛｾｯｻ命令」というと、ﾍｯﾀﾞﾌｧｲﾙの上の方に書かれていることが多く、
関数の中には書けない、というｲﾒｰｼﾞがあるかも知れませんが、
実際には、関数の中でも書けます。 ↓

```
void Kansuu()
{
```

```
  #ifdef _WIN64
    /* 64bitに対応した処理を、ここに書く。 */
  #elseif
    /* 64bitに対応していない場合の処理を、ここに書く。 */
  #endif

   return;
}
```

・「ﾌﾟﾘﾌﾟﾛｾｯｻ命令」は、ｺﾝﾊﾟｲﾙの直前に
ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞの「置換」を行う処理ですので、
条件に該当しなかった部分は、
ｺﾝﾊﾟｲﾙ中は無視されます。

・条件式で分岐させる場合は、次のように書きます。↓

```
#define TENSU 80

/* 中略 */

#if TENSU > 50
   #define OKOZUKAI 1000
#else
   #define OKOZUKAI 100
#endif
```

・「**ifdef**」というのは、「**if define**」の略語で、
「条件として指定した定数が、定義済みであるなら、
trueﾌﾞﾛｯｸを有効にしてｺﾝﾊﾟｲﾙする。」 という意味です。

・逆に、「**ifndef**」は、「**if not define**」の略語で、
「条件として指定した定数が、定義されていなければ、
trueﾌﾞﾛｯｸをｺﾝﾊﾟｲﾙする。」 という意味です。

・それから、
「ｲﾝｸﾙｰﾄﾞｶﾞｰﾄﾞ」のﾍﾟｰｼﾞで登場した「**#pragma**」には、
他にも、いろいろな使い方があります。↓

```
/* ｽﾀｯｸのﾁｪｯｸを、有効にする。 */
#pragma check_stack on

/* ｽﾀｯｸのﾁｪｯｸを、無効にする。 */
#pragma check_stack off

/* 自動ｲﾝﾗｲﾝ展開を有効にする。 */
#pragma auto_inline on

/* 自動ｲﾝﾗｲﾝ展開を無効にする。 */
#pragma auto_inline off

/* 最適化ｵﾌﾟｼｮﾝ「/Oi」を、有効にする。 */
#pragma optimize /Oi on

/* 最適化ｵﾌﾟｼｮﾝ「/Oi」を、無効にする。 */
#pragma optimize /Oi off
```

【ｺﾝﾊﾟｲﾗの最適化ｵﾌﾟｼｮﾝ】

/Oi ... 「組み込み関数」に置き換える。
/Ox ... ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞのｻｲｽﾞを考えずに、最大限の最適化を行う。
/Og ... ﾛｰｶﾙ、および、ｸﾞﾛｰﾊﾞﾙな最適化、自動ﾚｼﾞｽﾀ割り当て、
および、ﾙｰﾌﾟの最適化を行う。使用は推奨されていない。
/O1 ... ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞのｻｲｽﾞを、最小にする。
/O2 ... 最高速度。 (ﾘﾘｰｽﾋﾞﾙﾄﾞの既定値。)
/Od ... ﾃﾞﾊﾞｯｸ時に、最適化を行わない。 (ｺﾝﾊﾟｲﾙの速度が速くなる。)
/Os ... 実行ﾌｧｲﾙのｻｲｽﾞ縮小化を優先させる。
/Ot ... 実行速度を優先させる。
/Ob0

/Ob1

/Ob2

・部分的に「ｱｾﾝﾌﾞﾗ」で書きたい時は、
「ｲﾝﾗｲﾝｱｾﾝﾌﾞﾗ」を使います。 ↓

```
int Add_Asm( int a_ , int b_ )
{
    int c;

    __asm
    {
       mov eax , a_ ; eax = a_;
       add eax , b_  ; eax += b_;
       mov c , eax   ; c = eax;
     }

     return c;
}
```

・これはただの足し算で、
あまり速くはありません。

・というのも、わざわざ「ｱｾﾝﾌﾞﾗ」で書かなくても、
ｺﾝﾊﾟｲﾙされた時に、「最適化」によって
機械語に置き換えられるからです。

・「ｱｾﾝﾌﾞﾗ」は、「機械語」の命令番号 (ｵﾍﾟｺｰﾄﾞ) に
名前 (ﾆｰﾓﾆｯｸ) をつけて、少し読みやすくしたものです。

・「**mov**」とか「**add**」とかいうのがﾆｰﾓﾆｯｸで、
後に続いている引数を「ｵﾍﾟﾗﾝﾄﾞ」といいます。

・「**eax**」というのは、**CPU**の内部にある記憶領域で、
「ﾚｼﾞｽﾀ」といわれているものの一つです。

・CPUに計算させる時は、計算させたい数値を、
この「ﾚｼﾞｽﾀ」にｾｯﾄしてから、命令を呼び出します。

・**FPU**命令や、**SIMD**系の組み込み関数は、それなりに効果があります。↓

```
// ------------------------------------<OK><ﾃｽﾄ済み>
// FPU (浮動小数点演算処理装置) で乗算する。

float Mul_FPU( float f1_, float f2_ )
{
  float f3;

  __asm
  {
    fld f1_ ; 左辺値をFPUﾚｼﾞｽﾀに積む。
    fld f2_ ; 右辺値をFPUﾚｼﾞｽﾀに積む。
    fmul    ; 乗算する。
    fstp f3 ; 結果値を取り出す。
  }

  return f3;
}

// ------------------------------------<OK><ﾃｽﾄ済み>
// FPU (浮動小数点演算処理装置) で除算する。

float Div_FPU( float f1_, float f2_ )
{
  float f3;

  __asm
  {
    fld f1_
```

```
    fld f2_
    fdiv
    fstp f3
  }

  return f3;
}

// ------------------------------------
```

・それと、「ｲﾝﾗｲﾝｱｾﾝﾌﾞﾗ」は、
64bitｼｽﾃﾑでは使用できません。↓

http://msdn.microsoft.com/ja-jp/library/vstudio/hb5z4sxd(v=vs.100).aspx

・「**MASM64**」↓
http://www.isus.jp/article/introduction-to-x64-assembly/
http://doremifasort.web.fc2.com/testprogram/index.html
http://msdn.microsoft.com/ja-jp/library/vstudio/hb5z4sxd.aspx

・「**SIMD**」(並列演算)の組み込み関数を使う場合は、こちらをどうぞ。↓

http://nf.nci.org.au/facilities/software/intel-ct/13.1.117/Documentation/ja_JP/compiler_c/main_cls/GUID-712779D8-D085-4464-9662-B630681F16F1.htm

・**MMX**命令というのは、**Pentium**の頃のものですが、
同じ**Intel**系(**x86**)のCPUなら、たいていは使えます。

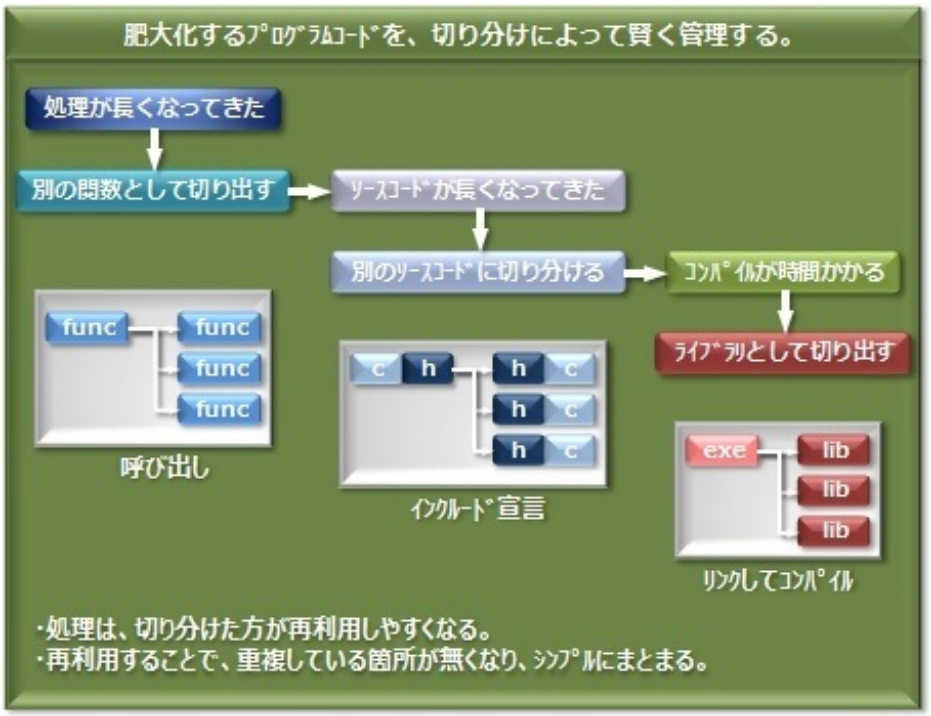
・その後継の**SSE**では、**SSE2**で浮動小数点の四則演算が定義されています。

・**SSE**は、'00年代の技術で、Intel系のほとんどのCPUで対応していますが、
'10年代以降は、**AVX2**と、後継のｼﾘｰｽﾞに移りつつあります。

・ﾍﾞｸﾄﾙ演算で並列処理しても、
賞味それだけ速くなるか、といえばそうでもないようですが、
「**OpenCV**」と比べても、かなり速くなるようです。

・ちなみに、「**OpenCV**」も、**SIMD**で実装されています。

ｽﾀﾃｨｯｸﾗｲﾌﾞﾗﾘ



・ﾌﾟﾛｸﾞﾗﾑが長くなってくると、
ｺﾝﾊﾟｲﾙする時に、
時間がかかるようになってきます。

・いま書いている箇所は、ひんぱんに書きなおしては
そのつど、ｺﾝﾊﾟｲﾙする必要がありますが、
すでに書き終わっていて、ある程度ﾃｽﾄもして、
動作が安定してる所まで毎回ｺﾝﾊﾟｲﾙするのは、時間の無駄です。

・ﾌﾟﾛｸﾞﾗﾐﾝｸﾞでは、こういったほとんど変更されない部分や、
別のﾌﾟﾛｸﾞﾗﾑでも使いまわせそうな普遍的な部分は、
「ﾗｲﾌﾞﾗﾘ」として切り分け、先にｺﾝﾊﾟｲﾙしておいて、
あとで「ﾘﾝｸ」させて使います。

・「ﾗｲﾌﾞﾗﾘ」の書き方は、通常のｱﾌﾟﾘｹｰｼｮﾝと同じですが、
次の**2**点については、少し異なります。↓

- 「ﾌﾟﾛｼﾞｪｸﾄの新規作成」では、「ｽﾀﾃｨｯｸﾗｲﾌﾞﾗﾘ」を選択する。
- 「ﾃﾞﾊﾞｯｸ」は、別の「ﾌﾟﾛｼﾞｪｸﾄ」でﾘﾝｸをして、呼び出す形で行う。

・「ﾗｲﾌﾞﾗﾘ」を「ﾘﾝｸ」して使う場合は、
呼び出し側のﾌﾟﾛｼﾞｪｸﾄで、次のように書きます。↓

```
#pragma comment( lib, "TestLibrary.lib" )
#include "TestLibrary.h"
```

・**1**行目は、「ﾗｲﾌﾞﾗﾘﾌｧｲﾙ」(*.lib)のﾘﾝｸです。

・ﾗｲﾌﾞﾗﾘ側のﾌﾟﾛｼﾞｪｸﾄをﾋﾞﾙﾄﾞすると、
そのﾌﾟﾛｼﾞｪｸﾄのﾌｫﾙﾀﾞ内に、作成されます。

・**2**行目は、ﾗｲﾌﾞﾗﾘ側のﾍｯﾀﾞﾌｧｲﾙのｲﾝｸﾙｰﾄﾞ宣言です。

・ﾌｧｲﾙﾊﾟｽは、ﾌﾙﾊﾟｽで書けば確実ですが、
面倒な場合は、よく参照するﾌｫﾙﾀﾞを、あらかじめ
「**VisualStudio**」で設定しておくこともできます。

・指定されたﾌｧｲﾙﾊﾟｽに不可分があったりして、
ﾘﾝｶが、ﾌｧｲﾙを見つけられない場合は、
「外部ｼﾝﾎﾞﾙが未解決」などといった
ﾘﾝｸｴﾗｰが表示されます。

・ﾗｲﾌﾞﾗﾘと同じｿﾘｭｰｼｮﾝ内に、
ﾃｽﾄ用のﾌﾟﾛｼﾞｪｸﾄを追加した場合は、
ｿﾘｭｰｼｮﾝのｱｲｺﾝを右ｸﾘｯｸして下さい。

・「ﾌﾟﾛｼﾞｪｸﾄの依存関係」をｸﾘｯｸすると、
ﾀﾞｲｱﾛｸﾞが表示されますので、
ここで、ﾗｲﾌﾞﾗﾘのﾌﾟﾛｼﾞｪｸﾄ名にﾁｪｯｸを入れて下さい。

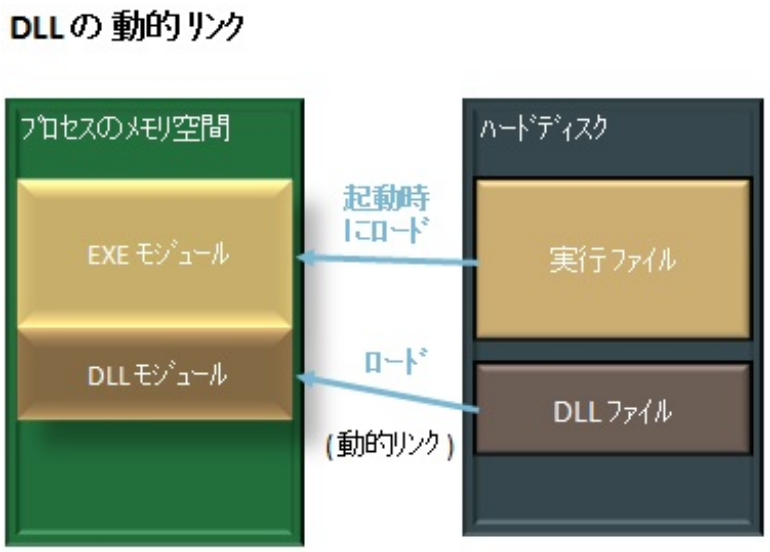
・それから、「ｽﾀｰﾄｱｯﾌﾟﾌﾟﾛｼﾞｪｸﾄの設定」をｸﾘｯｸして、

ﾃｽﾄ側のﾌﾟﾛｼﾞｪｸﾄ名にﾁｪｯｸを入れて下さい。

・あとは、#pragma comment(lib,"ﾗｲﾌﾞﾗﾘﾌｧｲﾙへのﾊﾟｽ")
と、ﾍｯﾀﾞﾌｧｲﾙのｲﾝｸﾙｰﾄﾞのﾊﾟｽが合っていれば、
他に原因がない限りは、前述のﾘﾝｸｴﾗｰは消えるはずです。

・この他にも、ﾗｲﾌﾞﾗﾘ側でｺﾝﾊﾟｲﾙｴﾗｰが残っていて、
ｺﾝﾊﾟｲﾙが完了していない場合もあります。

DLL (ﾀﾞｲﾅﾐｯｸﾘﾝｸﾗｲﾌﾞﾗﾘ)



・「ｽﾀﾃｨｯｸﾗｲﾌﾞﾗﾘ」は、
ｺﾝﾊﾟｲﾙ時にﾘﾝｸされていました。

・「ﾀﾞｲﾅﾐｯｸﾘﾝｸﾗｲﾌﾞﾗﾘ」は、
ﾌﾟﾛｸﾞﾗﾑの実行中に読み込んだり、(「ﾛｰﾄﾞ」という)
解放したりすることができます。(「ｱﾝﾛｰﾄﾞ」という)

・そのぶん、読み込み時間はかかりますが、
必要なところだけを読み込んで使いますから、
ﾒﾓﾘ効率はよくなります。

/* **DLL**側のヘッダファイル「 **dll_main.h** 」*/

```
#pragma once

#ifndef __DLL_MAIN_H__
#define __DLL_MAIN_H__

#include <targetver.h>
#include <windows.h>
#include <stdio.h>
```

```
/* DLLを呼び出す側からｲﾝｸﾙｰﾄﾞされた時は、「ｲﾝﾎﾟｰﾄ」に切り替わる。
   ( ※ ﾏｸﾛ定数「_EXPORTS」は、DLLのｺﾝﾊﾟｲﾙ時に、/D ｵﾌﾟｼｮﾝ で定義する。) */

#ifdef _EXPORTS
  #define _DLL_MEMBER __declspec(dllexport)
#else
  #define _DLL_MEMBER __declspec(dllimport)
#endif

extern "C"
{
  /* ｸﾞﾛｰﾊﾞﾙ変数のextern宣言。 */
   extern  _DLL_MEMBER int global_var;

   /* ｸﾞﾛｰﾊﾞﾙ関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言。 */
   _DLL_MEMBER void GlobalFunc( const wchar_t* p_text_ );
 }

   /* 「extern "C"」は、
            『C言語の構文規則を適用する』 という指示で、
           ｺﾝﾊﾟｲﾙ時に、関数や変数の名前が、自動的に変更されるのを
           抑制するために指定している。

           ( ※ これが適用された関数の宣言や実装内では、
              C++の文法で書いていると、ｺﾝﾊﾟｲﾙｴﾗｰになる。) */

  #endif
```

```
   /* DLL側のソースファイル 「dll_main.c 」 */

   #include "dll_main.h"

   /* ｸﾞﾛｰﾊﾞﾙ変数の宣言と初期化。 */
   int global_var = 0;

   /* ｸﾞﾛｰﾊﾞﾙ関数の実装。 */
   void GlobalFunc( const wchar_t* p_text_ )
```

```
  {
      printf( "%s", p_text_ );
  }

BOOL APIENTRY DllMain( HANDLE h_module_,
 DWORD ul_reason_for_call_, void* p_reserved_ )

{

        switch ( ul_reason_for_call_ )

        {

            case DLL_PROCESS_ATTACH: /* DLLがﾛｰﾄﾞされた。  */
            case DLL_THREAD_ATTACH: /* ｽﾚｯﾄﾞ開始。  */
            case DLL_THREAD_DETACH:  /* ｽﾚｯﾄﾞ終了。  */
            case DLL_PROCESS_DETACH: /* DLLがｱﾝﾛｰﾄﾞされた。*/
                  break;

        }

        return TRUE;

}
```

```
  /* 呼び出し側 TestCall.c */

#include <windows.h>
 #include "dll_main.h"

 typedef void (*GlobalFunc)(const wchar_t*);

 GlobalFunc p_func = NULL;

  int main( )
  {
      /* DLLを、動的にﾛｰﾄﾞする。 */
        HMODULE h_dll = LoadLibraryW( L"dll_main.dll" );

        if ( h_dll == NULL ) return 0; /* ﾛｰﾄﾞに失敗したら、ﾌﾟﾛｸﾞﾗﾑを終了する。 */

        /* DLL側で定義された関数のｱﾄﾞﾚｽを取得する。 */
```

```
        p_func = (GlobalFunc)
                GetProcAddress( h_dll, "GlobalFunc" );
        /* 引数2には、「序数」を渡してもいい。 */

        if ( p_func ) /* ｱﾄﾞﾚｽが取得できていたら、 */
          {
            (*p_func)( L"あいうえお¥0" ); /* 関数ﾎﾟｲﾝﾀから呼び出す。 */
        }

        FreeLibrary( h_dll ); /* DLLをｱﾝﾛｰﾄﾞする。 */

        return 0;
}
```

・**DLL** (動的ﾘﾝｸ) の方式は、2通りあります。

・ひとつは、「明示的ﾘﾝｸ」といって、上記のように、
ﾌﾟﾛｸﾞﾗﾑ実行時の任意のﾀｲﾐﾝｸﾞで、**DLL**を読み込み、ﾘﾝｸします。

・もうひとつは、「暗黙的ﾘﾝｸ」というもので、
これは、ﾌﾟﾛｸﾞﾗﾑの実行直後に、自動的に**DLL**が読み込まれ、
ﾘﾝｸされるというものです。

・この場合は、ｽﾀﾃｨｯｸﾗｲﾌﾞﾗﾘと同じように、
ﾌﾟﾛｸﾞﾗﾑのﾋﾞﾙﾄﾞ時に、**DLL**のﾗｲﾌﾞﾗﾘﾌｧｲﾙをﾘﾝｸします。

・少し違うのが、**DLL**ﾌｧｲﾙがあるという点で、
これは必ず、ｼｽﾃﾑから見える位置に置く必要があります。

(実行ﾌｧｲﾙが置かれているｶﾚﾝﾄﾌｫﾙﾀﾞに置いておけば確実。)

・**C++**から追加された構文を使用する場合は、
通常は、「**COM**」を使います。(後述)

「DEFﾌｧｲﾙ」と「序数」

・序数を設定しておくと、
　DLLを動的にﾘﾝｸする際に、
　ﾛｰﾄﾞ時間を短縮できます。

・関数の「序数」を指定するには、
　「DEFﾌｧｲﾙ」を書き換えます。↓

```
 ;DLL名の宣言↓(ﾛｰﾄﾞ時の先頭ｱﾄﾞﾚｽを指定するには、続けて「BASE=0x10000000」と書く。)
 LIBRARY KeisanDll

 ;ﾊﾞｰｼﾞｮﾝ番号↓(省略可。自分で採番する。)
 VERSION 1.0

 ;説明↓(省略可。ただのﾒﾓ書き。)
 DESCRIPTION "計算用の関数をまとめたﾗｲﾌﾞﾗﾘです。"

 EXPORTS

   ;関数の宣言↓(関数名の後の、「@」の後に、「序数」を書く。)
Tasu @1
Hiku  @2

   ;関数名を変更する場合は、↓
    Kakeru = Jousan

   ;変数の宣言↓
    hensu1 DATA
```

・「;」(ｾﾐｺﾛﾝ)から改行までがｺﾒﾝﾄです。

・「**EXPORTS**」ﾌﾞﾛｯｸ内の、宣言の並び順は、↓

関数名=別名 @序数 **NONAME PRIVATE DATA**

・関数名と序数は、省略できません。

・**NONAME**を指定すると、名無し関数になります。

(関数名では呼び出せなくなるが、ﾃｰﾌﾞﾙｻｲｽﾞは小さくなる。)

・**PRIVATE**を指定すると、ﾘﾝｸされない。

・**DATA**を指定すると、変数になる。

・**DEF**ﾌｧｲﾙを使う場合は、 **__declspec(dllexport)** は不要。

・ただし、 **__declspec(dllimport)** は必要。

・呼び出し側では、ﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞするだけで**OK**。

・実行中に、**DLL**を「ﾛｰﾄﾞ」する場合には、
序数を指定して、関数のｱﾄﾞﾚｽを取得する。

・大文字と小文字は区別される。

・ｽﾍﾟｰｽやｾﾐｺﾛﾝ (**;**) を含む長いﾌｧｲﾙ名は、
ﾀﾞﾌﾞﾙｸｫｰﾃｰｼｮﾝ (**"**) で囲む。

・ｼﾝﾎﾞﾙの間は、必ずｽﾍﾟｰｽで区切る。

・ｺﾛﾝ (**:**) やｲｺｰﾙ (**=**) の両側にも、ｽﾍﾟｰｽを入れる。

ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞ

・ﾌﾟﾛｼﾞｪｸﾄを作成する際に、
「ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞを使用する」を選択すると、
「**stdafx.h**」というﾌｧｲﾙが、ｲﾝｸﾙｰﾄﾞされるようになります。

・ﾌｧｲﾙを開いてみてみると、
stdio.h など、よく使うﾍｯﾀﾞﾌｧｲﾙがｲﾝｸﾙｰﾄﾞされています。

・これらのﾍｯﾀﾞﾌｧｲﾙとﾍﾟｱになっているｿｰｽﾌｧｲﾙは、
ﾌﾟﾛｼﾞｪｸﾄのｺﾝﾊﾟｲﾙが始まった直後に、まとめてｺﾝﾊﾟｲﾙされます。

・そのため、「**stdafx.h**」をｲﾝｸﾙｰﾄﾞしたﾌｧｲﾙでは、
ｺﾝﾊﾟｲﾙに要する時間が、大幅に短縮されます。

・ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞを使用する場合は、次のようにします。↓

---

1. ﾌﾟﾛｼﾞｪｸﾄに、下記のﾌｧｲﾙを追加する。(名前は自由)

pch.h
pch.cpp

---

2. ﾌﾟﾛｼﾞｪｸﾄの「ﾌﾟﾛﾊﾟﾃｨ」を開いて、
「ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞ」のところで、次のように入力する。↓

[ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞｰの作成/使用] [ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞｰﾌｧｲﾙを使用する (/Yu)]
[ﾌｧｲﾙを使用してPCHを作成/使用] [C:¥MyProject1¥**pch.h**]
[ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞｰﾌｧｲﾙ] [C:¥MyProject1¥**pch.pch**]

---

3. pch.cpp のｱｲｺﾝを右ｸﾘｯｸして、「ﾌﾟﾛﾊﾟﾃｨ」を開いて、

「ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞ」のところで、次のように入力する。↓

[ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞｰの作成/使用][ﾌﾟﾘｺﾝﾊﾟｲﾙ済みﾍｯﾀﾞｰﾌｧｲﾙを作成する (/Yu)]

---

4. 「**C/C++**」→「詳細」のところで、「必ずｲﾝｸﾙｰﾄﾞする」のところに、
pch.h を追加しておく。

---

5. pch.h を開いて、ﾌﾟﾘｺﾝﾊﾟｲﾙさせたいﾍｯﾀﾞﾌｧｲﾙを、ｲﾝｸﾙｰﾄﾞしておく。

```
#pragma once

#define _MY_PCH // ﾌﾟﾘｺﾝﾊﾟｲﾙを使わない場合は、この行をｺﾒﾝﾄｱｳﾄする。

#ifdef _MY_PCH
// ここにｲﾝｸﾙｰﾄﾞしていく。(自前のﾍｯﾀﾞﾌｧｲﾙは含めない。)
#endif
```

