C言語講座第[illegible]回

～それはとってもポインタだなって～

第５回でやる「ポインタ」はC言語の中でとても
重要な機能になります。

しかし、C言語を始めたばかりの人の大半がこの「ポインタ」
が理解できずにつまずきます。（そしてそのままにします）

なので、今回のみで無理に理解しようとしないで
分からないことが出来たらその度に部室に来るなりして
先輩に聞きに来てください
（別に恥ずべきことじゃないよ！！）

アドレス（address）

今までの講座で使ってきた「int～」や「char～」のような変数は全てメモリ上に一時的に記憶（保存）されている。

```
int   a;

char  b;
```

←いままでこんな感じで書いてるよね？

その保存されている場所（番地）を**アドレス**と言います。
（いきなりポインタと違う話じゃねーかと思うかも
しれませんが、アドレスという存在を知らないと
ポインタの説明が出来ないんだ☆）

アドレスを表示させる

アドレスを実際に使うには変数に「＆」をつけます。

```
#include<stdio.h>


int main(){
int    a  =  5;
printf("変数aのアドレス%p¥n",&a);
return 0;
}
```

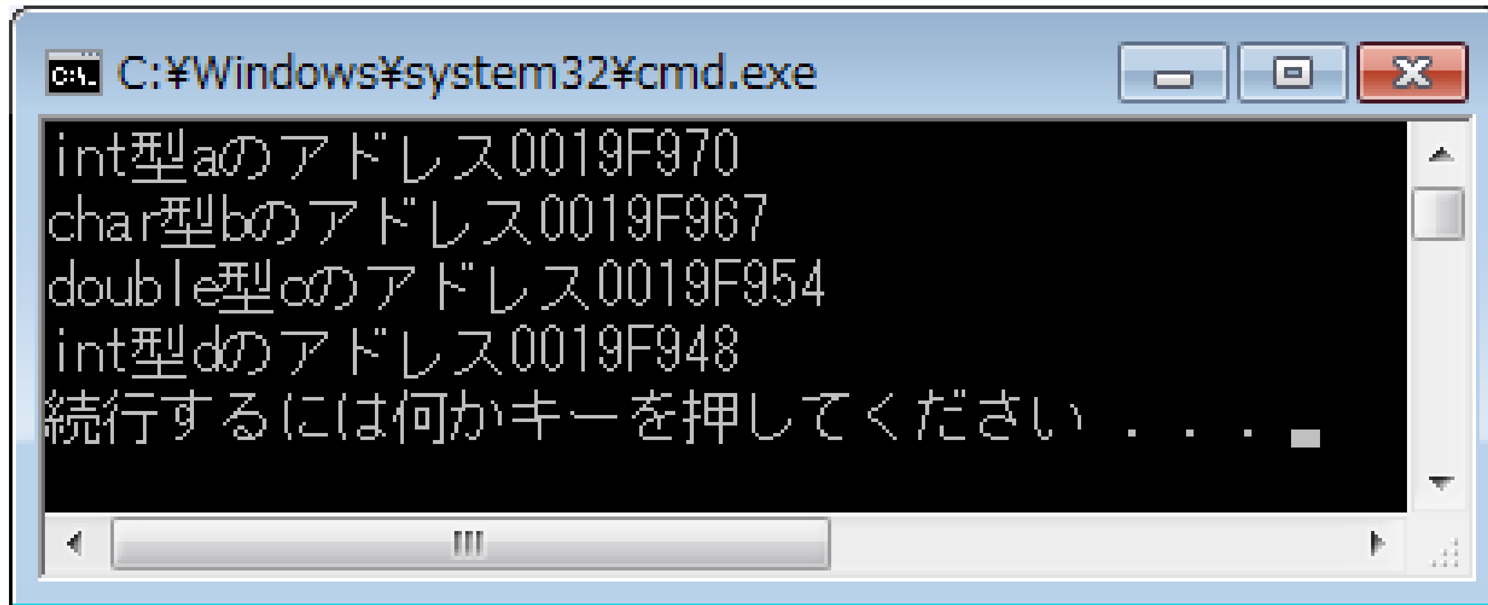
赤枠のところに注目！！（%pはアドレスを表示する）

で、この説明するための例（ちょっと汚いかも・・・）

```
#include<stdio.h>

int  main(){

int  a=5;

char b='a';

double c=1.0;

int d=5;

printf("int型aのアドレス%p¥n",&a);

printf("char型bのアドレス%p¥n",&b);

printf("double型cのアドレス%p¥n",&c);

printf("int型dのアドレス%p¥n",&d);

return 0;

}
```

