12 ポインタ

Newton-Raphson 法については今回説明するため、前回の課題の締切は 6/6(日)に変更。

実習 12.1. int 型変数 2 つの内容を交換する関数を作りたい。しかし、下の例の関数 xswap() ではうまく動作しないことを確かめよ。

動作確認中は、xswap() の中のコメントアウト部を適宜活かして確かめよ。

```
/*   xswap.c   2010-05-31   */
#include <stdio.h>

void xswap( int, int );

int main(int argc, char** argv)
{
    int x = 2010, y = 531;

    printf("x = %d, y = %d\n", x, y);
    xswap(x, y);
    printf("x = %d, y = %d\n", x, y);
}

void xswap( int a, int b )
{
    int c;

/*
    printf("a = %d, b = %d\n", a, b);
*/
    c = a;
    a = b;
    b = c;
/*
    printf("a = %d, b = %d\n", a, b);
*/

    return;
}
```

考察 12.1.1. 何故これではうまくいかないのか。

「そりゃそうだ」とも言える。だからこそ、関数内で心配なく引数の値を変更して良いのであった。

考察 12.1.2. 関数を呼び出すことによって 2 つの変数の値を交換するためには、関数の引数として何を渡せば良いか。

このようなことを考える為に、「変数」が計算機上で実際にどう扱われているかを改めて見てみよう。

12–1 ビットとバイト

よく「計算機は 0 と 1 しか判らない」という言い方をするが、これは電気的には「On か Off か」(正確には High か Low か) で判断していることに依存している。この「0 か 1 か」(「On か Off か」) という状態を 1 つ記憶しておく単位が計算機が扱える最小の単位 (というか情報量の最小単位) であり、**1 ビット (bit)** と言う。

電位が閾値より高ければ High、低ければ Low。

計算機にとっての最小単位はこのビットだが、対してそれを使う人間にとっての最小単位は「文字」であると考えると、まずは人間にとって必要な「文字」を計算機が扱えるようにしなければならない。「文字」が多数あるのに対し、1 ビットでは 2 種類の状態しか表現できないが、複数のビットを一組として考えれば、そのビット数に応じて多くの種類のデータを表現できる。現在の多くの計算機では 8 ビットを一組としており、この 8 ビットを **1 バイト (byte)** と言う。従って 1 バイトで記憶できる情報は $2^8 = 256$ 通り。アルファベットや数字などのみであれば 1 バイトで 1 文字を表せるが、日本語など文字の種類の多い言語は 2 バイト以上を用いて 1 文字を表現する (多バイト文字 (multi-byte character) という)。

bit = BInary digiT

CPU(Central Prrocessing Unit, 中央演算装置) は、一度に扱える (一回に計算できる) 情報量によって「n bit CPU」($n = 8, 16, 32, 64$ など) と呼ばれる。又、メモリ・ハードディスク・フロッピーディスク・CD・DVD などの記憶媒体の容量は、通常バイトを単位として、K(Kilo, キロ)・M(Mega, メガ)・G(Giga, ギガ)・T(Tera, テラ) などの接頭辞を用いて呼ぶ。

正確には、
1K バイト=1024($= 2^{10}$) バイト、
1M バイト=1024K バイト、
1G バイト=1024M バイト、
1T バイト=1024G バイト、
だが、1000 で計算してある場合もある。

ここに限らず、計算機の話では 2 の冪の値が頻繁に用いられる。2^{16} くらいまでの値はすぐ出て来ると良かろう。

12–2　メモリ

計算機には主記憶装置としてメモリ (Memory) が入っていて、磁気的にビット情報 (「0 か 1 か」) を記憶する。メモリには 1 バイトごとに通し番号が振られていて、その番号によって管理される。この番号を**アドレス (address)** 又は日本語で**番地**と言う。

12–3　変数とメモリ

C 言語に限らず、多くのプログラミング言語においては「変数」を用い、それにより数値や式の値を代入したり、逆にその値を参照したりする。変数の型により、その必要とするメモリの大きさが定められている。

さて、現行の計算機の多くでは、ディスク上にあるプログラムは実行時にまずメモリ上に読み込まれ、動き出すと必要な変数のための領域などもやはり同じメモリ上に割り当てられる。(**プログラム記憶方式・von Neumann 方式**という。) プログラムの実行が開始されると変数のテーブル (一覧表) が作成され、実行中に変数が宣言されると、まず必要な分のメモリ領域が確保され、その先頭アドレスや型、変数の有効域 (スコープ) などの情報が記録される。その変数の値の参照・代入では、まず変数のテーブルに問い合わせに行き、そこから得た情報を元に該当するメモリへのアクセス (読込・書込) を行う。

一般にはその大きさは処理系に依存するが、例えば上智のメディアセンターの Unix では、`int` は 4 バイト、`double` は 8 バイト など。
ということは、`int` 型変数が扱える数値の範囲は?