---

6. pch.cpp を開いて、**pch.h** のｲﾝｸﾙｰﾄﾞだけ書いておく。

```
#include "pch.h"
```

---

7. すべてのﾍｯﾀﾞﾌｧｲﾙで、ｲﾝｸﾙｰﾄﾞｶﾞｰﾄﾞの直後に、次のように書く。↓

```
#ifdef _MY_PCH
#include "pch.h"
#else
// ｲﾝｸﾙｰﾄﾞをここに書く。(自前のﾍｯﾀﾞﾌｧｲﾙは含めない。)
#endif
```

```
// 自前のﾍｯﾀﾞﾌｧｲﾙは、ここでｲﾝｸﾙｰﾄﾞする。
```

---

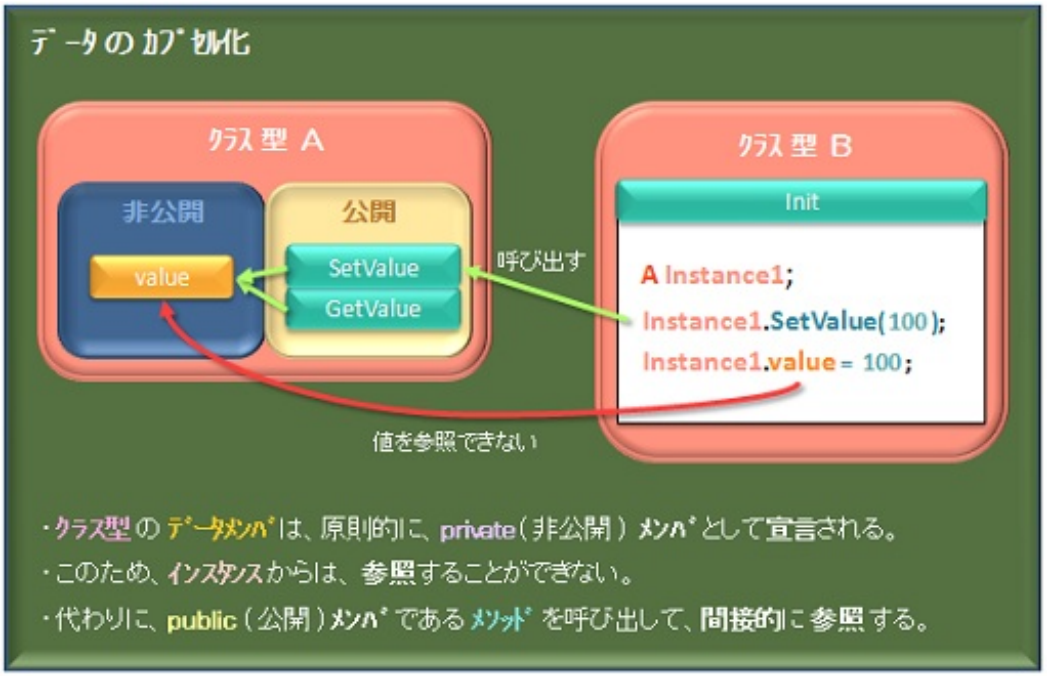
8. すべてのｿｰｽﾌｧｲﾙで、必ず最初に、次のように書く。↓

```
#include "pch.h"
#ifndef _MY_PCH
// ｲﾝｸﾙｰﾄﾞをここに書く。(自前のﾍｯﾀﾞﾌｧｲﾙは含めない。)
#endif

// 自前のﾍｯﾀﾞﾌｧｲﾙは、ここでｲﾝｸﾙｰﾄﾞする。
```

・手順2、3、4については、ﾋﾞﾙﾄﾞﾓｰﾄﾞごとに設定することになっています。

・ですから、たとえば、「DEBUG」を選択した状態で設定していても、
「RELEASE」に切り替えると、未設定のままだったりします。

C++のﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞを見てみよう。



・「C/C++」というﾌﾟﾛｸﾞﾗﾑ言語は、
Ｃ言語で「ｵﾌﾞｼﾞｪｸﾄ指向」のﾌﾟﾛｸﾞﾗﾑを書くために
開発された言語で、「ｸﾗｽ型」が使えます。

・前の章では、「C言語」が、
ﾒﾓﾘに直接ｱｸｾｽする言語であるということで、
ﾒﾓﾘｱｸｾｽの説明に重きを置いてきました。

・本章の「C/C++」では、
「ｸﾗｽ型」の書き方と、その使い方にあたる
「ｵﾌﾞｼﾞｪｸﾄ指向」についての説明が、最重要課題となっており、
これをある程度わかれば、あとはまあ
似たようなものですので、わりと簡単なはずです。

・人気のある「Objective-C」や「Java」、「C#」も、
この「ｵﾌﾞｼﾞｪｸﾄ指向」を使うための言語で、
基本的な書き方は、ほとんど同じです。

・Ｃ言語では、ﾒﾓﾘ上の配置を意識するだとか、
ｺﾝﾋﾟｭｰﾀ側の都合を優先して書いていました。

・まあ、C++も、その拡張言語ということで、

同じように気を配る必要があるんですが、
「ｵﾌﾞｼﾞｪｸﾄ指向」という考え方だとか、
「**Java**」などの、最近のﾓﾀﾞﾝな言語というのは、
人間寄りなところがあって、
ｺﾝﾋﾟｭｰﾀを意識しないで書けるというか、
意識しない方が、見通しが良くなるという
そういう世界なんですね。

・つまり、両方に目を配る必要があるということです。

```
// ----------------------------------------
// Control_Button.h

  #include "Control_ControlBase.h" // 親ｸﾗｽのﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞしておく。

  namespace Control // 名前空間
  {
   // 「Button」ｸﾗｽの宣言。（「ControlBase」ｸﾗｽからの派生ｸﾗｽ。）
   class Button : ControlBase
   {

    // ----------------------------------------
  protected: // 以下のﾒﾝﾊﾞは、このｸﾗｽの派生ｸﾗｽからはｱｸｾｽできる。

      HWND h_window; // ﾃﾞｰﾀﾒﾝﾊﾞ

    // ----------------------------------------
  public: // 以下のﾒﾝﾊﾞは、ｸﾗｽの外に公開されている。

      Button( );  // ｺﾝｽﾄﾗｸﾀ
      ~Button( ); // ﾃﾞｽﾄﾗｸﾀ

      HWND GetHandle( ); // Getterﾒｿｯﾄﾞ
      void SetHandle(HWND h_window_ ); // Setterﾒｿｯﾄﾞ

    // ----------------------------------------
```

```
    };

};

// ----------------------------------------
// Control_Button.cpp

#include "Control_Button.h"

namespace Control // 名前空間
{
    // ----------------------------------------
    // ｺﾝｽﾄﾗｸﾀ ( ｲﾝｽﾀﾝｽ化された直後に呼ばれる。 )

    Button::Button( ) : ControlBase( ) // 親ｸﾗｽのｺﾝｽﾄﾗｸﾀを呼ぶ。
    {
        // 初期化処理を、ｺｺに書く。
    }

    // ----------------------------------------
    // ﾃﾞｽﾄﾗｸﾀ ( ｲﾝｽﾀﾝｽが破棄される直前に呼ばれる。 )

    Button::~Button( )
    {
        // 解放処理を、ｺｺに書く。
    }

    // ----------------------------------------
    // Getterﾒｿｯﾄﾞ ( ﾃﾞｰﾀﾒﾝﾊﾞを返すﾒｿｯﾄﾞ )

    HWND Button::GetHandle( )
    {
        return h_window;
    }

    // ----------------------------------------
```

```
    // Setterﾒｿｯﾄﾞ (ﾃﾞｰﾀﾒﾝﾊﾞを渡すﾒｿｯﾄﾞ)

    void Button::SetHandle( HWND h_window_ )
    {
        h_window = h_window_;
    }

    // ----------------------------------------
}

// ｿｰｽﾌｧｲﾙ側には、中ｶｯｺの後に、「；」(ｾﾐｺﾛﾝ) が無い。
// ----------------------------------------
```

・「ｸﾗｽ型」というのは、構造体型の一種で、
「ﾒｿｯﾄﾞ」(手続き) という関数を、ﾒﾝﾊﾞとして持っています。

・「ｸﾗｽ型」のﾃﾞｰﾀﾒﾝﾊﾞは、原則的に、
class1.member1 = 10; というように、
直接ｱｸｾｽすることができません。

(できなくはないのですが、安全性のために
できないように書くのです。)

・ﾃﾞｰﾀﾒﾝﾊﾞにｱｸｾｽしたり、加工する場合は、
「ﾒｿｯﾄﾞ」の中で処理を書いておいて、
これを呼び出すことによって、
間接的に、ﾃﾞｰﾀﾒﾝﾊﾞを操作します。

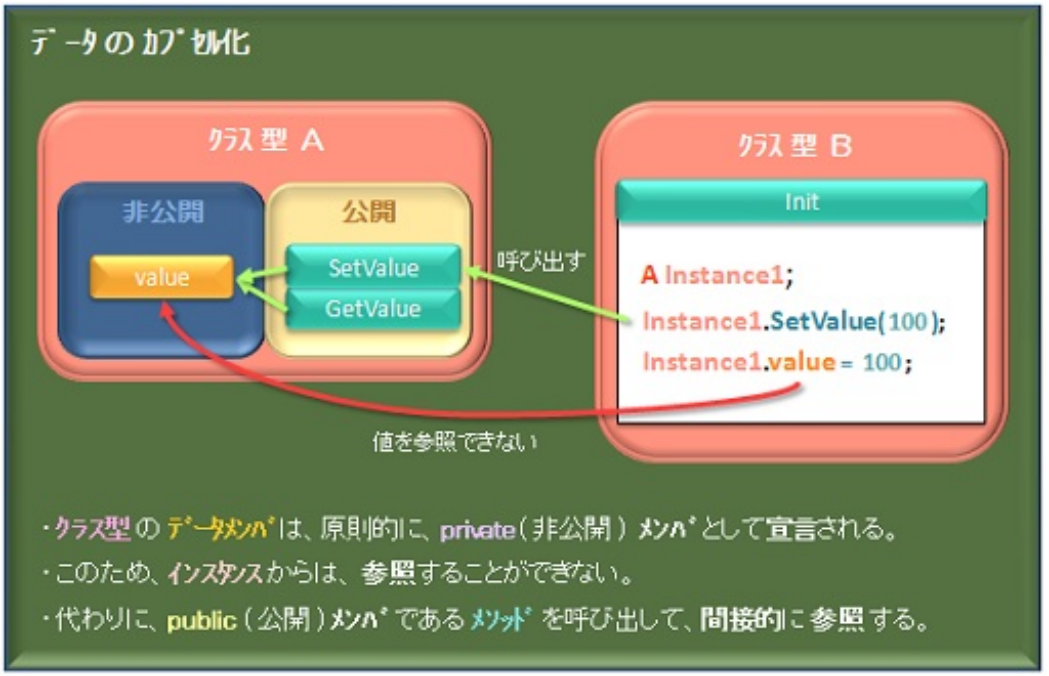
・このことを 「ｶﾌﾟｾﾙ化」 とか、
「隠蔽」(いんぺい) といいます。

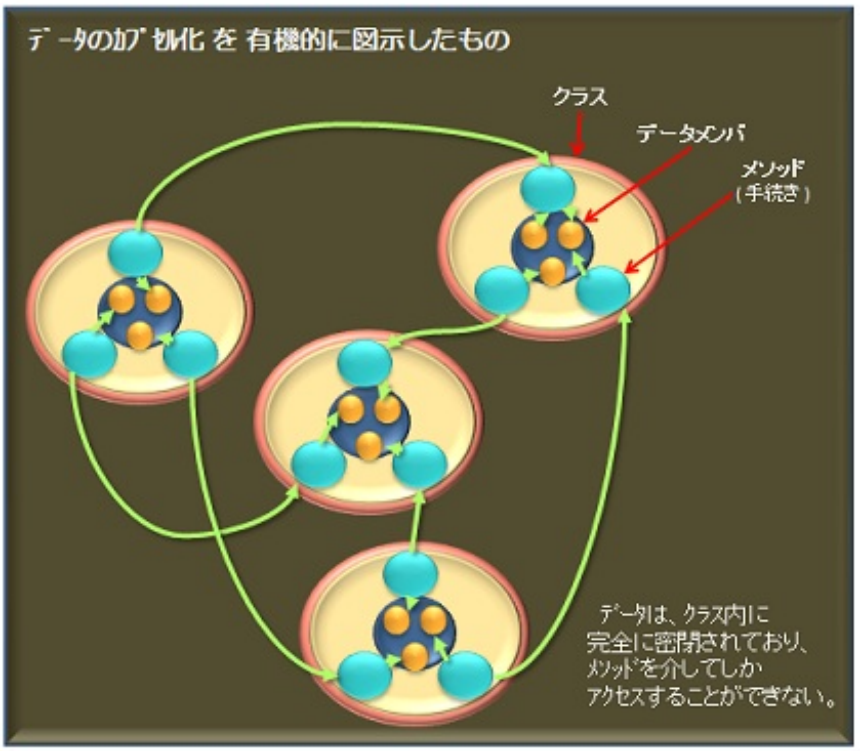
・このように、**C++**には、規則が多いんですが、
これは、**C**の有り余る自由度に、規制を加えることで、
生産性を高めるためのものです。

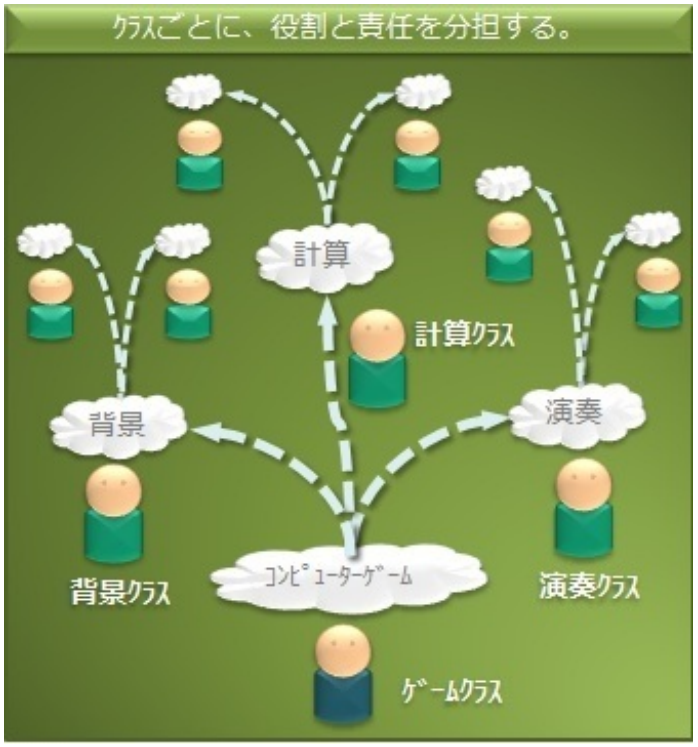
・詳細については、これからまた

順を追って、説明していきますが、
最初はわからないと思いますので、
とりあえず、ざっと読んでみて、
あとは、実際に、使いながら覚えて下さい。

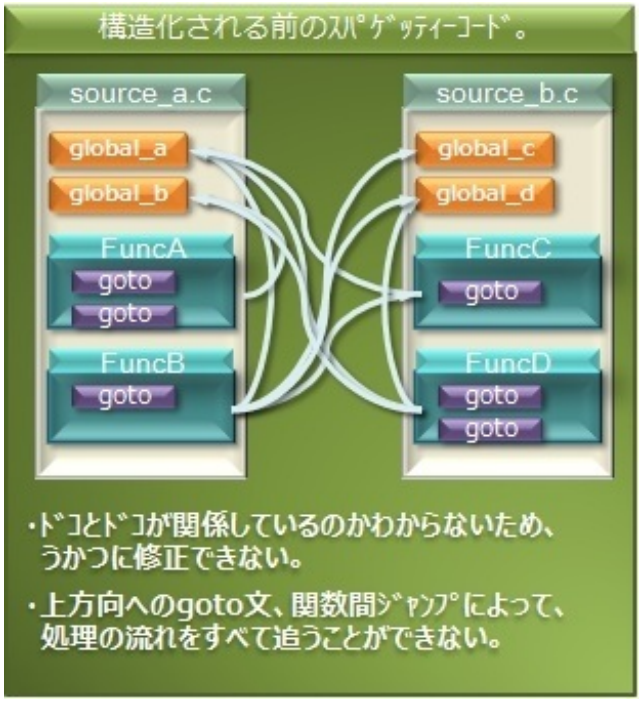
「ｵﾌﾞｼﾞｪｸﾄ指向」

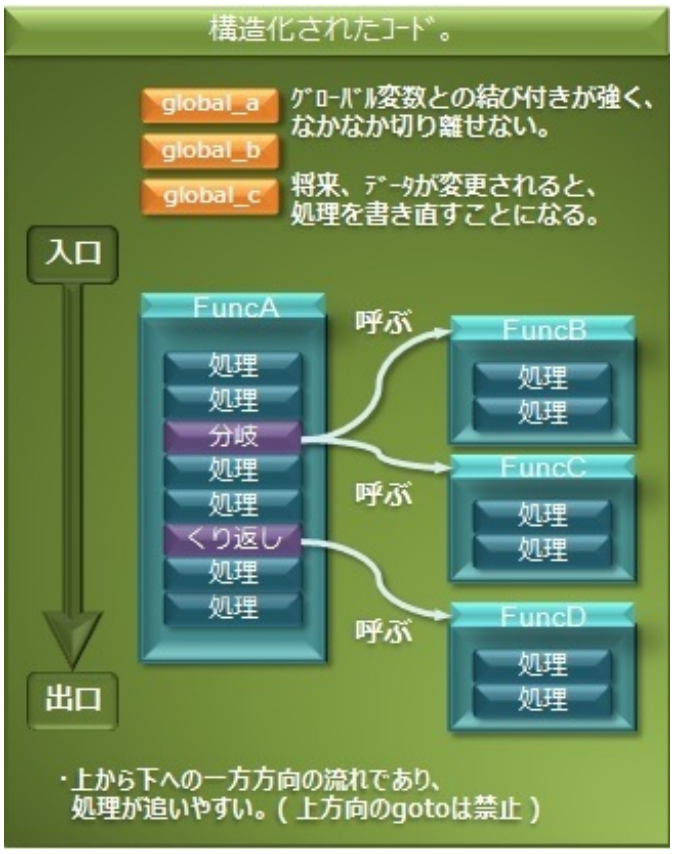

ﾃﾞｰﾀのｶﾌﾟｾﾙ化
ｸﾗｽ型 A
非公開
value
公開
SetValue
GetValue
呼び出す
ｸﾗｽ型 B
Init
A Instance1;
Instance1.SetValue(100);
Instance1.value = 100;
値を参照できない
・ｸﾗｽ型のﾃﾞｰﾀﾒﾝﾊﾞは、原則的に、private(非公開)ﾒﾝﾊﾞとして宣言される。
・このため、ｲﾝｽﾀﾝｽからは、参照することができない。
・代わりに、public(公開)ﾒﾝﾊﾞであるﾒｿｯﾄﾞを呼び出して、間接的に参照する。

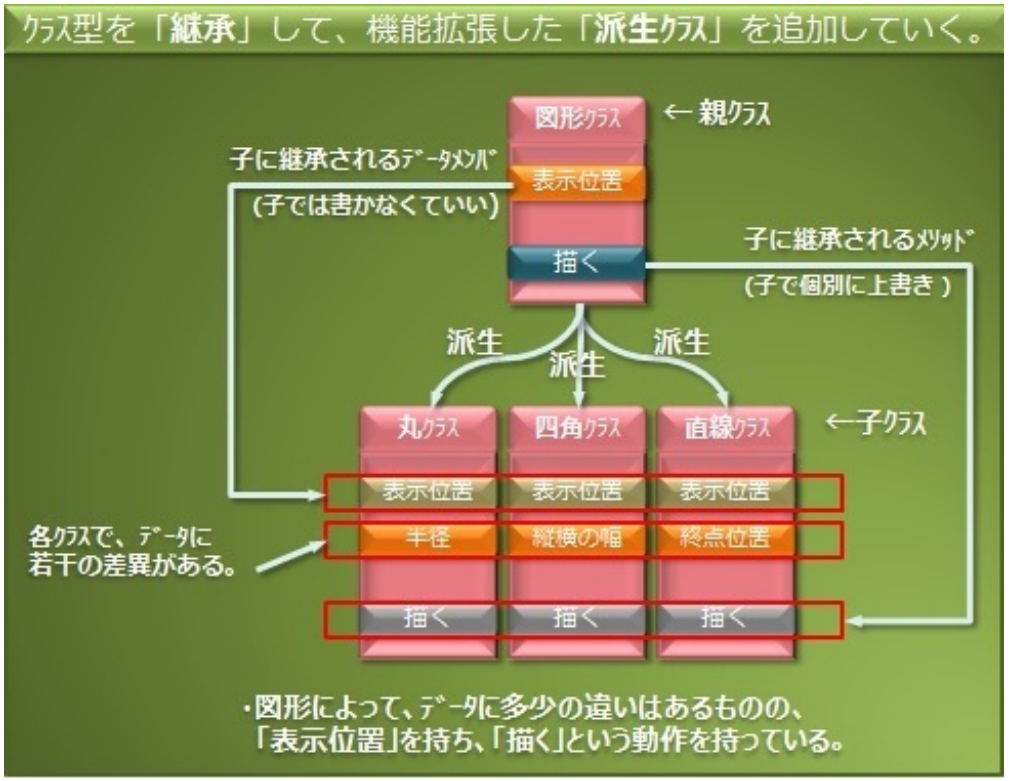

ﾃﾞｰﾀのｶﾌﾟｾﾙ化 を 有機的に図示したもの
クラス
データメンバ
メソッド
(手続き)
ﾃﾞｰﾀは、ｸﾗｽ内に
完全に密閉されており、
ﾒｿｯﾄﾞを介してしか
ｱｸｾｽすることができない。

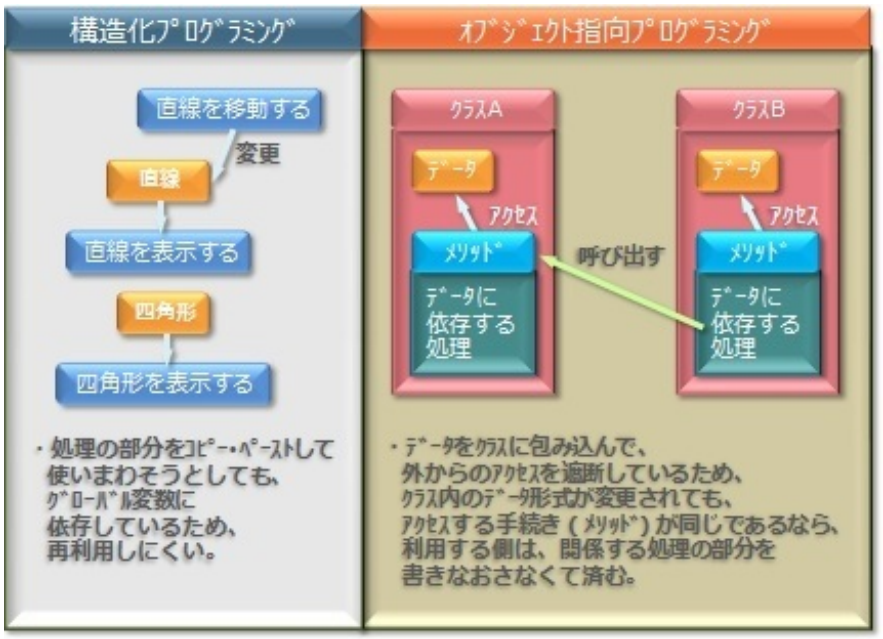

ｸﾗｽごとに、役割と責任を分担する。
計算
計算ｸﾗｽ
背景
演奏
背景ｸﾗｽ
ｺﾝﾋﾟｭｰﾀｰｹﾞｰﾑ
演奏ｸﾗｽ
ｹﾞｰﾑｸﾗｽ


引数が「注文」で、処理結果が「お料理」。
1番の曲を
演奏してくれ。
注文
1番の曲
PlayMusic ﾒｿｯﾄﾞ
よろ
こん
で一
(棒
秘
中はﾌﾞﾗｯｸﾎﾞｯｸｽ
←ｹﾞｰﾑｸﾗｽ
演奏ｸﾗｽ →
呼び出す
・PlayMusicﾒｿｯﾄﾞの処理内容は、将来、変更されるかもしれないが、
利用する際の手続き(ﾒｿｯﾄﾞ)が、いつも同じであれば、
客側は修正しなくていい。


構造化される前のｽﾊﾟｹﾞｯﾃｨｰｺｰﾄﾞ。
source_a.c
global_a
global_b
FuncA
goto
goto
FuncB
goto
source_b.c
global_c
global_d
FuncC
goto
FuncD
goto
goto
・ﾄﾞｺとﾄﾞｺが関係しているのかわからないため、
うかつに修正できない。
・上方向へのgoto文、関数間ｼﾞｬﾝﾌﾟによって、
処理の流れをすべて追うことができない。


構造化されたｺｰﾄﾞ。
global_a
global_b
global_c
ｸﾞﾛｰﾊﾞﾙ変数との結び付きが強く、
なかなか切り離せない。
将来、ﾃﾞｰﾀが変更されると、
処理を書き直すことになる。
入口
FuncA
処理
処理
分岐
処理
処理
くり返し
処理
処理
呼ぶ
呼ぶ
呼ぶ
FuncB
処理
処理
FuncC
処理
処理
FuncD
処理
処理
出口
・上から下への一方方向の流れであり、
処理が追いやすい。(上方向のgotoは禁止)





・「ｵﾌﾞｼﾞｪｸﾄ指向」というのは、
ﾌﾟﾛｸﾞﾗﾑ上で使うﾃﾞｰﾀの一つ一つを、
物質 (ｵﾌﾞｼﾞｪｸﾄ) として考えるという思想です。

・一般的な手法としては、まず、
ﾃﾞｰﾀの構成から書き始めます。

・そして実際に、
そのﾃﾞｰﾀを使って、どういう処理をさせよう

ということは、ﾃﾞｰﾀの構成を、
ある程度、まとめてから考えます。

・この世界は、いろんな物質で成り立っています。

・自動車とか、電車とか、いろいろな物質があって、
それらは、たくさんの部品で構成されています。

・しかし、その部品を、どんどん分解して、
ﾐｸﾛの世界へと降りて行くと、
分子だとか、素粒子だとか、
部品の種類はだんだん少なくなり、
単純な仕組みになっていきます。

