プログラミング１　第６回
ポインタ(3) -- ポインタの応用

- 関数の引数
- 配列を引数にする(前期教科書P241)
- main関数への引数(後期教科書P136)
- 動的メモリ割り当て(後期教科書P133)

この資料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

scanfの引数

- scanfにおいて変数iにつく&は「おまじない」として取り扱ってきた。又文字列の場合のみは配列名を書き、＆はつけないことになっていた。

```
scanf("%d",&i);
scanf("%s",str);
```

- また、これまで学んだことで以下の事が分かった。
  - 変数に&を付けるとその変数の「アドレス」になる
  - 配列名はその配列の先頭要素の「アドレス」である
- つまり&iは変数iのアドレスを、strは配列（文字列）strの先頭アドレスを表している
- 実はscanfの書式の後の引数は読み込む変数の「アドレス」を指定することになっている。通常の変数には＆をつけ、文字列（配列）の場合は＆をつけないのはこのためだった。

関数にアドレスを渡すとは？

- さて、それでは、scanfのように関数にアドレスを渡す意味は何だろうか？

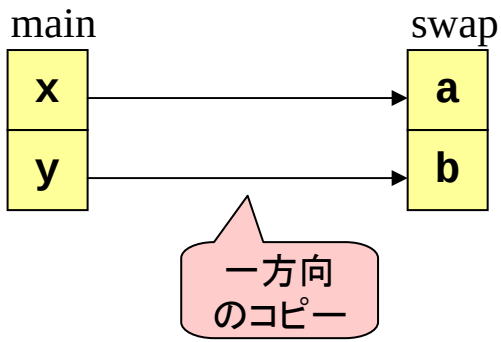
値渡し

- 通常関数を呼ぶ場合は引数の値が関数に渡される
- これを「値渡し」と呼ぶ
- 呼ぶ側の引数の値は決して変更されない

```
#include<stdio.h>
void swap(int, int);
main()
{
  int x = 5, y = 3;

  swap(x, y);
  printf("x=%d¥ty=%d¥n", x, y);
}

void swap(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
  printf("a=%d¥tb=%d¥n", a, b);
}
```

```
実行結果
s1000001{std1ss1}1: ./a.out
a=3       b=5
x=5       y=3
s1000001{std1ss1}1:
```

関数内では入れ替わったが、呼ぶ側の変数の値は入れ替わらない！



アドレス渡し

- アドレスを引数として関数に渡すことも可能である
- これを「アドレス渡し」と呼ぶ
- 呼ぶ側の変数の値はアドレスを介して関数側から変更出来る

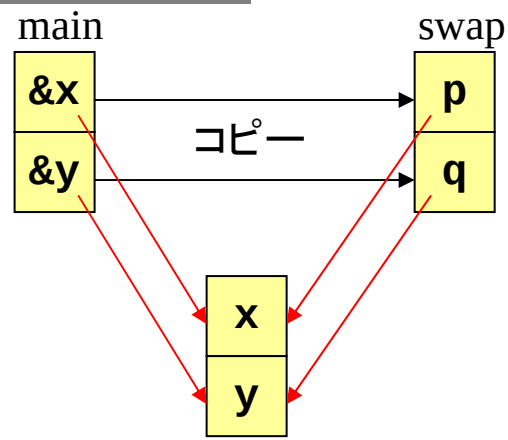
注：Cの場合、アドレスを「値渡し」するので、他言語のような「参照渡し」ではない

```
#include<stdio.h>
void swap(int *, int *);
main()
{
  int x = 5, y = 3;

  swap(&x, &y);
  printf("x=%d¥ty=%d¥n", x, y);
}

void swap(int *p, int *q)
{
  int temp;
  temp = *p;
  *p = *q;
  *q = temp;
  printf("*p=%d¥t*q=%d¥n",*p,*q);
}
```

関数の引数はポインタ

```
実行結果
s1000001{std1ss1}1: ./a.out
*p=3     *q=5
x=3      y=5
s1000001{std1ss1}2:
```

