

入門講習会 第8回

1. メモリについての軽い説明

ポインタを理解するために、まずメモリについて軽く触れておく。

メモリとは、データを一時的に保持しておくための記憶領域である。メモリからデータを取り出し、それを計算などに利用する。これは一時的な記憶領域であって、データをずっと保持しておくためのものではない。人間でいうと短期記憶、もので例えると計算に用いるメモ用紙にあたるものである。

さて、メモリにデータを保持しておくためには、どのデータがどの場所に保管されているのかを把握しておかなければならない。そこで、メモリの記憶スペースには、家でいう住所が割り当てられている。これを「アドレス」という。

これを踏まえて、メモリの構造を簡単に図式化すると右図のようになる。ここではメモリのごく一部の様子を示したが、勿論、記憶領域はこの図以上にたくさんある。アドレスは16進法(0~9の数字に加え、A, B, C, D, E, Fの文字を使って数字を表現する方法)で表記されることが多い。

Cでプログラムを書いているとき、変数を宣言することは日常茶飯事である。変数もこのメモリ上に確保される。しかし、普段私たちがアドレスについて気にすることはなかった。なぜなら、変数宣言がなされると、コンピュータが自動的に変数のための領域をメモリ上に確保してくれるからだ。

例を示そう。

```
char a;
```

という変数宣言がなされたとする。このとき、コンピュータがメモリ上で空いている領域を探してくれて、「ここが変数aの場所ですよ」と決めてくれる。

```
char b[5];
```

という宣言がなされても、同様の処理が行われる。ただし、配列の宣言のときは、メモリ上に連続して確保される。

さて、char型はふつう1バイトのデータを格納

アドレス	データ
0123	...
0124	...
0125	...
0126	...
0127	...
0128	...
0129	...
012A	...
012B	...
012C	...
012D	...
012E	...
012F	...
0130	...
0131	...
0132	...

	アドレス	データ
x {	0123	...
	(0124)	
	(0125)	
	(0126)	
a {	0127	...
	0128	...
	0129	...
	012A	...
a[0]	012B	...
a[1]	012C	...
a[2]	012D	...
a[3]	012E	...
a[4]	012F	...
	0130	...
	0131	...
	0132	...

できた。メモリ上の領域は一般的に 1 バイト区切りでアドレスが割り振られている。

```
int x;
```

と宣言されたとき、int 型は 4 バイトである。このとき、メモリ上では 4 つ分の領域を確保することになる。メモリ上での変数の様子の例を、図に示した。

変数に割り当てられているアドレスを取得するときは、アドレス演算子&を用いる

&変数名

という書式で、変数のアドレスが得られる。また、printf でアドレスを表示したい場合は%pを用いる。

以下は、int 型の変数 a のアドレスを出力するプログラムである。

```
#include <stdio.h>

int main(void)
{
    int a;
    printf("%p\n", &a);

    return 0;
}
```

実行結果は以下のようになった。アドレスは自動的に割り当てられるので、時と場合によって、表示される値は異なる。

```
0060FEAC
```

さて、ある変数があるスコープを抜けると、変数はもう必要なくなる。そのため、コンピュータは変数の割り当てを自動的に解除する。メモリの割り当てを解除することを「メモリの解放」ということがある。

ただし、malloc 関数などによってメモリを動的に確保した場合は、自動的に解放されない。プログラム中で free 関数を呼び出してメモリを解放してやらなければならないのですが、詳しくは割愛する。

2. ポインタ

ポインタとは、「アドレスを格納する特殊な変数」である。これを用いて、変数の値を間接的に得たり、変えたりできる。ポインタを用いて、変数を「遠隔操作」できる。

アドレスを格納する変数なのだが、「int 型変数のアドレスを格納するためのポインタ」「double 型変数のアドレスを格納するためのポインタ」など、何の変数のアドレスを格納するかによってポインタの種類が異なる。その理由については後で述べる。それぞ

れ, 「int 型へのポインタ」「double 型へのポインタ」などと呼ばれる。

ポインタを宣言するとき, 変数名の先頭に*をつける。例えば, int 型へのポインタを宣言するときには, 以下のように記述する。

```
int *p;
```