・ﾌﾟﾛｸﾞﾗﾑで使うﾃﾞｰﾀも同じことで、
分解していくと、単純にすることができ、
再利用されやすくなります。

・大きなｼｽﾃﾑを開発したり、管理していると、
あとで、ﾃﾞｰﾀの構成を変更することが多くなります。

・そして、そうなった時には、
そのﾃﾞｰﾀに関係する処理を
すべて洗い出して修正しなければならず、
大変な手間となります。

・また、すべての修正箇所を洗い出せなかった場合は、
整合性がとれなくなり、ﾃﾞｰﾀが壊れてしまいます。

・「ｵﾌﾞｼﾞｪｸﾄ指向」では、こうしたﾑﾀﾞを、少しでも省くために、
ﾃﾞｰﾀと処理を、なるべく切り離します。

・まず、ﾃﾞｰﾀに関係する処理を洗い出し、
ﾃﾞｰﾀとひとまとめにして、**1**つの物質とします。

・たとえば、効果音を鳴らす処理を書く場合は、
「**Sound**」とかいうｸﾗｽを作ります。↓

```
class Sound
{
  WaveData* p_data; // wave形式の音声ﾃﾞｰﾀ。
  bool Download( int number_ ); // 指定番号の曲を、ﾀﾞｳﾝﾛｰﾄﾞする。
  bool Play( int number_, int volume_ ); // 指定番号の音を鳴らず。
};
```

・そして、このｸﾗｽには、
　効果音を再生する「**Play**」とかいうﾒｿｯﾄﾞを作ります。

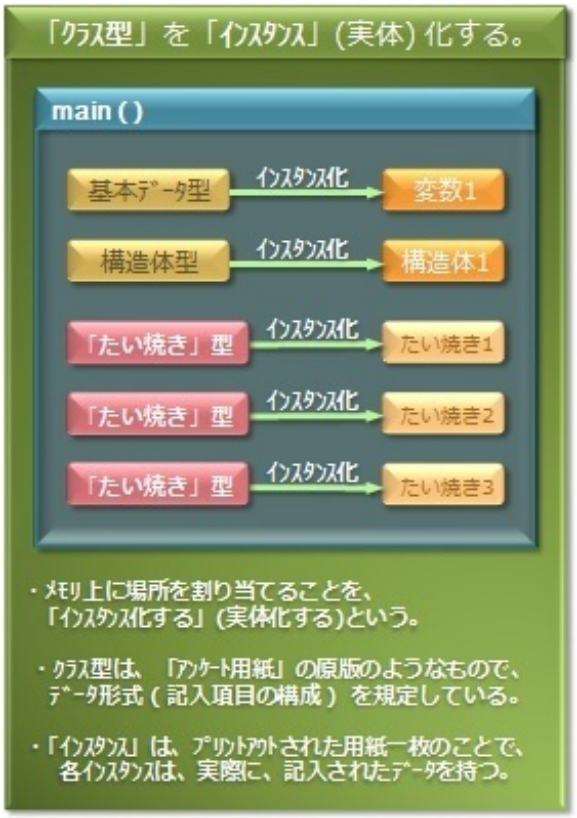
・これを利用する側のﾒｿｯﾄﾞでは、
　この「**Play**」ﾒｿｯﾄﾞに、
　音の番号や、音量を渡せば、
　効果音が鳴るようにします。↓

```
{
   sound1.Play( 1, 12 );  // 1番の音を鳴らす。
}
```

・こうしておけば、数年後に、効果音のﾃﾞｰﾀ形式が、
　変更されたとしても、「**Sound**」ｸﾗｽの中だけを修正すればよく、
　利用側のﾒｿｯﾄﾞでは、同じように使うことができます。

・「ｵﾌﾞｼﾞｪｸﾄ指向」の詳細については、後述の
　「ｸﾗｽの継承」、「仮想関数」あたりで説明します。

「ｸﾗｽの宣言」と「ｲﾝｽﾀﾝｽ化」



・まず、「ｸﾗｽの宣言」と「ｲﾝｽﾀﾝｽ化」について、
　単純な例を挙げて説明します。↓

```
// Taiyaki.h ( ﾍｯﾀﾞﾌｧｲﾙ )

class Taiyaki  // ｸﾗｽ型「Taiyaki」の宣言。
{

private: // 以降のﾒﾝﾊﾞを非公開とするｱｸｾｽ指定子。

    int anko; // ﾃﾞｰﾀﾒﾝﾊﾞ「anko」は、あんこの分量を、ｸﾞﾗﾑ単位で保持する。

public: // 以降のﾒﾝﾊﾞを公開とするｱｸｾｽ指定子。

    Taiyaki( );  // ｺﾝｽﾄﾗｸﾀ
    ~Taiyaki( ); // ﾃﾞｽﾄﾗｸﾀ
```

```
    void SetAnko( const int anko_ ); // Setterﾒｿｯﾄﾞ
    int GetAnko( );                  // Getterﾒｿｯﾄﾞ
};
```

```
// Taiyaki.cpp ( ｿｰｽﾌｧｲﾙ )

   Taiyaki::Taiyaki( ) // ｺﾝｽﾄﾗｸﾀ
   {
   }

   Taiyaki::~Taiyaki( ) // ﾃﾞｽﾄﾗｸﾀ
   {
   }

   void  Taiyaki::SetAnko(const int anko_ ) // Setterﾒｿｯﾄﾞ
   {
        anko = anko_;
   }

   int Taiyaki::GetAnko( ) // Getterﾒｿｯﾄﾞ
   {
      return anko;
   }
```

・ｸﾗｽ型を定義することを、
「ｸﾗｽを宣言する」といいます。

・構造体型の定義とよく似ていますが、
ｸﾗｽ型は、「ﾒｿｯﾄﾞ」といわれる関数を
ﾒﾝﾊﾞ に持っています。

(実をいえば、構造体型でもﾒｿｯﾄﾞを持てる。)

・次に、このｸﾗｽ型を使って、変数を宣言してみます。↓

```
// main.cpp

#include "Taiyaki.h"

 int main( )
 {
    Taiyaki* p_taiyaki1 = new Taiyaki( ); // ｸﾗｽのｲﾝｽﾀﾝｽ化。 (実体化)

    p_taitaki1->SetAnko( 100 );        // Setterﾒｿｯﾄﾞを呼び出し、ﾃﾞｰﾀﾒﾝﾊﾞに値を設定する。
    int anko = p_taiyaki1->GetAnko( ); // Getterﾒｿｯﾄﾞを呼び出し、ﾃﾞｰﾀﾒﾝﾊﾞの値を取得する。

    delete p_taiyaki1; // ｲﾝｽﾀﾝｽを解放する。
 }
```

・ｸﾗｽ型の変数のことを、
「ｲﾝｽﾀﾝｽ」といいます。

・また、ｸﾗｽ型とひっくるめて
「ｵﾌﾞｼﾞｪｸﾄ」と呼ぶこともあります。

・そして、それを宣言することを、
「ｲﾝｽﾀﾝｽ化」とか、「実体化」といいます。

・ﾃﾞｰﾀ型というのは、設計書のようなもので、
設計をした時点では、「実体」がありません。

・実際に、ﾌﾟﾛｸﾞﾗﾑが実行された時に、
ﾒﾓﾘ上に、ｲﾝｽﾀﾝｽの領域が割り当てられた時に
はじめて、「実体を持った」といえるのです。

・ｲﾝｽﾀﾝｽの宣言は、変数の宣言と同じように、
「Taiyaki taiyaki1」とすることもできますが、
(ﾛｰｶﾙ変数と同じで、ｽﾀｯｸ領域に確保される)
一般的には、「new」演算子を使って、
「ﾋｰﾌﾟﾒﾓﾘ」上に確保します。

・これは、**C**言語でよく使っていた
「**malloc**」関数に相当するもので、
最後の「**delete**」演算子は、
「**free**」関数に相当します。

・ﾒﾝﾊﾞへのｱｸｾｽは、構造体型の時と同じで、
変数の時は、「**.** 」(ﾋﾟﾘｵﾄﾞ)演算子を、
ﾎﾟｲﾝﾀの時は、「**->**」(ｱﾛｰ)演算子を使います。

・「ｺﾝｽﾄﾗｸﾀ」は、
各ｲﾝｽﾀﾝｽが持っているﾃﾞｰﾀﾒﾝﾊﾞの
初期化を行うﾒｿｯﾄﾞで、
ｲﾝｽﾀﾝｽ化の時に呼ばれます。(newしたとき)

・これは、「**Taiyaki taiyaki1**」というように
ﾛｰｶﾙ変数のように宣言した場合でも

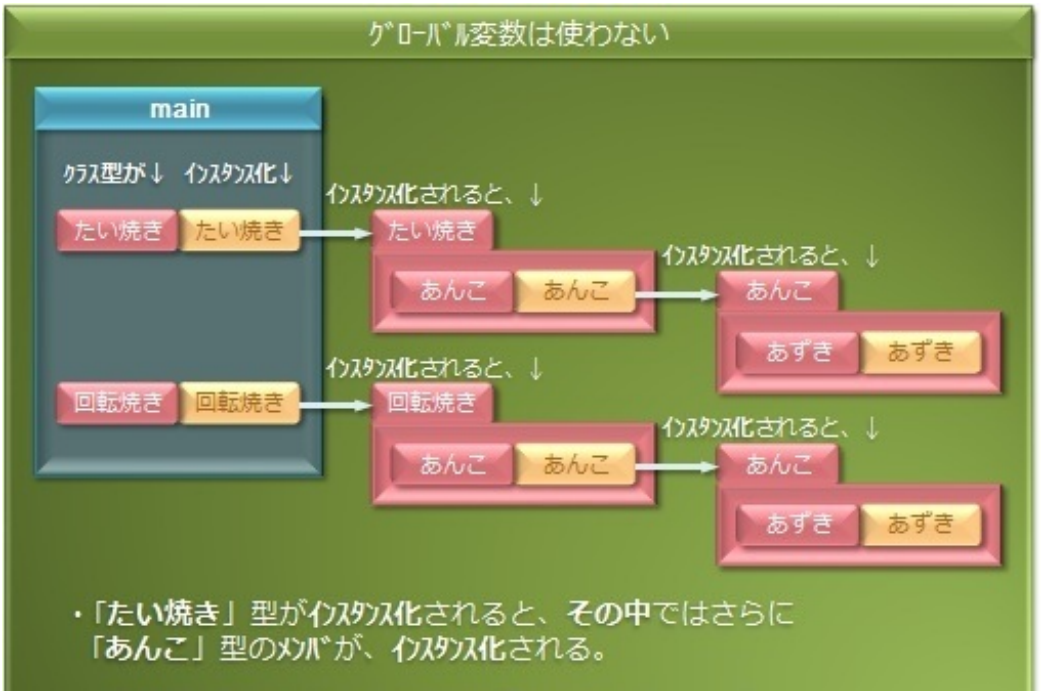
自動的に呼ばれます。

・「ｲﾝｽﾀﾝｽ」のﾒﾝﾊﾞとして
内部に「ｲﾝｽﾀﾝｽ」を持っている場合は、
この「ｺﾝｽﾄﾗｸﾀ」の中で、ｲﾝｽﾀﾝｽ化を行います。

・ﾃﾞｰﾀﾒﾝﾊﾞ 「**anko**」は、**int**型の数値ですが、
これが「**Anko**」というｸﾗｽ型の「ｲﾝｽﾀﾝｽ」なら、
上図のように、このﾀｲﾐﾝｸﾞでｲﾝｽﾀﾝｽ化します。

・「ﾃﾞｽﾄﾗｸﾀ」は、
使用したﾃﾞｰﾀの後始末を行うﾒｿｯﾄﾞで、
「ｲﾝｽﾀﾝｽ」が解放された時に呼ばれます。（deleteしたとき）

・「ｲﾝｽﾀﾝｽ」をﾒﾝﾊﾞとして持っている場合は、
この「ﾃﾞｽﾄﾗｸﾀ」中で、「**delete**」演算子を使って
「ｲﾝｽﾀﾝｽ」を解放します。



・それから、この「**delete**」演算子ですが、
配列の場合は、書き方が変わります。↓

```
Taiyaki* p_taiyakies = new Taiyaki[ 2 ]; // ｲﾝｽﾀﾝｽ配列を、ﾋｰﾌﾟに確保する。

delete []  p_taiyakies; // ｲﾝｽﾀﾝｽ配列を解放する。
```

・この配列の、各要素は、ﾎﾟｲﾝﾀではなく、ｲﾝｽﾀﾝｽです。

・このように、配列として、まとめてｲﾝｽﾀﾝｽ化する場合は、
ｺﾝｽﾄﾗｸﾀやﾃﾞｽﾄﾗｸﾀには、引数を渡せませんので、
引数の付いていないﾃﾞﾌｫﾙﾄｺﾝｽﾄﾗｸﾀと、
ﾃﾞﾌｫﾙﾄﾃﾞｽﾄﾗｸﾀが呼び出されます。

・ﾃﾞﾌｫﾙﾄのｺﾝｽﾄﾗｸﾀやﾃﾞｽﾄﾗｸﾀは、
自分で書くこともできますが、
書かなくてもよく、その場合は、
何もしないｺﾝｽﾄﾗｸﾀ、ﾃﾞｽﾄﾗｸﾀが
自動的に生成されます。

・ただし、引数付きのｺﾝｽﾄﾗｸﾀを
自分で書いた場合は、これが行われません。

・そして、この実装 (中ｶｯｺ内の処理部分) が
定義されていないﾒｿｯﾄﾞを呼び出すようにﾌﾟﾛｸﾞﾗﾑを書いていると、
「未解決の外部ｼﾝﾎﾞﾙ」とかいうｺﾝﾊﾟｲﾙｴﾗｰが出ます。

・引数付きのｺﾝｽﾄﾗｸﾀを使いたい時は、
ｲﾝｽﾀﾝｽへのﾎﾟｲﾝﾀの配列を作ります。↓

```
Taiyaki** pp_taiyakies = new Taiyaki*[ 2 ]; // ﾎﾟｲﾝﾀ配列を宣言する。

*pp_taiyakies = new Taiyaki( 100 ); // ｲﾝｽﾀﾝｽ化して、ﾎﾟｲﾝﾀを要素0に、代入する。
delete (pp_taiyakies[0]); // 要素0のｲﾝｽﾀﾝｽを解放する。

delete []  pp_taiyakies; // ﾎﾟｲﾝﾀ配列を解放する。
```

・「**malloc**」関数や「**free**」関数を使った場合は、
「ｺﾝｽﾄﾗｸﾀ」や「ﾃﾞｽﾄﾗｸﾀ」は、呼ばれません。

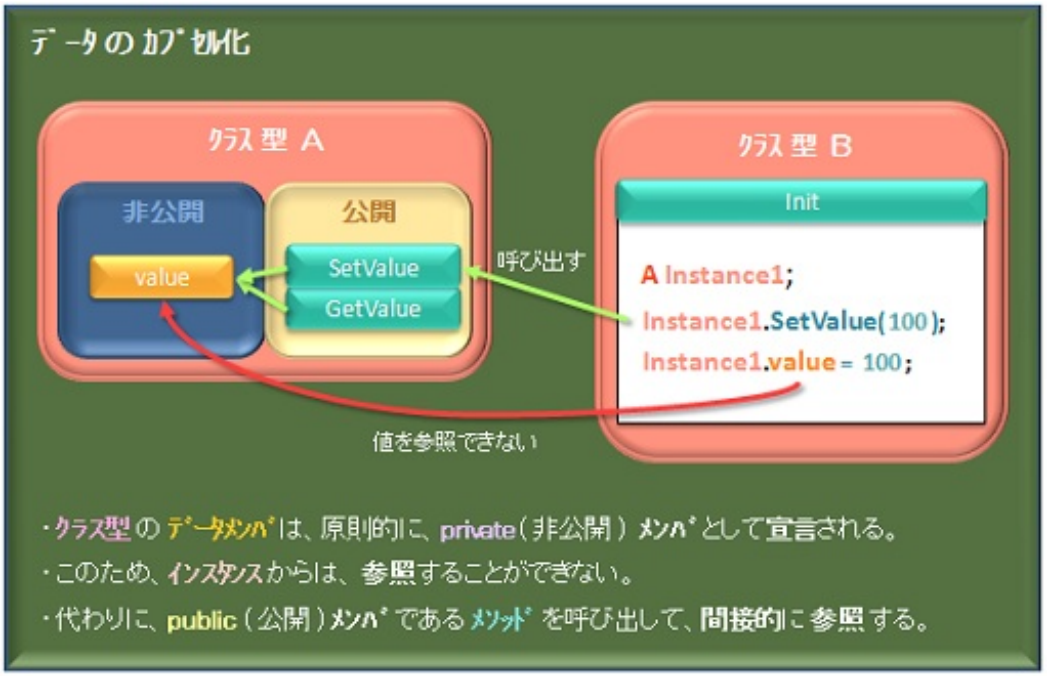
・ｸﾗｽ型であれば、ﾛｰｶﾙ変数として宣言した場合でも
「ｺﾝｽﾄﾗｸﾀ」や「ﾃﾞｽﾄﾗｸﾀ」が呼ばれます。

・しかし、構造体型の場合は、「**new**」や
「**delete**」をしても、呼ばれません。

・それと、**void**ﾎﾟｲﾝﾀに代入すると、
ﾃﾞｰﾀ型の情報が失われてしまい、
deleteしても、ﾃﾞｽﾄﾗｸﾀが呼ばれません。

・ﾃﾞｽﾄﾗｸﾀを呼ぶには、元のﾃﾞｰﾀ型にｷｬｽﾄしてから
呼ぶようにして下さい。

ｱｸｾｽ指定子 (public: private:)



・構造体型のﾒﾝﾊﾞは、
ｲﾝｽﾀﾝｽがあれば、
参照することができました。↓

```
Kouzou kouzou1;
kouzou1.member1 = 100;
int a = kouzou1.member2;
```

・一方、ｸﾗｽ型のﾒﾝﾊﾞは、
公開されているものと、
非公開のものとがありました。

・それを決めているのが、「ｱｸｾｽ指定子」です。

・この「ｱｸｾｽ指定子」というのは、
次のｱｸｾｽ指定子までに宣言されたﾒﾝﾊﾞを、
指定した公開ﾚﾍﾞﾙにしてくれます。↓

```
// A ｸﾗｽ の定義
```

```
class A
{
 private:    // 次のｱｸｾｽ指定子までのﾒﾝﾊﾞは、非公開ﾒﾝﾊﾞになる。
    int private_value;

 public:     // 次のｱｸｾｽ指定子までのﾒﾝﾊﾞは、公開ﾒﾝﾊﾞになる。
    int public_value;

 int GetPrivateValue( )
 {
      return private_value;
 }
};

// ------------------------
//  B ｸﾗｽ の定義

class B
{
   Hanasu( )
   {
       A a1;
      cout << "「Aの秘密は、" << a.GetPrivateValue( ) << "です。」" << endl;
   }
};
```

初期化子ﾘｽﾄ

・ｸﾗｽ型のｲﾝｽﾀﾝｽを、ﾒﾝﾊﾞ として持つ場合は、
ｸﾗｽ型がｲﾝｽﾀﾝｽ化された時に、
ﾒﾝﾊﾞ もｲﾝｽﾀﾝｽ化されます。 ↓

```
class HamBerger
{
    CheeseBerger cheese_berger;
};
```

・こういう場合に、
ﾒﾝﾊﾞ の、引数付きのｺﾝｽﾄﾗｸﾀを呼ぶには、
次のように書きます。 ↓

```
HamBerger::HamBerger( ) : cheese_berger( 78 )
{
    // newを使わない場合の初期化と同じ。 ↓
    // CheeseBerger cheese_berger( 78 );
}
```

・これを、「初期化子ﾘｽﾄ」 といいます。

・「初期化子ﾘｽﾄ」を使えば、
「const」によって、定数として宣言されているﾒﾝﾊﾞ を
初期化することができます。

・また、複数のﾒﾝﾊﾞ を初期化する場合は、
「, 」(ｶﾝﾏ) で区切って、続けて書くこともできます。 ↓

```
    // ・基本ﾃﾞｰﾀ型のﾒﾝﾊﾞでも初期化できる。
    // ・ﾍｯﾀﾞﾌｧｲﾙ側の、ﾌﾟﾛﾄﾀｲﾌﾟ宣言に書いてはいけない。
HamBerger::HamBerger( ) : patty( 20 ) , pickles( 5 ) , buns( 10 )
{

}
```

・また、すべてのﾃﾞｰﾀﾒﾝﾊﾞが 「public」 の場合は、
  配列のように初期化することもできます。 ↓

```
class Meishi
{
public:
  char namae[50];
  char jusyo[100];
};

void main( )
{
   Meishi meishi1 = { "ハム太", "かいわれ村" };
}
```

静的ﾒﾝﾊﾞ（ static ）

・**C++**では、ｸﾞﾛｰﾊﾞﾙな変数や関数は
原則、使わないことになっているんですが、
その代わりに、ｸﾗｽごとに持たせていたりします。

・ﾒﾝﾊﾞの宣言時に、「**static**」を付けると、
「静的ﾒﾝﾊﾞ」となります。

・この「静的ﾒﾝﾊﾞ」は、
特定のｲﾝｽﾀﾝｽには属しておらず、
最初にｱｸｾｽされた時点で、
ｸﾗｽ型ごとに確保されます。

・したがって、ｱｸｾｽする時も、
ｲﾝｽﾀﾝｽからではなく、ｸﾗｽ型の名前を使います。↓

```
// Color.h

  class Color // 「色」ｸﾗｽの宣言。
  {

 protected:

    // 通常のﾃﾞｰﾀﾒﾝﾊﾞ。（ｲﾝｽﾀﾝｽごとに持つ）
     unsigned char r; // Red値   （光の三原色）
     unsigned char g; // Green値（光の三原色）
     unsigned char b; // Blue値   （光の三原色）

  public:

     // 「静的ﾃﾞｰﾀﾒﾝﾊﾞ」   （黒色はよく使うので、これを使いまわす。）
     static  const Color Black;
```

```
    static  Color* p_default_color; // ﾃﾞﾌｫﾙﾄｶﾗｰへのﾎﾟｲﾝﾀ。

    // 「静的ﾒｿｯﾄﾞ」のﾌﾟﾛﾄﾀｲﾌﾟ宣言。 （「光の三原色」から、ｲﾝｽﾀﾝｽを作る。）
    static Color* FromRGB( unsigned char r_, unsigned char g_, unsigned char b_ );

    // 「ｺﾝｽﾄﾗｸﾀ」のﾌﾟﾛﾄﾀｲﾌﾟ宣言。
    Color(  unsigned char r_, unsigned char g_, unsigned char b_ );
};
```

// **Color.cpp**

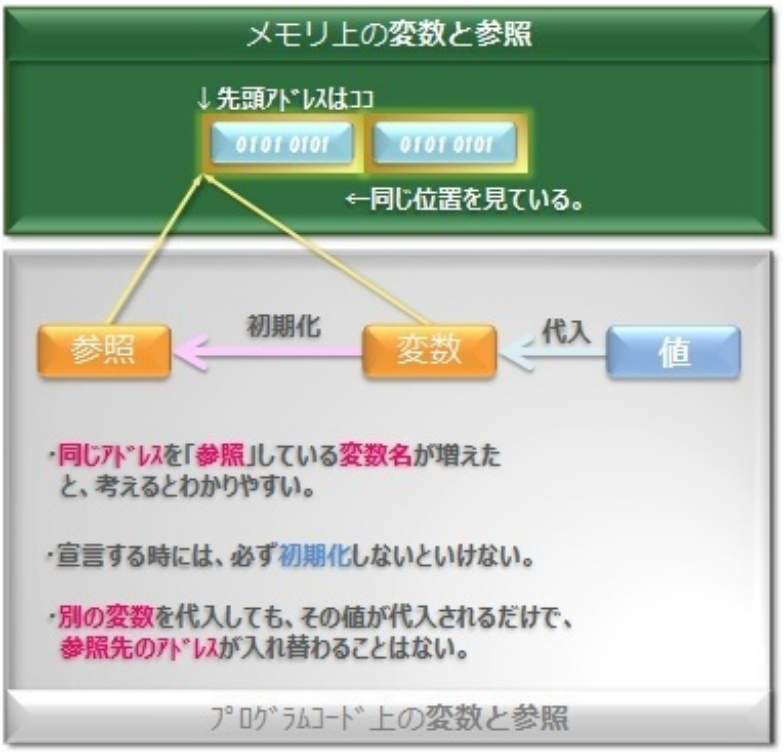
```
#include "Color.h"

    // 「静的ﾃﾞｰﾀﾒﾝﾊﾞ」を宣言し、初期化する。（※「static」は付けない。）
    const Color Color::Black(0, 0, 0);

     p_default_color = new Color(0, 0, 0); // ﾃﾞﾌｫﾙﾄｶﾗｰをｲﾝｽﾀﾝｽ化する。

    // 「静的ﾒｿｯﾄﾞ」の定義。（※「static」は付けない。）
    Color* Color::FromRGB( unsigned char r_, unsigned char g_, unsigned char b_ )
    {
            return new Color( r_, g_, b_ );
    }

    // 「ｺﾝｽﾄﾗｸﾀ」の定義。
    Color::Color( unsigned char r_, unsigned char g_, unsigned char b_ )
    {
        r = r_;
        g = g_;
        b = b_;
    }
```

// **main.cpp**

```
#include "Color.h"
```

```
void main( )
{
    Color* p_red = new Color( 255, 0, 0 ); // ｸﾗｽ型をｲﾝｽﾀﾝｽ化する。

    Color* p_yellow = Color::FromRGB( 255, 255, 0 ); // 「静的ﾒｿｯﾄﾞ」を呼び出す。

    Color black = Color::Black; // 「静的ﾃﾞｰﾀﾒﾝﾊﾞ」を代入する。

    Color* p_default_color = *Color::p_default; // ﾃﾞﾌｫﾙﾄｶﾗｰへのﾎﾟｲﾝﾀを取り出す。

}
```

「参照」 (別名)



・「参照」は、
「ﾎﾟｲﾝﾀ」とよく似ています。↓

```
int hensu1 = 100; // 変数を宣言し、「100」で初期化する。
```

```
int& sansyo1 = hensu1; // 「参照」に、変数を代入する。
// int* p_hensu1 = &hensu1; と同じ。

int hensu2 = sansyo1;   // 別の変数に、「参照」を代入する。
// int hensu2 = *p_hensu1; と同じ。    ( どちらも、「100」が代入される。 )

sansyo1= 200; // 「参照」に、「200」を代入する。
// *p_hensu1 = 200; と同じ。    ( どちらも、参照元の変数の値が、「200」になる。 )
```

