プログラミング１　第11回

関数の復習

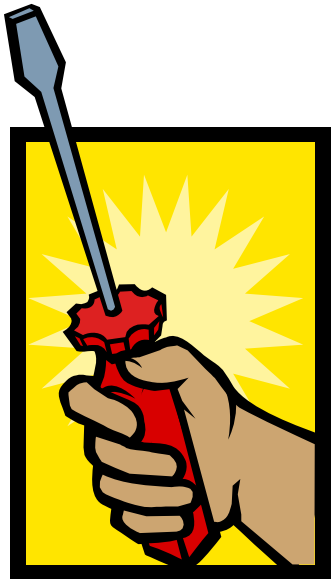
- 関数とは？
- 引数
- 戻り値
- 再帰

ここにあるサンプルプログラムは
/home/course/prog1/public_html/2007/HW/lec/sources/
下に置いてありますから、各自自分のディレクトリにコピーして、コンパイル・実行してみてください

関数とは？

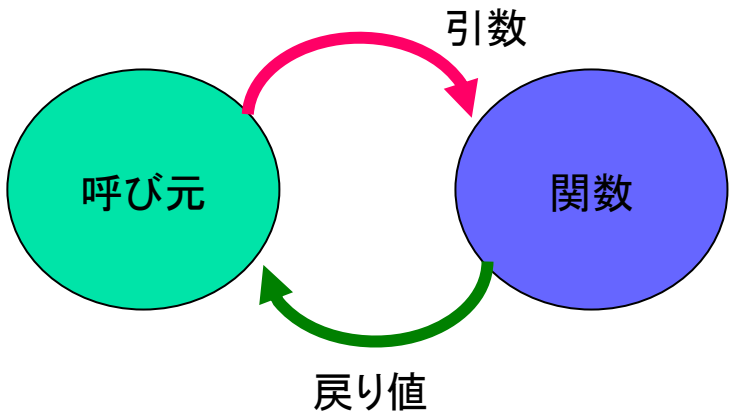
- 何度も同じ処理をする必要がある時に、その部分を独立したプログラムにする。
- それを何度も呼ぶことでプログラムを見やすくする（見やすいプログラムはバグが少ない）
- 関数はプログラミング言語によっては
  - プロシジャ(procedure)、手続き
  - サブルーチン(subroutine)

  などとも呼ばれる
- CやFortranは関数（手続き）を呼ぶことでプログラムするので、「手続き型言語」と呼ばれる



関数のまとめ

- 引数が仮引数にコピーされる
- 関数内で仮引数の値に従って計算（処理）
- 戻り値が関数の値となる
- mainの前にプロトタイプ宣言
- mainの後ろに関数本体の宣言




```
#include <stdio.h>

float nijou(float);

main()
{
  float a = 1.73 , b , c ;
  b = nijou(a);
  c = nijou(1.41);
  ...（以下略）
}

float nijou( float x )
{
  float y;
  y = x * x;
  return y ;
}
```