ポインタはアドレスを格納するための変数であるため, 「ポインタ変数」と呼ばれることもある。

ポインタにアドレスを代入するときは, 通常の代入と同じように書く。例えば, int 型の変数 a のアドレスを p に代入するときは, 以下のように書く。

```
p = &a;
```

これによって, p には a のアドレスが入ったため, p というポインタはメモリ上での変数 a の場所を把握していることになる。

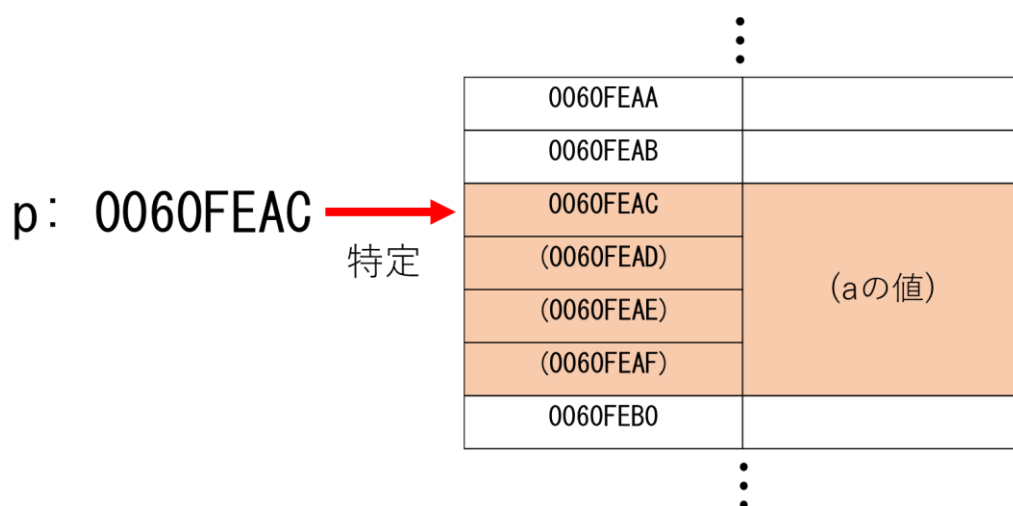
p は a の場所を知っているため, p は a の値を操作できる。p を介して a の値を操作するためには間接演算子*を用いる。例えば, ポインタを介して a の値に 10 を代入したい場合は, 以下のように書ける。これは a = 10 と同義である。

```
*p = 10;
```

b = a + 1 のような単純な計算も, ポインタを用いて次のように書ける。

```
b = *p + 1;
```

a のアドレスが p に格納されている図を以下に示す。ここでは a のアドレスが 0060FEAC だった場合を想定している。p = &a という代入処理によって, p には 0060FEAC が格納される。間接演算子によって, アドレスの番地 0060FEAC に入っている値, すなわち a の値を知ることができる。



わざわざ b = a + 1 としないで, b = *p + 1 と書く意味がどこにあるのかと思うかもしれない。応用例については後で述べる。

3. ポインタ演算

ポインタの値はアドレスである。アドレスに 1 加えれば, アドレスは 1 つ分動くと思うかもしれない。例えばアドレス 0060FEAC に 1 を加えれば, 0060FEAD となりそうである。しかし実際は異なる。試しに, 以下のプログラムを作ってみよう。

```
#include <stdio.h>

int main(void)
{
    int a;
    double b;
    int *ip = &a;
    double *dp = &b;

    printf("ip-1: %p\n", ip - 1);
    printf("ip  : %p\n", ip);
    printf("ip+1: %p\n", ip + 1);

    printf("dp-1: %p\n", dp - 1);
    printf("dp  : %p\n", dp);
    printf("dp+1: %p\n", dp + 1);

    return 0;
}
```

int 型へのポインタと, double 型へのポインタを作った。その値(=アドレス)に 1 加えたり引いたりした結果を printf で出力している。

実行結果は以下のようになる。

```
ip-1: 0060FEA0
ip  : 0060FEA4
ip+1: 0060FEA8
dp-1: 0060FE90
dp  : 0060FE98
dp+1: 0060FEA0
```