別に型がなんでも
アドレスを扱うなら
「＆」をつけるよ。

前のページの例の実行結果



「～型～のアドレス・・・」の「・・・」がこんな感じのものが表示されます。これが、その変数がメモリ上に保存されている場所、つまり**「アドレス」**です。

（このアドレスは環境によって違うので同じ必要はないです

　とりあえずこんな感じのものが出ればOK）

ちなみに配列でやってみると・・・

配列を使った例

```
#include<stdio.h>

int main(){

int a[3]={1,2,3};

printf("a[0]のアドレス%p¥n",&a[0]);

printf("a[1]のアドレス%p¥n",&a[1]);

printf("a[2]のアドレス%p¥n",&a[2]);

retrn 0;

}
```

C:¥Windows¥system32¥cmd.exe

```
a[0]のアドレス002DFC60
a[1]のアドレス002DFC64
a[2]のアドレス002DFC68
続行するには何かキーを押してください . . .
```

黒い画面に注目するとさっきと違って下２桁の数値が４ずつ増えているのがわかると思います。

なんか配列の説明にもなりそう・・・

最初の「配列を使用しない」場合では、
バラバラなアドレスになっている（詳しくは実行結果ね）

次の「配列を使用する」場合では、
４ずつ増加したアドレスになっている（これも結果を）

これをみると変数は別に順番に置かれているわけではない
ということがわかります。
（配列は一気に確保しているから連続しているよ）
ちなみに４ずつ増加する理由は、第１回でやったけど
「int」の大きさは「4byte」だから4byte分確保しています。
（もちろん「char」なら「１」、「double」なら「８」）

ポインタ（pointer）

ポインタとは「変数のアドレスを入れる変数」
まぁ・・・何をいってるのか分からないと思うので
少しずつ例を使いながら説明していこうと思います。

まずポインタを使うには先ほどの「アドレス」の時の
ように決まった書き方があります（覚えてね）

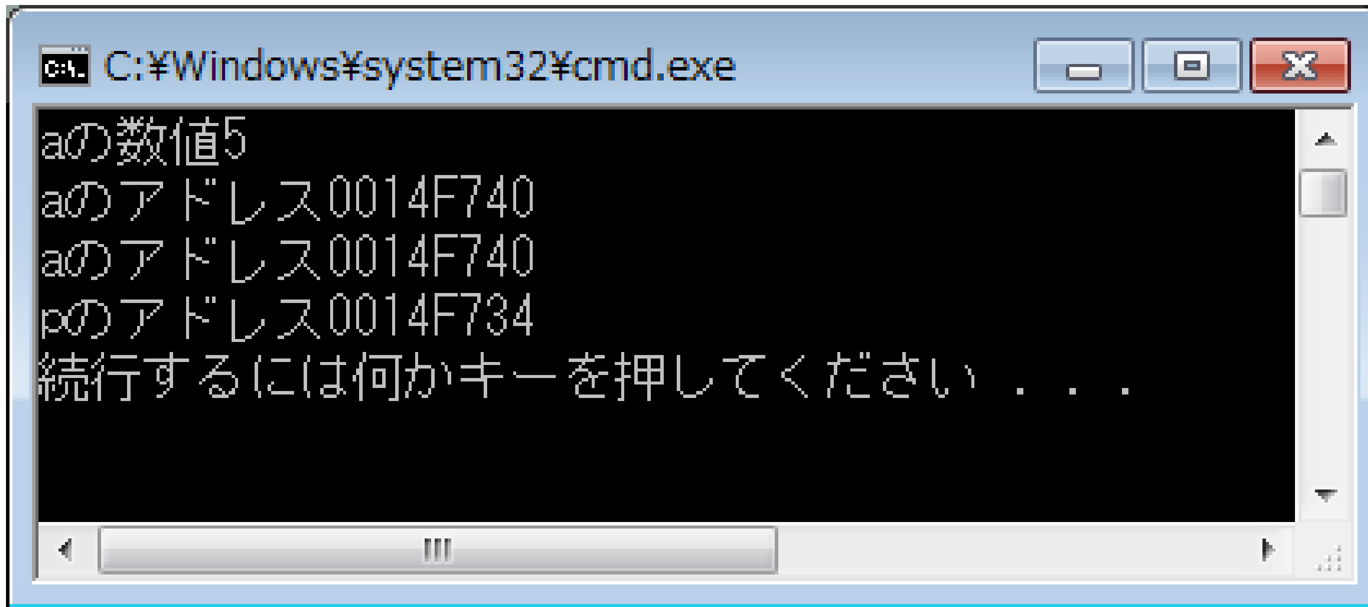
```
int a;        →        int *p;
```

といっても、上の図のように変数の前に「＊」
をつければいいだけです。

で、ポインタがどのようなものか見てみましょう（例でね）

```
#include<stdio>
int main(){
int a = 5;
int *p; //ポインタ（[pointer]の頭文字から[p]だよ）
p = &a;
printf("aの数値%d",a)
printf("aのアドレス%p¥n",&a);
printf("aのアドレス%p¥n",p);
printf("pのアドレス%p¥n",&p);
return 0;
}
```