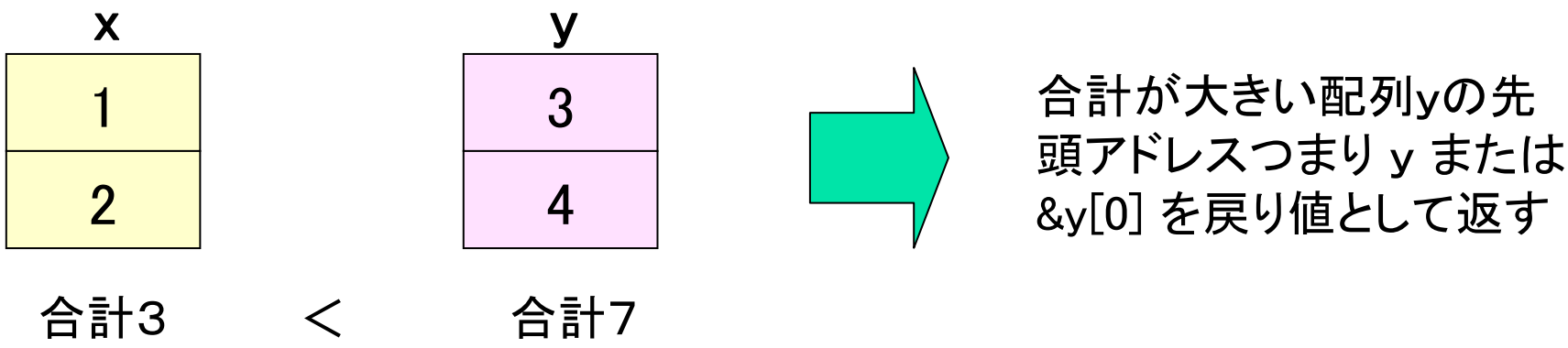
引数(ひきすう)と戻り値

- 引数の種類
  - 普通の型 (**int , float** など)
  - 構造体 (**struct xy** など)
  - ポインタ (**int *, struct adr *** など)
  - 配列 (**int []** など):実際にはポインタと同じ
  - 自分で**typedef**した型 (**Height** など)

- 戻り値の種類
  - 普通の型 (**int , float** など)
  - 構造体 (**struct xy** など)
  - ポインタ (**int *, struct adr *** など)
  - 自分で**typedef**した型 (**Height** など)

サンプル1

- 要素の和の大きい方（つまり**平均の大きい方**）の配列の先頭アドレスを返す関数



配列(ポインタ)を引数 ポインタを戻り値

- 関数は
  `int *max(int *x, int *y)`
  とも書けるし
  `int *max(int x[N], int y[N])`
  のように要素数を指定しても良い
  (但し要素数は実質的には何の意味もない)
- いずれにしても関数の仮引数x、yはポインタである

```
実行結果:
s1000001{std0ss0}1:  ./a.out
a[0] = 1
a[1] = 2
b[0] = 3
b[1] = 4
p[0] = 3
p[1] = 4
s1000001{std0ss0}2:
```

```
#include<stdio.h>
#define N 2
int *max(int[] , int[]);
main(){
  int a[N] = {1, 2}, b[N] = {3, 4}, i, *p;

  for(i = 0 ; i < N ; i++)
    printf("a[%d] = %d\n",i,a[i]);
  for(i = 0 ; i < N ; i++)
    printf("b[%d] = %d\n",i,b[i]);
  p = max(a, b);
  for(i = 0 ; i < N ; i++)
    printf("p[%d] = %d\n",i,p[i]);
}
int *max(int x[], int y[]){
  int i, sumx = 0, sumy = 0;
  for(i = 0 ; i < N ; i++){
    sumx += x[i]; /* sumx+= *(x + i)でも可 */
    sumy += y[i];
  }
  if (sumx > sumy)
    return x;
  else return y;
}
```

問題

- プログラムを少し変えてみた。
- このプログラムをコンパイルすると警告(warning)が出る。
- とりあえず実行は出来たが、結果がおかしい。
- このプログラムのどこがいけないのか？

```
s1000001{std0ss0}1: gcc lec12-2.c
warning: function returns address of
local variable
s1000001{std0ss0}2: ./a.out
a[0] = 1
a[1] = 2
b[0] = 3
b[1] = 4
p[0] = 3
p[1] = 67404
s1000001{std0ss0}3:
```

無理やり実行すると

```
#include<stdio.h>
#define N 2
int *max(int * , int *);
main(){
  int a[N] = {1, 2}, b[N] = {3, 4}, i, *p;

  for(i = 0 ; i < N ; i++)
    printf("a[%d] = %d\n",i,a[i]);
  for(i = 0 ; i < N ; i++)
    printf("b[%d] = %d\n",i,b[i]);
  p = max(a, b);
  for(i = 0 ; i < N ; i++)
    printf("p[%d] = %d\n",i,p[i]);
}

int *max(int *x, int *y){
  int i, sumx = 0, sumy = 0 , z[N];
  for(i = 0 ; i < N ; i++){
    sumx += x[i];
    sumy += y[i];
  }
  if (sumx > sumy)
    for(i = 0 ; i < N ; i++) z[i] = x[i];
  else
    for(i = 0 ; i < N ; i++) z[i] = y[i];
  return z;
}
```

