担当：富井尚志 (tommy@ynu.ac.jp)

# 第 2 回「基本的制御構造」の復習

## ☆ 選択計算(条件判断)

・ **if 文** : ある条件に応じて別々の計算を行う．式が真（0 以外）であれば文 1 が実行され，偽であれば文 2 が実行される．なお，else 以下は省略可能である．

```
if (式)
    文１
else
    文２
```

例）
```
if ( s1 = = s2 )
    printf("s1 と s2 は同じだよ¥n");
else
    printf("s1 と s2 は違うよ¥n");
```
“等しい”は==

## ☆ 反復計算

・ **while 文** : 式が真の時に文が実行される．これを式が偽になるまで実行する．

```
while (式)
    文
```

例）
```
s1=0;
i = n;
while(i>0) {
    s1 += i;      /* s1=s1+i; と等価 */
    i--;          /* i=i-1; と等価 */
}
```
1 から n までの和を求めている．

・ **for 文** : 最初に一度だけ式 1 が実行される．式 2 が真の時に文が実行される．その後に，式 3 が実行される．これを式 2 が偽になるまで繰り返す．

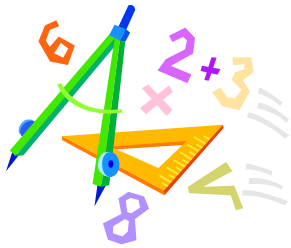
```
for(式 1; 式 2; 式 3)
    文
```

例）
```
s2=0;
for(i = n; i>0; i--) {
    s2 += i;
}
```
1 から n までの和を求めている．

・ **do ～ while 文** : まず文を実行する．次に式が計算される．これを式が偽になるまで繰り返す．

```
do
    文
while (式);
```

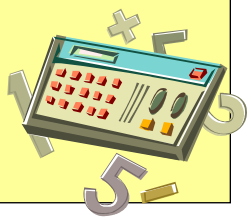
例）
```
s3=0;
i = n;
do {
    s3 += i;
    i--;
} while (i>0);
```
1 から n までの和を求めている．



## ◎ 関数定義の一般形:

```
得られる値の型
関数名(引数型 1 仮引数 1, 引数型 2 仮引数 2,... )
{
    変数型 1 変数 11,変数 12,...;
    変数型 2 変数 21,変数 22,...;

    文 1
    文 2
      :
    return 戻り値;
}
```



例)

```
int
summation( int n1, int n2 )
/* 整数 n1 から n2 までの和を */
/* 求めて返す関数          */
{
    int  i, sum = 0;

    for ( i = n1; i <= n2; i ++ ){
        sum = sum + i;
    }
    return sum;
}
```

## ◎ 関数呼出

・関数内部に記述された計算を行ない値を求める.

```
関数名(値 1,値 2,値 3,...)
```

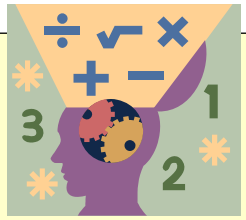
例)

```
int main(void)
{
    int n;
    ・・・・・・
    n = summation( 1, 100 );
    ・・・・・・
    return 0;
}
```

## ◎ 前回の出席票の小テストの解答

キーボードから１つの自然数を入力させ，その数の２乗を表示するプログラムを，関数を用いて作りなさい．ただし，入力された数が３の倍数であるときは再入力させること.

★次の不完全なプログラムを正しく動作するものに直すことによって回答せよ.




```
#include<stdio.h>

int func1( int n );
/* 引数の２乗を計算して返す関数（プロトタイプ宣言部） */

int main(void)
{
    int num;

    do {
        printf("input a number :" );
        scanf("%d",&num);
    } while ( num % 3 == 0 );
    printf("%d×%d = %d¥n", num, num, func1( num ) );
    return 0;
}

/* 関数 func1 の本体 */
int func1( int n )
{
    return n * n;
}
```

宣言部では，セミコロン（;）を最後に付けます.

10 進数フォーマット(%d)で, num という変数をキーボードから入力します．scanf は標準関数です．num の前に ＆ を忘れずに.

画面には例えば 10×10=100 などと表示されます(num=10 の場合).

引数(=n)を２乗した結果を返し（return）ます．例えば,

```
    int answer;

    answer = n * n;
    return answer;
```

などとしてももちろん可です.

