C 言語のサンプルプログラム (3)

計算機応用　5E 2003.05.08

1．データ型

1.1　データ型とは何か

C 言語では、変数は定義してから用いなくてはなりません。FORTRAN のように、暗黙の型宣言はありません。型を指定することにより、変数を定義できます。これは、コンパイラーがプログラムを実行するときに、必要な領域を確保するためです。

型と言っても、大きく分けると、

- 文字型
- 整数型
- 浮動小数型

しかありません。FORTRAN にあった複素数型が無いのは、非常に残念です。数値計算を行う上で、この複素数型は良く利用するのですが、C 言語にはありません。そのため、複素数を使うときには、工夫が必要です。また、C 言語には論理型もありません。

変数の定義は、今までのサンプルプログラムにあるように、

```
#include <stdio.h>
main(){
  char aa; bb; cc;
  int u1, v2b,wwc;
  double x, y, z;


}
```

のように、データの型を書き、そのあとに変数名を書きます。簡単ですね。

1.2 データ型のサイズ

それでは、データ型のサイズを調べて見ましょう。ANSI 規格では、型のサイズは決められていないものもあります。コンパイラーに依存します。一般的には、その CPU がもっとも効率よくデータを処理するサイズになっています。

サンプルプログラム(datasize.c)

```
#include <stdio.h>
main(){

   printf("             char   %d\n",sizeof(char));
   printf("      signed char   %d\n",sizeof(signed char));
   printf("    unsigned char   %d\n",sizeof(unsigned char));

   printf("              int   %d\n",sizeof(int));
   printf("       signed int   %d\n",sizeof(signed int));
   printf("     unsigned int   %d\n",sizeof(unsigned int));
   printf("        short int   %d\n",sizeof(short int));
   printf("  signed short int   %d\n",sizeof(signed short int));
   printf("unsigned short int   %d\n",sizeof(unsigned short int));
   printf("         long int   %d\n",sizeof(long int));
   printf("   signed long int   %d\n",sizeof(signed long int));
   printf(" unsigned long int   %d\n",sizeof(unsigned long int));

   printf("            float   %d\n",sizeof(float));
   printf("           double   %d\n",sizeof(double));
   printf("      long double   %d\n",sizeof(long double));

}
```

実行結果

以下に実行結果を示します。その横に、呼称を示します。これら、いろいろな型がありますが、通常使うのは、char と int、double です。これ以外を使うことは、希です。

型	バイト数	呼称
char	1	文字型
signed char	1	同上
unsigned char	1	符号なし文字型
int	4	整数型
signed int	4	同上
unsigned int	4	符号なし整数型
short int	2	短長整数型
signed short int	2	同上
unsigned short int	2	符号なし短長整数型
long int	4	倍長整数型
signed long int	4	上同
unsigned long int	4	符号なし倍長整数型
float	4	実数型
double	8	倍精度実数型
long double	8	拡張倍精度実数型

1.3 変数の適用範囲

C 言語では、変数の適用範囲は、厳密に決められています。FORTRAN の場合、COMMON 文やら関数の引数の受け渡しなど、変数の適用範囲が思わぬところで広がることがあります。

その点、C 言語では、関数内で定義された変数は、定義された関数内でしか利用できないという制限があります。関数の外で定義された変数は、他の関数からも参照できます。C 言語の詳細な仕様を見ると、いろいろな定義の方法があり、かなり柔軟に変数を使うことが出来ます。しかし、ここで我々が勉強する上で、必要なことは、次の 2 つだけです。

自動変数　関数の中で定義され、その関数の中だけで使用できる。関数がコールされるとメモリー上に変数が配置される。その関数の処理が終わるとその変数は消滅する。

外部変数　関数の外で定義され、どの関数でも使用できる。プログラムが起動されるとメモリー上に変数が配置される。プログラムが終了するまで、変数は維持される。

次ページにサンプルプログラムを示します。3 行目の関数外で定義されている ev が、外部変数です。したがって、main 関数からでも、func 関数からでも参照できます。

main 関数や、func 関数で定義されている iv は自動関数で、各々の関数の中でしか使えません。同じ名前ですが、異なるメモリー領域にデータは格納されています。

2 行目の

```
void func(void) ;
```

は、func 関数のプロトタイプ宣言です。戻り値と引数の型を宣言しています。最初の void が戻り値がなし、括弧内の void は引数なしと言う意味です。

例えば、戻り値が int で、引数が double と int がそれぞれ 1 個の関数 hogehoge のプロトタイプ宣言は、

```
int hogehoge(double a, int b);
```

となります。

サンプルプログラム(scope.c)

```
#include <stdio.h>
void func(void);
int ev;
/* ====================================*/
/* =   main                           */
/* ====================================*/
main(){
   int iv;

   iv = 11111;
   ev = 22222;

   printf("iv = %d   ev = %d\n",iv,ev);

   func();
   printf("iv = %d   ev = %d\n",iv,ev);

}

/* ====================================*/
/* =   func                           */
/* ====================================*/
void func(void){
   int iv;

   iv = 33333;

   printf("iv = %d   ev = %d\n",iv,ev);

   ev = 44444;

}
```

実行結果

```
iv = 11111   ev = 22222
iv = 33333   ev = 22222
iv = 11111   ev = 44444
```

2．ポインター

2.1　ポインターとは何か

今までのサンプルプログラムは、FORTRAN と置き換えが可能です。対応する FORTRAN の命令があり、理解は容易でしょう。しかし、ここで説明するポインター(pointer)は FORTRAN にない概念です。これこそ、C と FORTRAN の大きな違いで、C 言語のもっとも大きな特徴です。そのため、C 言語の勉強で、ここで挫折する人が多く居ます。みなさん、がんばって勉強してください。覚えることなど無く、その内容を理解すれば、ポインターなんか難しくありません。