・書き方が、少し簡単になっているだけではなく、
　挙動も少し違います。

・例えば、次のように書くと、
　ｺﾝﾊﾟｲﾙｴﾗｰになります。 ↓

```
int& sansyo1; // 「参照」を宣言する。 ( ※初期化していないのでﾀﾞﾒ )
```

・つまるところ、「参照」というのは、
　「変数の別名」ですから、
　参照する相手が決まってないと、
　宣言できないわけです。

・ですから、宣言する時には、
　必ず、同じ型の変数を代入しておく、
　すなわち初期化しておく必要があるのです。

・参照は、**NULLで初期化することができない**ため、
　NULLであるかをﾁｪｯｸする必要がありません。

・ﾎﾟｲﾝﾀでは、
　別の変数のｱﾄﾞﾚｽを代入すると、
　参照する変数を変更することができました。

・しかし、「参照」は違っていて、
　初期化した時に代入した変数を
　参照し続けます。↓

```
int hensu3 = 500;     // さらに、別の変数を宣言する。
sansyo1 = hensu3;    // 「参照」に代入してみる。(ｱﾄﾞﾚｽではなく、値が代入される。)

// ・つまり、ｱﾄﾞﾚｽが代入されるのは、宣言の時だけです。
// ・値を代入していることを見てもわかるように、変数の分身としてふるまいます。
```

・「参照」がよく使われているのは、
　「引数」と「戻り値」です。

・戻り値は、ﾎﾟｲﾝﾀの時と同じで、
　ﾛｰｶﾙ変数を返してはいけません。↓

```
int& Func( int& param_ )
{
    param_ = 100; // 引数に代入してみる。
    // 「param_」は、「a」の「参照」なので、
    // 「a」の値も、「100」になる。

    int local = 200; // ﾛｰｶﾙ変数を宣言する。

    return local; // ﾛｰｶﾙ変数の「参照」を返す。
}
```

```
int main( )
{
   int a = 0;
```

```
    int& b = Func( a );
    // 「b」は、「local」の「参照」だが、
    // 「local」が、すでに解放されているため、値がおかしい。

    return 0;
}
```

・要するに、「参照」で返してもいいのは、
解放されていないものだけです。

(**static**のﾃﾞｰﾀﾒﾝﾊﾞ、ｸﾞﾛｰﾊﾞﾙ変数、
ﾋｰﾌﾟﾒﾓﾘ上に確保されたｲﾝｽﾀﾝｽのﾃﾞｰﾀﾒﾝﾊﾞ など。
これは、ﾎﾟｲﾝﾀを返す場合と同じ。)

・参照は、入力引数でよく使います。 (**const** int& **src_** など)

・理由はいくつかあって、
constﾎﾟｲﾝﾀ引数よりも、やや速いということ、
また、渡される値も、定数(ﾘﾃﾗﾙ) か、
初期化された変数への参照であることが
保証されていて安全であるためです。

・それから、上記の例のように、
「参照」同士の代入では、
ｺﾋﾟｰｺﾝｽﾄﾗｸﾀは呼ばれません。

・ｺﾋﾟｰｺﾝｽﾄﾗｸﾀの引数が「参照」なのも、
この性質を利用したものです。

・まとめると、「参照」というのは、
『変数の別名』であり、
『参照先を変更できないﾎﾟｲﾝﾀ』 (正体は、実行時ｱﾄﾞﾚｽ)
と見ることもできます。

配列の参照

・これは使い道がないんですが、いちおう説明しておきます。

```
int a1[2] = {0,1};  // 配列を宣言する。
int (&r1)[2] = a1; // その参照を宣言する。
int v1 = r1[1];      // v1 には、1 が代入される。
```

・引数や、戻り値にも使えます。 ↓

```
int (&Func1(      int (&r1_)[2]      ))[2]
{
    return r1_;
}

void main()
{
  int a1[2] = {0,1};  // 配列を宣言する。
  int (&r1)[2] = Func1( a1 ); // その参照を宣言する。
  int v1 = r1[1];      // v1 には、1 が代入される。
}
```

・いくつか留意点があるんですが、
  まず、こうした参照渡しでは、
  配列の要素数を明記する必要があります。

・ちなみに、要素数を省略した場合は、ポインタ渡しとみなされます。↓

```
int* Func2( int p1_[] )
{
    return p1_;
}

void main()
{
  int a1[2] = {0,1};  // 配列を宣言する。
  int* p1 = Func2( a1 ); // その参照を宣言する。
  int v1 = p1[1];      // v1 には、1 が代入される。
}
```

・また、ポインタで渡した場合は、ポインタ演算ができますが、
参照渡しでは、これができません。

・それと、参照渡しでは、ポインタの配列が渡せません。

・以上のことから、参照渡しは不要といえます。

演算子の「ｵｰﾊﾞｰﾛｰﾄﾞ」

・構造体型のｲﾝｽﾀﾝｽは、
同じ構造体型のｲﾝｽﾀﾝｽに
代入することができました。

・ｸﾗｽ型のｲﾝｽﾀﾝｽでも、
これと同じことができます。 ↓

```
Hamuta hamuta1; // ｲﾝｽﾀﾝｽ1
Hamuta hamuta2; // ｲﾝｽﾀﾝｽ2

hamuta2 = hamuta1; // 同じｸﾗｽ型に代入をすると、すべてｺﾋﾟｰされる。

// ・構造体と同じで、すべてのﾃﾞｰﾀﾒﾝﾊﾞの値がｺﾋﾟｰされてしまう。
```

```
// ・ﾎﾟｲﾝﾀや参照で初期化した場合は、ｺﾋﾟｰされない。

Hamuta* p_hamuta1 = new Hamuta(); // ｲﾝｽﾀﾝｽ化して、ｱﾄﾞﾚｽをﾎﾟｲﾝﾀに代入する。
p_hamuta2 = p_hamuta1;   // ﾎﾟｲﾝﾀからﾎﾟｲﾝﾀへ代入する。
```

・ﾃﾞｰﾀﾒﾝﾊﾞを、たくさん持っているｲﾝｽﾀﾝｽの場合は、
すべてｺﾋﾟｰしていたのでは、時間がかかり過ぎます。

・また、そのﾃﾞｰﾀﾒﾝﾊﾞの中に、
ﾎﾟｲﾝﾀ型のものが含まれていると、
ｱﾄﾞﾚｽがｺﾋﾟｰされてしまいます。

・これをﾃﾞｽﾄﾗｸﾀでdeleteしていた場合は、
両方のﾃﾞｽﾄﾗｸﾀで、同じｲﾝｽﾀﾝｽが
2重にdeleteされてしまいます。

・代入時の処理を変更するには、
「代入演算子」を、「ｵｰﾊﾞｰﾛｰﾄﾞ」します。↓

```
class Hamta
{

protected:

   int value; // ﾃﾞｰﾀﾒﾝﾊﾞ。

public:

   // ﾃﾞｰﾀﾒﾝﾊﾞを返すﾒｿｯﾄﾞ。
   int GetValue()
   {
      return value;
   }

   // 「代入演算子」を、ｵｰﾊﾟｰﾛｰﾄﾞする。
   Hamta& operator=( Hamta& other_ )
  {
      // 単純に、ﾃﾞｰﾀﾒﾝﾊﾞの値をｺﾋﾟｰする
      // という、既定の動作を、再現してみる。↓
      value = other_.GetValue();
   }

};
```

ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ (と explicit)

・前回は、
代入時の処理を変更するために、
「代入演算子」を「ｵｰﾊﾞｰﾛｰﾄﾞ」しました。

・今回は、初期化をした場合に呼ばれる
「ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ」を、書き直してみます。

・下記のように、同じｸﾗｽ型のｲﾝｽﾀﾝｽを代入して
初期化をすると、「ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ」が呼ばれます。 ↓

```
Hamuta hamuta1;
Hamuta hamuta2 = hamuta1; // 同じｸﾗｽ型で初期化すると、ｺﾋﾟｰｺﾝｽﾄﾗｸﾀが呼ばれる。
```

```
// ・ﾎﾟｲﾝﾀや参照で初期化した場合は、
// ｲﾝｽﾀﾝｽ化はされていないので、呼ばれない。

Hamuta* p_hamuta1 = new Hamuta();
Hamuta* p_hamuta2 = p_hamuta1;
```

```
hamuta2 = hamuta1; // 初期化ではないので、代入演算子の既定の実装か、その上書きが呼ばれる。
```

・「ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ」を、自前で再定義するには、
次のように書きます。 ↓

```
// ・いちおう、ｺﾝｽﾄﾗｸﾀなので、ｸﾗｽと同じ名前にして、戻り値は書かない。
// ・引数は、同じｸﾗｽ型のconst参照にする。

class Hamuta
```

```
{

protected:

    Obj* p_obj; // ﾎﾟｲﾝﾀﾒﾝﾊﾞ

public:

  // ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ
  Hamuta( const Hamuta& hamuta_ ) // 引数は、同じｸﾗｽ型のconst参照。
  {
      if ( this !=&hamuta_ ) // 自身へのｱﾄﾞﾚｽでなければ、
                    delete p_obj; // 自身がﾒﾝﾊﾞとして持つﾎﾟｲﾝﾀを解放する。

      p_obj = hamuta_->GetObj( ); // 渡されたｲﾝｽﾀﾝｽがﾒﾝﾊﾞとして持つﾎﾟｲﾝﾀを受け取る。
  }

  // 自身がﾒﾝﾊﾞとして持つﾎﾟｲﾝﾀを返すﾒｿｯﾄﾞ。
  Obj* GetObj()
  {
     return p_obj;
  }

};
```

・それから、「ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ」とよく似たものに、
「引数が1つ付いたｺﾝｽﾄﾗｸﾀ」というのがあります。 ↓

```
class Hamuta
{
protected:
   char* p_message;
public:
     Hamuta( char* p_message_ ) : p_message(p_message_) { }
```

```
};
```

・これもやはり、初期化をした時に、呼ばれるんですが、↓

```
// 別の型の値で初期化すると、対応するｺﾝｽﾄﾗｸﾀが呼ばれる。

   Hamuta hamuta1 = "やっほっほー"; /* 同じｸﾗｽ型以外で初期化している。 */
```

・この書き方を禁止する方法というのがあるんですが、↓

```
  explicit Hamuta( char* p_message_ ) : p_message( p_message_ ) { }
```

・この「**explicit**」指定子は、
ｲﾝｽﾀﾝｽが初期化された時に、
自動的に行われる「暗黙的な型ｷｬｽﾄ」を禁止するもので、
引数が**1**つだけ付いたｺﾝｽﾄﾗｸﾀでしか使いません。

・初期化した場合は、ｺﾝﾊﾟｲﾙ時にｴﾗｰが出ます。

・これを「自動変換拒否」といいます。

・「**explicit**」指定子は、ｺﾋﾟｰｺﾝｽﾄﾗｸﾀでは必須ともいえるのですが、
その他の箇所では、あまり使うことは無いと思います。^^;

・ちなみに、この、初期化時の自動変換というのは、
たとえば、int型の引数を持つｺﾝｽﾄﾗｸﾀに、short型の値を渡すと、
自動的に型ｷｬｽﾄが行われて、ｺﾝｽﾄﾗｸﾀが呼ばれるというもので、
int型の値しか渡せなくしたい場合に使用します。

```
// ﾊﾞｲﾄ配列を作成して解放するｸﾗｽ「CMaker」を定義してみる。↓

class CMaker
{

public:
    BYTE*  p_buf; // ﾊﾞｲﾄ配列へのﾎﾟｲﾝﾀ。
    size_t  length; // 配列の要素数。

    CMaker() : p_buf( NULL ) , length(0) {}; // ｺﾝｽﾄﾗｸﾀ

    CMaker( size_t length_ ) // 引数付きのｺﾝｽﾄﾗｸﾀ
    {
      length = length_;
      p_buf = new BYTE[ length ]; // 配列を作成する。
      ::memset( p_buf, 0, length ); // ｾﾞﾛ埋めする。
    };

     explicit CMaker( const CMaker& src_ ) // ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ
     { *this = src_; } // 代入演算子が呼ばれ、ﾃﾞｰﾀﾒﾝﾊﾞの値がｺﾋﾟｰされる。

     void operator=(  const CMaker& src_ ) // 代入演算子
     {
       length = length_;
       p_buf = new BYTE[ length ]; // 配列を作成する。
       ::memcpy_s( p_buf, length, src_.p_buf, length ); // 要素値をｺﾋﾟｰする。
     };

      virtual ~CMaker() // ﾃﾞｽﾄﾗｸﾀ
      { if ( p_buf != NULL ) delete [] p_buf; }; // 配列を解放する。
```

```
#include <stdio.h>

int main(void)
```

```
{
  CMaker* p_maker1 = new CMaker( 10 ); // ｲﾝｽﾀﾝｽ1を作成する。

  int i;
  for ( i = 0; i < 10; i++ )  p_maker1->p_buf[ i ] = i; // 0~9 を格納する。

  CMaker* p_maker2 = new CMaker( *p_maker1 ); // ｲﾝｽﾀﾝｽ2を作成する。

  for ( i = ; i < 10; i++ ) // 要素値を表示する。
      ::printf( "[%d] = %d\n", p_maker2->p_buf[ i ] ); // 0~9 が表示される。

  delete p_maker1; // ｲﾝｽﾀﾝｽ1を解放する。
  delete p_maker2; // ｲﾝｽﾀﾝｽ2を解放する。

  return 0;
}
```

・ﾃﾞｽﾄﾗｸﾀは、virtual (仮想関数)として宣言した方がいいでしょう。

・ｲﾝｽﾀﾝｽのｻｲｽﾞが、ﾎﾟｲﾝﾀ1つ分だけ増えますが、
子ｸﾗｽ側でﾃﾞｽﾄﾗｸﾀを上書きしておけば、
親ｸﾗｽのﾎﾟｲﾝﾀでｱﾄﾞﾚｽを格納していても、deleteした時には、
上書きした子ｸﾗｽのﾃﾞｽﾄﾗｸﾀだけが呼ばれます。

・ｺﾝｽﾄﾗｸﾀについては、仮想関数にできませんので、
上書きする場合は、初期化処理は、ｲﾝｽﾀﾝｽ化したあとに別のﾒｿｯﾄﾞで
行うようにした方がいいと思います。

・ｺﾋﾟｰｺﾝｽﾄﾗｸﾀと、代入演算子も、明示的に定義した方がいいでしょう。

・定義されていない場合は、ｺﾝﾊﾟｲﾗによって自動的に生成されますが、
上記の例のように、ﾋｰﾌﾟﾒﾓﾘ上の配列へのﾎﾟｲﾝﾀを、
ﾃﾞｰﾀﾒﾝﾊﾞとして持たせている場合は、ﾎﾟｲﾝﾀだけがｺﾋﾟｰされるだけで、
そのﾎﾟｲﾝﾀが指しているﾋｰﾌﾟﾒﾓﾘ上の配列の要素値がｺﾋﾟｰされません。

・この場合、同じｱﾄﾞﾚｽを指すﾎﾟｲﾝﾀが2つできるわけで、
　どちらかのｲﾝｽﾀﾝｽで解放されると、使えなくなります。

いろいろな演算子の「ｵｰﾊﾞｰﾛｰﾄﾞ」

・今回は、前々回のつづきで、
　その他の演算子も
　「ｵｰﾊﾞｰﾛｰﾄﾞ」してみます。↓

```
class Hamta
{

protected:

   int value; // ﾃﾞｰﾀﾒﾝﾊﾞ。

public:

  // ｺﾝｽﾄﾗｸﾀ ( 引数付き )
  Hamta( int value_ ) :value( value_ )
  {
  }

   // ﾃﾞｰﾀﾒﾝﾊﾞを返すﾒｿｯﾄﾞ。
   int GetValue()
   {
       return value;
   }

   // 「=」演算子を、ｵｰﾊﾟｰﾛｰﾄﾞする。 ( 戻り値 = ( 自身 = その他 ) )
   Hamta& operator=( Hamta& other_ )
  {
       value = other_.GetValue();

       return *this;
  }

   // 「+=」を、ｵｰﾊﾟｰﾛｰﾄﾞする。 ( 戻り値 = ( 自身 += その他 ) )
```

```
Hamta& operator+=( Hamta& other_ )
{
    value += other_.GetValue();

    return *this;
}

// 「+」演算子を、ｵｰﾊﾞｰﾛｰﾄﾞする。（戻り値＝(自身＋その他)）
Hamta operator+( Hamta& other_ )
{
    return Hamta( value + other_.GetValue() );

    // ・戻り値の型が「参照」だと、ﾃﾞｽﾄﾗｸﾀが呼ばれる。

    // ・ﾎﾟｲﾝﾀをﾃﾞｰﾀﾒﾝﾊﾞとして持っている場合は、
    //　ここで解放されてしまうため、注意が必要。

}

// 前置の「++」演算子を、ｵｰﾊﾞｰﾛｰﾄﾞする。（戻り値＝(++自身)）
Hamta& operator++()
{
    ++value;

    return *this;
}

// 後置の「++」演算子を、ｵｰﾊﾞｰﾛｰﾄﾞする。（戻り値＝(自身++)）
Hamta& operator++( int )// int型の無名引数を付けると、「後置」とみなされる。
{
    ++value;

    return *this;
}

// 「==」演算子を、ｵｰﾊﾞｰﾛｰﾄﾞする。（戻り値＝(自身 == その他)）
bool operator==( Hamta& other_ )
```

```
  {
      return ( value == other_.GetValue() );
  }

};
```

・「[]」演算子の場合は、
ﾃﾞｰﾀﾒﾝﾊﾞが、ｲﾝｽﾀﾝｽ配列なら、
そのままの感じで使えます。↓

```
class ArrayItem // 配列項目ｸﾗｽの宣言。
{

protected:

    int value; // ﾃﾞｰﾀﾒﾝﾊﾞ。

public:

    // ﾃﾞｰﾀﾒﾝﾊﾞを返すﾒｿｯﾄﾞ。
    int GetValue()
    {
        return value;
    }

    // 「=」演算子を、ｵｰﾊﾞｰﾛｰﾄﾞする。
    ArrayItem& operator=( ArrayItem& other_ )
  {
        value = other_.GetValue();

        return *this;
  }

};
```

```
class Array // 配列ｸﾗｽの宣言。
{

protected:

   ArrayItem* p_items; // ﾃﾞｰﾀﾒﾝﾊﾞ。

public:

  // ｺﾝｽﾄﾗｸﾀ ( 引数付き )
  Array( const size_t size_ )
  {
      p_items = newArrayItem[ size_ ];
  }

  // ﾃﾞｽﾄﾗｸﾀ
  ~Array()
  {
      if ( p_items != NULL )
        delete [] p_items;
  }

   // 「[]」演算子を、ｵｰﾊﾞｰﾛｰﾄﾞする。 ( 戻り値 = 自身[ index_ ] )
   ArrayItem& operator [](  const int index_ )
  {
       return p_items[ index_ ];
  }

};
```

・基本ﾃﾞｰﾀ型の配列だと、
「 int value = array1[i]; 」 はできますが、
「array1[i] = value; 」 はできません。

・それから、「new」や「delete」ですが、
「static」 か、ｸﾞﾛｰﾊﾞﾙ空間 (ｸﾗｽ外) でなら、
「ｵｰﾊﾞｰﾛｰﾄﾞ」できます。 ↓

```
// 「new」演算子を、ｵｰﾊﾟｰﾛｰﾄﾞする。 ( Hamta* p1 = (Hamta*) new( sizeof(Hamta) ) Hamta; )
void* operator new( const size_t size_ )
{
    return MyAllocator::New( size_ );
}
```

```
// 「delete」演算子を、ｵｰﾊﾟｰﾛｰﾄﾞする。 ( operator delete( p1 ); )
void  operator delete( void* p_ )
{
    MyAllocator::Delete( p_ );
}
```

・「new」演算子の引数2以降は、
自由に決めることができます。

・つづいて、配列版。 ↓

```
// 「new」演算子を、ｵｰﾊﾟｰﾛｰﾄﾞする。 ( Hamta* p1 = (Hamta*) new( sizeof(Hamta) ) Hamta[10];
)
void* operator new[]( const size_t size_ )
{
    return MyAllocator::New( size_ );
}
```

```
// 「delete」演算子を、ｵｰﾊﾟｰﾛｰﾄﾞする。 ( operator delete[]( p1 ); )
void  operator delete[]( void* p_ )
{
```

```
    MyAllocator::Delete( p_ );
}
```

型ｷｬｽﾄの「ｵｰﾊﾞｰﾛｰﾄﾞ」

```
// ｸﾗｽ型の宣言

class Int32
{
int value; // ﾃﾞｰﾀﾒﾝﾊﾞ

public:

    // ｺﾝｽﾄﾗｸﾀ ( 初期化用 )
Int32( int value_ ) : value( value_ ) { }

    // 型ｷｬｽﾄの「ｵｰﾊﾞｰﾛｰﾄﾞ」
operator double( )
    {
        return(double) value;
    }
};
```

```
int  main( )
{
Int32 value( 10 ); // ｲﾝｽﾀﾝｽ化する。

double real = value; // ｵｰﾊﾞｰﾛｰﾄﾞされた型ｷｬｽﾄが呼ばれる。

return 0;
}
```

ｸﾗｽの「継承」

・ｸﾗｽ型を「継承」すると、
元になるｸﾗｽ型が持っているすべてのﾒﾝﾊﾞが、
継承先のｸﾗｽに引き継がれます。

（「**Java**」などでは、「ｸﾗｽの拡張」と呼ぶこともあります。）

・元になるｸﾗｽのことを 「親ｸﾗｽ」 といい、
継承先のｸﾗｽのことを 「子ｸﾗｽ」 といいます。

・正式な呼び方としては、
元になるｸﾗｽのことを 「基底ｸﾗｽ」（base class）で、
継承先のｸﾗｽのことを 「派生ｸﾗｽ」（inherited class）といいます。

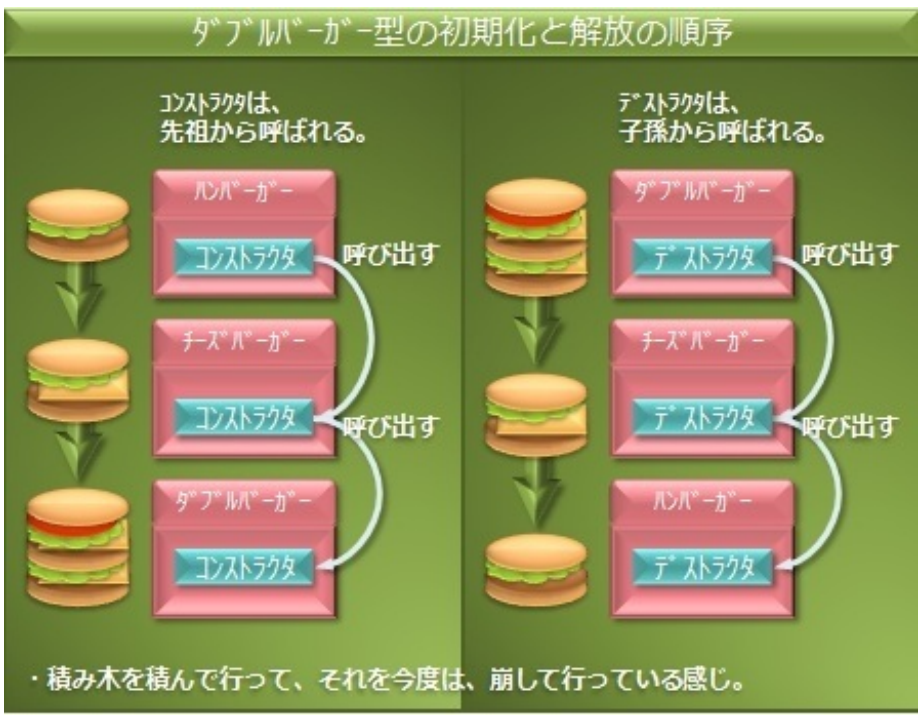
```
// ----------------------------
// HamBerger.h

#include <iostream>
using namespace std;

class HamBerger // ﾊﾝﾊﾞｰｶﾞｰｸﾗｽ。
```

```
{

protected: // 以降のﾒﾝﾊﾞは、子ｸﾗｽからのみｱｸｾｽできる。

    int patty;    // ﾊﾝﾊﾞｰｸﾞ
    int pickles;  // ﾋﾟｸﾙｽ
    int buns;     // ﾊﾟﾝ
    int ketchup; // ｹﾁｬｯﾌﾟｿｰｽ
    int mustard; // ﾏｽﾀｰﾄﾞｿｰｽ

public: // 以降のﾒﾝﾊﾞは、公開されている。

    HamBerger( ); // ｺﾝｽﾄﾗｸﾀ。
    ~HamBerger( ); // ﾃﾞｽﾄﾗｸﾀ。

    int GetCost( ); // 原価の合計を返すﾒｿｯﾄﾞ。
};
// ---------------
class CheeseBerger : public HamBerger // ﾁｰｽﾞﾊﾞｰｶﾞｰｸﾗｽ。
{

protected: // 以降のﾒﾝﾊﾞは、子ｸﾗｽからのみｱｸｾｽできる。

   // 書いてはいないが、親ｸﾗｽのﾃﾞｰﾀﾒﾝﾊﾞを継承している。

    int cheese;   // ﾁｰｽﾞ    ( 追加されたﾃﾞｰﾀﾒﾝﾊﾞ )

public: // 以降のﾒﾝﾊﾞは、公開されている。

    CheeseBerger( ); // ｺﾝｽﾄﾗｸﾀ。
    ~CheeseBerger( ); // ﾃﾞｽﾄﾗｸﾀ。

    int GetCost( ); // 原価の合計を返すﾒｿｯﾄﾞ。 ( 親ｸﾗｽのﾒｿｯﾄﾞを上書き )
};

// ----------------------------
// HamBerger.cpp
```

```
#include "HamBerger.h"

HamBerger::HamBerger( ) // ｺﾝｽﾄﾗｸﾀ。
{
       cout << "ﾊﾝﾊﾞｰｶﾞｰのｺﾝｽﾄﾗｸﾀが呼ばれました。" << endl;

      patty     = 20; // ﾊﾝﾊﾞｰｸﾞ
      pickles   = 5;  // ﾋﾟｸﾙｽ
      buns      = 10; // ﾊﾟﾝ
      ketchup   = 2;  // ｹﾁｬｯﾌﾟｿｰｽ
      mustard = 2;  // ﾏｽﾀｰﾄﾞｿｰｽ
}
// ---------------
CheeseBerger::CheeseBerger( ) // ｺﾝｽﾄﾗｸﾀ。
{
    // 親ｸﾗｽのｺﾝｽﾄﾗｸﾀが、 先に呼ばれている。

      cout<<"ﾁｰｽﾞﾊﾞｰｶﾞｰのｺﾝｽﾄﾗｸﾀが呼ばれました。" << endl;

     cheese = 30; // ﾁｰｽﾞが追加された。
}
// ---------------
HamBerger::~HamBerger( ) // ﾃﾞｽﾄﾗｸﾀ。
{
   cout << "ﾊﾝﾊﾞｰｶﾞｰのﾃﾞｽﾄﾗｸﾀが呼ばれました。" << endl;
}
// ---------------
CheeseBerger::~CheeseBerger( ) // ﾃﾞｽﾄﾗｸﾀ。
{
   cout << "ﾁｰｽﾞﾊﾞｰｶﾞｰのﾃﾞｽﾄﾗｸﾀが呼ばれました。" << endl;
}

// ---------------
int HamBerger::GetCost( ) // 原価の合計を返すﾒｿｯﾄﾞ。
{
    cout << "ﾊﾝﾊﾞｰｶﾞｰのGetCostﾒｿｯﾄﾞが呼ばれました。" << endl;
```

```
    return ( patty + pickles + buns + ketchup + mustard );
}
// ---------------
int CheeseBerger::GetCost( ) // 原価の合計を返すﾒｿｯﾄﾞ。（親ｸﾗｽのﾒｿｯﾄﾞを上書き）
{
    cout << "ﾁｰｽﾞﾊﾞｰｶﾞｰのGetCostﾒｿｯﾄﾞが呼ばれました。" << endl;

    // ﾃﾞｰﾀﾒﾝﾊﾞに、「ﾁｰｽﾞ」が追加されたため、
    // 合算の処理も、変更されている。
    return ( HamBerger::GetCost() + cheese );

   // ↑ 親ｸﾗｽのｸﾗｽ名を指定して、ﾒｿｯﾄﾞを呼び出している。

   // 「 typedef HamBerger base; 」 としておくと、
   // 後でｸﾗｽ名が変更されても、書きなおさなくて済む。
}

// ----------------------------
// main.cpp

#include "HamBerger.h"

#include <iostream>
using namespace std;

int main( )
{

  HamBerger* p_ham = new HamBerger(); // ﾊﾝﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。
  CheeseBerger* p_cheese = new CheeseBerger(); // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。

  // 両方の値段を合算する。
  int total_cost = p_ham->GetCost() + p_cheese->GetCost();

  cout << "原価の合計は、" << total_cost << "円です。" << endl;
```

```
 delete p_ham;     // ﾊﾝﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを解放する。
 delete p_cheese; // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを解放する。

 return 0;
}

// ----------------------------
```