第 3 回の始まり

# 変数のスコープルール・関数

## ☆ 有効範囲(スコープ)

- プログラミング言語においては，手続きや変数の有効範囲が厳密に決められる
- 大域的 ⟷ 局所的

## ◎ C における名前（関数，変数）の有効範囲の一般規則

「プログラムの字面の範囲」でみた有効範囲
^^^^ 「どのように関数呼出が組み合わされるか」という動的な振舞は無関係．
プログラムを見るだけでわかる．

- ブロック先頭で宣言される名前 →「内部変数」：ここで定義される変数

重要：ブロックの内部だけで有効　→ ブロック内は「外から見えない」
ブロックの内外に同じ変数があるときには，「内側の変数だけがみえる」

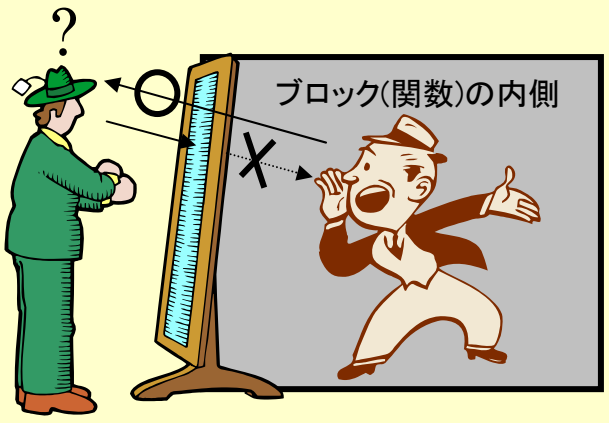
```
{宣言
   文 1
   文 2
   :
}
```

例）
```
int func1( int n1,int n2 )
{
     int a, b;
      ・・・
}
```

この関数の外からは代入や参照をすることが出来ない．

※ 関数の本体はブロック　→　関数の始めで定義される変数 は 関数内部だけで有効

ブロックは
内側から外は見えるが，外から内側が見えないようにする“ハーフミラー”




- **関数自身**：ソースファイル（プログラムを記述したファイル）内で宣言された点からそのファイルの終わりまで

  - C ではある関数で局所的に定義される関数はない
  - 宣言されていない関数は，int 型の関数として暗黙の宣言がなされる
    cf. 関数のプロトタイプ宣言
  - 関数の仮引数（パラメータ）は関数内部だけで有効

- 外部変数：どの関数にも属さない共通の変数．ソースファイル内で宣言された点からファイルの終わりまで．

## ◎ 自動変数(内部変数)と外部変数の違い　～「時間軸」で見た有効範囲の差 ～

- **自動変数**：関数の始め（ブロックの始め）で定義される変数．**通常その関数（ブロック）に制御が移るたびに生成され，関数（ブロック）を終了すると，「消滅」** (cf. static 宣言)
- **外部変数**：すべての関数の外で定義される変数．一回だけ生成され，「常に存在」

> ※ static 宣言(静的変数，静的関数)
> - 内部変数に対する宣言．ブロックを抜けても「**消滅せず，値を保持**」.
> - 外部変数，関数に対する宣言
> - 宣言以降に限定し，別ファイルからは見えなくなる．

## ◎ 宣言と定義

- 外部的な名前（変数や関数）に関しては区別される

> **宣言**：性質のみを述べる．「実体はなし」.
> **定義**：実体を伴う．「記憶割当」．一つの名前に対して一度だけ

※ 通常の外部変数の宣言では定義が伴う．宣言だけを行なう場合（複数のファイルに共通する変数など）は extern 宣言を行なう．

- 内部変数では定義と宣言が同一（定義という概念がない）

## ◎ 変数の初期化

宣言において変数の値を設定できる．

例）
```
int   a = 10;
float   f = 1.1;
```

---

## ☆ スコープの例 --- scope.c

```
1  /*****************************************************************
2        アルゴリズムとデータ構造
3        サンプルプログラム  scope.c
4        <<変数のスコープ>>
5        copyright (c) 1995,96,97    T.Mori <mori@forest.dnj.ynu.ac.jp>
6    *****************************************************************/
7  #include <stdio.h>
8
9  /*  外部変数の定義  */ /*  一番外側にある  */
10 int a = 3;
11 /*  関数の宣言  *//*  一番外側にある  */
12 int f( int x );
13 void f_auto(void);
14 void f_static(void);
15
16 /*  関数  main  の定義  *//*  一番外側にある  */
17 int main(void)
18 {
19     int b;
20
21     a = a + 1;
22     b = 3;
23     printf("main: a == %d, b == %d¥n", a, b);
24     { int a;   /*  自動変数  *//* main の中の一つのブロックにある  */
25
26       a = 2;
27       printf("main:block: a == %d, b == %d¥n", a, b);
28       printf("main:block: f(a) == %d¥n", f(a));
29     }
```

a = 4 になった.