問題の答え

- 関数maxは関数内の配列zの先頭アドレスをリターンしている
- ところが変数zは「自動変数」で、関数終了と共に生存期間が終了する。
- つまり関数maxは既に存在していない自動変数（又はローカル変数、局所変数）のアドレスをリターンしている点がまずい。
- よく間違うのでまとめておく。
  - ローカル変数の値自体をリターンする ➡ ○
  - ローカル変数のアドレスをリターンする ➡ ×

```
int *max(int *x, int *y)
{
  int i, sumx = 0, sumy = 0 , z[2];
  for(i = 0 ; i < 2 ; i++){
    sumx += x[i];
    sumy += y[i];
  }
  if (sumx > sumy)
    for(i = 0 ; i < 2 ; i++) z[i] = x[i];
  else
    for(i = 0 ; i < 2 ; i++) z[i] = y[i];
  return z;
}
```

zをリターンするのがまずい！

配列の要素数を渡す

- 全部の要素を使用しない場合の配列の要素数の渡し方2例

引数として渡す方法

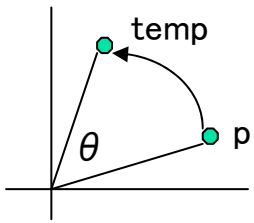
```
#include<stdio.h>
#define N 1000

int total(int *, int);

main(){
  int data[N], i, t, n;

  for(n = 0 ; n < N ; n++)
    if(scanf("%d",&data[n]) != 1) break;

  t = total(data, n);
  printf("total = %d¥n",t);
}

int total(int *a, int n){
  int i, sum = 0;
  for(i = 0 ; i < n ; i++){
    sum += a[i];
  }
  return sum;
}
```

外部変数として渡す方法

```
#include<stdio.h>
#define N 1000

int total(int *);
int n;

main(){
  int data[N], i, t;

  for(n = 0 ; n < N ; n++)
    if(scanf("%d",&data[n]) != 1) break;

  t = total(data);
  printf("total = %d¥n",t);
}

int total(int *a){
  int i, sum = 0;
  for(i = 0 ; i < n ; i++){
    sum += a[i];
  }
  return sum;
}
```

構造体を引数
構造体を戻り値

- ２次元の座標を回転させるプログラム
- 平面の点を表す構造体と角度を渡し、戻り値として回転後の点の構造体を受け取る



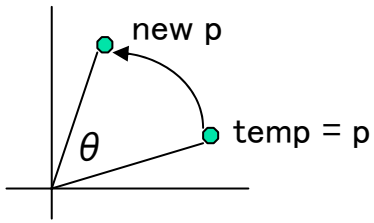

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159265358979323846  /* PIの定義 */

struct xy {
  double  x; /* x座標 */
  double  y; /* y座標 */
};

struct xy conv(double, struct xy); /* 関数プロトタイプ */

main(){
  struct xy p1 = {1.0,0.0} ,p2;

  printf("BEFORE  : (%f, %f)＼n",p1.x,p1.y);
  p2 = conv(PI/4.0, p1);   /* 関数の呼出(角度 45度) */
  printf("ROTATED : (%f, %f)＼n",p2.x,p2.y);
}

struct xy conv(double theta, struct xy p){
  struct xy temp;/* ワーク用構造体 */
  /* 座標変換 */
  temp.x = p.x * cos(theta) + p.y * (-sin(theta));
  temp.y = p.x * sin(theta) + p.y * cos(theta);
  return temp;
}
```

角度

構造体ポインタを引数

- 同じプログラムを構造体ポインタで作成
- 構造体ポインタと角度を渡し、構造体ポインタで指し示されている場所の値を直接更新する

- 問題:
  なぜtempにコピーする必要があるのか？？