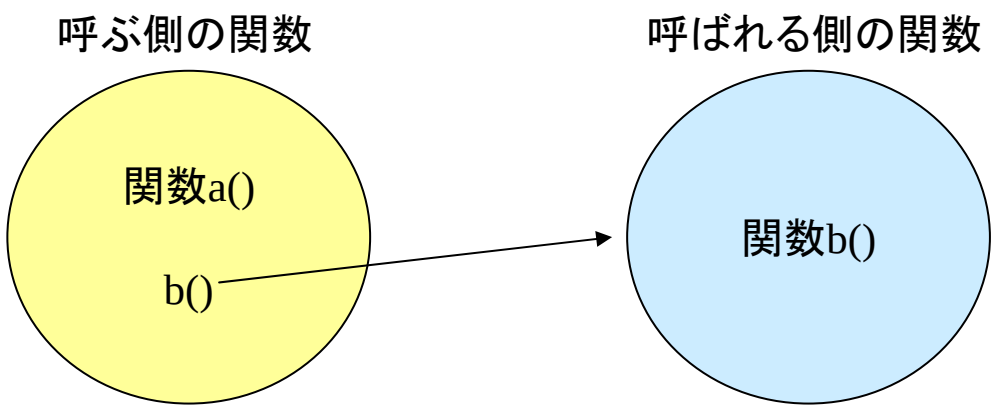
入れ替わった！



同一の物参照（同一アドレス）

アドレス渡しの利点と注意点

- 関数を呼ぶ側の変数の値を、呼ばれる側からも変更出来る
  - 呼ぶ側の変数に値を代入したい場合（**scanf**など）に便利
  - 複数の戻り値が欲しい場合などにも利用出来る
- 誤って、呼ぶ側の関数の変数を破壊する可能性もあるので、注意が必要



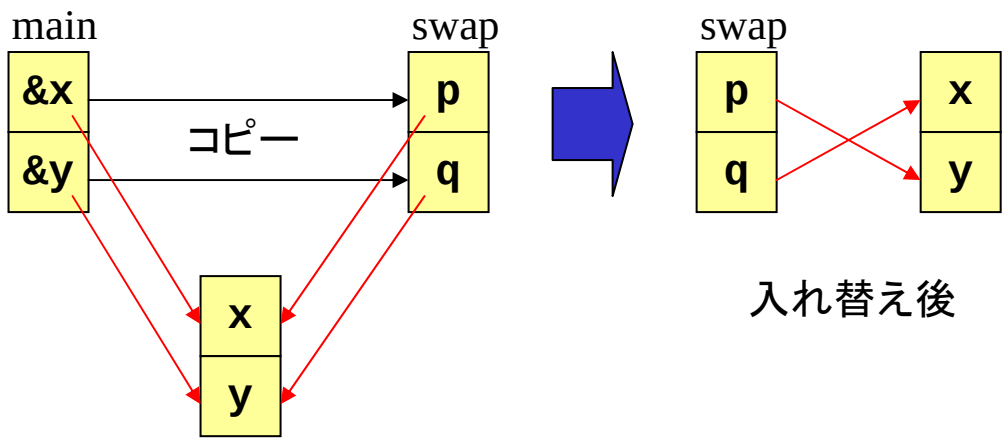
例えば関数内で、
`if(*p == 0)`
と書くべき所を、
`if(*p = 0)`
とすると、ポインタpが指すアドレスの変数に0が代入されてしまう！

クイズ

- 先ほどのアドレス渡しで値を入れ替えたプログラムを書き直し
  関数で値ではなく、アドレスを入れ替えるように変更した
- 結果はどうなるか推測してみよう

```
#include<stdio.h>
void swap(int *, int *);
main()
{
  int x = 5, y = 3;

  swap(&x, &y);
  printf("x=%d¥ty=%d¥n", x, y);
}

void swap(int *p, int *q)
{
  int *temp;
  temp = p;
  p = q;
  q = temp;
  printf("*p=%d¥t*q=%d¥n",*p,*q);
}
```