これらは異なる変数！

```
1       printf("main: a == %d, b == %d¥n", a, b);
2
3       /*  自動変数と静的変数の違い  */
4       f_auto();
5       f_auto();
6       f_static();
7       f_static();
8       return 0;
9   }
10
11  /*  関数  f  の定義  */
12  int f( int b ) /*  仮引数  b  は  関数 f の中で新たに宣言  */
13  {
14      printf("f: a == %d, b == %d¥n", a, b);      /* a  は外部変数の方  */
15      b = b + a;
16      a = a + 5;
17      printf("f: a == %d, b == %d¥n", a, b);
18
19      return b;   /* b の値を関数の値として返す  */
20  }
21
22
23  /*  自動変数を内部に持つ関数  */
24  void f_auto(void)
25  {
26      int x = 0; /*  自動変数  x  の初期化  */
27
28      x = x + 1;
29      printf("f_auto: x == %d¥n", x);
30  }
31
32  /*  静的変数を内部に持つ関数  */
33  void f_static(void)
34  {
35      static int x = 0; /*  静的変数  x  の初期化  */
36
37      x = x + 1;
38      printf("f_static: x == %d¥n", x);
39  }
40
41  【実行結果】
42  main: a ==      , b ==
43  main:block: a ==      , b ==
44  f: a ==      , b ==
45  f: a ==      , b ==
46  main:block: f(a) ==
47  main: a ==      , b ==
48  f_auto: x ==
49  f_auto: x ==
50  f_static: x ==
51  f_static: x ==
```

第 52～55 行の結果に対応している.

この代入文で外部変数の値が変わる！

この関数が呼ばれるたびに初期化される！

だからここでは毎回 f_auto:x == 1 と表示される.

この関数が初めて呼ばれたときだけ初期化される.（static）

前回呼ばれたときのこの変数 x の値に 1 が加わる. だから 2 回目に呼ばれたときは 2 になる.

**自分で考えて，書き入れて下さい.**

次の図は scope.c の変数のスコープが分るように書いた図です．参考にして下さい．

```
変数宣言　a

関数　ｍａｉｎ　()
    変数宣言　b

    ａ　＝　ａ　＋　１；
    ｂ　＝　３；
      変数宣言　a

      ａ　＝　２；
      ｐｒｉｎｔｆ（...,　ａ，ｂ）；
      ｐｒｉｎｔｆ（...,　ｆ（ａ））；
    ｐｒｉｎｔｆ（...,　ａ，　ｂ）；
    ｆ＿ａｕｔｏ（）；
    ｆ＿ａｕｔｏ（）；
    ｆ＿ｓｔａｔｉｃ（）；
    ｆ＿ｓｔａｔｉｃ（）；

関数　ｆ　（変数宣言　ｂ）
    ｐｒｉｎｔｆ（...,　ａ，　ｂ）；
    ｂ　＝　ｂ　＋　ａ；
    ａ　＝　ａ　＋　５；
    ｐｒｉｎｔｆ（...,　ａ，　ｂ）；

関数　ｆ＿ａｕｔｏ　()
  変数宣言　ｘ　＝　０；

  ｘ　＝　ｘ＋１；
  ｐｒｉｎｔｆ（...,　ｘ）；

関数　ｆ＿ｓｔａｔｉｃ　()
  変数宣言（静的）　ｘ　＝　０；

  ｘ　＝　ｘ＋１；
  ｐｒｉｎｔｆ（...,　ｘ）；
```

## ☆ 再帰呼び出し

- ある手続き/関数の中から自分自身を呼び出す．
- 自動変数があるので可能．呼出のたびに別の領域が変数に割り当てられる．
- 数学的帰納法などにより解を求める場合などに使う

## ☆ 例 --- fact.c

- 再帰を使って階乗を求める．(再帰に関するアルゴリズムはあらためて取り扱う)

```
/*****************************************************************

      アルゴリズムとデータ構造
      サンプルプログラム　fact.c
      <<簡単な再帰呼出>>

      copyright (c) 1995,96,97　T.Mori <mori@forest.dnj.ynu.ac.jp>
 *****************************************************************/
#include <stdio.h>

int fact( int n );

int main(void)
{
   int k=3;

   printf("fact(%d) == %d¥n", k, fact(k) );
   return 0;
}

/* 階乗を求める */
int fact(int n)
{
   int fn;

   printf("Call:    fact(%d)¥n",n);

   if (n>0)
      fn = n*fact(n-1);
   else
      fn = 1;

   printf("Return: fact(%d) == %d¥n",n,fn);
   return fn;
}
```