```
#include <stdio.h>
#include <math.h>
#define PI 3.14159265358979323846  /* PIの定義 */

struct xy {
  double  x; /* x座標 */
  double  y; /* y座標 */
};

void conv(double, struct xy *); /* 関数プロトタイプ */

main(){
  struct xy p1 = {1.0,0.0};

  printf("BEFORE  : (%f, %f)＼n",p1.x,p1.y);
  conv(PI/4.0 , &p1);   /* 関数の呼出(角度 45度) */
  printf("ROTATED : (%f, %f)＼n",p1.x,p1.y);
}

void conv(double theta , struct xy *p){
  struct xy temp;       /* ワーク用構造体 */
  temp = *p;            /* tempに引数の構造体をコピー */
  /* 座標変換 */
  p->x = temp.x * cos(theta) + temp.y * (-sin(theta));
  p->y = temp.x * sin(theta) + temp.y * cos(theta);
}
```

問題の答え

- 問題:
  なぜtempにコピーする必要があるのか？？
- 答え
  下のソースのようにtempを使わない場合、x座標の計算は正しく出来るが、y座標の計算は誤った答えとなる。

  なぜなら、①の計算をした時点で、x座標(p->x)の値が更新されてしまうので、②の計算では更新後のp->xを基に計算してしまうからである。

```
void conv(double theta , struct xy *p)
{
  /* 座標変換 */
  p->x = p->x * cos(theta) + p->y * (-sin(theta)); ①
  p->y = p->x * sin(theta) + p->y * cos(theta);    ②
}
```

既に更新されている

文字列の比較

- 関数mystrcmpは文字列を2つ受け取りアスキーコードで比較する
  - 全く同じなら0を
  - 前者の方が大きければ1を
  - 後者の方が大きければ-1を返す
  - **"abc"と"abd"、"abcd"と"abd"、"abcd"と"abce"、"Abc"と"abc"**はどちらも後者が「大きい」

- main部分
  - 2つの文字列をscanfで読み込む
  - 関数の戻り値によってどちらの文字列が長いか判定の表示をする

```
#include<stdio.h>
#include<stdlib.h>
#define LEN 100

int mystrcmp(char *, char *);

main(){
  char a[LEN], b[LEN];
  int status;
  while(1){
    status = scanf("%s%s",a,b);
    if(status != 2) break;
    switch (mystrcmp(a, b)){
    case -1:
      printf("%s, %s : latter is larger\n",a, b);
      break;
    case 0:
      printf("%s, %s : both are equal\n",a, b);
      break;
    case 1:
      printf("%s, %s : latter is smaller\n",a, b);
      break;
    default:
      printf("%s, %s : error\n",a, b);
      exit(1);
    }
  }
}
```

1/2

ポインタを引数、整数を戻り値

```
int mystrcmp(char *s1, char *s2){
  while (*s1 == *s2){
    if(*s1 == '\0') return 0;
    s1++;
    s2++;
  }
  if (*s1 < *s2) return -1;
  else return 1;
}
```

ヌルが来るまで全ての文字が等しければ0をリターン

アスキーコードでの大小比較

2/2

ポインタを引数 ポインタを戻り値

- 小さい方の文字列のアドレスを戻り値として返す（文字列の大小はmystrcmpと同じ方式とする）
- なお、両方の文字列が同じである時は、戻り値としてNULLを返す。