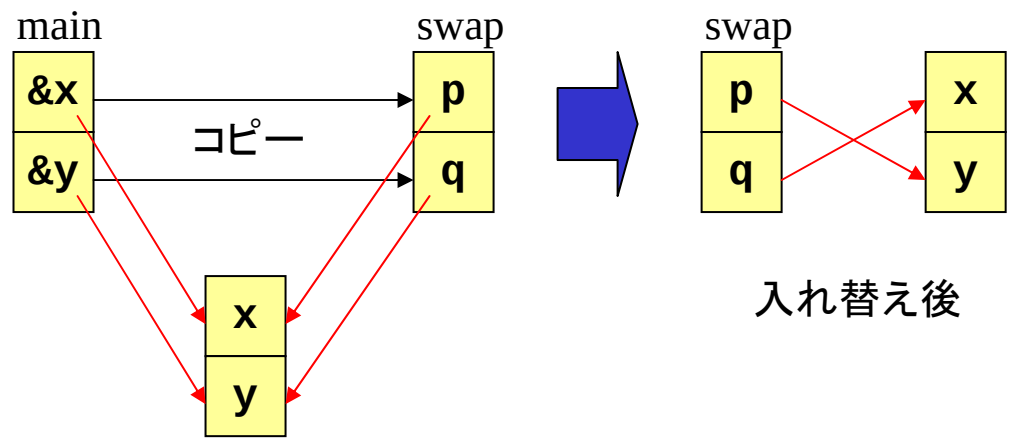
```
実行結果
s1000001{std1ss1}1: ./a.out
*p=3     *q=5
x=?      y=?
s1000001{std1ss1}2:
```



クイズの解答

- 先ほどのアドレス渡しで値を入れ替えたプログラムを書き直し関数で値ではなく、アドレスを入れ替えるように変更した
- これは、結局最初pがxを指していたのが、yを、qがyを指していたのがxを指すようになる。
- 結局mainの変数x,yの値は入れ替わらない

```
実行結果
s1000001{std1ss1}1: ./a.out
*p=3      *q=5
x=5       y=3
s1000001{std1ss1}2:
```



配列の引数

- 配列の先頭アドレスを「アドレス渡し」することで配列を引数にすることが出来る
- 関数側の引数はポインタ
- この関数は配列の各々の要素を2倍する

```
#include<stdio.h>
void x2(int *);
main()
{
  int a[] = {1 , 2 , 3 , 4} , i;
  for(i = 0 ; i < 4 ; i++)
    printf("a[%d] = %d\n",i,a[i]);
  x2(a);
  for(i = 0 ; i < 4 ; i++)
    printf("a[%d] = %d\n",i,a[i]);
}

void x2(int *x)
{
  int i;
  for(i = 0 ; i < 4 ; i++)
    x[i] = 2 * x[i];
  return;
}
```

呼び出し前の要素値の表示

配列aを引数にして関数x2を呼ぶ

呼び出し後の要素値の表示

ポインタを配列風に使用している

実行結果

```
s1000001{std1ss1}1: ./a.out
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[0] = 2
a[1] = 4
a[2] = 6
a[3] = 8
s1000001{std1ss1}2:
```

関数呼び出し前

関数呼び出し後

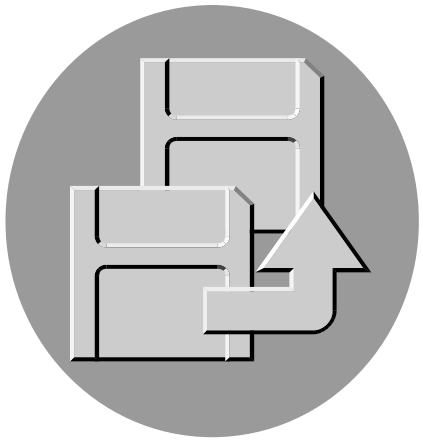
配列の引数

```
#include<stdio.h>
void x2(int []);
main()
{
  int a[] = {1 , 2 , 3 , 4} , i;
  for(i = 0 ; i < 4 ; i++)
    printf("a[%d] = %d\n",i,a[i]);
  x2(a);
  for(i = 0 ; i < 4 ; i++)
    printf("a[%d] = %d\n",i,a[i]);
}

void x2(int x[])
{
  int i;
  for(i = 0 ; i < 4 ; i++)
    x[i] = 2 * x[i];
  return;
}
```

- 関数の引数を配列風に書く事も出来る
- これは前ページの例と全く同じ動作をする
- 関数x2の仮引数xは配列ではなく、ポインタである

配列の引数サンプル

- 以下は今日出てきた関数へのアドレス渡し・配列の引数を利用したプログラムである
- かなり実用的なプログラムであるので、是非サンプルプログラムを各自のディレクトリにコピーし、色々と変更して動作を試してみて欲しい



main関数への引数

- これまでmain関数はただ、main()と書いてきたが、本当は以下のようになる。（戻り値のintは省略しても構わない。引数も使用しない場合は省略出来る）

```
int main(int argc, char *argv[])
```