実行結果（アドレスは環境によって違うよ）



ここで例のプログラムをみると「printf」で「&a」と「p」なのに表示するアドレスが同じだということがわかる。
そしてその下の「&p」では「int *p」のアドレスが表示している。（結局「p」も「int」の型だからアドレスがある。）
で、なぜ「*p」と書いたり「p」と書いたりするのか？
（「&」についてはアドレスでやったからいいよね？）

実は最初の構文としてポインタを使うには「int *p」と書くと説明しましたが、別に「*p」という変数ではない。
正確にいうと「int p」という変数に「*」をつけることによって他の変数のアドレスを入れて使用できるようになるといった考え方をしてください。

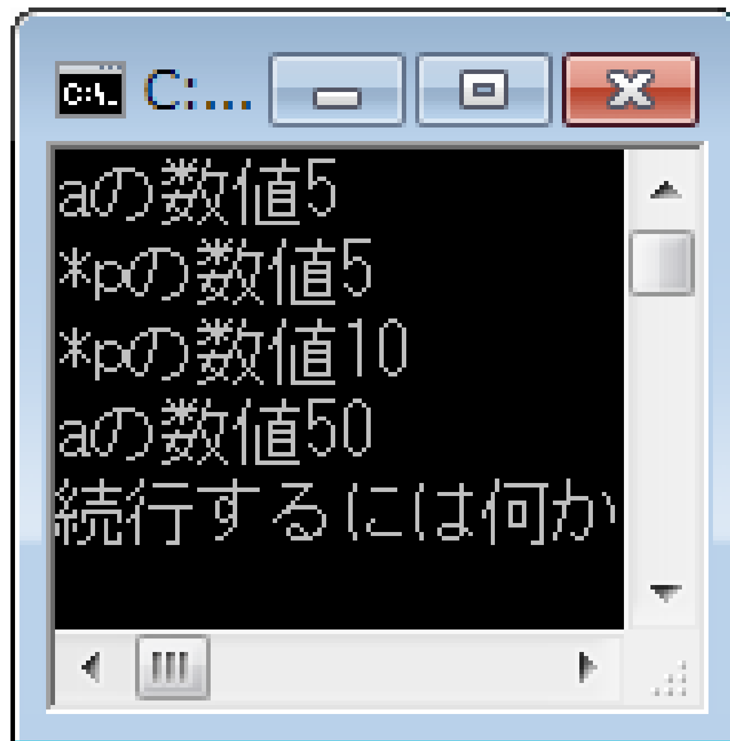
そう考えると、「a」のアドレスは「p」に入れたいので、「p=&a」としているわけです。
だから「a」のアドレスを表示する際は「p」になっています。

では「*」の意味はなんなのかと思いますが、こいつは「自分（ここではp）に入っているアドレスを指す」という意味になっています。（次で説明するよ）

では「＊」を使った例を見ましょう

```
#include<stdio.h>
int main(){
int a = 5;
int *p;
p = &a;
printf("aの数値%d¥n",a);
printf("*pの数値%d¥n",*p);

a = 10;
printf("*pの数値%d¥n",*p);

*p = 50;
printf("aの数値%d¥n",a);

return 0;
}
```

左の例の実行結果↓



上の例を見ると「p」に何も数値を入れてないのに
「printf」で「*p」を使うと「a」と同じ数値を表示しています。
まさにこれが「*」つまりポインタの効果（効果でいいの？）
「*」のついた時は**「＊のついている変数（ここではp）に入っているアドレス（ここでは&a）にある中身を参照する」**
といった働きがあります。
だから上の例では「p」の得ているアドレス「&a」の中身「５」を得て表示しているというわけです。
また「*」は使う直前のアドレスの中身を得るので、
「a = 10」のようにアドレスの中身を変えると「*p」で
得られる数値も同じ「10」になるわけです。
逆に「*」の数値を「*p=50」のようにかえると、参照しているアドレスの中身を変えます。
なので、最後の「a」の数値が「50」になるわけです。