・さて、今回のｻﾝﾌﾟﾙｺｰﾄﾞは、
ﾊﾝﾊﾞｰｶﾞｰの製造ｺｽﾄを計算するｸﾗｽです。

・商品ごとにｸﾗｽ型が定義されていて、
計算する項目 (ﾃﾞｰﾀﾒﾝﾊﾞ) が 少し異なっていて、

そのため、計算を行う処理 (ﾒｿｯﾄﾞの中身) も、少し異なっています。

・しかし、3つの製品は、
同じような項目(ﾃﾞｰﾀﾒﾝﾊﾞ)で構成されており、
ｺｽﾄを計算をするという機能 (ﾒｿｯﾄﾞ) も、
すべての商品ｸﾗｽで共通しています。

・このように、少しずつ項目が増え、
機能拡張されていく場合には、
ｸﾗｽ型の「継承」が、効果を発揮します。

・それから、細かいところを少し説明しておくと、
ｸﾗｽ型は、ﾒｿｯﾄﾞ (関数) を持つことができて、
ｺﾝｽﾄﾗｸﾀやﾃﾞｽﾄﾗｸﾀがあるんですが、
構造体型にはありません。

・したがって、構造体型をｲﾝｽﾀﾝｽ化しても、
ｺﾝｽﾄﾗｸﾀは呼ばれません。

・ｸﾗｽ型のｲﾝｽﾀﾝｽであっても、
malloc関数を使って作った場合は、
ｺﾝｽﾄﾗｸﾀは呼ばれません。

・**free**関数で解放した場合も、
やはり、ﾃﾞｽﾄﾗｸﾀは呼ばれません。

・ｸﾗｽ型のｲﾝｽﾀﾝｽのｱﾄﾞﾚｽを、
「**void***」 (voidﾎﾟｲﾝﾀ型) に代入して
deleteした場合も、ﾃﾞｽﾄﾗｸﾀは呼ばれません。

・これはﾅｾﾞなのかというと、
voidﾎﾟｲﾝﾀに代入してしまうと、
どのﾃﾞｰﾀ型のｱﾄﾞﾚｽだったのかが、
わからなくなってしまうからです。

・あと、それから、親ｸﾗｽの定義のところで、

「class」の前に付いている「ｱｸｾｽ指定子」なんですが、
これは「public」 と書くのが普通です。

・しかしこれを、もし、「protected」にした場合は、
どうなるのかというと、
親ｸﾗｽの「public」ﾒﾝﾊﾞが、「protected」扱いになり、
親ｸﾗｽか、子ｸﾗｽのﾒｿｯﾄﾞでしか使えなくなります。

・これを、さらに「private」にした場合は、
親ｸﾗｽのすべてのﾒﾝﾊﾞが「private」扱いになり、
子ｸﾗｽのﾒｿｯﾄﾞからも使えなくなります。

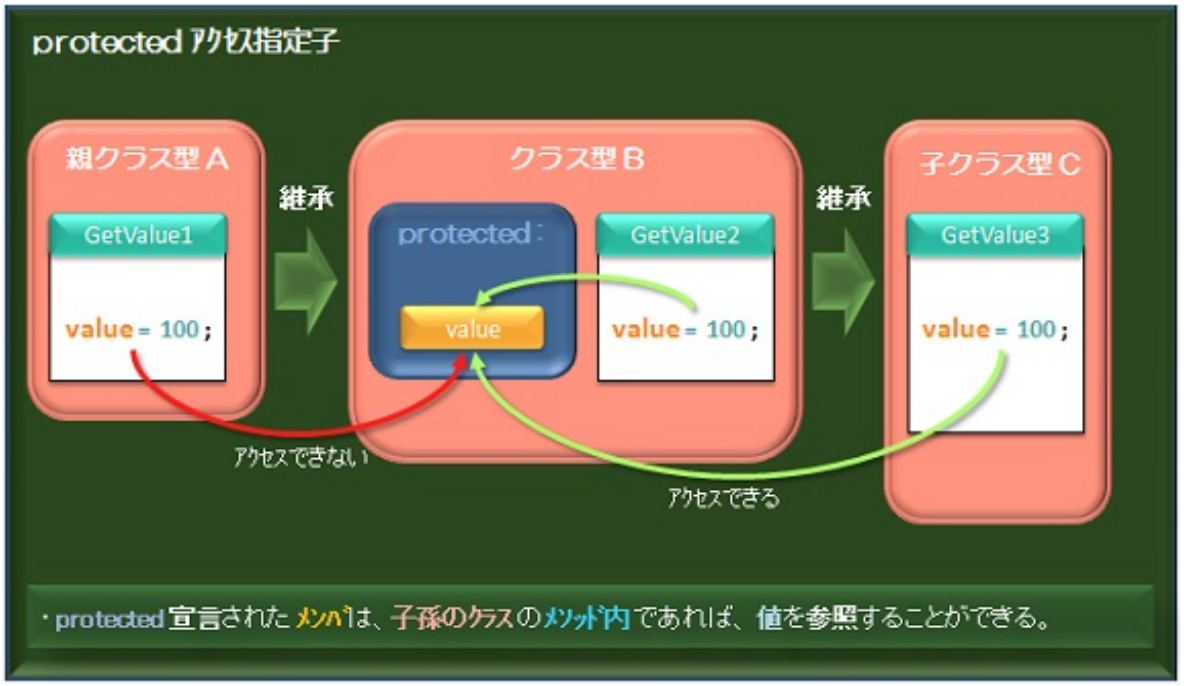
・それから、ﾀﾞﾌﾞﾙﾊﾞｰｶﾞｰ型は、こんな感じです。↓

```
// ----------------------------
// HamBerger2.h

#include <iostream>
using namespace std;

#include "HamBerger.h"

class DoubleBerger : public CheeseBerger // ﾀﾞﾌﾞﾙﾊﾞｰｶﾞｰｸﾗｽ。
{

protected: // 以降のﾒﾝﾊﾞは、子ｸﾗｽからのみｱｸｾｽできる。

    int momato; // ｽﾗｲｽしたﾄﾏﾄ

public: // 以降のﾒﾝﾊﾞは、公開されている。

    DoubleBerger( ); // ｺﾝｽﾄﾗｸﾀ。
    ~DoubleBerger( ); // ﾃﾞｽﾄﾗｸﾀ。

    int GetCost( ); // 原価の合計を返すﾒｿｯﾄﾞ。
```

```
};
// ---------------

// ----------------------------
// HamBerger2.cpp

#include "HamBerger2.h"

// ---------------
DoubleBerger::DoubleBerger( ) // ｺﾝｽﾄﾗｸﾀ。
{
    // 親ｸﾗｽのｺﾝｽﾄﾗｸﾀが、 先に呼ばれている。

cout << "ﾀﾞﾌﾞﾙﾊﾞｰｶﾞｰのｺﾝｽﾄﾗｸﾀが呼ばれました。" << endl;

       patty *= 2;  // ﾊﾝﾊﾞｰｸﾞが、2つに増えた。
      buns += buns / 2;  // ﾊﾟﾝが、1.5倍になった。

      cheese *= 2; // ﾁｰｽﾞが、2つに増えた。

       tomato = 30; // ﾄﾏﾄが追加された。
}
// ---------------
DoubleBerger::~DoubleBerger( ) // ﾃﾞｽﾄﾗｸﾀ。
{
cout << "ﾀﾞﾌﾞﾙﾊﾞｰｶﾞｰのﾃﾞｽﾄﾗｸﾀが呼ばれました。" << endl;
}
// ---------------
int DoubleBerger::GetCost( ) // 原価の合計を返すﾒｿｯﾄﾞ。 ( 親ｸﾗｽのﾒｿｯﾄﾞを上書き )
{
    cout << "ﾀﾞﾌﾞﾙﾊﾞｰｶﾞｰのGetCostﾒｿｯﾄﾞが呼ばれました。" << endl;

    // ﾃﾞｰﾀﾒﾝﾊﾞに、 「ﾄﾏﾄ」 が追加されたため、
    // 合算の処理も、変更されている。
    return ( CheeseBerger::GetCost() + tomato );
}
```

```
// ---------------------------
```

protected ｱｸｾｽ指定子



・**protected** ｱｸｾｽ指定子は、すこし特殊で、
これだけは「継承」が関係しています。

```
// 親ｸﾗｽ ( 最初の親ｸﾗｽのことを、「基底ｸﾗｽ」という。 )

class Oya
{
 private:    // 次のｱｸｾｽ指定子までのﾒﾝﾊﾞ は、このｸﾗｽの中からしか、ｱｸｾｽできない。
    int himitsu1;

 protected: // 次のｱｸｾｽ指定子までのﾒﾝﾊﾞ は、このｸﾗｽの子孫ｸﾗｽからしか、ｱｸｾｽできない。
    int himitsu2;

 public:     // 次のｱｸｾｽ指定子までのﾒﾝﾊﾞ は、このｸﾗｽの子孫ｸﾗｽからしか、ｱｸｾｽできない。
             // また、間接的にｱｸｾｽするのであれば、どこからでもｱｸｾｽできる。
    int teisai;

 Hanasu( )
 {
     cout << "親 「私の秘密は、" << himitsu1 << "です。」" << endl;
 }
```

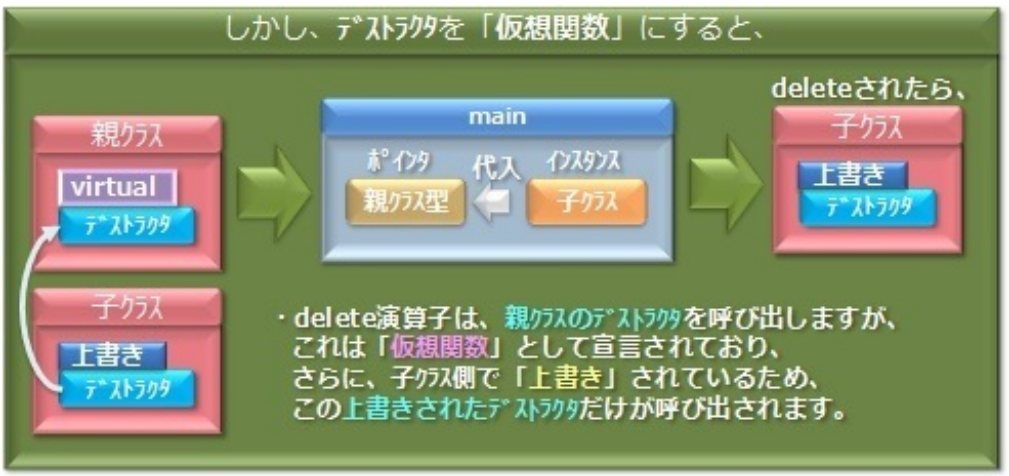
```
};

// ------------------------
// 子ｸﾗｽ

class Ko : public Oya
{
  Hanasu( )
  {
    cout << "子供 「親の秘密は、" << Oya::himitsu2 << "です。」" << endl;
  }
};
```

・対応表にしてみると、次のようになります。↓

ｱｸｾｽ指定子	親ｸﾗｽ型のﾒｿｯﾄﾞ内から	ｸﾗｽ型のﾒｿｯﾄﾞ内から	子ｸﾗｽ型のﾒｿｯﾄﾞ内から
private	直接ｱｸｾｽ不可	直接ｱｸｾｽ可	直接ｱｸｾｽ不可
protected	直接ｱｸｾｽ不可	直接ｱｸｾｽ可	直接ｱｸｾｽ可
public	直接ｱｸｾｽ不可	直接ｱｸｾｽ可	直接ｱｸｾｽ可

・**protected** ﾒﾝﾊﾞ は、
子孫のｸﾗｽ型の中で定義されたﾒｿｯﾄﾞの中でなら、
直接ｱｸｾｽすることができます。

「仮想関数」(virtual)



・子ｸﾗｽ側で、上書きしそうなﾒｿｯﾄﾞには、
「**virtual**」指定子を付け、
「仮想関数」(上書きされる関数)として宣言しておきます。

・こうしておけば、
ｲﾝｽﾀﾝｽへのｱﾄﾞﾚｽを
親ｸﾗｽ型のﾎﾟｲﾝﾀ変数に代入している場合でも、
子ｸﾗｽ側の上書き(ｵｰﾊﾞｰﾗｲﾄﾞ)したﾒｿｯﾄﾞが呼ばれます。↓

```
// ----------------------------
// HamBerger.h

#include <iostream>
using namespace std;

class HamBerger // ﾊﾝﾊﾞｰｶﾞｰｸﾗｽ。
{

protected: // 以降のﾒﾝﾊﾞは、子ｸﾗｽからのみｱｸｾｽできる。

    int patty;    // ﾊﾝﾊﾞｰｸﾞ
    int pickles;  // ﾋﾟｸﾙｽ
    int buns;     // ﾊﾟﾝ
    int ketchup; // ｹﾁｬｯﾌﾟｿｰｽ
    int mustard; // ﾏｽﾀｰﾄﾞｿｰｽ

public: // 以降のﾒﾝﾊﾞは、公開されている。

    HamBerger( ); // ｺﾝｽﾄﾗｸﾀ。
    virtual ~HamBerger( ); // ﾃﾞｽﾄﾗｸﾀ。

    virtual int GetCost( ); // 原価の合計を返すﾒｿｯﾄﾞ。
};

// ---------------

class CheeseBerger : public HamBerger // ﾁｰｽﾞﾊﾞｰｶﾞｰｸﾗｽ。
{

protected: // 以降のﾒﾝﾊﾞは、子ｸﾗｽからのみｱｸｾｽできる。

   // 書いてはいないが、親ｸﾗｽのﾃﾞｰﾀﾒﾝﾊﾞを継承している。

    int cheese;   // ﾁｰｽﾞ    ( 追加されたﾃﾞｰﾀﾒﾝﾊﾞ )
```

```
public: // 以降のﾒﾝﾊﾞは、公開されている。

        CheeseBerger( ); // ｺﾝｽﾄﾗｸﾀ。
        virtual ~CheeseBerger( ); // ﾃﾞｽﾄﾗｸﾀ。

        virtual int GetCost( ); // 原価の合計を返すﾒｿｯﾄﾞ。（親ｸﾗｽのﾒｿｯﾄﾞを上書き）
};

// ----------------------------
// HamBerger.cpp

#include "HamBerger.h"

HamBerger::HamBerger( ) // ｺﾝｽﾄﾗｸﾀ。
{
         cout << "ﾊﾝﾊﾞｰｶﾞｰのｺﾝｽﾄﾗｸﾀが呼ばれました。" << endl;

        patty    = 20; // ﾊﾝﾊﾞｰｸﾞ
       pickles  = 5;   // ﾋﾟｸﾙｽ
       buns     = 10; // ﾊﾟﾝ
       ketchup = 2;  // ｹﾁｬｯﾌﾟｿｰｽ
       mustard = 2;  // ﾏｽﾀｰﾄﾞｿｰｽ
}

// ---------------

CheeseBerger::CheeseBerger( ) // ｺﾝｽﾄﾗｸﾀ。
{
    // 親ｸﾗｽのｺﾝｽﾄﾗｸﾀが、先に呼ばれている。

       cout << "ﾁｰｽﾞﾊﾞｰｶﾞｰのｺﾝｽﾄﾗｸﾀが呼ばれました。" << endl;

      cheese = 30; // ﾁｰｽﾞが追加された。
}

// ---------------
```

```
HamBerger::~HamBerger( ) // ﾃﾞｽﾄﾗｸﾀ。
{
    cout << "ﾊﾝﾊﾞｰｶﾞｰのﾃﾞｽﾄﾗｸﾀが呼ばれました。" << endl;
}

// ---------------

CheeseBerger::~CheeseBerger( ) // ﾃﾞｽﾄﾗｸﾀ。
{
cout << "ﾁｰｽﾞﾊﾞｰｶﾞｰのﾃﾞｽﾄﾗｸﾀが呼ばれました。" << endl;
}

// ---------------

int HamBerger::GetCost( ) // 原価の合計を返すﾒｿｯﾄﾞ。
{
    cout << "ﾊﾝﾊﾞｰｶﾞｰのGetCostﾒｿｯﾄﾞが呼ばれました。" << endl;

    return ( patty + pickles + buns + ketchup + mustard );
}

// ---------------

int CheeseBerger::GetCost( ) // 原価の合計を返すﾒｿｯﾄﾞ。 ( 親ｸﾗｽのﾒｿｯﾄﾞを上書き )
{
    cout << "ﾁｰｽﾞﾊﾞｰｶﾞｰのGetCostﾒｿｯﾄﾞが呼ばれました。" << endl;

    // ﾃﾞｰﾀﾒﾝﾊﾞに、「ﾁｰｽﾞ」が追加されたため、
    // 合算の処理も、変更されている。
    return ( HamBerger::GetCost() + cheese );

   // ↑ 親ｸﾗｽのｸﾗｽ名を指定して、ﾒｿｯﾄﾞを呼び出している。

   // 「 typedef HamBerger base; 」 としておくと、
   // 後でｸﾗｽ名が変更されても、書きなおさなくて済む。
}
```

```
// ----------------------------
// main.cpp

#include "HamBerger.h"

#include <iostream>
using namespace std;

int main( )
{

  HamBerger* p_ham = new HamBerger(); // ﾊﾝﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。
  HamBerger* p_cheese = new CheeseBerger(); // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。

  // 両方の値段を合算する。
  int total_cost = p_ham->GetCost() + p_cheese->GetCost();

  cout << "原価の合計は、" << total_cost << "円です。" << endl;

  delete p_ham;     // ﾊﾝﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを解放する。
  delete p_cheese; // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを解放する。

  return 0;
}

// ----------------------------
```

・ﾃﾞｽﾄﾗｸﾀも、ちゃんと
　子ｸﾗｽ側のﾒｿｯﾄﾞが呼ばれています。

・上記の例のように、
　親ｸﾗｽ型のﾎﾟｲﾝﾀ変数から
　子ｸﾗｽ側のﾒｿｯﾄﾞを呼び出す場合は多いので、
　ﾃﾞｽﾄﾗｸﾀは、念のため、
　「**virtual**」にしておくことをｵｽｽﾒします。

・呼び出しｺｽﾄは、少し高めですが、
・「仮想関数」にしておけば、

後から、新商品が追加された時でも、
次のような、利用側のｺｰﾄﾞを書き直さなくて済みます。↓

```
// ----------------------------
// main.cpp

#include "HamBerger.h"

#include <iostream>
using namespace std;

int main( )
{

  HamBerger** pp_order =new HamBerger*[2]; // 親ｸﾗｽ型のﾎﾟｲﾝﾀ配列を作る。

  pp_order[0] = new HamBerger();      // ﾊﾝﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。
  pp_order[1] = new CheeseBerger(); // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。

  int total_cost = GetTotalCost( pp_order ); // 合計原価を返す関数を呼び出す。

  delete (pp_order[0]); // ﾊﾝﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを解放する。
  delete (pp_order[1]); // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを解放する。

  delete [] pp_order; // ﾎﾟｲﾝﾀ配列を解放する。

  return 0;
}

// ----------------------------
// 原価の合計を返す関数。（この処理は、修正しなくて済む。）

void GetTotalCost( HamBerger** pp_order_ ) // この引数だとかも、変更しなくていい。
{
  // 両方の値段を合算する。
  int total_cost = 0;
```

```
  for (unsigned int =0;i < 2; i++)
  {
      total_cost += pp_order_[ i ]->GetCost(); // ｺｽﾄを取り出し、加算する。
   }

  cout << "原価の合計は、" << total_cost << "円です。" << endl;

 return total_cost;

}

// ----------------------------
```

・それから、
子ｸﾗｽ側から、親ｸﾗｽ側のﾒｿｯﾄﾞを呼ぶ時は、
「HamBerger::GetCost();」というように、
ｸﾗｽ名を指定して書きます。

・「仮想関数」は、「抽象ｸﾗｽ」でよく使います。
その場合は、中ｶｯｺ内の処理を書きません。

・ｲﾝｽﾀﾝｽの場合は、
構造体変数とは少し違うみたいなんですが、
わかりやすく書けば、こんな感じでしょうかね。

・ﾒﾓﾘ上の、ﾃﾞｰﾀの並び方が同じだから、
親ｸﾗｽ型のﾎﾟｲﾝﾀ変数であっても、
子ｸﾗｽの上書きしたﾒｿｯﾄﾞに
ｱｸｾｽできるんですね。

・ﾌﾟﾛｸﾞﾗﾑが実行されている時は、
ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞ (機械語) も、

ﾒﾓﾘ上に配置されていて、
そのｱﾄﾞﾚｽが、親の「GetCost」と、
子で上書きされた「GetCost」とでは別の位置で、
ただ、そのｱﾄﾞﾚｽを配置している位置が一緒だから、
同じように呼び出せるんですね。

・それと、ｺﾝｽﾄﾗｸﾀの中では、仮想関数は使えません。

・これはﾅｾﾞかというと、
上記の関数ﾎﾟｲﾝﾀ(隠しﾒﾝﾊﾞ)が
まだ準備されていないからです。

this ﾎﾟｲﾝﾀ

・「this ﾎﾟｲﾝﾀ」は、自身へのﾎﾟｲﾝﾀです。

・通常、ｸﾗｽ内で定義されたﾒｿｯﾄﾞの中から、
ｸﾗｽ内で宣言されているﾒﾝﾊﾞに対してｱｸｾｽする場合は、
value1 = 100;　と書きますが、
これは、this->value1 = 100;
と書くこともできます。

```
// ------------------------------
// 「ﾊﾑ」ｸﾗｽ の宣言。

class Ham
{
      int ham;

      Ham( )
      {
        ham = 20;
     }

      ~Ham( )
      {
     }

     int GetCost( )
      {
            return ham;
      }
};

// ------------------------------
// 「ﾊﾑｶﾂ」ｸﾗｽ の宣言。
```

```
class HamKatsu
{
    Ham* p_ham;

    int koromo;

    HamKatsu( )
    {
        p_ham = new Ham();
       koromo = 5;
     }

    ~HamKatsu( )
    {
         delete p_ham;
     }

    int GetCost( )
    {
       // ・「this」ﾎﾟｲﾝﾀには、このｲﾝｽﾀﾝｽへの
       //  ｱﾄﾞﾚｽが代入されている。
       //
       //  ( ふつうは、わざわざ書かないが、
       //    他ｸﾗｽの持つ同じ名前のﾒﾝﾊﾞと、区別したい場合に使う。 )
       //
       return ( p_ham->GetCost( ) + this->GetCost( ) );

      // この場合でも、thisは省略できるが、
      // ぱっと見で、分かりやすいため、あえて書いている。
    }
};

// -------------------------------
```