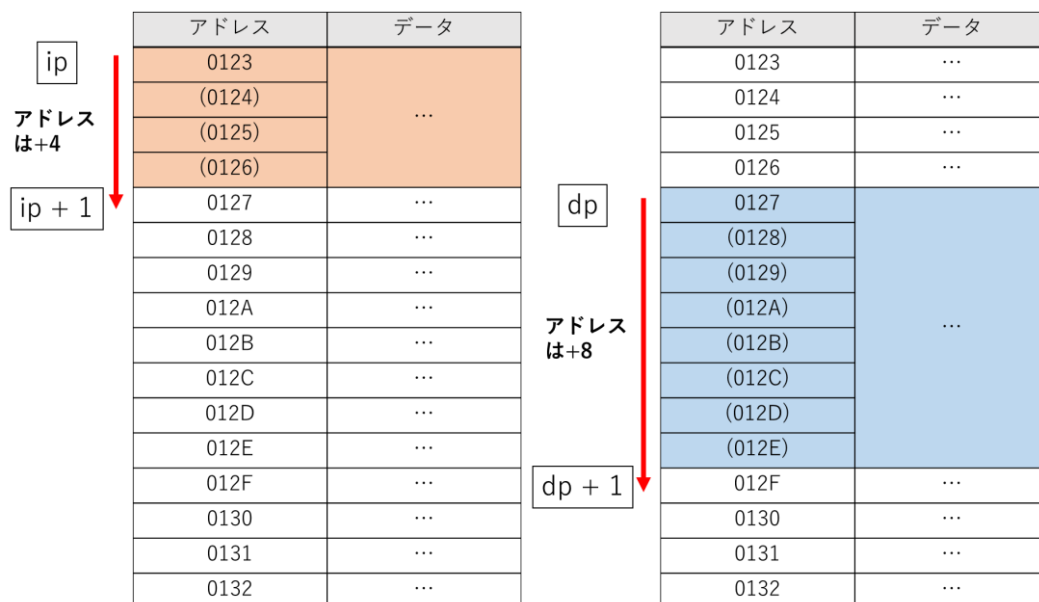
奇妙に思うかもしれない。ip の場合は 1 加えたり引いたりすると, アドレスの値は 4 増えたり減ったりしている。dp の場合は 1 加えたり引いたりすると, アドレスの値は 8 増えたり減ったりしている。

実は, ポインタ演算において, その値を+iするとアドレスは(型のバイト数)×i分動く。これは, 型によってメモリ上に確保される領域の大きさが異なることが理由である。

例えば int 型変数 a のアドレスが 0123 としよう。int 型の変数はふつう 4 バイトであるから、メモリ上には 0123~0126 の領域が変数 a として確保されている。

ip=&a によって ip にはアドレス 0123 が格納される。次に ip + 1 という演算を行うと、アドレスは+4 される。変数 a は 4 つ分のアドレスで一つの値を表現しているため、その領域にかぶらないように、うまくアドレスが移ったと考えればよい。

double 型の場合は 8 バイトなので、dp+1 すると、double 型変数の領域にかぶらないように 8 バイト分移動する。



このように、ポインタの加算/減算の結果は何型へのポインタかによって異なる。だから、ポインタの宣言時に型を明示する必要がある。

しかしこの性質が何の役に立つのかについては、後で述べる。

4. ポインタの応用例

ポインタは変数を「遠隔操作」する、と先に述べた。この遠隔操作が実際に役立つ一例は、関数にポインタを渡す場合である。二つ例を示す。

[swap 関数]

swap は、二つの変数を入れ替える処理である。関数を用いない場合は、例えば次のように書ける。

```
#include <stdio.h>

int main(void)
{
    int x = 1;
    int y = 10;
```

```

int tmp;

printf("before... x: %d, y: %d\n", x, y);

tmp = x;
x = y;
y = tmp;

printf("after... x: %d, y: %d\n", x, y);

return 0;
}

```

tmp という関数に一度 x の値を退避したあと, y の値を x に代入, その後 y には退避しておいたもとの x の値を代入して, 交換が完了する。

さて, この

```

tmp = x;
x = y;
y = tmp;

```

の部分関数化して,

```
swap(x, y)
```

と書けないだろうかと考えてみよう。もしかすると次のような関数が考えられるかもしれない。

```

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