補足・・・

いままで、

```
int a;

int *p;

p = &a;
```

のように書いていましたが。

最後の代入を一括にし

```
int a;

int *p = &a;
```

このように書くことが出来ます。

最初のアドレスを入れるとかの話のために分離していましたが
別にどちらを使ってもいいです。

補足2･･･

ポインタを使う時に注意することがあります。

```
#include<stdio.h>

int main(){

int a;

int *p;

printf("%d",*p);

return 0;

}
```

←絶対に実行しないでね。
（まぁこれは問題ないと思うけど）

なぜやってはいけないのかというと、
ポインタはアドレスを得てその中身を間接的に書き換えたり参照する物になっている。

では、前のページの例のようにアドレスを渡さずに「*p」を使うと「*p」が勝手どこかのアドレスを参照してしまいます。

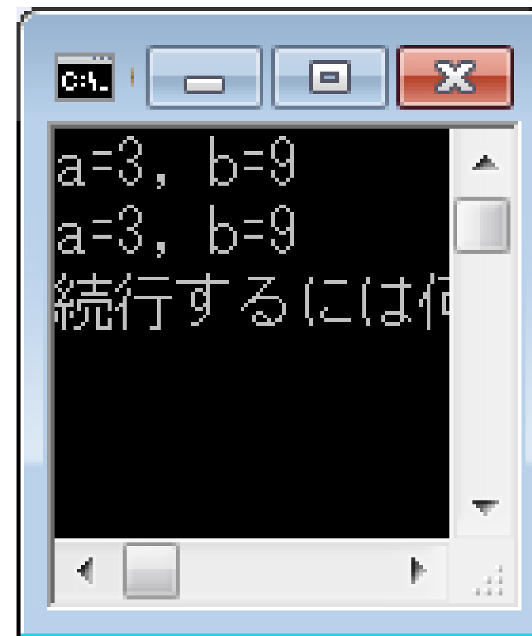
これだけ聞くと「別に問題なさそう・・・」と思うかもしれませんがその参照したアドレスが他のシステムが使っている場所でもしそこを書き換えたりするとシステム自体がクラッシュします。

だから、アドレスを渡さずにポインタを使用するのは
「絶対にしないでください」

ではポインタの例としてよくある数値の入れ替えをしましょう

```
#include <stdio.h>
void swap(int x, int y) {
int tmp = x;
x = y;
y = tmp;
}
int main() {
int a = 3;
int b = 9;
printf("a=%d, b=%d¥n",a,b);
swap(a, b);
printf("a=%d, b=%d¥n",a,b);
return 0;
}
```

実行結果↓



このプログラムを見ると、「aとbの値を入れ替える」というのが
ちゃんと出来そうに見えます。

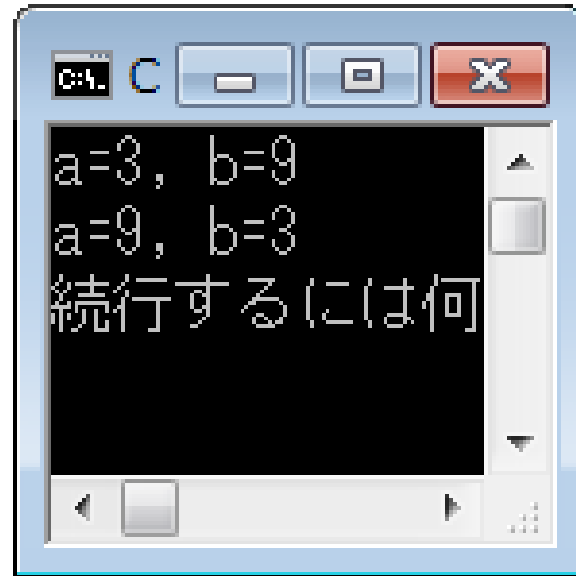
しかし、実行結果を見てみると、入れ替える前も、「swap」関数を
よんで入れ替える処理をした後でも「a」と「b」の結果（中身）は同じ
になっています。

なぜこのようになるのかというと
swap関数の引数「int x,int y」に「a」と「b」の数値を入れていますが
この「x」と「y」はそれぞれ「a」と「b」のコピーになっています。
コピーということになり「int x」と「int y」は別のアドレスを持っており
この２つの中身をいくら変えても最終的に表示するのは「a,b」なので
最終的にみても値が交換されていないというわけです。（分かる？）

次は、ちゃんと入れ替えが起こるように書いた例です。

```
#include<stdio.h>
void swap(int *x,int *y){
int tmp = *x;
*x = *y;
*y = tmp;
}
int main(){
int a=3;
int b=9;
printf("a=%d,b=%d¥n",a,b);
 swap(&a,&b);
printf("a=%d,b=%d¥n",a,b);
return 0;
}
```