型ｷｬｽﾄ (ｸﾗｽ型)

```
class Oya { }; // 親ｸﾗｽの宣言。
class Ko : public Oya { }; // 子ｸﾗｽの宣言。
```

```
void main( )
{
   // (親ｸﾗｽ) ← (子ｸﾗｽ)    ( 自動的にｷｬｽﾄされる。 )
   Oya* p_oya = new Ko( );

   // (子ｸﾗｽ) ← (親ｸﾗｽ)    ( 危険なﾀﾞｳﾝｷｬｽﾄ。 )
   Ko* p_ko = (Ko*) new Oya( );

   // ↑ 子ｸﾗｽのﾒﾝﾊﾞを呼ぼうとすると、無いので、ｺｹる。
}
```

```
  // ｷｬｽﾄできない時は、NULLが代入される。
  Ko* p_ko1=  dynamic_cast<Ko*>( new Oya( ));

  if ( p_ko1 == NULL) // NULLが代入されたら、
  {
    cout << "dynamic_castの結果: ｷｬｽﾄ失敗" << endl; // NULLなので、表示される。
  }
```

```
  // どんなﾃﾞｰﾀ型でもｷｬｽﾄする。     ( 別系統のﾎﾟｲﾝﾀ型同士や、整数型とﾎﾟｲﾝﾀ型など )
  Ko* p_ko2 =  reinterpret_cast<Ko*>( new Oya( ) );

  if ( p_ko2 != NULL) // NULLが代入されていなければ、
  {
```

```
    cout << "reinterpret_castの結果:ｷｬｽﾄ成功" << endl; // NULLでないので、表示される。
}
```

・基本ﾃﾞｰﾀ型などの型ｷｬｽﾄでは、
「static_cast」をよく使いますが、
これはｺﾝﾊﾟｲﾙ時にｷｬｽﾄが試みられるというもので、
危険なｷｬｽﾄを行った場合は、ｴﾗｰが表示されます。

・これをよく使うのは、親ｸﾗｽのﾎﾟｲﾝﾀを、子ｸﾗｽのﾎﾟｲﾝﾀにｷｬｽﾄする
「ﾀﾞｳﾝｷｬｽﾄ」の時で、使わないと、ｺﾝﾊﾟｲﾙｴﾗｰになります。

・まったく親子関係にないｸﾗｽのﾎﾟｲﾝﾀ同士でもｷｬｽﾄはされますが、
当然ながら、正常に機能する保証はありません。

・「dynamic_cast」の場合は、
ﾌﾟﾛｸﾞﾗﾑの実行時に 型ｷｬｽﾄを行い、
危険なｷｬｽﾄの場合には、代わりに、NULLを返します。

・実行速度が遅いので、ﾃﾞﾊﾞｯｸ時に使う
くらいにしておいた方がいいと思います。

・「 reinterpret_cast」は、
まったく系統の違うﾃﾞｰﾀ型であっても
強引にｷｬｽﾄしてしまいます。

・C言語では、強引にｷｬｽﾄしても、
ﾎﾟｲﾝﾀは無事でしたが、
C++では、壊れる可能性があります。

・「 reinterpret_cast」による危険なｷｬｽﾄや、
「void*」に入れてから、他の型にｷｬｽﾄする場合は
特に注意が必要です。

・delete演算子は、ﾎﾟｲﾝﾀを解放する際に、

ﾎﾟｲﾝﾀが内部的に隠し持っている型情報を利用していますが、
ひとたびvoid*に代入してしまうと、これが失われてしまい、
正常に解放できなくなります。

・「void*」に入れたｱﾄﾞﾚｽは、
いったん、元の型に戻してから
ｷｬｽﾄするようにして下さい。

・この他にも、
「const」を解除するための
「const_cast」などがあります。

・「const_cast」を使えば、
「参照」や「volatile」を解除することもできます。

・また、「仮想関数」が含まれているｸﾗｽであれば、
ﾃﾞｰﾀ型の「ID」や、「名前」を取得することもできます。↓

```
#include <typeinfo> // 「type_info」ｸﾗｽと、取得用の「typeid」関数が定義されている。
#include <iostream>

using namespace std;

void main( )
{

   // 親ｸﾗｽ型の 【ﾎﾟｲﾝﾀ変数】 に、
  // 子ｸﾗｽ型の 【ｲﾝｽﾀﾝｽ】 (ｱﾄﾞﾚｽ) を代入した場合、 ↓
   HamBerger* p_cheese = new CheeseBerger(); // ﾁｰｽﾞﾊﾞｰｶﾞｰ型のｲﾝｽﾀﾝｽを作る。

   // 【ｲﾝｽﾀﾝｽ】 を渡すと、子ｸﾗｽの型になる。↓
   // 結果は、「class CheeseBerger」
   cout << typeid(*p_cheese).name( ) << endl;
```

```
    // 【ﾎﾟｲﾝﾀ変数】を渡すと、ﾎﾟｲﾝﾀ変数の型 ( 親ｸﾗｽのﾎﾟｲﾝﾀ型 ) になる。↓
    // 結果は、「class HamBerger *」
    cout << typeid( p_cheese ).name( ) << endl;

    // 【ﾃﾞｰﾀ型】を、渡すこともできる。↓
    // 結果は、「class HamBerger」
    cout << typeid( HamBerger ).name( ) << endl;

    // 「type_info」ｸﾗｽには、ｲﾝｽﾀﾝｽ同士を比較するために、
    // 「==」「!=」演算子がｵｰﾊﾞｰﾛｰﾄﾞされています。↓
    bool is_same = ( typeid( HamBerger ) == typeid( CheeseBerger ) );

}
```

・ﾃﾝﾌﾟﾚｰﾄｸﾗｽの場合は、
ﾃﾝﾌﾟﾚｰﾄﾊﾟﾗﾒｰﾀのﾃﾞｰﾀ型も同じでないと、
同じﾃﾞｰﾀ型とは見なされません。

・これもやはり、やけに重いので、
なるべく使わない方がいいと思います。

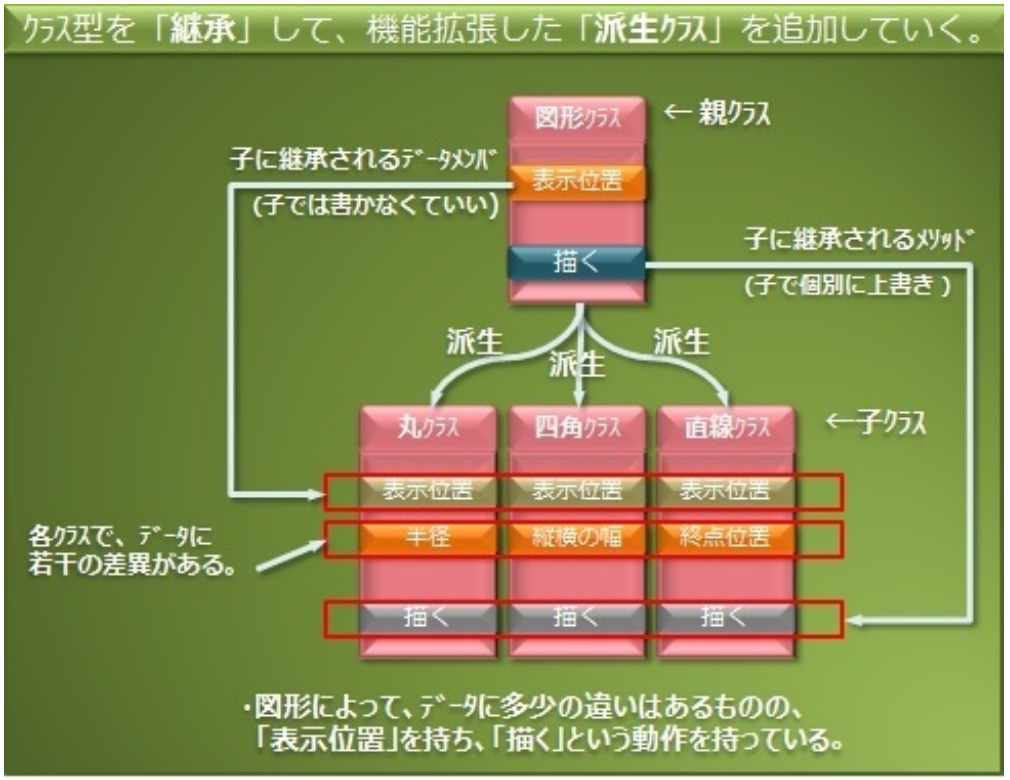
・「dynamic_cast」が重いのも、
これを使っているからです。

・ﾌｧｸﾄﾘｰｸﾗｽのﾕﾆｯﾄﾃｽﾄなどで利用する場合もありますが、
使う必要に迫られた場合は、設計を見直すべきです。

・ﾃﾞｰﾀ型によって、処理を分岐させる場合は、
仮想関数によるｵｰﾊﾞｰﾛｰﾄﾞを使用して下さい。

・使えない場合は、ｺﾝﾊﾟｲﾙｽｲｯﾁ「/GR」が、
指定されているか、確認して下さい。

「抽象ｸﾗｽ」 (と、純粋仮想関数)



・ｸﾗｽ型「図形」は、どんな形をしているんでしょうか。

・形が決まってないものは、もちろん表示もできません。

・このように、ほとんど何も決まっていない親ｸﾗｽのことを
「抽象ｸﾗｽ」といいます。

・「抽象ｸﾗｽ」では、子ｸﾗｽ側で
上書きされるであろうﾒｿｯﾄﾞを、
「仮想関数」として宣言しておきます。↓

```
// Zukei.h

class Zukei // 抽象ｸﾗｽ「図形」の宣言
{
    virtual ~Zukei() = 0; // ﾃﾞｽﾄﾗｸﾀを 「純粋仮想関数」 として宣言する。
    virtual void Hyouji() = 0; // 表示ﾒｿｯﾄﾞを 「純粋仮想関数」 として宣言する。
};
```

```
// Zukei.cpp

#include "Zukei.h"

Zukei::~Zukei() {} // これを書いてないと、ﾘﾝｸｴﾗｰになる。
```

・仮想関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言の後で、
「0」が代入されています。

・これを「純粋仮想関数」といいます。

・処理内容は、この時点では決まっていないので、
基本的には、書かなくてもいいんですが、
ﾃﾞｽﾄﾗｸﾀの定義だけは書いておかないと、
ﾘﾝｸ時に、「外部ｼﾝﾎﾞﾙ****は未解決です。」というｴﾗｰが出ます。

関数の「ｵｰﾊﾞｰﾛｰﾄﾞ」

・**C++**では、同じ名前の関数やﾒｿｯﾄﾞを
いくつも定義することができます。

・ただし、「ｼｸﾞﾈﾁｬ」(引数の構成) は、
それぞれ違ったものでないといけません。

・ｺﾝﾊﾟｲﾗは、このｼｸﾞﾈﾁｬを見て、
どの関数が呼ばれたのかを特定しています。

```
#include <iostream>
using namespace std;

int Tasu( int a, int b ) { return ( a + b ); }          /* 引数が2つの「Tasu」関数 */
int Tasu( int a, int b, int c ) { return ( a + b + c ); } /* 引数が3つの「Tasu」関数 */
float Tasu( float a, float b ) { return ( a + b ); }    /* 引数がfloatの「Tasu」関数 */

void main( )
{
    // 下記の、3つの呼び出しは、それぞれｼｸﾞﾈﾁｬが違っている。
    cout << "1 + 1 = " << Tasu( 1, 1 ) << endl;
    cout << "1 + 1 + 1 = " << Tasu( 1, 1, 1 ) << endl;
    cout << "1.0f + 1.0f = "  << Tasu( 1.0f, 1.0f ) << endl;
}
```

ｺﾝﾎﾟｼﾞｼｮﾝｸﾗｽと派生ｸﾗｽの使い分け

・さて、ｸﾗｽ型を継承させると、
　親ｸﾗｽのﾒﾝﾊﾞを、子ｸﾗｽへと引き継ぐことができました。↓

```
class ClassA
{
  int member1;
  int GetMethod1() const;

  int member2;
  int GetMethod2() const;
};

class ClassB : public ClassA
{
  // ClassAの、ﾃﾞｰﾀﾒﾝﾊﾞとﾒｿｯﾄﾞが継承されている。
};
```

・このようにして、ﾃﾞｰﾀの形式や、機能を
　使い回すことができるのは、大変便利なことですが、
　子ｸﾗｽによっては、必要がないﾒﾝﾊﾞもあるわけです。

・そうした場合には、親ｸﾗｽのﾒﾝﾊﾞを、
　もう少し切り分けて、別々のｸﾗｽとして宣言し、
　そのｲﾝｽﾀﾝｽを、子ｸﾗｽのﾃﾞｰﾀﾒﾝﾊﾞとして持たせます。↓

```
class ClassA1
{
```

```
  int member1;
  int GetMethod1() const;
};

class ClassA2
{
  int member2;

  int GetMethod2() const;
};

class ClassB1
{
   ClassA1 member1;
};

class ClassB2
{
   ClassA2 member1;
};
```

・このように、入れ子にしたｸﾗｽを、
「ｺﾝﾎﾟｼﾞｼｮﾝｸﾗｽ」と言います。

・ｸﾗｽ型を部品化するという点では、ｺﾝﾎﾟｼﾞｼｮﾝに軍配が上がりますが、
ﾒｿｯﾄﾞのｵｰﾊﾞｰﾛｰﾄﾞ (実装の上書き) を利用して、
ｲﾝｽﾀﾝｽごとに別の処理をさせるような場合には、
継承が必要となります。

「ｲﾝﾀｰﾌｪｲｽ」 (__interface)

・「ｲﾝﾀｰﾌｪｲｽ」は、「抽象ｸﾗｽ」を
さらにｼﾝﾌﾟﾙにしたものです。

・ﾃﾞｰﾀﾒﾝﾊﾞは いっさい持たず、
ﾒｿｯﾄﾞのﾌﾟﾛﾄﾀｲﾌﾟ宣言しか書きません。 ↓

```
// 「ｲﾝﾀｰﾌｪｲｽ」の宣言。 ↓

__interface Zukei
{
public: // ﾒﾝﾊﾞは、すべて公開。

    void Hyouji( ); // ﾒｿｯﾄﾞのﾌﾟﾛﾄﾀｲﾌﾟ宣言のみ行う。 (「純粋仮想関数」と同じ扱いとなる。 )
};
```

```
// 「ｲﾝﾀｰﾌｪｲｽ」を「実装」するｸﾗｽの宣言。 ↓

class Sankaku : public Zukei
{
   void Hyouji( )
  {
       // 表示処理をｺｺに書く。 ( 処理を書くことを、「実装する」という。 )
  }
};
```

・「ｲﾝﾀｰﾌｪｲｽ」は、
C/C++で書いたﾗｲﾌﾞﾗﾘを、
「**COM**」や「**.Net**」から利用する場合など、
他言語で書かれたﾌﾟﾛｸﾞﾗﾑと連絡をとる際に使います。

DLLのクラス型を呼び出す

・**DLL** を 動的 ﾘﾝｸ で ﾛｰﾄﾞ する場合は、
C++ の ｸﾗｽ 型 を 直接利用することはできません。

・しかし、前述の「ｲﾝﾀｰﾌｪｲｽ」を使えば、
間接的に ｲﾝｽﾀﾝｽ を生成して、
その ﾒｿｯﾄﾞ を呼び出すことができます。

```
// DLL側のヘッダファイル「dll_main.hpp」

#pragma once

#ifndef __DLL_MAIN_H__
#define __DLL_MAIN_H__

#include <targetver.h>
#include <windows.h>

#include <stdio.h>

// ----------------------------------------------------
// ※ ﾏｸﾛ定数「_EXPORTS」は、DLLのｺﾝﾊﾟｲﾙ時に、/D ｵﾌﾟｼｮﾝ で定義する。

#ifdef _EXPORTS
  #define _DLL_MEMBER __declspec(dllexport)
#else
  #define _DLL_MEMBER __declspec(dllimport)
#endif

// ｸﾗｽの公開ﾒﾝﾊﾞ をｴｸｽﾎﾟｰﾄするには、
// ｸﾗｽの宣言のところで、__declspec(dllimport)を指定するか、
// ( ﾒﾝﾊﾞ ごとにｴｸｽﾎﾟｰﾄ宣言する必要はない。)
// または、ﾓｼﾞｭｰﾙ定義ﾌｧｲﾙ (*.def) を独自に作成し、
// 呼び出し側のﾘﾝｸ時に、/DEF ｵﾌﾟｼｮﾝで指定する。
```

```
// ( ｴｸｽﾎﾟｰﾄ宣言は、ﾒﾝﾊﾞごとに必要。)

// ---------------------------------------------------
// ｴｸｽﾎﾟｰﾄするｸﾗｽのｲﾝﾀｰﾌｪｲｽ宣言。

// ( ｸﾗｽのｲﾝｽﾀﾝｽは、DLL側で生成・解放する。
//  呼び出し側では、このｲﾝﾀｰﾌｪｲｽへのﾎﾟｲﾝﾀで操作する。)
//
__interface _DLL_MEMBER IClass1
{
    // ----------------------------------
    // ﾒｿｯﾄﾞ

public:

    int  GetMethod1(void) const;          // ﾒﾝﾊﾞの値を取得する。
    void SetMethod1( const int value_ ); // ﾒﾝﾊﾞの値を変更する。

    // ----------------------------------

};

// ---------------------------------------------------
// ｸﾗｽの宣言。( これはｴｸｽﾎﾟｰﾄされない。)

class Class1 : public IClass1
{

    // ----------------------------------
    // ﾃﾞｰﾀﾒﾝﾊﾞ

protected:

    int member1; // ﾃﾞｰﾀﾒﾝﾊﾞ。

    // ----------------------------------
    // ﾒｿｯﾄﾞ
```

```
public:

    Class1();           // ｺﾝｽﾄﾗｸﾀ。
    virtual ~Class1(); // ﾃﾞｽﾄﾗｸﾀ。( ※ virtual 必須。)

   // ｲﾝﾀｰﾌｪｲｽで定義されているﾒｿｯﾄﾞは、必ず実装する。
   int  GetMethod1(void) const;
   void SetMethod1( const int value_ );

   // ----------------------------------

}; // end class

// ------------------------------------------------
// ここで C ﾘﾝｹﾞｰｼﾞ にすることによって、ｴｸｽﾎﾟｰﾄされる
// 関数や変数の名前が、自動的に変更されることを防ぐ。

extern "C"
{
// ----------------------------------
// ｸﾞﾛｰﾊﾞﾙ変数のexport宣言。

  extern _DLL_MEMBER int global_var1;

// ----------------------------------
// ｸﾞﾛｰﾊﾞﾙ関数のexport宣言。

  _DLL_MEMBER int GlobalFunc1( const int value1_, const int value2_ );

  _DLL_MEMBER IClass1* CreateInstance(void);
  _DLL_MEMBER void     DeleteInstance( IClass1** pp_instance_ );

}

// ------------------------------------------------
// DLLのｴﾝﾄﾘｰﾎﾟｲﾝﾄ。
```

```
BOOL APIENTRY DllMain( HANDLE h_module_,
  DWORD ul_reason_for_call_, void* p_reserved_ );

// ------------------------------------------------

#endif
```

```
// DLL側のソースファイル「dll_main.cpp」

#include "c:¥dll_test¥dll_main.hpp"

// ----------------------------------
// ｺﾝｽﾄﾗｸﾀ

   Class1::Class1(void)
   {
       member1 = 0; // ﾃﾞｰﾀﾒﾝﾊﾞを初期化しておく。
   }

// ----------------------------------
// ﾃﾞｽﾄﾗｸﾀ

   Class1::~Class1(void)
   {
   }

// ----------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞの値を返す。

   int Class1::GetMethod1( void ) const
   {
       return member1;
   }

// ----------------------------------
```

```
// ﾃﾞｰﾀﾒﾝﾊﾞの値を変更する。

   void Class1::SetMethod1( const int value_ )
   {
        member1 = value_;
   }

// ####################################
// ｸﾞﾛｰﾊﾞﾙ変数の宣言
// ####################################

  int global_var1 = 0; // 初期化しておく。

// ####################################
// ｸﾞﾛｰﾊﾞﾙ関数の宣言
// ####################################
// ----------------------------------
// 渡された値1と値2の和を返す。

   int GlobalFunc1( const int value1_, const int value2_ )
   {
       return ( value1_ + value2_ );
   }

// ----------------------------------
// ｲﾝｽﾀﾝｽを作成する。( ｲﾝﾀｰﾌｪｲｽへのﾎﾟｲﾝﾀを返す。)

   IClass1* CreateInstance(void)
   {
       return (IClass1*) new Class1();
   }

// ・仮に、DLLが、別のﾌﾟﾛｾｽ上でﾛｰﾄﾞされていても、
//   常にそのDLL側でｱﾄﾞﾚｽが参照されるのであれば、
//   ｱﾄﾞﾚｽ空間は同じなので、問題はない。

// ----------------------------------
```

```
// ｲﾝｽﾀﾝｽを解放する。( ｲﾝﾀｰﾌｪｲｽへのﾎﾟｲﾝﾀを渡す。)

   void DeleteInstance( IClass1** pp_instance_ )
   {
      // ※ ｸﾗｽへのﾎﾟｲﾝﾀ型に型ｷｬｽﾄしてからでないと、
      //   使用領域のｻｲｽﾞが不明なため、正しく解放されない。

      delete ( (Class1*) *pp_instance_ );

      *pp_instance_ = NULL;
   }

// ------------------------------------
// DLL側のｴﾝﾄﾘｰﾎﾟｲﾝﾄ

BOOL APIENTRY DllMain( HANDLE h_module_,
DWORD ul_reason_for_call_, void* p_reserved_ )
{

  switch ( ul_reason_for_call_ )
  {
   case DLL_PROCESS_ATTACH: // DLLがﾛｰﾄﾞされた。
   case DLL_THREAD_ATTACH:  // ｽﾚｯﾄﾞが開始された。
   case DLL_THREAD_DETACH:  // ｽﾚｯﾄﾞが終了された。
   case DLL_PROCESS_DETACH: // DLLがｱﾝﾛｰﾄﾞされた。
         break;
   }

   return TRUE;
}

// ------------------------------------
```

// 呼び出し側のヘッダファイル「**main.hpp**」

```
#pragma once
```

```
#ifndef __TEST_MAIN_H__
#define __TEST_MAIN_H__

#include <targetver.h>
#include <windows.h>

#include <stdio.h>

// ※DLL側のｸﾗｽのｲﾝﾀｰﾌｪｲｽを宣言しているので、ｲﾝﾎﾟｰﾄに必要。
#include "c:¥dll_test¥dll_main.hpp"



 // DLL側のｸﾞﾛｰﾊﾞﾙ関数の型を、再定義しておく。
typedef int (__cdecl*GlobalFunc1Type)(const int, const int);
typedef IClass1* (__cdecl*CreateInstanceType)(void);
typedef void (__cdecl*DeleteInstanceType)(IClass1**);

      // それを用いて、関数ﾎﾟｲﾝﾀを
      // ｸﾞﾛｰﾊﾞﾙ変数としてextern宣言しておく。
      extern GlobalFunc1Type     p_func1;
      extern CreateInstanceType p_create_func;
      extern DeleteInstanceType p_delete_func;

#endif
```

```
 // 呼び出し側のソースファイル「main.cpp」

#include "c:¥dll_test¥main.hpp"

// ------------------------------------
// ｸﾞﾛｰﾊﾞﾙ変数の宣言。

GlobalFunc1Type     p_func1         = NULL;
CreateInstanceType p_create_func = NULL;
DeleteInstanceType p_delete_func = NULL;
```

```
// ------------------------------------
// 呼び出し側のｴﾝﾄﾘｰﾎﾟｲﾝﾄ。

int main(void)
{

   // --------------
    // DLLをﾛｰﾄﾞする。

  HMODULE h_dll =
    LoadLibraryW( L"c:¥¥dll_test¥¥dll_main.dll" );

   if ( h_dll == NULL )
   { printf( "%s¥n", "DLL Load Error !" ); return -1; }

   // --------------
   // ｸﾞﾛｰﾊﾞﾙ変数のｱﾄﾞﾚｽを取得してみる。

   int* p_global_var1 =
     (int*) GetProcAddressW( h_dll, L"global_var1" );

   if ( p_global_var1 == NULL )
   { printf( "%s¥n","p_global_var1 Address Error !" ); return -1; }

   printf( "global_var1 : %d¥n", *p_global_var1 ); // ﾎﾟｲﾝﾀから値を取得する。

   // --------------
   // ｸﾞﾛｰﾊﾞﾙ関数のｱﾄﾞﾚｽを取得してみる。

   p_func1 = (GlobalFunc1Type) GetProcAddressW( h_dll, L"GlobalFunc1" );

   // 引数2には、序数を渡してもいい。
   // p_func1 = (GlobalFunc1Type) GetProcAddress( h_dll, (LPCSTR) 3 );

   //
   // ・序数は、/DEF ｵﾌﾟｼｮﾝを使って、独自のﾓｼﾞｭｰﾙ定義ﾌｧｲﾙを
   //    指定しない場合は、自動的に割り振られる。
```

```
//
// ・序数を調べるには、 DUMPBIN /EXPORTS DLLへのﾊﾟｽ を実行する。

if ( p_func1 == NULL )
{ printf( "%s¥n","GlobalFunc1 Address Error !" ); return -1; }

printf( "func1: %d¥n", (*p_func1)( 10, 20 ) ); // 関数ﾎﾟｲﾝﾀから呼び出す。

// --------------
// 次に、 DLL側ｸﾗｽのｲﾝｽﾀﾝｽを生成・解放してくれる2つの
// ｸﾞﾛｰﾊﾞﾙ関数のｱﾄﾞﾚｽも取得してみる。

p_create_func = (CreateInstanceType)
   GetProcAddressW( h_dll, L"CreateInstance" );

if ( p_create_func == NULL )
{ printf( "%s¥n", "CreateInstance Address error !" ); return -1; }



p_delete_func = (DeleteInstanceType)
  GetProcAddressW( h_dll, L"DeleteInstance" );

if ( p_delete_func == NULL )
{ printf( "%s¥n", "DeleteInstance Address error !" ); return -1; }

// --------------
// DLL側ｸﾗｽのｲﾝﾀｰﾌｪｲｽﾎﾟｲﾝﾀを取得する。

IClass1* p1 = (*p_create_func)(); // ｲﾝｽﾀﾝｽを生成する。

p1->SetMethod1(100);                          // ﾒﾝﾊﾞの値を変更してみる。
printf( "member1: %d¥n",  p1->GetMethod1() ); // ﾒﾝﾊﾞの値を取得してみる。

(*p_delete_func)( &p1 ); // ｲﾝｽﾀﾝｽを解放する。

 // --------------
FreeLibrary( h_dll ); // DLLをｱﾝﾛｰﾄﾞする。
```

```
    // --------------

    return 0;
}

// -----------------------------------
```

・ｸﾗｽを扱うDLLについては、前節の、「DLL」のﾍﾟｰｼﾞでも
　少し触れていましたが、通常は、「COMｻｰﾊﾞｰ」として作成します。

・COMについては、次節の「COM」で詳しく扱いますが、
やはりｲﾝﾀｰﾌｪｲｽを用いて間接的に操作するというもので、
このﾍﾟｰｼﾞで紹介した方法とよく似ています。