```

実は, これではうまくいかない。なぜなら, swap(x, y)としても, x, y の値が a, b に代入されるだけで, x, y には何の変化も及ぼさないからだ。

x, y の値を外部の関数から書き換えられるようにしたい。そこで, ポインタを用いて以下のように書く。

関数呼び出しがあると
x, yの値がa, bに代入される

```

swap(x, y);

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

```

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

ただし, x, y のアドレスを関数内に渡さなくてはならないため, この関数を呼び出すには swap(&x, &y) と書く。アドレスを渡したため, ポインタ a, b を介して x, y の値を遠隔操作できる。よって, x, y の値は実際に交換できたことになる。

関数呼び出しがあると x, y の アドレス が a, b に 代入 される

```
swap(&x, &y);
```

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

[商と余りを取得する関数]

return 文はひとつの値を返すことしかできない。そこでポインタを利用すれば, return を使わずに 2 つ以上の値を受け取ることができる。

a を b で割った商と余りを取得する関数 divide を次のように定義する。

```
void divide(int a, int b, int *pq, int *pr) {
    *pq = a / b;
    *pr = a % b;
}
```

*pq, *pr がそれぞれ商, 余りを表す。return で値を返す代わりに, 取得したい値をポインタとして仮引数にとっている。実際に関数を呼び出すときは, あらかじめ呼び出し元で商と余りを受け取るための変数を用意しておいて,

```
divide(a, b, &q, &r);
```

と書く。q, r の値を divide を用いて遠隔操作をしているイメージを持つと良い。divide 関数を用いて, 遠隔的に q に a/b を, r に a%b を代入している。

divide 関数を用いたプログラムは次のように書ける。

```
#include <stdio.h>

void divide(int a, int b, int *pq, int *pr) {
    *pq = a / b;
    *pr = a % b;
}
```

```

int main(void)
{
    int a = 16;
    int b = 3;
    int q, r;

    divide(a, b, &q, &r);
    printf("%d %d\n", q, r);

    return 0;
}

```

5. 配列とポインタ

もしかしたら驚くべきことかもしれないが、

配列名は配列の先頭要素へのポインタ

である。つまり、

```
int a[3]
```

という配列が宣言されたとき、`a` そのものはポインタである。しかも `a` には配列の先頭要素のアドレス、つまり `&a[0]`が入っている。

配列名がポインタなら、勿論 `*a` と `a[0]`は同等である。さらに、先に説明した配列のメモリ上での位置、ポインタ演算の話と合わせると次のことが分かる。

$$*(a + i) \Leftrightarrow a[i]$$

実際にそうになっているのかを確認するためのプログラムを作ってみると次のようになる。

```

#include <stdio.h>

int main(void)
{
    int a[3] = {81, 24, 68};

    printf("a      : %p, &a[0]: %p\n", a, &a[0]);
    printf("a + 1: %p, &a[1]: %p\n", a + 1, &a[1]);
    printf("a + 2: %p, &a[2]: %p\n", a + 2, &a[2]);
}

```



```
printf("*a      : %d, a[0]: %d¥n", *a      , a[0]);
printf("*(a + 1): %d, a[1]: %d¥n", *(a + 1), a[1]);
printf("*(a + 2): %d, a[2]: %d¥n¥n", *(a + 2), a[2]);

return 0;
}
```

実行結果は以下のようになる。

```
a      : 0060FEA4, &a[0]: 0060FEA4
a + 1: 0060FEA8, &a[1]: 0060FEA8
a + 2: 0060FEAC, &a[2]: 0060FEAC
*a      : 81, a[0]: 81
*(a + 1): 24, a[1]: 24
*(a + 2): 68, a[2]: 68
```

$a + i$ と $\&a[i]$ は同じアドレスを示し、 $*(a + i)$ と $a[i]$ は同じ値を示している。

メモリ上の位置関係を見てみよう。配列はメモリ上に連続して配置されるのだったから、次のようになるはずである。 a は $\&a[0]$ 、すなわち 0060FEA4 が入っている。ポインタ演算の性質により、 $a + 1$ とすると int 型ではアドレスが 4 つ分動く。つまり $a + 1$ の値は 0060FEA8 となり、 $\&a[1]$ を示していることになる。 $a + 2$ の場合についても同様である。