実行結果↓



今度はしっかりと入れ替えが起こっていると思います。
で、なぜ「swap関数」に「&a」と「&b」を渡していて
「swap関数」の宣言では「int *x」と「int *y」なのか？
と思うかもしれません。（突然こうだから分かりにくいよね）

でも、実際は一つずつ見ていけば非常に簡単に出来ています。

まずは、「int *x,int *y」に対して「&a,&b」を渡すのかを
考えていきましょう。

これはそれぞれ「int *x = &a」と「int *y = &b」だと
考えます。（次のページで図に示しています）

つまりこう考えればよい（よいのか？）

```
#include<stdio.h>
void swap(int *x,int *y){
int tmp = *x;
*x = *y;
*y = tmp;
}
int main(){
int a=3;
int b=9;
printf("a=%d,b=%d¥n",a,b);
swap(&a,&b);
printf("a=%d,b=%d¥n",a,b);
return 0;
}
```

int *x = &a;　　int *y = &b;

こう見ると「補足」
でやった形と同じだね
つまり
「x」,「y」に
アドレスを入れてる。

これは今までやってた
「p」に「a」の
アドレスを入れる
というのと同じこと。

つぎに「swap関数」の中身について見よう
実は今まで書いてきた「＝」の式というのは
「＝」の左辺、右辺によって同じ変数であっても意味が違う
・左辺はその変数の領域（つまりその場所自体）
・右辺はその変数の中身（つまり変数に入れた数字や文字）
を見る。（ちょっと分かりずらいかも・・・）
つまり
「int tmp = *x」というのは「tmpという変数がある場所に
*xの中身を入れる」という意味になっている。
で、「*」がついてるので「x」に入っているアドレスの先に
ある中身（この例では「a」の中身つまり「3」）を得る。
この次の「*x = *y」という式も「*x」がある場所に
「*y」の中身（この場合は「9」）を入れるという事です。

配列とポインタ

配列での場合は今までと微妙に違います。
まず配列の場合は「&」をつけません。（やったね！！）
ポインタにいれてあげるのは今までと同じで

```
int a[4] = {1,2,3,4};

int *p = a;
```

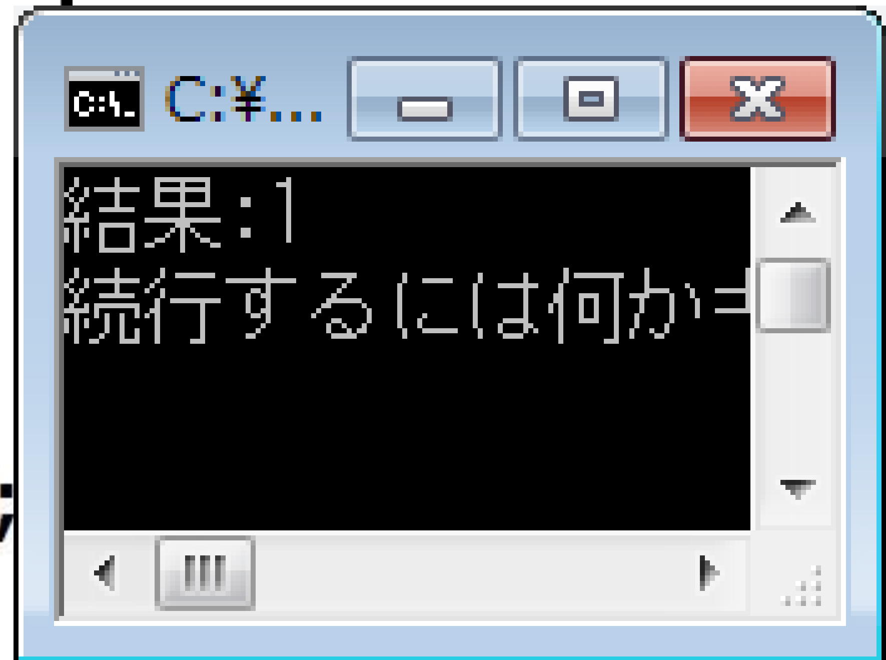
このように書きます。（「a」に「&」がついてません）

で、これ以外に実際に呼び出した時も少し結果が違います

配列を使った時の違い

```
#include<stdio.h>
int main(){
int a[4] = {1,2,3,4};
int *p = a;
printf("結果:%d¥n",*p);
return 0;
}
```

実行結果↓



結果をみてみると「1」という結果が出ています。
なぜこのような結果になるかというと
「a」というのはその配列の先頭のアドレスを示すポインタ
（つまり「a[0]」に入っている物を示す）となっています。