引数の初期値

・関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言で、
　引数に初期値を設定しておくと、
　関数の呼び出し時に、引数へ渡す値が省略された場合に、
　この初期値が渡されます。↓

```
// Test.h

// Tasu関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言。↓
int Tasu( int a = 100, int b = 100 ); // 初期値を、「100」に設定してみた。
```

```
// Test.cpp

#include <iostream>
using namespace std;

// Tasu関数の実装定義。↓(※ こちら側に初期値を書いてはいけない。)
int Tasu( int a, int b ) { return ( a + b ); }

void main( )
{
    cout << "200 + 100 = " << Tasu( 200 ) << endl; // 引数2を省略。
    cout << "100 + 100 = " << Tasu( ) << endl; // 引数1と2を省略。
}
```

・非常に便利なんですが、決まりがあって、
　初期値が設定された引数以降の引数には、
　初期値を設定する必要があります。↓

```
int Tasu( int a = 100, int b );// これはﾀﾞﾒ。
```

・さらに、途中の引数の
　初期値を省略することもできません。↓

```
int Tasu( int a , int b = 100 );// これもﾀﾞﾒ。
```

・ということですから、
　引数に初期値を設定する場合は、
　かならず最初の引数から
　続けて設定するようにして下さい。

constﾒｿｯﾄﾞと「mutable」

・ﾒｿｯﾄﾞの中で、値を変更することがない引数は、
「const」を付けて、定数にしておきます。↓

```
int GetValue( const int index_ ) { return values[ index_ ]; };
```

・ﾒｿｯﾄﾞの中で、ﾃﾞｰﾀﾒﾝﾊﾞの値を変更しない場合は、
「const」を付けて、「const ﾒｿｯﾄﾞ」 にしておきます。↓

```
int GetValue( const int index_ ) const { return values[ index_ ]; };
```

・ｲﾝｽﾀﾝｽが「const」で宣言されていた場合は、
通常であれば、ﾃﾞｰﾀﾒﾝﾊﾞの値が変更できないのですが、
この、「const ﾒｿｯﾄﾞ」だけは、呼び出すことができます。↓

```
class Hamuta
{

protected:

  int values[10];

public:

  // ﾌﾟﾛﾄﾀｲﾌﾟ宣言の後ろに「const」を付けると、constﾒｿｯﾄﾞになる。
  // ( ※ただし、すべての引数が、constでなければならない。 )
  int GetValue( const int index_ ) const { return values[ index_ ]; };
```

```
  // こちらは、ﾃﾞｰﾀﾒﾝﾊﾞへの変更を行う通常のｾｯﾀｰﾒｿｯﾄﾞ。
  void SetValue ( const int index_, const int value_ ) { values[ index_ ] = value_; };
};
```

```
void main( )
{
   const Hamuta hamuta1; // const指定されたｲﾝｽﾀﾝｽを宣言する。
   // constで宣言されたｲﾝｽﾀﾝｽのﾒﾝﾊﾞには、ｱｸｾｽできない。

   int value = hamuta1.GetValue( 0 ); // constﾒｿｯﾄﾞは呼び出せる。

   hamuta1.SetValue( 0, 100 );  // constではないﾒｿｯﾄﾞだと、ｴﾗｰが出る。
}
```

・ｲﾝｽﾀﾝｽが、
「const」で宣言されている場合は、
「const ﾒｿｯﾄﾞ」から、
ﾃﾞｰﾀﾒﾝﾊﾞの値を変更することは、
できないことになっていますが、
「mutable」を付けたﾃﾞｰﾀﾒﾝﾊﾞであれば、
値を変更することができます。↓

```
class Hamuta
{
protected:
      mutable int values[10];
};
```

・ﾎﾟｲﾝﾀの宣言時に、後置でconstを付けても
delete演算子で解放できますし、
mutableで解除することもできますから、
変更を完全に防ぐことはできません。

・つまり、ﾎﾟｲﾝﾀを操作されたくない場合は、
ｸﾗｽの外に出さないようにするしかありません。

・それから、ﾎﾟｲﾝﾀの宣言時に、
後置で**const**を付けた場合は、
ﾎﾟｲﾝﾀへのｱﾄﾞﾚｽの代入はできませんが、
constでないﾒｿｯﾄﾞを呼ぶことができます。↓

```
int* p_ham1 const = new Hamta();
```

・そして、前置と後置の両方をつけると、
両方の制約を適用することができます。　↓

```
const int* p_ham1 const = new Hamta();
```

・ちなみに、前置した場合は、ｱﾄﾞﾚｽ代入ができます。

集約ｸﾗｽとｺﾝﾎﾟｼﾞｯﾄｸﾗｽ

・さて、上の図に登場するｸﾗｽ**B**は、
他の複数のｸﾗｽから利用されていて、
ｸﾗｽ**A**や**C**から見れば、部品の**1**つであると
考えることができます。↓

```
class ClassA
{
 ClassB* p_data; // ｸﾗｽBのｲﾝｽﾀﾝｽを、ﾃﾞｰﾀﾒﾝﾊﾞとして持っている。
};
```

・ｸﾗｽとｸﾗｽとの間には、親子の関係もありましたが、
会社や組織のような関係もあるんですね。

・ｺﾝﾎﾟｼﾞｯﾄｸﾗｽ**B**は、ｸﾗｽ**A**専属の社員ですが、
集約ｸﾗｽ**B**は、いろんな会社の仕事をこなす
ﾌﾘｰﾗﾝｽのﾗｲﾀｰのようなものです。

・機械の部品でいえば、

特定の機械にしか組み込めない特注の部品と、
広く流通している一般的な部品とがあるようなものです。

・**USB**端子たどか、印刷用紙は、集約ｸﾗｽで、
NASAのﾛｹｯﾄに使うｴﾝｼﾞﾝなどは、
ｺﾝﾎﾟｼﾞｯﾄｸﾗｽです。

・どちらも必要なもので、
目的に応じて、作られるものです。

・おぼえ方としては、
あるｸﾗｽの中に、
別のｸﾗｽのｲﾝｽﾀﾝｽを持つことを「集約」といい、
中でも、特定のｸﾗｽにだけ持たせる物を
「ｺﾝﾎﾟｼﾞｯﾄ」というということです。

・おさらいしておくと、
「ｺﾝﾎﾟｼﾞｼｮﾝ」というのは、入れ子になったｸﾗｽのことで、
「ｺﾝﾎﾟｼﾞｯﾄ」は、その中でも、特定のｸﾗｽでしか使われない
専用のｸﾗｽのことです。

「名前空間」

・たくさんのﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞしていると、
　名前が衝突することがあります。

・「名前空間」を設定しておけば、
　名前の衝突を防ぐことができます。↓

```
// ------------------------------
// Mac.h

#include <iostream>
using namespace std;

namespace IT // 名前空間「IT」のﾌﾞﾛｯｸ。（この中ｶｯｺ内の名前は、この名前空間に属する。）
{
  class Mac
  {
      Mac( );
      ~Mac( );
  };
}; // ｸﾗｽのﾌﾞﾛｯｸと同じで、ｾﾐｺﾛﾝが付いている。

namespace FC // 名前空間「FC」のﾌﾞﾛｯｸ。（この中ｶｯｺ内の名前は、この名前空間に属する。）
{
  class Mac
  {
      Mac( );
      ~Mac( );
  };
}; // ｸﾗｽのﾌﾞﾛｯｸと同じで、ｾﾐｺﾛﾝが付いている。

// ------------------------------
// Mac.cpp
```

```
#include "Mac.h"

 namespace IT
 {

   Mac::Mac( )
   {
      cout << "IT業界のMacです。" << endl;
   }

   Mac::~Mac( ) { }

 } // ﾒｿｯﾄﾞのﾌﾞﾛｯｸと同じで、ｾﾐｺﾛﾝが付いていない。

 namespace FC
 {
   Mac::Mac( )
   {
      cout << "FC業界のMacです。" << endl;
   }

   Mac::~Mac( ) { }

 } // ﾒｿｯﾄﾞのﾌﾞﾛｯｸと同じで、ｾﾐｺﾛﾝが付いていない。

// ------------------------------
// main.cpp

#include "Mac.h"

using namespace IT;  // 名前空間「IT」を使いますよ、という宣言。( 名前空間を省略できる )
using namespace FC; // 名前空間「FC」を使いますよ、という宣言。( 名前空間を省略できる )

 int main( )
 {
    IT::Mac mac1; // 名前を使う時は、必ず、所属する名前空間を指定する。
    FC::Mac mac2; // 名前を使う時は、必ず、所属する名前空間を指定する。
```

```
    return 0;
}

// ----------------------------
```

・「名前空間」のﾌﾞﾛｯｸ内で定義した名前は、
使う時には必ず、名前空間を指定します。

・「**using namespace**」を使えば、省略できますが、
その場合は、どの空間の名前なのかがｱｲﾏｲになり、
ｺﾝﾊﾟｲﾙ時に、警告が表示されることがあります。

・それから、名前空間とﾃﾞｰﾀ型の間の「**::**」ですが、
これは、「ｽｺｰﾌﾟ演算子」といいます。

・それと、名前空間の中で宣言されたｼﾝﾎﾞﾙを前方宣言する場合は、
次のようにして、名前空間で囲みます。↓

```
namespace IT { class Mac; }
```

・入れ子になっている場合は、同じように入れ子にして書きます。↓

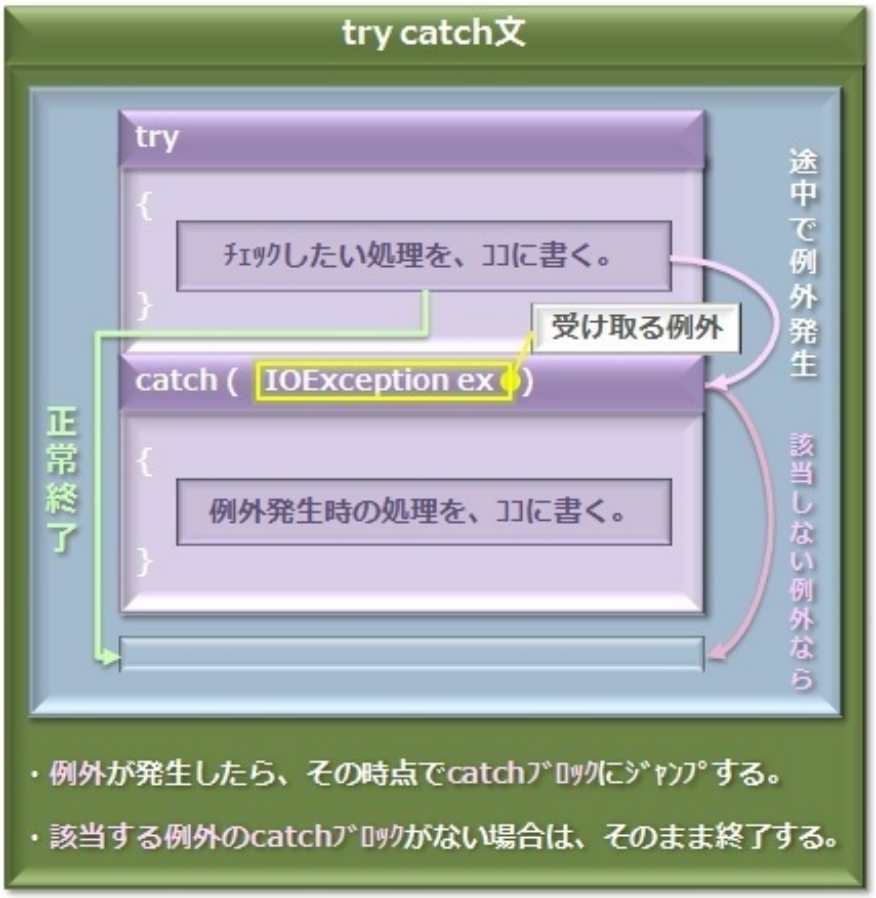
```
namespace TOKYO { MINATOKU { class Akasaka; } }
```

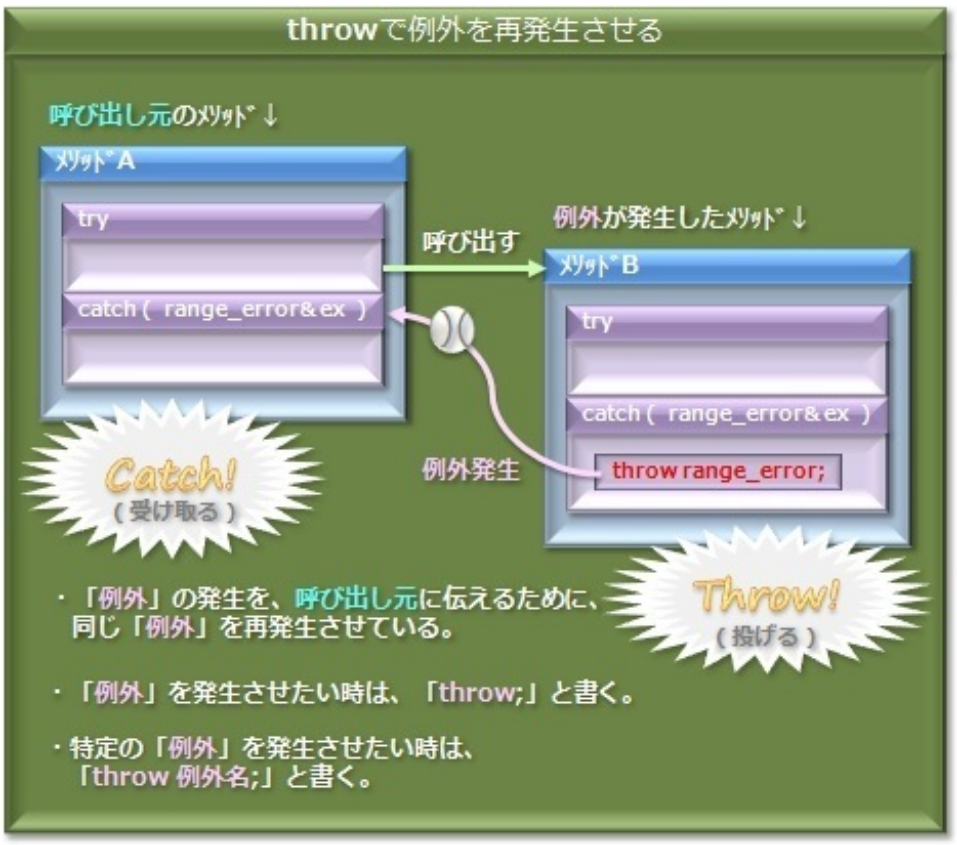
・さらに、**typedef**したﾃﾞｰﾀ型を使う場合は、
次のようにして、名前空間の中で、再度、**typedef**する必要があります。↓

```
namespace IT
{
  class Mac; // 元のﾃﾞｰﾀ型の前方宣言をしておく。
  typedef Mac MacPC; // 別名定義する。

  class MacPC;  // 前方宣言する。
}
```

・これで、「**IT::MacPC**」ｸﾗｽが使えるようになります。

例外処理 (try catch throw)





・ﾌﾟﾛｸﾞﾗﾑが実行されている間に、
ｼｽﾃﾑの不調によって

ｴﾗｰが発生することがあります。

・**C**言語では、
たとえば、ﾌｧｲﾙを開くために、
「**fopen_s**」関数を呼び出した場合は、
次のようにして結果をﾁｪｯｸして、
ｴﾗｰ処理を行なってしていました。↓

```
errno_tresult = fopen_s( &p_file, "C:\\00_Doc\\ansi.txt", "r" ); /* ﾌｧｲﾙを開く。 */

if ( p_file == NULL ) return 0; /* ﾌｧｲﾙﾎﾟｲﾝﾀが取得できているか、ﾁｪｯｸする。 */
if ( result != 0 ) return 0;      /* 戻り値を、ﾁｪｯｸする。*/
```

・しかし、こうした処理が続く場合は、
同じようなｴﾗｰ処理を、何度も書くことになり、
ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞが読みづらいものになっていました。

・**C++**では、
こうしたｴﾗｰ処理を、わかりやすく書くために、
「例外」という機能を導入しています。↓

```
try // 「例外」(ｴﾗｰ) が発生しそうな処理↓
{
    int array[1];             // 要素数1の配列を宣言する。
    int value = array[100]; // 要素番号100の値を取り出す。 ( → 例外が発生する )
}
catch( out_of_range& ex_ )// 例外発生時の処理↓
{
   cout << "範囲外の要素番号が指定されました。" << endl;
    abort(); // ﾌﾟﾛｸﾞﾗﾑを強制終了する。
}
```

・「try」ﾌﾞﾛｯｸには、
「例外」(ｴﾗｰ報告)が発生しそうな処理
を書きます。

・「catch」ﾌﾞﾛｯｸには、
「例外」が発生した時の処理
を書きます。

・「catch」ﾌﾞﾛｯｸは、
いくつも続けて書くことができ、
引数の型は、何でも構いません。

・「throw」 (ｽﾛｰ) は、
わざと「例外」を発生させる時に使います。

・たとえば私が、
「Reigai」という例外型を宣言して、
それをｽﾛｰしたとします。

・ｽﾛｰされたものは、「例外」とみなされますから、
「Reigai」型の引数を持った「catch」ﾌﾞﾛｯｸがあれば、
この「Reigai」例外はｷｬｯﾁされて、
例外処理が行われます。 ↓

```
class Reigai  // まずは、独自の例外ｸﾗｽを、宣言してみる。 ↓
{
 public:
  char* p_message; // ｴﾗｰﾒｯｾｰｼﾞ を持っている。
  Reigai( char* p_message_ ) : p_message( p_message_ ) { } // ｺﾝｽﾄﾗｸﾀ
};
```

```
// さっそくｽﾛｰしてみる。↓

try
{
    Reigai ex( "よくわからないｴﾗｰです。" ); // 例外ｸﾗｽをｲﾝｽﾀﾝｽ化する。

   throw ex; // 例外をｽﾛｰする。
}
catch( Reigai& ex_ ) // 「Reigai」例外をｷｬｯﾁする例外ﾊﾝﾄﾞﾗ。
{
cout << ex_.p_message << endl; // ｴﾗｰﾒｯｾｰｼﾞを表示する。
}
```

・単に、「throw;」と書いた場合は、
　引数のないcatchﾌﾞﾛｯｸだけがｷｬｯﾁします。

・「例外」を使う場合は、
　下記のように書いておけば、
　使う人がわかりやすくて便利です。↓

```
// 例外が発生しないﾒｿｯﾄﾞは、
// 次のように書いておくと、わかりやすい。↓
bool SafeMethod( ) throw();
```

```
// 例外が発生するﾒｿｯﾄﾞは、
// 発生する例外を、ｶﾝﾏで区切って書く。↓
bool DengerMethod( int index_ , int length_ ) throw( out_of_range , range_error );
```

・「**Visual C++**」では、
　例外型のﾁｪｯｸは行っていないようですが、
　利用者への伝達事項として、

書いておいた方がいいでしょう。

・呼び出し元の関数やﾒｿｯﾄﾞでは、例外がｽﾛｰされた時に備えて、
catchﾌﾞﾛｯｸに、失敗した時の処理を書いておく必要があります。

・「例外」は、**goto**文によるｴﾗｰ処理を
扱いやすくしたものです。

・しかし、呼び出しｺｽﾄが高いため、
実行速度を重視する場合には
やや不向きといえます。

・ﾃﾞﾊﾞｯｸ時のｴﾗｰ検出には役立ちますが、
呼び出しが多岐に及ぶ場合は、
どういった例外がｽﾛｰされるのかを
あらかじめ把握して対応するのは困難であるため、
使用されていない場合が多いようです。

【ﾒﾘｯﾄ】
・ｺﾝｽﾄﾗｸﾀが失敗したことを知ることができる。

【ﾃﾞﾒﾘｯﾄ】
・throwを追加する時に、呼び出し元をすべて調べないといけない。
・制御の流れを理解しにくくなる。(不正な入力があった場合に、throwしてはいけない)
・ﾊﾞｲﾅﾘが肥大化し、ﾒﾓﾘ空間を圧迫する。(ｺﾝﾊﾟｲﾙも少し遅くなる)

「ﾌﾚﾝﾄﾞｸﾗｽ」と「ﾌﾚﾝﾄﾞ関数」(friend)

```
class Hamko; // 使用するｸﾗｽ型の前方宣言。

class Hamta
{

  friend class Hamko; // ﾌﾚﾝﾄﾞｸﾗｽの宣言。
  // これによって、「Hamko」ｸﾗｽは、
  // 「Hamta」ｸﾗｽの持つ、すべてのﾒﾝﾊﾞに、
  // ｱｸｾｽできるようになった。

  friend void Warikan( int okaikei_ , Hamta&  a_ ); // ﾌﾚﾝﾄﾞ関数の宣言。
  // これによって、「Warikan」関数は、
  // 「Hamta」ｸﾗｽの持つ、すべてのﾒﾝﾊﾞに、
  // ｱｸｾｽできるようになった。

  // あとは、いつもと一緒。

 private:
    int osaifu; // 非公開ﾒﾝﾊﾞ

};
```

```
// 「Hamta」ｸﾗｽから、ﾌﾚﾝﾄﾞ指定された
// 「Hamko」ｸﾗｽの宣言。 ↓

class Hamko
{
   Hamko( Hamta& ham_ )
  {
     ham_.osaifu = -10000; // 非公開ﾒﾝﾊﾞでも、ｱｸｾｽできる。
  }
```

```
};
```

```
// 「Hamta」ｸﾗｽから、ﾌﾚﾝﾄﾞ指定された
// 「Warikan」関数の定義。↓

void Warikan( intokaikei_ , Hamta&  a_ )
{
   a_.osaifu - = okaikei_; // 非公開ﾒﾝﾊﾞでも、ｱｸｾｽできる。
}
```

・ﾌﾚﾝﾄﾞ宣言は、「**Hamta**」にのみ適用されるもので、
その派生ｸﾗｽでは無効となります。

・ﾌﾚﾝﾄﾞ宣言は、ｱｸｾｽ指定子を無視して、
ｶﾌﾟｾﾙ化の壁を、すり抜けます。

・とまあ、身もﾌﾀもない代物なんですが、
ｸﾗｽ継承も、万能ではありませんから、
どうしても使いたいという特殊なｹｰｽがたまにあります。

・同じｿｰｽﾌｧｲﾙ上でだけ、限定して使うのであれば、
処理の流れも追えますし、混乱することはないでしょう。

「ﾃﾝﾌﾟﾚｰﾄ」 (template typename)

・「ﾃﾝﾌﾟﾚｰﾄ」というのは、
部分的に、ﾃﾞｰﾀ型を抽象化して書く書き方です。

・『ﾃﾞｰﾀ型は違うけど、処理はまったく同じ』
というｸﾗｽや関数を、
たくさん作らないといけない場合、
あるいは、将来作るかもしれないという場合に、
威力を発揮します。↓

```
// Value.h

// ﾃﾝﾌﾟﾚｰﾄｸﾗｽの宣言 ↓ ( ﾍｯﾀﾞ側で実装する )
template<typename T>   // ← ﾃﾝﾌﾟﾚｰﾄ引数ﾘｽﾄ  ( ｸﾗｽ宣言の直前に付ける )
class Value
{
    T value; // ﾃﾞｰﾀﾒﾝﾊﾞ  ( ﾃﾞｰﾀ型の「T」は、何型なのか、まだ決まってない )

   // ｺﾝｽﾄﾗｸﾀ  ( 初期化用の引数付き )
   Value( T value_ ) : value( value_ )
   {

   }

    // Getterﾒｿｯﾄﾞ
    T GetValue( )
    {
        return value;
    }

    // Setterﾒｿｯﾄﾞ
    void SetValue( T value_ )
    {
        value = value_;
```

```
    }

};
```

```
// main.cpp

#include "Value.h"

void main( )
{
    Value<int> v1( 100 );        // 「T」を、int型に置換して、ｲﾝｽﾀﾝｽ化する。
    Value<double> v2( 0.005 ); // 「T」を、double型に置換して、ｲﾝｽﾀﾝｽ化する。
}
```

・「ﾃﾝﾌﾟﾚｰﾄ引数」を、
　複数使う場合は、
　ｶﾝﾏで区切って並べます。 ↓

```
template<typename TIndex, typename TValue> // ← ﾃﾝﾌﾟﾚｰﾄ引数ﾘｽﾄ
class Value
```

・さらに、ﾃﾞﾌｫﾙﾄのﾃﾞｰﾀ型を設定しておけば、
　使用する時に、ﾃﾞｰﾀ型を省略して書くことことができます。 ↓

```
template<typename TIndex =unsigned int, typename TValue> // ← ﾃﾝﾌﾟﾚｰﾄ引数ﾘｽﾄ
class Value
```

・また、「ﾃﾝﾌﾟﾚｰﾄ」は、
　ｸﾞﾛｰﾊﾞﾙ関数にも使えます。 ↓

```
// Keisan.h

// ﾃﾝﾌﾟﾚｰﾄ関数の定義 ↓ ( やはり、ﾍｯﾀﾞ側で実装する )
template<typename T>   // ← ﾃﾝﾌﾟﾚｰﾄ引数ﾘｽﾄ  ( 関数定義の直前に付ける )
T Tasu( T a_ , T b_ )
{
   return a_ + b_;
}
```

```
// main.cpp

#include "Keisan.h"

void main( )
{
    int kotae1 = Tasu<int>( 10 , 20 );          // 「T」を、int型に置換して、関数を呼び出す。
    double kotae2 = Tasu<double>( 0.1 , 0.2 ); // 「T」を、double型に置換して、関数を呼び出す。
}
```

・上の例では、
「ﾃﾝﾌﾟﾚｰﾄ引数」に、
基本ﾃﾞｰﾀ型を渡してきました。

・ｸﾗｽ型や構造体型が渡された場合、
足し算とかできるのかというと
必ずしも、そうではありません。

・ｸﾗｽ型で、足し算をするには、
「+」演算子を、
「ｵｰﾊﾞｰﾛｰﾄﾞ」しておく必要があります。

・ともあれ、
　何かと便利な「ﾃﾝﾌﾟﾚｰﾄ」ですが、
　使い過ぎると、ｺｰﾄﾞｻｲｽﾞが肥大化する
　という、ﾃﾞﾒﾘｯﾄもあります。

・これは、ｺﾝﾊﾟｲﾙ時に、ｺﾝﾊﾟｲﾗが
　ｸﾗｽ型や関数を、自動生成しているからです。

・ｺﾝﾊﾟｲﾗは、この時に、
　ﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞ上で使用されたﾃﾝﾌﾟﾚｰﾄ引数ごとに、
　ｸﾗｽ型や関数を生成します。