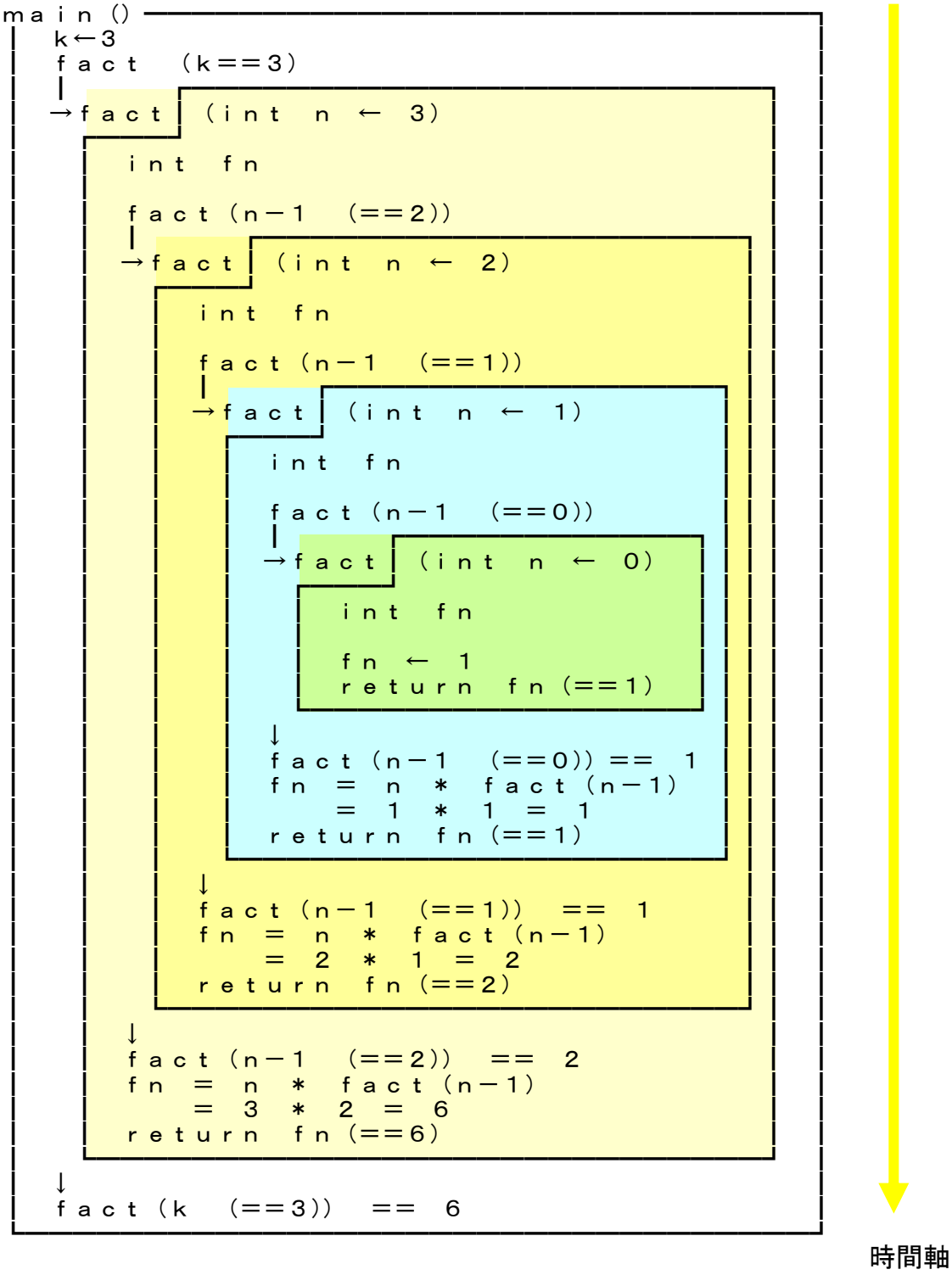
fact( 3 ) で関数 fact に行く．すなわち引数 n=3 である．n > 0 なので，fact( 3 – 1 )すなわち fact(2)が実行される．
まだ n > 0 なので fact(1)が実行される．
まだ n > 0 なので fact(0)が実行される．
このときは n > 0 ではないので，fn = 1 を戻す．いもづる式に 3 × 2 × 1 をしたことになる．

fn = 3 * fact( 2 )
　= 3 * ( 2 * fact( 1 ) )
　= 3 * ( 2 * ( 1 * fact( 0 ) )
　= 3 * ( 2 * ( 1 * 1 ) )　　∵ fact(0) = 1
　= 3 * ( 2 * 1 )
　= 3 * 2 * 1
　= 6