だから「*p」に渡したのは先頭アドレスだけであり
そこから先の「a[1]」以降はありません。

なので、次は各配列の部分を呼び出すのをしましょう

配列の任意の場所を呼び出す方法（見にくいのはすいません）

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

for(i=0;i<4;i++){
    printf("%d",a[i]);
}

printf("¥n");
for(i=0;i<4;i++){
    printf("%d",p[i]);
}
```

```
printf("¥n");
for(i=0;i<4;i++){
    printf("%d",*(a+i));
}

printf("¥n");
for(i=0;i<4;i++){
    printf("%d",*(p+i));
}
printf("¥n");

return 0;
}
```

実行結果↓

結果を見ると全部同じになっています。
しかし、赤枠で囲った部分に注目してみてください。
全て記述が異なっているのがわかると思います。

実はこれは全部同じ意味合いになっています。
まず最初の「a[i]」となっているのは単純に「配列のa[i]番にある中身を出す」となっています。（「p[i]」も同じかな）
次に、「*(a+i)」となっています、
これは「配列の先頭アドレスにそのデータ型のバイト数×iのアドレスの中身を示す」というものになっています。（分かりにくい？）
つまり「i」が「1」なら「a[1]の中身（つまり「２」）」を示しているという事になっています（「*(p+1)」も同じ）

でも実際は・・・

うえでなんだとごちゃごちゃと書き方を教えているけど
このような例（main以外の関数がない）なら素直に
（「a[~]」）という普通の配列の書き方でいいです
（むしろそのほうがいい）

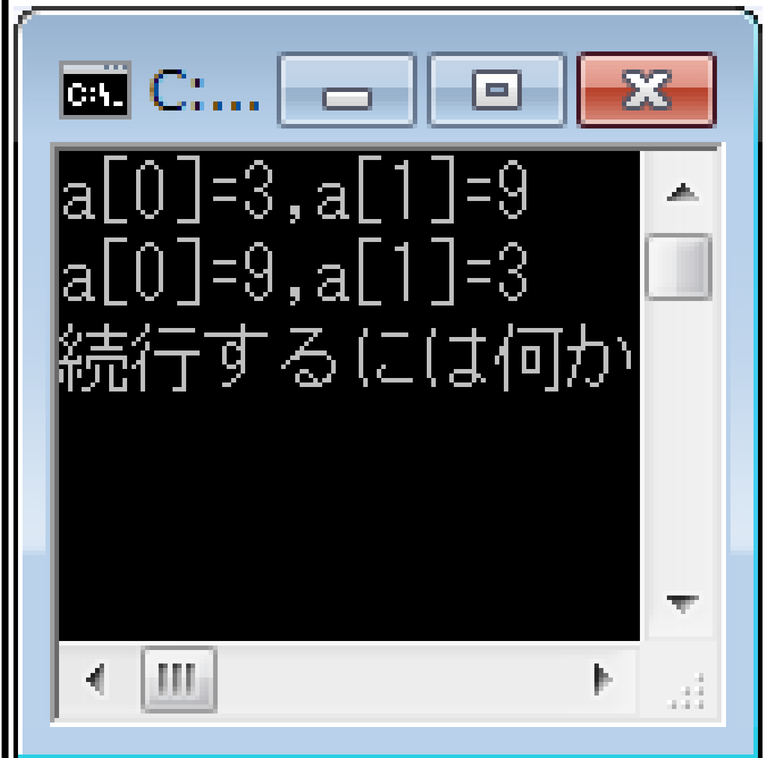
まぁ・・・つまり
こういう書き方も出来るとだけ覚えておいてください

次に最初のほうでやった数値の入れ替えを配列でやってみましょう。（配列の場合の関数への渡し方を見よう）

中身の入れ替え（配列ver）

```
#include<stdio.h>
void swap(int *x){
int tmp = *x;
*x = *(x+1);
*(x+1) = tmp;
}