・技術的には、「ﾏｸﾛ関数」や
　「ｲﾝﾗｲﾝ展開」に近い機能といえます。
(ｺﾝﾊﾟｲﾙの直前に行われるﾌﾟﾘﾌﾟﾛｾｯｻ処理)

・具体的な使用例については、
　続巻の「**STL編**」を、参照して下さい。

「ｲﾝﾗｲﾝ展開」（inline）

・処理が数行しかない短めの関数は、
　ｺﾝﾊﾟｲﾙの直前に、「ｲﾝﾗｲﾝ展開」されます。

・たとえば、次のような場合、↓

```
// main.h

inline int Tasu( int a_ , int b_ )
{
   return a_ + b_;
}
```

```
// main.cpp

#include "main.h"

void main( )
{
    int banana = 10;
    int orange =20;

     int kotae = Tasu( banana, orange );
     kotae += Tasu( banana, orange );
}
```

・この「Tasu」関数は、
　「ﾏｸﾛ関数」と同じように、置換されて、
　次のようになります。↓

```
void main( )
{
    int banana = 10;
    int orange = 20;

     int kotae = banana + orange; /* 「ｲﾝﾗｲﾝ展開」により、処理部が置換された。 */
     kotae += banana + orange;   /* 「ｲﾝﾗｲﾝ展開」により、処理部が置換された。 */
}
```

・「ｲﾝﾗｲﾝ展開」されると、
関数では無くなったことにより、
引数の受け渡しや、戻り値を返すなどの
「呼び出しｺｽﾄ」が無くなります。

・「**inline**」指定子は、これを促すものですが、
関数の処理が長い場合は、無視されます。

・なぜかといえば、
置換によって、ｺｰﾄﾞ量が増えることで、
結果的に遅くなるなどの
副作用が生じるからです。

・「ｲﾝﾗｲﾝ関数」は、置換されるだけのものですから、
「ﾏｸﾛ関数」同様に、ﾍｯﾀﾞﾌｧｲﾙ側に処理を書きます。

・「**inline**」指定子は、
「ﾒｿｯﾄﾞ」にも、もちろん使用できますが、
「ｺﾝｽﾄﾗｸﾀ」や「ﾃﾞｽﾄﾗｸﾀ」には指定できません。

・それから、「**VisualC++**」で定義されている
「**__inline**」 と 「**__forceinline**」 (強制ｲﾝﾗｲﾝ) は、
C言語のｺｰﾄﾞでも使えます。

ﾎﾟｲﾝﾀと参照の使い分け

```
class CData // ﾃﾞｰﾀｸﾗｽ。
{

protected:

   int hp;
   int mp;

public:

  CData(){}; // ｺﾝｽﾄﾗｸﾀ
  CData( const int hp_, const int mp_ ) : hp(hp_), mp(mp_){}; // 引数付きｺﾝｽﾄﾗｸﾀ
   ~CData(){}; // ﾃﾞｽﾄﾗｸﾀ

   CData( const CData& src_ ) : hp(src_.hp), mp(src_.mp){}; // ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ

   void operator=( const CData& src_ )
      { hp = src_.hp; mp = src_.mp; }; // 代入演算子の上書き

   int GetHP(void) const { return hp; };
   int GetMP(void) const { return mp; };

};
```

```
// ﾃﾞｰﾀへのﾎﾟｲﾝﾀを管理するｸﾗｽ。

template<typename T>
class CManager
{

protected:
```

```
    const T* p_data; // ﾃﾞｰﾀ領域へのﾎﾟｲﾝﾀ。

public:

    CManager( const T* p_src_ ) : p_data( p_src_ ) {}; // 引数付きｺﾝｽﾄﾗｸﾀ
    ~CManager(void){ delete p_data; }; // ﾃﾞｽﾄﾗｸﾀ ( 解放は、唯一ここでのみ行う。 )

    const GetRef(void) const { return *p_data; }; // 参照を返す。

private: // 呼び出せないように非公開にする。 ↓

    CManager(){}; // ｺﾝｽﾄﾗｸﾀ
    CManager( const p_src_ ) : p_data( src_.p_data ) {}; // ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ
    void operator=( const  p_src_ ) {}; // 代入演算子の上書き

};
```

```
// ﾃﾞｰﾀを利用する側のｸﾗｽは、 参照を持つ。( constではない参照。再設定可能。)

template<typename T>
class CCustomer
{

protected:

    const T data; // ﾃﾞｰﾀ領域への参照。
    // ( constなので、 値は変更できないが、 値が変更された場合は、 反映される。 )

public:

    CCustomer( const T& src_ ) : data( src_ ){}; // 引数付きｺﾝｽﾄﾗｸﾀ
    ~CCustomer(void) {};                         // ﾃﾞｽﾄﾗｸﾀ

    const T& GetRef(void) const { return data; }; // 参照を返す。
    // ( 初期化時にしか参照先を設定できないため、 SetRefﾒｿｯﾄﾞは無い。 )
```

```
};
```

```
int main(void)
{
  CManager<CData>* p_mng1 = new CManager<CData>( new CData(140,80) );
  CCustomer<CData>* p_cst1 = new CCustomer<CData>( p_mng1->GetRef() );

  delete p_cst1;
  delete p_mng1; // ﾃﾞｰﾀへのﾎﾟｲﾝﾀが解放される。

  return 1;
}
```

・CManagerｸﾗｽは、ﾎﾟｲﾝﾀの破棄を行うｸﾗｽで、
ｽﾏｰﾄﾎﾟｲﾝﾀのような役まわりです。

・ｽﾏｰﾄﾎﾟｲﾝﾀは、速度的に遅いので、
独自に実装される方が多いようです。

・CCustomerｸﾗｽは、ﾃﾞｰﾀを利用するｸﾗｽです。

・実際には、ﾃﾞｰﾀごとに、このようなｸﾗｽを作るのではなく、
一つのｸﾗｽの中で、複数の参照を保持して利用したり、
また、ﾎﾟｲﾝﾀの管理を行ったりもします。

Cの関数を呼び出す (extern "C")

・C言語で書かれたDLLを使おうとして、
DLLのﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞしてｺﾝﾊﾟｲﾙすると、
「外部ｼﾝﾎﾞﾙが未解決です。」
というｴﾗｰが、出ることがあります。

・これは、C++のｺﾝﾊﾟｲﾗが、
関数の名前を、勝手に改名しているからで、
これを抑制するには、DLLのﾍｯﾀﾞﾌｧｲﾙ側で、
次のように書いておきます。↓

```
// ・C++のｺﾝﾊﾟｲﾗでｺﾝﾊﾟｲﾙされる場合は、
//    「extern "C"」ﾌﾞﾛｯｸが有効となる。

// ・これは、ﾌﾞﾛｯｸ内の関数を、Cの関数として扱うように
// ｺﾝﾊﾟｲﾗに指示するためのもので、このﾌﾞﾛｯｸ内で
// C++の文法が使用された場合は、ｺﾝﾊﾟｲﾙｴﾗｰとなる。

#ifdef __cplusplus
  extern "C"
  {
#endif

    /* Cの関数のﾌﾟﾛﾄﾀｲﾌﾟ宣言。↓*/
    int Tasu( int a_ , int b_ );
    int Hiku( int a_ , int b_ );

#ifdef __cplusplus
  }
#endif
```

・Cのｺﾝﾊﾟｲﾗでｺﾝﾊﾟｲﾙされた時は、

「**extern "C"**」ﾌﾞﾛｯｸは、読み飛ばされて
無効となります。

ﾒﾝﾊﾞ関数ﾎﾟｲﾝﾀ

・ﾒｿｯﾄﾞのｱﾄﾞﾚｽを、
「関数ﾎﾟｲﾝﾀ」に代入して使う場合は、
書き方が少し違います。

・まず、静的ﾒｿｯﾄﾞを呼び出す場合は、↓

```
// ﾃｽﾄ用のｸﾗｽを宣言しておく。↓

class Value
{
protected:
   static int value; // ﾃﾞｰﾀﾒﾝﾊﾞ

public:
   static int GetValue( ) { return value; } // Getterﾒｿｯﾄﾞ
   static void SetValue( int value_ ) { value = value_; } // Setterﾒｿｯﾄﾞ
};


int Value::value = 0; // 静的ﾃﾞｰﾀﾒﾝﾊﾞの宣言と初期化。


// 関数ﾎﾟｲﾝﾀ型の定義。↓
typedef int (*Getter)(void);
typedef void (*Setter)(int);


// main関数の定義。↓
int main( )
{

  Setter p_setter = Value::SetValue; // 静的ﾒｿｯﾄﾞのｱﾄﾞﾚｽを、関数ﾎﾟｲﾝﾀに代入する。
```

```
  (*p_setter)( 100 ); // 関数ﾎﾟｲﾝﾀから呼び出す。

  Getter p_getter = Value::GetValue; // 静的ﾒｿｯﾄﾞのｱﾄﾞﾚｽを、関数ﾎﾟｲﾝﾀに代入する。
  int value = (*p_getter)(); // 関数ﾎﾟｲﾝﾀから呼び出す。

  return 0;
}
```

・次に、ｲﾝｽﾀﾝｽのﾒｿｯﾄﾞを呼び出す場合。↓

```
// ﾃｽﾄ用のｸﾗｽを宣言しておく。↓

class Value
{
protected:
   int value;

public:
   int GetValue( ) { return value; }
   void SetValue( int value_ ) { value = value_; }
};


// 関数ﾎﾟｲﾝﾀ型の定義。↓
typedef int (Value::*Getter)(void);
typedef void (Value::*Setter)(int);


// main関数の定義。↓
int main( )
{

  Value v;
```

```
  Setter p_setter = &(Value::SetValue); // 静的ﾒｿｯﾄﾞのｱﾄﾞﾚｽを、関数ﾎﾟｲﾝﾀに代入する。
  (v.*p_setter)( 100 ); // 関数ﾎﾟｲﾝﾀから呼び出す。

  Getter p_getter = &(Value::GetValue); // 静的ﾒｿｯﾄﾞのｱﾄﾞﾚｽを、関数ﾎﾟｲﾝﾀに代入する。
  int value = ((&v)->*p_getter)(); // 関数ﾎﾟｲﾝﾀから呼び出す。

  return 0;
}
```

・静的ﾒｿｯﾄﾞの場合と異なるのは、次の**2**点です。↓

　・関数ﾎﾟｲﾝﾀ型の定義で、
　　ｸﾗｽ型の指定が必要であること。

　・呼び出しに、ｲﾝｽﾀﾝｽが必要なこと。

・それから、ｲﾝｽﾀﾝｽを、
　ﾋｰﾌﾟに確保する場合は、
　ﾒｿｯﾄﾞを呼ぶことはできるんですが、
　ﾃﾞｰﾀﾒﾝﾊﾞにｱｸｾｽしょうとすると、
　ｱｽｾｽ違反ｴﾗｰとなります。

protected継承

・ｸﾗｽを継承する際に、
　子ｸﾗｽ側のｱｸｾｽ指定子に「**protected**」を指定すると、
　親ｸﾗｽから継承した**public**ﾒﾝﾊﾞが、すべて**protected**ﾒﾝﾊﾞになります。↓

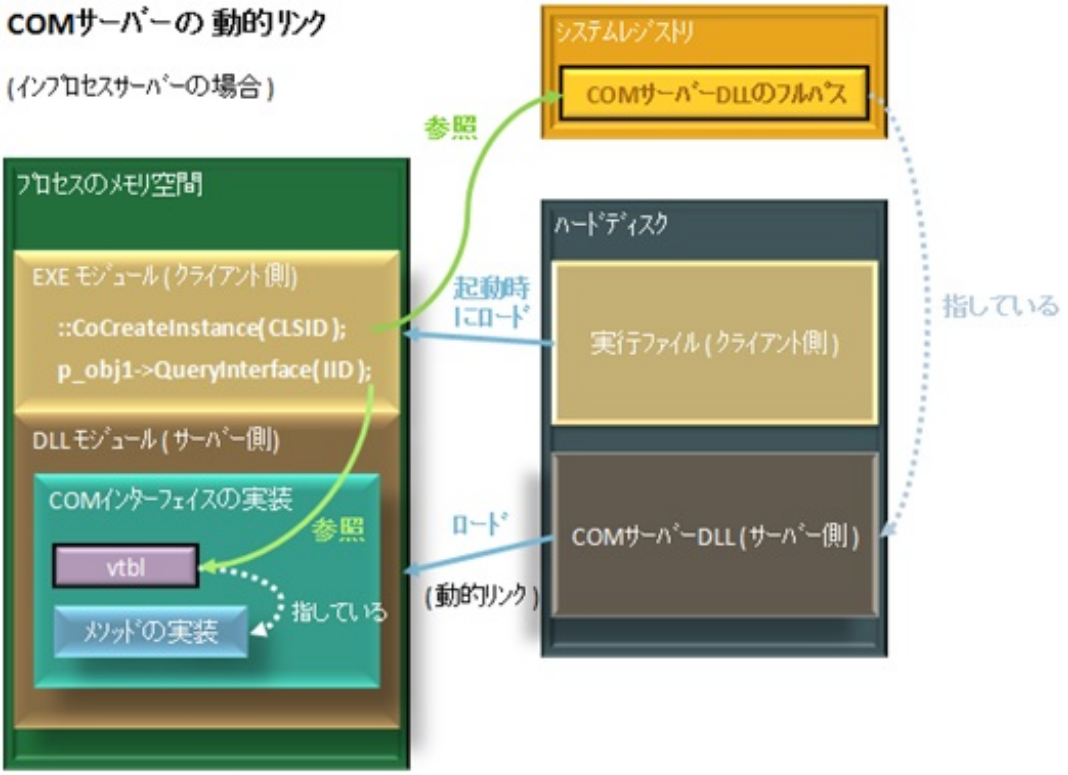
```
class Point2D
{

public:

  int x;
  int y;
};
```

```
class Point3D : protected Point3D
{
protected:

  int z;

public:

  int GetX(void) const { return x; };
  int GetY(void) const { return y; };
  int GetZ(void) const { return z; };

};
```

```
// あまりいい方法ではありませんが、こんなこともできます。↓
```

```
int main(void)
{

  Point2D* p_2d = new Point2D();

  p_2d->x = 100;
  p_2d->y = 200;

  Point3D* p_3d = (Point3D*) p_2d;

  int x = p_3d->GetX();
  int y = p_3d->GetY();

  return 0;
}
```

COMの仕組みを見てみよう



```
int main( void )
{
	// ----------------------
	 // COMｸﾗｽﾗｲﾌﾞﾗﾘの初期化を行う。

	::CoInitialize(NULL);
	// ・この関数は、ｽﾚｯﾄﾞごとに呼び出す必要がある。
	//
	// ・この関数を呼ぶことで、ｽﾚｯﾄﾞは、ｱﾊﾟｰﾄ(STA) に入り、
	//   ｵﾌﾞｼﾞｪｸﾄを操作できるようになる。
	//
	// ・MTA(STAとは別のｱﾊﾟｰﾄ)に入る場合は、
	//    CoInitializeEx関数を呼ぶ。

	// ----------------------
	// COMｵﾌﾞｼﾞｪｸﾄを作成する。( 成功すると、参照ｶｳﾝﾄが1になる。)

	IUnknown*p_obj1 = NULL;

	HRESULT hr = ::CoCreateInstance( CLSID_ComClassObjectA, NULL,
	                                 CLSCTX_INPROC_SERVER,
```

```
                                        IID_PPV_ARGS(  &p_obj1  ) );

if ( FAILED(hr) )
{
    ::CoUninitialize(void);// COMに関連するすべてのﾘｿｰｽを解放する。

    return 0;
}

//･ﾚｼﾞｽﾄﾘの中から、CLSIDに対応するCOMｺﾝﾎﾟｰﾈﾝﾄ
//   ( EXE or DLL ) のﾌﾙﾊﾟｽを取り出し、
//    ﾌﾟﾛｾｽのﾒﾓﾘ上か、独立したﾌﾟﾛｾｽのﾒﾓﾘ上に、ﾛｰﾄﾞする。
//
//･ｱｳﾄﾌﾟﾛｾｽｻｰﾊﾞｰの場合は、CLSCTX_LOCAL_SERVERを指定する。

// ----------------------
// ･このｸﾗｽｵﾌﾞｼﾞｪｸﾄが、IID のｲﾝﾀｰﾌｪｲｽを
//   実装しているかを確認し、あれば取得する。
//
//･成功すると、参照ｶｳﾝﾄが1つ増えて、2になる。

IComInterfaceA* p_interface1 = NULL;

hr = p_obj1->QueryInterface( IID_ComInterfaceA,
                             (void **) &p_interface1 );

if ( FAILED(hr) )
{
     p_obj1->Release(); // COMｸﾗｽｵﾌﾞｼﾞｪｸﾄを、解放する。

    ::CoUninitialize(void);// COMに関連するすべてのﾘｿｰｽを解放する。

    return 0;
}

// ----------------------
// COMｲﾝﾀｰﾌｪｲｽの、ﾒｿｯﾄﾞを呼び出す。

p_interface1->MethodA();

// ----------------------
// COMｲﾝﾀｰﾌｪｲｽを、解放する。

p_interface1->Release();

// この呼び出しで、参照ｶｳﾝﾄが、1になる。
```

```
        // ----------------------
        // COMｸﾗｽｵﾌﾞｼﾞｪｸﾄを、解放する。

        p_obj1->Release();

        // この呼び出しで、参照ｶｳﾝﾄが、0になり、
        //ｵﾌﾞｼﾞｪｸﾄが破棄される。

        // ----------------------
        // COMに関連するすべてのﾘｿｰｽを解放する。

         ::CoUninitialize(void);

        // ----------------------
          return 1;
}
```

COMｻｰﾊﾞｰ　(いわゆる COM ｺﾝﾎﾟｰﾈﾝﾄ)

|

+- **COMｸﾗｽｵﾌﾞｼﾞｪｸﾄ** (いわゆる ｵﾌﾞｼﾞｪｸﾄ。実体。)

　|

　+- **COMｲﾝﾀｰﾌｪｲｽ** (IUnknown ｲﾝﾀｰﾌｪｲｽの派生。)

　　|

　　+- ﾒｿｯﾄﾞ

・**CLSID** ... ｸﾗｽｵﾌﾞｼﾞｪｸﾄの**GUID**。

・**IID**　... ｲﾝﾀｰﾌｪｲｽの**GUID**。

「BSTR」

・「COM」では、「BSTR」という文字列型を使います。

・「BSTR」の正体は、「wchar_t*」なんですが、
最初の4byte(2文字分)に、文字数を記録しています。

(※この文字数には、終端のNULL文字は含まれていません。)

・「BSTR」は、必ず次のようにして使います。 ↓

```
BSTR bstr = ::SysAllocString( L"Hello" ); /* BSTRを生成する。 */

/* BSTRを使用する。 */

size_t length = ::SysStringLen( bstr ); /* 文字数を取り出す。 */

::SysFreeString( bstr ); /* BSTRを解放する。 */
```

・次のように自作することもできます。 ↓

```
/* BSTRを作成して返す。 */

wchar_t* CreateBstr( const wchar_t* p_src_text_ )
{

  size_t src_length = ::wcslen( p_src_text_ ); /* 文字数を求めておく。*/

  /* ｻｲｽﾞで 4byte使うんで 2 と、 NULL文字 ( 2byteなんで 1 ) を足して、 +3文字。 ↓ */
  wchar_t* p_bstr = ( wchar_t* ) ::malloc( sizeof( wchar_t ) * ( src_length + 3 ) );

  /* ﾊﾞｲﾄｻｲｽﾞを求める。 ( 2byte文字なので、2倍している。 ) */
```

```
    size_t src_size = src_length << 1;

    /* 先頭4ﾊﾞｲﾄはﾊﾞｲﾄ数。( 終端の0x0000は含まない。 ) */
    *((unsigned int*)p_bstr) = src_size;

    /* 文字列をｺﾋﾟｰする。*/
    ::memcpy( ((unsigned int*)p_bstr) + 1, p_src_text_, src_size );

    /* 終端2ﾊﾞｲﾄは0。( +2 は、ｻｲｽﾞの4byteを読み飛ばすため。 ) */
     p_bstr[ src_length + 2 ] = 0;

    return (p_bstr + 2); /* ｻｲｽﾞの4byte ( 2文字分 相当 ) を進めている。 */
}
```

```
/* BSTRの文字数を返す。 */

unsigned int GetBstrLength( const wchar_t* p_bstr_ )
{
    /* ｻｲｽﾞの4byte分を戻して、ｻｲｽﾞを取り出し、2で割っている。↓ */
    return ( *( ( (unsigned int*) p_bstr_ ) -1 ) >> 1 );
}
```

```
/* BSTRを解放する。 */

void DeleteBstr( wchar_t* p_bstr_ )
{
     /* ｻｲｽﾞの4byte ( 2文字分 相当 ) を引いている。 */
     ::free( ( (unsigned int*)p_bstr_ ) -1 );
}
```

・文字数は、4byte固定ですので、
「size_t」ではなく、「unsigned int」にしています。

ｺﾏﾝﾄﾞﾗｲﾝでﾋﾞﾙﾄﾞする

・「ｺﾏﾝﾄﾞﾌﾟﾛﾝﾌﾟﾄ」というのは、いわゆる「**DOS**画面」のことで、
真っ黒な画面に、文字を打ち込んで、ﾊﾟｿｺﾝを操作していくというものです。

・この節では、「**Visual Studio**」を起動せずに、
付属のｺﾏﾝﾄﾞﾗｲﾝ用のﾌﾟﾛｸﾞﾗﾑを実行することによって、
ｺﾝﾊﾟｲﾙとﾘﾝｸを行う方法を説明して行きます。

・ｺﾏﾝﾄﾞﾌﾟﾛﾝﾌﾟﾄの実行ﾌｧｲﾙは、「**cmd.exe**」という名前であり、
「C:¥Windows¥**System32**」ﾌｫﾙﾀﾞの中に置いてあります。

・このﾌﾟﾛｸﾞﾗﾑは、これからよく使いますので、
ｼｮｰﾄｶｯﾄを作成して、ﾀｽｸﾊﾞｰに置いておくと便利です。

・「**Visual Studio**」に付属しているｺﾝﾊﾟｲﾗの実行ﾌｧｲﾙは、
「**cl.exe**」という名前で、「**Visual Studio**」のﾌｫﾙﾀﾞ内にあります。

・ｺﾏﾝﾄﾞﾌﾟﾛﾝﾌﾟﾄが起動したら、まず最初に、
ｶﾚﾝﾄﾃﾞｨﾚｸﾄﾘを、ｺﾝﾊﾟｲﾗのあるﾌｫﾙﾀﾞに移動させます。↓

cd c:¥VisualStudio¥bin

・こうしておくと、実行ﾌｧｲﾙ名を入力するだけで、
上記ﾌｫﾙﾀﾞ内のﾌﾟﾛｸﾞﾗﾑを実行させることができます。↓

cl /c test.c /**Fo**C:¥test¥test.obj

・次に、同じﾌｫﾙﾀﾞ内にある「**vcvars32.bat**」という
ﾊﾞｯﾁﾌｧｲﾙを実行して下さい。↓

vcvars32.bat

・「**/c**」というのは、「ｺﾝﾊﾟｲﾙｵﾌﾟｼｮﾝ」の一つで、
C言語のｿｰｽｺｰﾄﾞをｺﾝﾊﾟｲﾙするよう指示するものです。

・次の「**/Fo**」は、ｺﾝﾊﾟｲﾗから出力される「ｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙ」の
出力先のﾌｫﾙﾀﾞﾊﾟｽを指定するためのものです。

（※ **/Fo** ｵﾌﾟｼｮﾝと、ﾊﾟｽとの間に、ｽﾍﾟｰｽがないことに注意して下さい。）

・さて、次に、ｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙをﾘﾝｸして、実行ﾌｧｲﾙを作成します。

link /**out:**c:¥test¥test.exe test.obj

・「**/out**」は、ﾘﾝｶのｵﾌﾟｼｮﾝで、出力される実行ﾌｧｲﾙの
出力先を指定するためのものです。

・ちなみに、ﾘﾝｶの実行ﾌｧｲﾙは、「**link.exe**」という名前で、
これもやはり同じﾌｫﾙﾀﾞ内にあるため、ﾊﾟｽは不要です。

・複数のｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙをﾘﾝｸする場合は、次のように書きます。↓

link /**out:**c:¥test¥test.exe test1.obj test2.obj test3.obj

・しかし、このようにｵﾌﾞｼﾞｪｸﾄﾌｧｲﾙが増えていくと、
ｺﾏﾝﾄﾞﾗｲﾝで入力するのが面倒になってきます。

・そうした場合には、上記のようなｺﾏﾝﾄﾞﾗｲﾝへの入力を、
「ﾒｲｸﾌｧｲﾙ」というﾃｷｽﾄﾌｧｲﾙに書いておいて、
ﾒｲｸを行うことで、一連の入力作業を省略することができます。↓

nmake -f makefile1.mak

・「**-f**」というのは、ﾒｲｸﾌｧｲﾙを指定するためのｵﾌﾟｼｮﾝです。

・ﾒｲｸﾌｧｲﾙの名前は何でもいいんですが、
ﾌｧｲﾙ拡張子は、**nmake** の場合は、「**mak**」にして下さい。

・ﾒｲｸﾌｧｲﾙの書き方については、次のﾍﾟｰｼﾞから説明して行きます。

奥付

「ﾊﾑ太と作るC++ライブラリ0 -構文編-」

http://p.booklog.jp/book/74354

著者：ﾈｲｷｯﾄﾞはむ太
著者プロフィール：http://p.booklog.jp/users/haseham/profile

図表の作成に使用したﾂｰﾙ: 「PowerPoint 2007」

感想はこちらのコメントへ
http://p.booklog.jp/book/74354

ブクログ本棚へ入れる
http://booklog.jp/item/3/74354

電子書籍プラットフォーム：ブクログのパブー（http://p.booklog.jp/）
運営会社：株式会社ブクログ