Chapter 3

ポインタと関数

3.1　ポインタは苦手ですか？

「ポインタ」と聞いて「えーっ」と引いた君。気持ちはわからなくもないが、とても

もったいない

とも思う。

なぜか。

いわゆる高水準言語（ハードウェアに近いのが低水準言語、人間の言葉・考え方に近い書き方ができるのが高水準言語）であるのだから、「メモリ上のアドレス」なんていうハードウェア的なことばが顔をだすと違和感を感じるかもしれない。これは、確かにその通りではある。高水準言語は車にたとえていえば、オートマ車。ギア操作(=ハードウェアの操作）を全自動でやってくれる。楽ちんである。

では、その「メモリ上のアドレス」が無用のものかというと、そうでもない。再び車にたとえていえば、マニュアル車のように運転者（＝プログラマ）の意のままにギア操作やアクセルワークに対してのレスポンス(=パフォーマンス）を発揮できる仕組みである。使いこなせれば武器になるのだ。だから、使わないともったいない。

ついでに言うと、このあたりがわかっているとJavaなど他の言語を扱うときにも入りやすい。

たとえ話はさておき

まずは、ポインタの仕組みからおさらいしよう。int型のポインタ変数の宣言およびその初期化は次の通りだった。

```
int *x;
x=(int *)malloc(sizeof(int));
*x=10;
```

ポインタに慣れていればなんてことはないが、苦手な口もいることとおもうので、あえて日本語に翻訳すると

1. int型のデータを置く場所(メモリ上のアドレス)をxと名付けよう
2. xにちょうどint型のデータが入るサイズ(sizeof(int))の領域をメモリ上にとっておこう
3. xという置き場に10という値をいれておこう

となるわけだ。
「いや、まてよ。それだけなら

```
int x=10;
```

と何が違うの？わざわざメモリに領域なんて明示的につくらなくったって、10という値を持っておけているじゃないか[1]。わざわざmallocなんていう関数[2]を持ち出して、何がしたいんだよ？」
ポインタの存在意義についての

疑問はここにある

のではなかろうか？
まず、ポインタは、

メモリ上のデータの置き場（=アドレス）を変数として扱える

[1] しかも「参照」を使って$\&a$なんてすると、値が格納されているメモリの番地もとれるしさ。
[2] あ、malloc関数を使うときには、stdlib.hをインクルードすることをお忘れなきよう。あと、例のようにキャストをするのも忘れずに。

点が重要であることを思い出そう。[3]。普通の変数の「参照」である$\&a$は確かに変数が格納されているメモリ上の番地（アドレス）を与えるが、たとえば、変数xのメモリ上のアドレスを変数aの番地に置き換えることは残念ながらできない。つまり、変数xと変数aのデータの置き場をおなじところにしようと思って

```
int x;
int a=10;
&x=&a;
```

なんてことをしようとおもっても出来ないのである[4]。しかし、ポインタを使えばもちろん、

```
int *x;
int a=10;
x=&a;
```

なんてことができる[5]。

ここで注意すべきは、ポインタは「単に変数のアドレスを持っているだけの変数」ではないということだ。メモリ領域を割り当てているので、その領域のサイズについての情報も持っているのであることを忘れてはならない。これを忘れるとエラー[6]が発生するか誤動作することが落ちである。これは、毎年、みんな引っかかるところである。気をつけよう。

ぴんとこない？そりゃそうだ。この威力を発揮する場面をみたいなら、一つの変数分の領域ぐらいじゃ不足だ。ここで、例えば、C言語の配列とポインタの関係を思いだそう。次の二つのコードは全く同じことを表している。

```
int[100] z;
for(i=0; i<100; i++){
        z[i]=0;
}
```

[3]当たり前すぎる？

[4]実際、linuxのgccでこんなコードをコンパイルしようとすると「エラー: 代入の左側の被演算子として左辺値が必要です」などといってコンパイラにおこられてしまう。

[5]当然、このあと$*x$の値を取得すると10が得られる。

[6]segmentation fault　（セグメントエラー）が起こるに違いない。

```
int *z=(int *)malloc(sizeof(int)*100);
for(i=0; i<100; i++){
         *(z+i)=0;
}
```

であるので、配列をポインタを使って表してみよう。いま、次のように用意される二つの配列

```
int d[1000] ,g[1000];
for(i=0; i<1000; i++){
     d[i]=i+1;
     g[i]=1000-i;
}
```

を入れ替えることを考えよう。ポインタを使ってこれを書き換えると

```
int *d=(int *)malloc(sizeof(int)*1000);
int *g=(int *)malloc(sizeof(int)*1000);
for(i=0; i<1000; i++){
     *(d+i)=i+1;
     *(g+i)=1000-i;
}
```

となるだろう。入れ替え方法をもっとも単純に考えると

```
int k;
for(i=0; i<1000; i++){
     k=*(d+i);
     *(d+i)=*(g+i);
    *(g+i)=k;
}
```