int main(){
int a[2]={3,9};
printf("a[0]=%d,a[1]=%d¥n",a[0],a[1]);
swap(a);
printf("a[0]=%d,a[1]=%d",a[0],a[1]);
return 0;
}
```

実行結果↓



上の実行結果を見るとちゃんと入れ替わっているのが分かる。
では、最初の方でやった入れ替えとどこが違っているのか
見てみよう。

まず「main関数」のなかで「swap関数」を呼び出しているが
引数が「a」とアドレスがついていないのが分かります。
（これは「配列とポインタ」の最初の方でやりました）
そしてその「配列a」の先頭アドレス（つまり「a[0]」）を
「int *x」に入れています。
「swap関数」内の書き方は基本的に普通の入れ替えと同じです。
変更点は「*y」と宣言していた場所を「*(x+1)」としていることです。（「*(x+1)」は「a[1]」をさしている）

で、結局・・・

こいつの利点ってなに？と思うかもしれませんが
今は入れ替えの問題なんで引数２つだけでいいですが

もっとたくさん（極端に言えば１００個とか）になると
最初のやり方では変数を１００個、さらにそれに応じた
関数の引数を１００個宣言しないといけなくなります。

しかし、このように配列を使えば最初に「a[100]」として
中身を初期化すればいいだけになり、さらにそれに応じた
関数の引数は１個だけで済みます。
（中身の書き方が変わるけど・・・）
このようにプログラムは出来るだけ楽にしましょう

じゃあ演習だ（面倒かも・・・）

次のページのプログラムの空白部分にプログラムを
書き込み、このプログラムを完成させなさい。
このプログラムのやりたいこと
・配列に入れた数値を大きい順に並び替えて表示をする。

また下記の条件のもとで行うこと
・すでに書いてある文書を書き換えてはだめ。
・自作関数を増やすことはだめ。
・最終的な結果の出力は「main関数」内で行う、つまり
　並べ替えをする「sort関数」で出力をしてはいけない

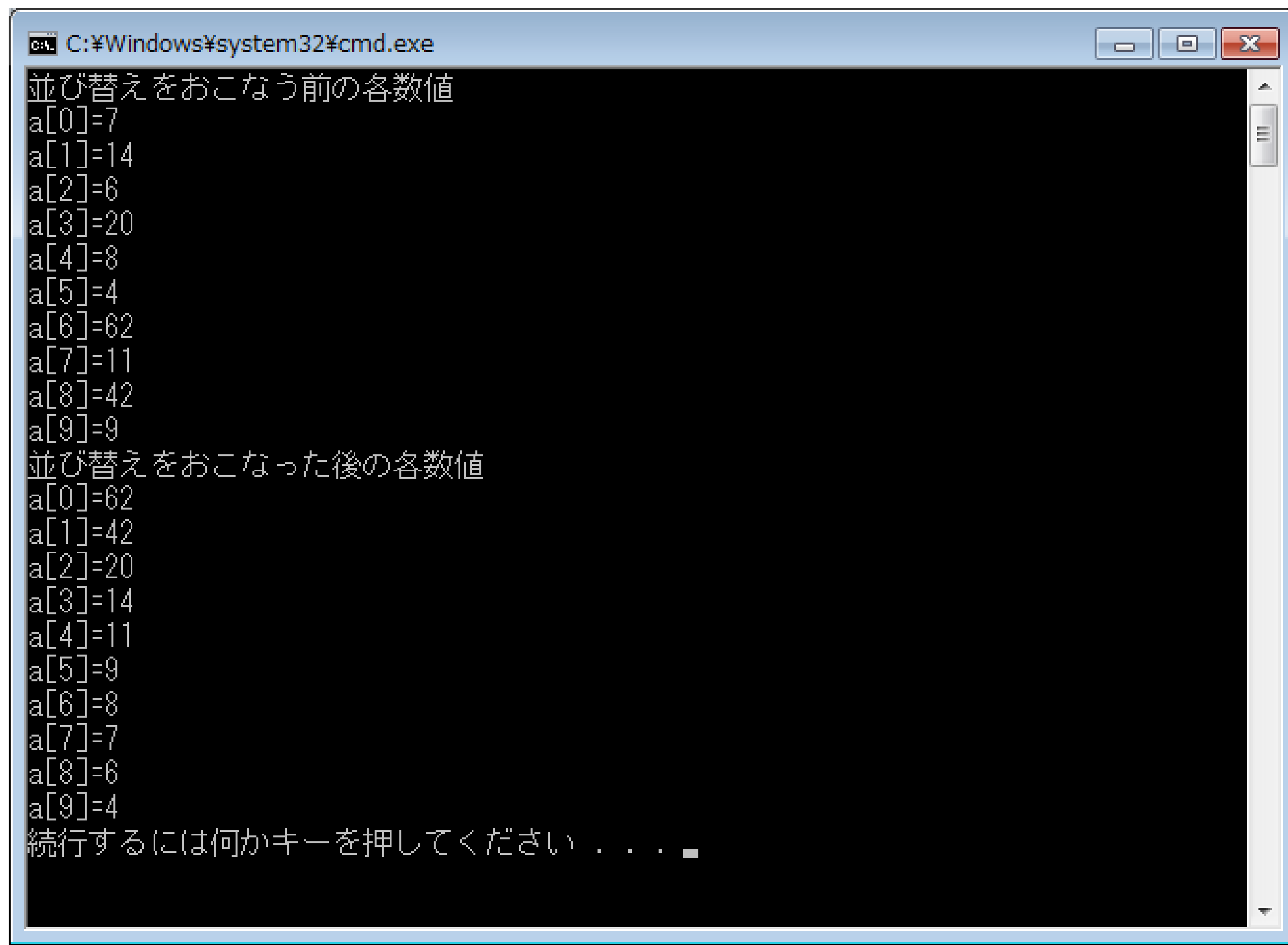
空白の部分を埋めること