- *argv[]は第五回授業の最後に出てきたポインタの配列（文字列定数の配列）である
- main関数への引数は実行時のオプションとして与えることが出来る。
- 例えば、プログラムの実行モジュール名がprogexだとして、コマンドラインから、

```
./progex abc def gdsf t47jk
```

のように入力すると以下のように数値（文字列）が代入される

1. argcには自分(progex)を含んだ引数の数（この場合5）
2. argv[0]にはコマンド自身の文字列（"./progex"）
3. argv[1]には1番目の引数の文字列（"abc"）
4. argv[2]には2番目の引数の文字列（"def"）
5. argv[3]には3番目の引数の文字列（"gdsf"）
6. argv[4]には4番目の引数の文字列（"t47jk"）

main関数の引数

- 以下のプログラムによってargvに入った文字列をすべて出力することが可能である。
- main()関数への引数によって、プログラムにいろいろな指示を与えることが出来る。例えば以下のようなプログラムが考えられる。
  - **prog2 100** などのように（ループ回数のような）渡したい情報を与える。
  - **prog3 test/test.data** などのように使いたいファイル名を指定してもらう
  - **prog1 -a -bcd** などのようなオプションを入力し、プログラムの動作を変える

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  for (i = 0 ; i < argc ; i++)
    printf("argv[%d] : %s\n",i,argv[i]);
}
```

i番目の文字列

実行結果

```
s1000001{std1ss1}1: ./progex abc def gdsf t47jk
argv[0] : ./progex
argv[1] : abc
argv[2] : def
argv[3] : gdsf
argv[4] : t47jk
s1000001{std1ss1}2:
```

```c
#include <stdio.h>
#include <stdlib.h>
int my_strlen(char *);
int main(int argc, char *argv[])
{
  char buf[100];
  int count = 0, acu_leng = 0;
  FILE *fp;

  if(argc != 2) {
    printf("Usage: %s filename\n",argv[0]);
    exit(8);
  }
  if((fp = fopen(argv[1],"r")) == NULL){
    printf("File %s not found\n",argv[1]);
    exit(8);
  }
  while(fscanf(fp,"%s",buf) == 1){
    count++;
    acu_leng += my_strlen(buf);
  }
  printf("total %d word(s), average length = %f\n",
     count, (double)acu_leng/count);
}

int my_strlen(char *buf)
{
  int i;
  char *p = buf;

  for(i = 0 ; p[i] != '\0' ; i++);
  return i;
}
```

サンプル

実行結果:

```
std1ss1{s1000000}1: cat in1.data
In 1984 Apple Computer introduced the Macintosh
desktop computer with a very "friendly" graphical
user interface. Graphical user interfaces(GUIs) began
to change the complexion of the software industry.
std1ss1{s1000000}2: ./a.out  in1.data
total 28 word(s), average length = 6.250000
std1ss1{s1000000}3: wc in1.data
      4      28     207 in1.data
std1ss1{s1000000}4:
```

このプログラムは以下の特徴がある。
(オリジナル：lec04-5.cを参照)

- **main**引数を利用してファイル名を入力
- 関数への配列引数を利用して文字列の長さを知る関数**my_strlen**を作成
- **scanf(”%s”)**を使用して、文中の単語の数を数える

無駄な配列要素

- 右は引数の文字列を数字とみなして、データの個数を指定し、最大1000個までのデータをscanfで読み込むことが出来るプログラムである。
- ところが例えば要素数が5個なら配列の995個の要素は無駄になる
- なお、右のプログラム中、関数atoiは数字の文字列（例えば"234"など）をint型の数字に変換する関数である(stdlib.h に定義されている)