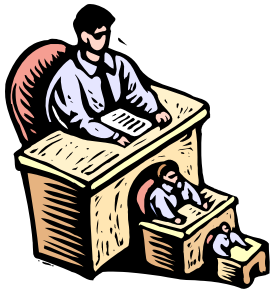
Call:　fact(　　)
Call:　fact(　　)
Call:　fact(　　)
Call:　fact(　　)
Return: fact(　　) ==　　
Return: fact(　　) ==　　
Return: fact(　　) ==　　
Return: fact(　　) ==　　
fact(　　) ==　　

**自分で考えて，書き入れて下さい．**

注意：　下の図の枠は先ほどのブロックの入れ子と違うので注意. 関数の実行順序に従って変数の値を示したものである. 自動変数の「時間軸」でみた有効範囲が理解の鍵である.




再帰呼び出しは，慣れるまではなかなか理解できずに，頭がパニックになってしまうかも知れません．最初は誰でもそうなのです．
何回か使ううちに，使い方がわかるようになります．習うより慣れろです．



# C言語のプログラムって，大体こんな感じです．．．

```
#include<stdio.h>
#include<math.h>
・・・・・・
```

必要なヘッダファイル（.h で終わるファイル）のインクルードなど．

```
int number;
・・・・・・
```

外部変数の宣言（常に存在し，どこからでも参照・代入することができる）

```
int func1( int num );
float func2( int n1, double d1 );
・・・・・・
```

関数のプロトタイプ宣言（ここで用いる関数のリスト．関数の冒頭部分に；をつけたもの）

main 関数

プログラム中に 1 つだけある主関数

```
int main(void)
{
        int a, b;
        float f1;
        double d1;
        ・・・・・
        a = func1( b );
        f1 = func2( a, d1 );
        ・・・・・・
        return 0;
}
```

これらの変数はその関数内でしか使えない変数．

```
int func1( int num )
{
        int p1,p2;
        ・・・・・・
        return p1;
}
```

1 番目の関数

```
float func2( int n1, double d1 )
{
        float c1;
        ・・・・・・
        return c1;
}
```

2 番目の関数

3 番目の関数

・
・
・
・
・

必要な関数が必要な数だけ並んでいる．

プログラムを書くときに，**どのような関数を使えば良いのかは，対象となる問題に応じてプログラマが考えることです**．慣れれば，どのような部分を関数にすれば**構造的に分かり易い**プログラムになるかわかるようになります．とにかく，**たくさんプログラムを書いて下さい**．プログラミングは語学に似ています．色々な表現が身につけば，話すのも楽に成ります．それにはとにかく場数を踏むことです．

**関数はこうしなければならない**のではなく，**こんな風にすることもできる**，と考えましょう．要は道具の一つと思えばよいのです．

**演習問題１**（この中からランダムに出題しますのでワークシートに回答して下さい）

１－１　２つの引数 n1，n2（ともに int 型）を与えると，それらの値の差（正の数）を返す関数 dif を絶対値関数 abs は使わないで作りなさい（ヒント：なし）

１－２　任意の正の整数 n(int 型)を与えると，フィボナッチ数 $F_n$ を返す関する fibo を再帰を用いて作りなさい．
（ヒント：フィボナッチ数 $F_n$ とは，フィボナッチ数列を作る数であり，n=1 または n=2 のとき $F_n$=1，n が 3 以上のとき $F_n = F_{n-1} + F_{n-2}$ で定義される数である）

１－３　1-2 のプログラムを使って，n=1 から n=10 までのフィボナッチ数 $F_n$ を画面に表示するプログラムを作りなさい．（ヒント：なし）

１－４　乱数で 100 以下の正の整数を発生させて，それを当てる“数当てゲーム”を作りなさい．例えば実行時は次のような感じです．

```
number = 40
小さいよ
number = 80
大きいよ
・・・・・
number = 55
当たり！
```

（ヒント：stdlib.h と time.h をインクルードすると，次のようにして，int 型変数 ans に 100 以下の正の整数がランダムに入力することができます．

```
int ans;

srand(time(NULL));
ans = (int)(100.0 * rand() / 32768.0);
```

次回「配列を扱うアルゴリズムの基礎(1)．最大値，最小値」