```
#include<stdio.h>
void sort('空白');//プロトタイプ宣言

int main(){
    int i;
    int a[10]={7,14,6,20,8,4,62,11,42,9};//配列の宣言と初期化
    printf("並べ替える前の数値の確認¥n");
    for(i = 0;i < 10;i++){
        printf("a[%d]=%d¥n",i,a[i]);
    }

    sort('空白'); //sort関数の呼び出し

    printf("並べ替えた後の数値の確認¥n");
    for(i = 0;i < 10;i++){
        printf("a[%d]=%d¥n",i,a[i]);
    }
    return 0;
}
```

```
//並べ替えをおこなう場所
void sort('空白'){
    空白
}
```

空白 で囲っている場所を完成させる

結果はこうなる（はず...）



答え（すぐ見ちゃだめだよ）

また全部を表示するのはあれなんで
空白の部分のみを表記します。
まず「void sort(“空白”)」の部分は

```
void sort(int *x);
```

次にmain関数内の「sort(“空白”)」の部分は

```
sort (a);
```

これでいい
これはさっきの配列とポインタでやったからわかるよね？

次のsort関数の中身の「空白」は少し面倒
（難しくはありません）

ようはアルゴリズムをつかえばいい
（少なくとも応情の人はやってると思います。）

```
void sort(int *x){
    for(int i=0;i<10;i++){
        for(int j=i+1;j<10;j++){
            if(x[i]<x[j]){
                int tmp = x[i];
                x[i] = x[j];
                x[j] = tmp;
            }
        }
    }
}
```

←この赤枠の部分

この演習の補足

さて、無事に演習が終わったと思いますが
このプログラム・・・
はっきりいって使えないです。なんでかというと

・配列の個数を変更すると全体の記述の変更が面倒
・文字を表示するためのforループを２回もやってる
・最初に初期化しているので個数が増えると面倒

といった感じに、これ専用のプログラムになっています。
プログラムは使えるところは何度も流用したほうがいいので
でその点を踏まえて改造してみましょう。

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define N 10//定数Nの宣言

//プロトタイプ宣言
void sort(int *x,int n);
void random(int *x,int n);
void put(int *x,int n);

int main(){
   int a[N];
   random(a,N);
   printf("並べ替えを行う前の数値¥n");
   put(a,N);
   sort(a,N);
   printf("並べ替えを行った後の数値¥n");
   put(a,N);
   return 0;
}
```

```
void sort(int *x,int n){
        for(int i=0;i<n;i++){
                for(int j=i+1;j<n;j++){
                    if(x[i]<x[j]){
                        int tmp = x[i];
                        x[i]  =  x[j];
                        x[j]  =  tmp;
                    }
                }
        }
}

void random(int *x,int n){
            srand((unsigned) time(NULL));
            for(int i = 0;i<n;i++){
                    x[i]=rand()%101;
            }
}

void put(int *x,int n){
        for(int i=0;i<n;i++){
                printf("a[%d]=%3d¥n",i,x[i]);
        }
}
```

上の例では毎回、配列の中身の値はランダムになります。
（本当は「scanf」使って入力してもいいんですが面倒）
が、結局並べ替えがキチンと行われています。

このプログラムは最初の「#define N 10」と書いてある部分の「10」を違う数字にするだけで、どのような大きさの配列でも同じ処理を行えます。（あんまり大きのはダメだけど）

なんか、見てると「最初よりごちゃごちゃしてる」と思うかもしれませんが、基本的にプログラムでは「main関数」の中に、直接処理を書くのはあまりよくないのです。
（理由としては流用出来なくなるというのがあります。）

なので、行う処理ごとに関数を作るようにしてください。

これにて終了（長かった・・・）

これでポインタの説明は以上になります。
でも、本当は・・・
ポインタ自身も配列を持てたり
ポインタに対するポインタ（ダブルポインタ）
関数ポインタ（ようは関数を参照するポインタ）
と、説明してないことが多くありますがこれ以上
やると、長いうえに分かりづらくなるので
今回は説明しません。
（別に今知らないとやばいものではないよ）

では、以上になります。　お疲れ様でした～