```
#include<stdio.h>
#define LEN 100

char *strsmaller(char *, char *);

main(){
  char a[LEN], b[LEN], *str;
  int status;
  while(1){
    status = scanf("%s%s",a,b);
    if(status != 2) break;
    str = strsmaller(a, b);
    if(str == NULL) printf("%s, %s : both are same\n",a,b);
    else printf("%s, %s : smaller string is %s\n", a, b, str);
  }
}

char *strsmaller(char *s1, char *s2) {
  int i;

  for(i = 0; s1[i] == s2[i]; i++){
    if(s1[i] == '\0') return NULL;
  }
  if (s1[i] < s2[i]) return s1;
  else return s2;
}
```

外部変数の利用(1)

- この例は引数でデータを受け渡す、サンプルである。
- mainでデータ個数を入力し、データ個数分配列を用意する。
- 関数inputdataでは配列にデータを入力する。この際配列と要素数は引数として受け取る
- 関数outputdataはデータを表示する。関数inputdataと同様に、配列と要素数は引数として受け取る

```
#include<stdio.h>

void inputdata(int *, int);
void outputdata(int *, int);
main(){
  int *data, numData;

  printf("Enter number -> ");
  scanf("%d",&numData);
  data = malloc(numData * sizeof(int));

  inputdata(data,numData);
  outputdata(data,numData);
}
void inputdata(int *dt, int num){
  int i;
  for(i = 0 ; i < num ; i++){
    scanf("%d",&dt[i]);
  }
}
void outputdata(int *dt, int num){
  int i;
  for(i = 0 ; i < num ; i++){
    printf("data[%d] : %d¥n",i,dt[i]);
  }
}
```

外部変数の利用(2)

- 配列のアドレス(data)と配列の大きさ(numData)を外部変数(グローバル変数)とする
- 各関数で、dataとnumDataを直接使用することが可能である
- 外部変数に関してはハンドアウトLec02-6を参照のこと

```
#include<stdio.h>

void inputdata(void);
void outputdata(void);
int *data, numData;
main(){
  printf("Enter number -> ");
  scanf("%d",&numData);
  data = malloc(numData * sizeof(int));

  inputdata(data);
  outputdata(data);
}
void inputdata(void){
  int i;
  for(i = 0 ; i < numData ; i++){
    scanf("%d",&data[i]);
  }
}
void outputdata(void){
  int i;
  for(i = 0 ; i < numData ; i++){
    printf("data[%d] : %d\n",i,data[i]);
  }
}
```

複数の戻り値を得たい場合

- 以下の3つを場合によって使い分けると良い
  - 戻り値に構造体を使用(10ページ)
    - 関連するデータを構造体にまとめることで、分かりやすいプログラムに出来る(ただし、無理にまとめるぐらいなら他の方法を考えよう)
  - 引数をポインタとし、ポインタの指し示す変数を直接書き換える(11ページ)
    - 一般的に配列の受け渡しに良く利用される
    - 引数とは違うデータ型を戻したい場合、引数を変更したくない場合は別途戻り値用の引数を利用する
  - 外部変数を使用する(16～17ページ)
    - 複数の関数で共用する変数を更新する時に便利
    - ローカル変数との混同などでバグが出やすく、保守が大変となるので多用は勧められない。

再帰(1)

- 例えば「階乗」の漸化式は(n >= 1の時)

  n > 1 の時　n! = n * (n - 1)!

  n = 1 の時　n! = 1

- 5! の値を計算してみる

```
5! = 5 * 4!
   = 5 * (4 * 3!) = 5 * 4 * 3!
   = 5 * 4 * (3 * 2!) = 5 * 4 * 3 * 2!
   = 5 * 4 * 3 * (2 * 1!) = 5 * 4 * 3 * 2 * 1!
   = 5 * 4 * 3 * 1 = 60
```

再帰(2)

- Cプログラムでの「再帰」とは、関数中から同じ関数を呼び出すこと
- 前頁の階乗の漸化式は

  n > 1 の時　n! = n * (n - 1)!

  n = 1 の時　n! = 1
- 階乗を計算する関数factorial(n)の定義は以下のようになる

  factorial(n) = n * factorial(n - 1)　（n > 1の時）

  factorial(n) = 1　（n = 1の時）
- 関数は以下のようになる