	アドレス	データ
$*a \Leftrightarrow a[0]$	0060FEA4	81
	(0060FEA5)	
	(0060FEA6)	
	(0060FEA7)	
$*(a + 1) \Leftrightarrow a[1]$	0060FEA8	24
	(0060FEA9)	
	(0060FEAA)	
	(0060FEAB)	
$*(a + 2) \Leftrightarrow a[2]$	0060FEAC	68
	(0060FEAE)	
	(0060FEAF)	
	(0060FEB0)	
	0060FEB1	...
	0060FEB2	...
	0060FEB3	...
	0060FEB4	...

このようにして、ポインタ演算の奇妙な性質によって、配列とポインタはほとんど似たようなものとみなせるようになった。

[補足]

$*(a + i) \Leftrightarrow a[i]$ の \Leftrightarrow の記号は、単なる比喩的なものではなく、本当に数学的な意味での同値関係が成り立つ。一方はもう一方の書き換えに過ぎない。

[補足]

厳密には、配列とポインタは完全に同じものではない。このあたりの詳しい話については割愛する。

6. ポインタの応用例(続き)

ポインタと配列との関係を述べたところで、それを利用した応用例について紹介しよう。

配列を関数に渡したいという状況を考える。実は、配列の値そのものを関数の引数として渡すことは文法的にできない。配列の代わりにポインタを引数として渡す。

[配列の要素を出力する]

以下は、配列の内容を出力する関数 `showArray` を実装したプログラムである。

```
#include <stdio.h>

void showArray(int *a, int n)
{
    int i;
    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main(void)
{
    int x[5] = {2, 4, 6, 8, 10};
    showArray(x, 5);
    return 0;
}
```

配列名は配列の先頭要素へのポインタだったことを思い出そう。すると、

```
showArray(x, 5);
```

において、実引数 `x` は `&x[0]` を表していることになる。よって、仮引数 `a` には `&x[0]` の値がコピーされる。配列はメモリ上に連続して配置されているので、`*a` で `x[0]`, `*(a + 1)` で `x[1]`, `*(a + 2)` で `x[2]`... にアクセスできる。`*(a + i) ⇔ a[i]` であったから、`a[0]` で `x[0]`, `a[1]` で `x[1]`, `a[2]` で `x[2]`... にアクセスできる。

いくつか注意点を述べる。

一つは

```
void showArray(int *a, int n)
```

の部分は、`*a` を `a[]` として


```
void showArray(int a[], int n)
```

と書いても同義である。配列の要素数を間に挟まずに `[]` と書くことに違和感を覚えるかもしれないが、これは C 言語の文法的に許されていることなので仕方ない。要素数を

書いてもよいが、その数字は一切意味を持たない。

二つ目。配列を関数に渡すときは要素数を引数に取ることが多い。a に渡されたものはあくまで配列の先頭要素のアドレスなので、それは渡した配列の要素数の情報を持ってないからである。なので、showArray 関数では要素数を受け取るために整数 n を仮引数としてとっている。

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

a[]と書いても同じ
要素数を渡しておく

三つ目。配列の先頭要素へのアドレスを a に渡しているということは、配列を「遠隔操作」していることになる。なので、もし関数内でポインタの指す値をいじることがあれば、渡した配列の値も書き換わる。これを利用した例については後で述べる。

四つ目。実引数として配列を渡しても、仮引数として受け取ったのはあくまでポインタである。呼び出された関数にとっては、受け取ったのはポインタでしかない。つまり、その関数は受け取ったものが配列であるかどうか分かっていない。

五つ目。いままでは一次元配列を関数に渡すときを例に述べたが、多次元配列を引数にとるときは少し注意しなければならない。多次元配列を引数に取るようなケースはあまりないため、ここでは詳しく述べない。

[文字列を操作する関数]

配列を引数にとるような関数を作るときは、要素数も引数に指定することが多い、と先に述べた。しかし、文字列には要素数の代わりにヌル文字が利用できる。文字列の場合は要素数を指定する必要はない。