```
実行結果
s1000001{std1ss1}1: ./a.out 5
array = 4000 bytes
2 4 6
8 10
2 4 6 8 10
s1000001{std1ss1}2:
```

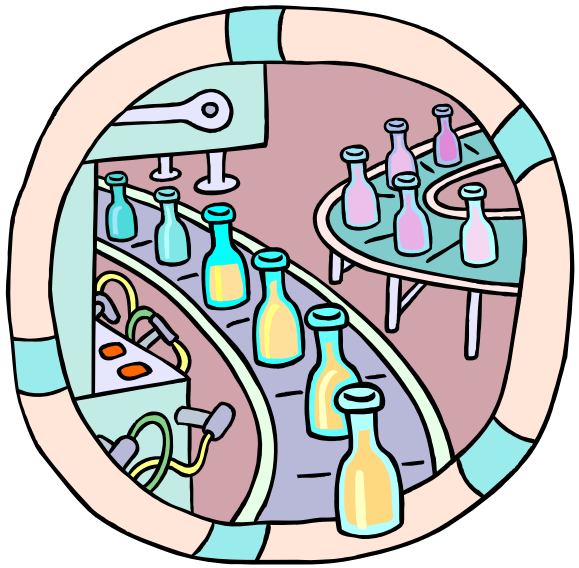
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000

int main(int argc, char *argv[])
{
  int  i, array[MAX],elem;

  if(argc != 2){
    printf("Parameter error. Usage:¥n");
    printf(" %s element-su¥n",argv[0]);
    exit(1);
  }

  elem = atoi(argv[1]);
  printf("array = %d bytes", sizeof(array));

  for(i = 0 ; i < elem ; i++)
    scanf("%d",&array[i]);

  for(i = 0 ; i < elem ; i++)
    printf("%d ",array[i]);

  printf("¥n");
}
```

動的メモリ割り当て

- 大きさを読み込んで（又は計算して）から配列を取りたい場合が出てくる。
- プログラムが作動しだしてから状況に応じて配列の大きさを変更することからこれを「動的メモリ割り当て」と呼ぶ。
- しかし、以下のようなコードを書くことはCでは許されない（エラーになる）

```
int n;
scanf("%d",&n);
int a[n];
```

- そこで、動的メモリ割り当てを関数とポインタによって実現している
（この関数はライブラリ stdlib.h に含まれる）



動的メモリ割り当て関数(1)

- 動的メモリ割り当てのための関数として、代表的なものはmallocとfree（次頁）の2つである
- **`void *malloc(size)`**
  これは size バイト分の領域をメモリ上に確保して、その先頭アドレスを戻り値として返すものである。例えば、int型の領域を5個確保してその場所をarrayという名前にしたいなら、以下のようにする。

```
int *array;
array = malloc(5 * sizeof(int));
```

  ここで、**5 * sizeof(int)** はその配列全体のバイト数を意味している。

関数mallocの戻り値は、「void *」（void型ポインタ）という型で、「どんな型でもないポインタ」と言う意味である。確保した領域をint型として使用する場合は、**`array = (int *)malloc(..)`** のようにint型ポインタ(int *)でキャストするか、上のように「暗黙のキャスト」を使用してそのまま代入する。

動的メモリ割り当て関数(2)

- **`void free(void *p)`**
  これは引数で指定されたポインタが指す領域（正確にはポインタは領域の先頭アドレスを指す）を解放することを意味する。上の例の配列 array を解放するには以下のようにすれば良い。
  ```
  free(array);
  ```
- 動的メモリ割り当ての手順は以下のようになる
  - 配列名となるポインタを宣言
  - **`malloc`**で配列となる領域を確保、領域の場所をポインタに代入
  - ポインタを配列名として配列を使用
  - 使用が終了したら**`free`**で領域を解放

無駄なく配列を使う

- 右のプログラムは先ほどの例を動的メモリ割り当てを使用して書き換えたもので、この方法では余分な配列要素を作成しないので、無駄が無い。

実行結果

```
s1000001{std1ss1}1: ./a.out 5
array = 20 bytes
2 4 6
8 10
2 4 6 8 10
s1000001{std1ss1}2:
```

注：動的メモリ割り当ての場合、**sizeof(array)**は配列の大きさではなく、ポインタの大きさ（4）を返すので注意！

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int   i, *array,elem;

  if(argc != 2){
    printf("Parameter error. Usege:\n");
    printf(" %s element-su\n",argv[0]);
    exit(1);
  }

  elem = atoi(argv[1]);
  array = malloc(elem * sizeof(int));
  printf("array = %d bytes",elem * sizeof(int));

  for(i = 0 ; i < elem ; i++)
    scanf("%d",&array[i]);

  for(i = 0 ; i < elem ; i++)
    printf("%d ",array[i]);

  printf("\n");
  free(array);
}
```