というところだろう。これは、片っ端から対応する各要素を入れ替えることに相当する。しかし、実際には次のほうが同じことが素早く行える。

```
     int *h;
     h=d;
     d=g;
     g=h;
```

これは結局、配列の要素の型、要素数が同じであれば、配列のありかの情報を交換してやれば、いやもっと端的にいえば、（この例でいえば、dやらgという）ポインタが持つ（指す）配列の（先頭の）アドレスを値を取り替えてやればよい、というわけである。配列の要素という「実物」をわっせわっせと入れ替えるよりも、配列のありか（アドレス）だけをいれかえてやるだけですめば、処理は早くおわる。当然だね[7]。

これ以外にもポインタがおおいに役立つ場合がある。それは、関数の引数にもちいるときである。関数はご存じの通り、戻り値（返り値ともいう）はひとつしか返せない。が、実用上、複数の値を返したいことがある。例えば、小学校のときにおなじみの「つるかめ算」をする関数を定義しよう。「つるかめ算」は、鶴と亀の頭数($heads$)と脚数($feet$)が与えられた時、鶴が何羽、亀が何匹いるかを計算するんだった。そうすると、そのプログラムは次のようになるであろう。

```
void turukame(int feet, int heads, int *turu, int *kame){
        *kame=(feet-2*heads)/2;
        *turu=heads-kame;
}
```

関数の出力として、鶴の羽数($turu$)や亀の匹数($kame$)をセットで返したいときには、このような方法をとることがもっぱらである。こんなとき、関数の戻り値（上の例ではvoidになっているが）は、処理がうまくいったときに1、うまくいかなかったときに0を返すように利用されることが多い。引数にポインタ変数を利用するということは、メモリ上の値の置き場（アドレス）を指定し、かつ関数外部と共有していることになるので、この置き場を経由して関数外部は計算結果をうけとることが出来るわけだ[8]。C言語を使ったプログラムを作るときに、これはとてもよく使う。この場でぜひ覚えてしまってほしい。

では、演習に移ろう。また、次回（第3回）までに、セクション3.3まで完成させ、セクション3.2〜3.3のソースコード、出力、苦労した点をレポートとしてまとめ提出せよ。次回の授業開始時にレポートを集めるので、開始までにはレポートを仕上げておき、提出できるようにしておくこと。

[7]言い換えれば名前を付け替えているだけとも言える。

[8]そうすることで、Pascalのプロシージャ(procedure) やBasicのサブルーチン(subroutine)と同様のことができることになる。

3.2 ポインタを使って配列を入れ替えてみる

配列$array1$および$array2$を以下のように定義する。これを次のように定義する。

- 配列$array1$は、奇数の列とする。すなわち、各要素は$array1[i] = 2 * i - 1$となるようにする。要素数は10000とする。
- 配列$array2$は、偶数の列とする。すなわち、各要素は$array2[i] = 2 * i$となるようにする。要素数は10000とする。

これを使って、以下の各プログラムを作成し、実行時間を比較せよ。

1. 「$array1$および$array2$の各要素を素直に全ていれかえる」ことを2000回行うプログラムを作成せよ。入れ替えたあとに、「終了」という文字列を画面に表示するものとする。(prog2-1a.c)　コンパイル後の実行プログラムはprog2-1aとすること。(a.outのままにしない。以下、同様)
2. 「$array1$および$array2$を配列の先頭アドレスだけを使って配列をいれかえる」ことを2000回行うプログラムを作成せよ。入れ替えたあとに、「終了」という文字列を画面に表示するものとする。(prog2-1b.c)　コンパイル後の実行プログラムはprog2-1bとすること。
3. できあがったプログラムをコンパイルし、実行時間を計測せよ（計測前に要素の値が正しく入れ替わっているかはあらかじめ確認しておくこと)。時間の計測は、Linuxのtimeコマンドを利用すること[9]。timeコマンドはreal、user、sysと三種類の時間を画面に出力するが、実行時間はそれらの総和としてよい。prog2-1aとprog2-1bの実行時間を実測して比較せよ。

3.3 関数の引数としてポインタを使う

1. 行列$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$とベクトル$\begin{pmatrix} p \\ q \end{pmatrix}$が与えられているとして、次の線型一次方程式を解く関数$solve$を作成せよ。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix} \tag{3.1}$$

[9] tcshシェルを使っている場合、/usr/bin/time -p “コマンド”　を使おう。

ただし、*solve*の引数には、double型のa、b、c、d、p、qが含まれるとし、double型のxおよびyの値も引数を通して返されるものとする。なお、行列式が0のときには、画面に「det=0」と表示されることとし、関数の戻り値として0を返す。行列式は0出ない場合は、戻り値は1とする。なお、関数*solve*の戻り値が1の場合、*main*関数では「$x = \cdots, y = \cdots$」という形で解を出力するものとする。レポートには

$$\begin{pmatrix} 1 & -1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 7 \end{pmatrix} \tag{3.2}$$

を解いた場合の出力をのせよ。