12–4　アドレス演算子 `&` と `sizeof` 演算子

変数はその型によって定められた複数のバイトを組にして一つの値を表すので、メモリ上の変数にアクセスするためには、**先頭アドレス**とその変数の**型、または大きさ**との組が必要となる。C 言語では、或る変数に対してそれが割り当てられているメモリのアドレスや大きさの情報を得るための演算子が用意されている。

変数が割り当てられたメモリの先頭アドレスを得るには、**アドレス演算子** `&` を用いる。変数 `a` に対して `&a` の値、つまり変数 `a` に割り当てられたメモリ上の領域の先頭アドレスを `a` **へのポインタ (pointer)** と言う。

また、その変数のための領域の大きさを得るには `sizeof` **演算子**を用いる (値は `int` 型)。

pointer: 指し示すもの
ポインタは単にアドレス (番地) の値だけでなく、その指し示す変数の型を含んだ概念であることに注意しよう。

実習 12.2. int 型・double 型の変数に対して、アドレス演算子 & や sizeof 演算子を使ってみる。

```
/*   addr.c   2010-05-31   */
#include <stdio.h>

int main(int argc, char** argv)
{
    int i = 2010;
    double d = 5.31;

    printf("i=%d, d=%f\n", i, d);
    printf("&i=%p, &d=%p\n", &i, &d);
    printf("sizeof(i)=%d, sizeof(d)=%d\n", sizeof(i), sizeof(d));
}
```

printf() でポインタの値 (変数の先頭アドレス) を表示するには変換文字 %p を用いる (表示は処理系依存。ここのシステムでの表示は 16 進数)。実際に用いるのは動作確認 (や仕組みの勉強!!) などのみであろう。

処理系に依存しない移植性の高いプログラムを書くには、sizeof 演算子はしばしば必要である。

実行例 . &i, &d の値は各人毎、或は実行時毎に異なるかも知れない。

```
=> cc -o addr addr.c
=> ./addr
i=2010, d=5.310000
&i=ffbffba0, &d=ffbffb98
sizeof(i)=4, sizeof(d)=8
=>
```

12–5 ポインタ型変数・間接演算子 *

変数の先頭アドレス、即ち変数へのポインタを値として持つ「ポインタ型」の変数を用いることが出来る。ポインタ型変数の値はアドレスであるが、変数に正しくアクセスするには、同時に型 (大きさ) の情報も必要である。従って、ポインタ型変数は指し示す変数の型を指定して、「**〜型変数へのポインタ**」として宣言される。

ポインタ型変数の宣言の書式は次の通り。

型名　*変数名;

実習 12.3. ポインタ型変数や間接演算子 (間接参照演算子) * を使ってみる。

```
/*   ptr1.c   2010-05-31   */
#include <stdio.h>

int main(int argc, char** argv)
{
    int i=2010, j=531;
    int *p;

    printf("&i = %p, &j = %p, &p = %p\n", &i, &j, &p);

    p = &i;
    printf("p = %p, *p = %d\n", p, *p);

    p = &j;
    printf("p = %p, *p = %d\n", p, *p);

    *p = i;
    printf("j = %d\n", j);
}
```

まず int *p; によって p が「int 型変数へのポインタ」型の変数 (今後は単に「int へのポインタ」のように言う) として宣言されている。p は「int 型変数の先頭アドレスを値をして持つ変数」であるので、int 型変数 i や j の先頭アドレス &i, &j を値をして代入することが出来る。

ポインタ変数が指し示している (その値であるアドレスを先頭とする変数の) 内容を値として得るには**間接演算子 (間接参照演算子・逆参照演算子)** * を用いる。間接演算子 * は代入式の左辺に用いることも出来る。

```
*p = i;
```

で、ポインタ変数 p が指し示している内容 (その値であるアドレスを先頭とする変数の値) に i の値を代入。

考察 12.3.1. 変数 j の値はどこで変わったのか。

例えば、
int *p;
で、「int 型へのポインタ」という型を持つ変数 p が宣言される。

int *p という宣言の表記は (int *) p、即ち、int * 型の変数 p と見た方が判り易いかも。しかし、同じ型のポインタ変数を複数まとめて宣言する場合は、
int *p, *q;
としなくてはいけないことに注意。逆に、
int a, *p;
で、int 型変数 a と int *型変数 p との宣言がまとめて出来る。

判り難ければ、途中の i や j の値も適宜表示させてみよ。

int *p; と宣言したら、
「*p と書いたら int 型の式になる」
と思うと判り易いかも。

実習 12.4. ポインタ型変数を引数に取る関数を作成し、変数のポインタを引数として渡す例。

```
/*   ptr2.c   2010-05-31   */
#include <stdio.h>

void xincr( int );
void incr( int * );

int main(int argc, char** argv)
{
    int i = 2010;
/*
    printf("&i=%p\n", &i);
*/
    printf("before: i = %d\n", i);

    xincr(i);
    printf("after xincr: i = %d\n", i);

    incr(&i);
    printf("after incr: i = %d\n", i);
}
```

```
void xincr( int n )
{
/*
    printf("&n=%p\n", &n);
*/
    n++;

    return;
}

void incr( int *p )
{
/*
    printf(" p=%p\n", p);
*/
    (*p)++;

    return;
}
```

考察 12.4.1. xincr() と incr() でそれぞれ何が起きているのかを考えよ。実行後に main() で宣言された変数 i の値が変化しているのはどちらか。

incr() 内の (*p)++; の優先順位に注意。

今回は新たな課題無し。今までの課題提出が滞っている人は、この機会に追い付くべし。