例を示そう。W, w を M, m に相互に変換する関数 changeWM は次のように書ける。

```
void changeWM(char p[])
{
    int i;

    for(i = 0; p[i] != '¥0'; i++) {
        if(p[i] == 'W') p[i] = 'M';
        else if(p[i] == 'M') p[i] = 'W';
    }
}
```

```

        else if(p[i] == 'w') p[i] = 'm';
        else if(p[i] == 'm') p[i] = 'w';
    }
}

```

文字列をポインタとして受け取ったら, p[i]の形で渡した配列にアクセスできる。p[i]がヌル文字¥0になるまで, for ループを行っている。

受け取った p は配列ではなくポインタでしかないことに注目すると, 以下のように changeWM を簡潔に書くことができる。

```

void changeWM(char *p)
{
    while(*p != '¥0') {
        if(*p == 'W') *p = 'M';
        else if(*p == 'M') *p = 'W';
        else if(*p == 'w') *p = 'm';
        else if(*p == 'm') *p = 'w';
        p++;
    }
}

```

前のコードでは, i の値を配列の添え字としていたが, 今回のコードでは, p の値を直接インクリメントしている。変数 i を宣言する必要がなくなり, プログラムがやや簡潔になった。

ただし, プログラムが簡潔になることと, そのコードが分かりやすくなることは別である。今回の changeWM の実装例は単純なので読みやすい。しかし一般的に, 簡潔だからといってポインタを濫用したプログラムを書くと, コードが非常に読みづらくなる。ただ, ポインタをうまく使った書き方があることは知っていて, それが読めるようにはなるべきである。

これを踏まえて, 入力文字列の W, w を M, m に相互に変換して出力するプログラムは次のように書ける。

```

#include <stdio.h>

void changeWM(char *p)
{
    while(*p != '¥0') {
        if(*p == 'W') *p = 'M';
        else if(*p == 'M') *p = 'W';
        else if(*p == 'w') *p = 'm';
    }
}

```

```

        else if(*p == 'm') *p = 'w';
        p++;
    }
}

int main(void)
{
    char str[1000];

    scanf("%s", str);
    changeWM(str);
    printf("%s¥n", str);

    return 0;
}

```

実行結果は以下のようになる。

Hello, World! (入力)
Hello, Morld!
Don' tworry. (入力)
Don' tmorry.
wombat (入力)
mowbat

最後に、長く使ってきた scanf について少し述べよう。

scanf 関数を利用するとき、変数名には&をつけて、文字列名には&はつけない、と第 1 回で述べた。この理由についていままで保留にしていた。

変数名につけていた&は、アドレスである。例えば、

```
scanf("%d", &x);
```

という記述があったとき、scanf 関数には x のアドレスを渡していたのである。ポインタを利用して、キーボードからの入力値を x の値に遠隔的に入れていたのである。

文字列の場合も同様である。

```
scanf("%d", str);
```

という記述があったとする。文字列 str はあくまで char 型の配列に過ぎないため、str そのものは配列 str の先頭要素へのポインタを意味している。結局&x も str も、どちらもアドレスを scanf に渡していたということになる。

7. 構造体へのポインタ

構造体へのポインタも、いままでのポインタと同様の使い方ができる。

前回に作った構造体 Human を例にして説明しよう。構造体 Human 型へのポインタは次のように宣言できる。

```
struct Human *p;
```

struct Human 型の変数 a のアドレスを p に代入するときは, p=&a のように書く。p の指すデータのメンバにアクセスするためには, (*p).メンバ のように記述する。

```
p = &a;
```

```
(*p).x = 2;
```

```
(*p).y = 10;
```

実は, (*p).メンバ のような書き方の省略として, p->メンバ という書き方がある。->のことをアロー演算子という。省略記法というよりも, 「ポインタの指すデータのメンバにアクセスするための演算子」として覚えておいたほうが良い。

```
p->x = 2;
```

```
p->y = 10;
```

8. 終わりに

競技プログラミングにおいては, ポインタをほとんど使わない。競プロの世界においてはポインタを覚えておく必要はないのだが, 教養としてポインタの知識を持っておいて損はない。また, 学校の授業では扱うので一応講習会で説明するに至った。