```
int factorial(int n)
{
  if(n == 1) return 1; /* factorial(1) */
  else return factorial(n - 1) * n;
}
```

漸化式の部分

再帰(3)

- 分かりやすいように、関数factorial()にprintf文を加えてみた

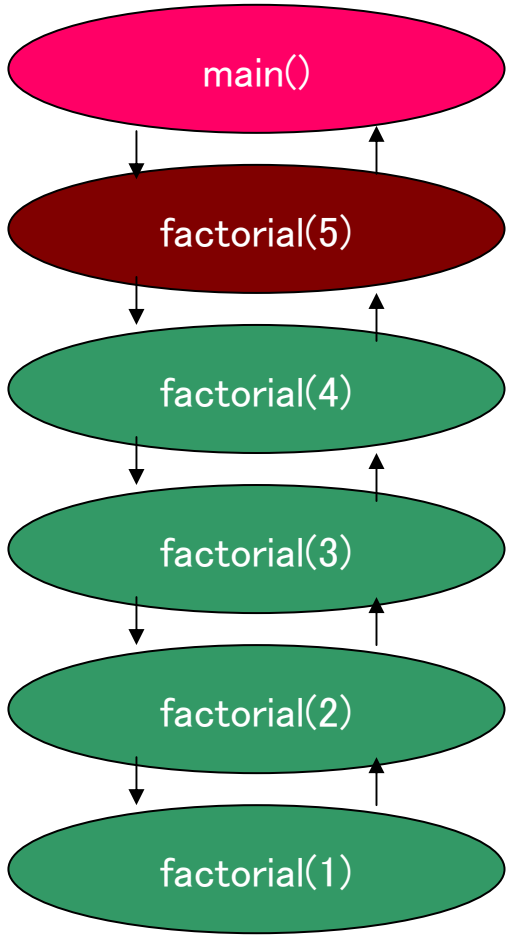
```
s1000001{std0ss0}1: ./a.out
Now in main
factorial(5) called
factorial(4) called
factorial(3) called
factorial(2) called
factorial(1) called
factorial(1) : 1
factorial(2) : 2
factorial(3) : 6
factorial(4) : 24
factorial(5) : 120
Now back in main
5! = 120
s1000001{std0ss0}1:
```

```
#include<stdio.h>

int factorial(int);

main()
{
  int ans, n = 5;
  printf("Now in main¥n");
  ans = factorial(n);
  printf("Now back in main¥n");
  printf("%d! = %d¥n",n,ans);
}

int factorial(int n)
{
  int fact;

  printf("factorial(%d) called¥n",n);
  if(n <= 1) fact = 1;
  else fact = factorial(n - 1) * n;

  printf("factorial(%d) : %d¥n",n,fact);
  return fact;
}
```

再帰

再帰(4)

呼ばれる順番



計算・終了する順番

(5)
(4)
(3)
(2)
(1)

```
s1000001{std0ss0}1: ./a.out
Now in main
factorial(5) called ①
factorial(4) called ②
factorial(3) called ③
factorial(2) called ④
factorial(1) called ⑤
factorial(1) : 1    (1)
factorial(2) : 2    (2)
factorial(3) : 6    (3)
factorial(4) : 24   (4)
factorial(5) : 120  (5)
Now back in main
5! = 120
s1000001{std0ss0}1:
```

再帰を使用する際の注意

- 再帰では関数呼び出しの思わぬ無限ループに陥りやすいので、注意する
- 例えば関数factorialで、

```
if(n <= 1) return 1;
```

の条件がなく、ただ

```
return factorial(n - 1) * n;
```

だとすると無限ループとなってしまう。

再帰を使うべきかどうかを良く考えてから使用する

- 再帰は漸化式をそのままプログラミング出来、直感的に理解しやすいプログラムを作成できるが、通常のループによる計算に比べて計算機資源（メモリなど）や時間を多量に消費する場合が多い
- １０！の計算を再帰とループ両方の方法で100万回計算して実行時間を比較するプログラムが以下の場所にあるので、興味のある人は実行してみると良い

```
/home/course/prog1/public_html/2007/HW/lec/sources/lec11-9b.c
```