普通の変数、サンプルプログラムの v1, v2, v3 は、整数を格納する変数です。コンパイラーが確保した領域に整数を格納します。コンピューターのメモリには、すべて、アドレス(番地)がついています。各アドレスには、1 バイト(=8 ビット)のデータが対応しています。このアドレスに従い、メモリー内のデータや命令を参照しています。プルグラムの実行時は、v1 等の変数が呼ばれると、それに対応したアドレスが呼ばれ、データの参照をします。

これら普通の変数のアドレスを参照したいときは、&v1 のように、変数の前に & をつけます。これで、その変数の先頭アドレスを直接見ることが出来ます。例えば、int 型の変数であれば、4 バイト[1]のメモリー領域が必要なので、連続した 4 つのアドレスのメモリー領域を使用します。その先頭アドレスを&v1 のようにして参照します。

定義	`int v1;`
データの参照	`v1`
アドレスの参照	`&v1`

一方、ポインター、サンプルプログラムの p1,p2,p3 はアドレスを表す変数です。通常の変数とは異なり、コンパイラーにより確保された領域にアドレスを格納します。アドレス、即ち、値の場所を示すものだから、ポインターといいます。ポインターもアドレスというデータを格納するため、連続したメモリー領域が必要です。秋田高専の IBM の AIX は 32 ビットアドレッシングのため、4 バイトのメモリー領域が必要です。

プログラム内で*p1 のように呼ばれると、p1 が示しているアドレスの値を参照します。p1 は先頭アドレスを示しているので、型に応じたバイト数を呼び出すことになります。そのため、ポインターにも、型が必要になります。

プログラム内で p1 と呼ぶと、そのポインターが示しているアドレスを参照します。また、&p1 と呼び出すと、ポインターの先頭アドレスを参照します。

定義	`int *p1;`
ポインターが示しているデータの参照	`*p1`
ポインターが示してるアドレスの参照	`p1`
ポインターのアドレスの参照	`&p1`

ポインターに関係する演算子を下表に示します。

演算子	機能
*	ポインターが指しているアドレスの内容を取り出す
&	変数が格納されているアドレスを取り出す

例えば、ポインターの勉強のために、次のサンプルプログラムを実行しましょう。実行結果と、メモリーの内容を次ページ以降に示します。

---

[1] int が 2 バイトの場合もある。int のバイト数は、処理系に依存する。秋田高専にある IBM の AIX の場合、int は 4 バイトである。

サンプルプログラム(ptrtest.c)

```
#include <stdio.h>
main()
{
  int v1, v2, v3, *p1, *p2, *p3;
  char *ip;
  int i;

  v1=-1;
  v2=2;
  v3=3;

  p1=&v1;
  p2=&v2;
  p3=&v3;

  printf("\n");

  printf("\n ----- address hexadecimal -------\n");

  printf("&v1=%x\n",&v1);
  printf("&v2=%x\n",&v2);
  printf("&v3=%x\n",&v3);
  printf("&p1=%x\n",&p1);
  printf("&p2=%x\n",&p2);
  printf("&p3=%x\n",&p3);

  printf("\n ----- value decimal -------\n");

  printf(" v1=%d\n",v1);
  printf(" v2=%d\n",v2);
  printf(" v3=%d\n",v3);
  printf("*p1=%d\n",*p1);
  printf("*p2=%d\n",*p2);
  printf("*p3=%d\n",*p3);

  printf("\n ----- value hexadecimal -------\n");

  printf(" v1=%x\n",v1);
  printf(" v2=%x\n",v2);
  printf(" v3=%x\n",v3);
  printf("*p1=%x\n",*p1);
  printf("*p2=%x\n",*p2);
  printf("*p3=%x\n",*p3);
```

```
        printf("\n ----- pointer hexadecimal -------\n");

        printf("p1=%x\n",p1);
        printf("p2=%x\n",p2);
        printf("p3=%x\n",p3);

        printf("\n ---------------------------------\n");
        printf("\n   address    data\n\n");

        for(i=0;i<=(int)&p3-(int)&v1+3;i++){
             ip = (char *)&v1+i;
             printf("   %08x   %02x\n",ip,*ip);
        }

        printf("\n");

      }
```

実行結果

以下に実行結果を示します。アドレスの絶対値は、異なっているかもしれませんが、アドレス、ポインター、データの関係は、同じはずです。次ページのメモリー内容と比較して、ポインターを理解してください。

```
  ----- address hexadecimal -------    ---------------------------------
&v1=2ff220b0
&v2=2ff220b4                            address    data
&v3=2ff220b8
&p1=2ff220bc                            2ff220b0   ff
&p2=2ff220c0                            2ff220b1   ff
&p3=2ff220c4                            2ff220b2   ff
                                        2ff220b3   ff
 ----- value decimal -------            2ff220b4   00
 v1=-1                                  2ff220b5   00
 v2=2                                   2ff220b6   00
 v3=3                                   2ff220b7   02
*p1=-1                                  2ff220b8   00
*p2=2                                   2ff220b9   00
*p3=3                                   2ff220ba   00
                                        2ff220bb   03
 ----- value hexadecimal -------       2ff220bc   2f
 v1=ffffffff                            2ff220bd   f2
 v2=2                                   2ff220be   20
 v3=3                                   2ff220bf   b0
*p1=ffffffff                            2ff220c0   2f
*p2=2                                   2ff220c1   f2
*p3=3                                   2ff220c2   20
                                        2ff220c3   b4
 ----- pointer hexadecimal -------     2ff220c4   2f
p1=2ff220b0                             2ff220c5   f2
p2=2ff220b4                             2ff220c6   20
p3=2ff220b8                             2ff220c7   b8
```

メモリーの内容

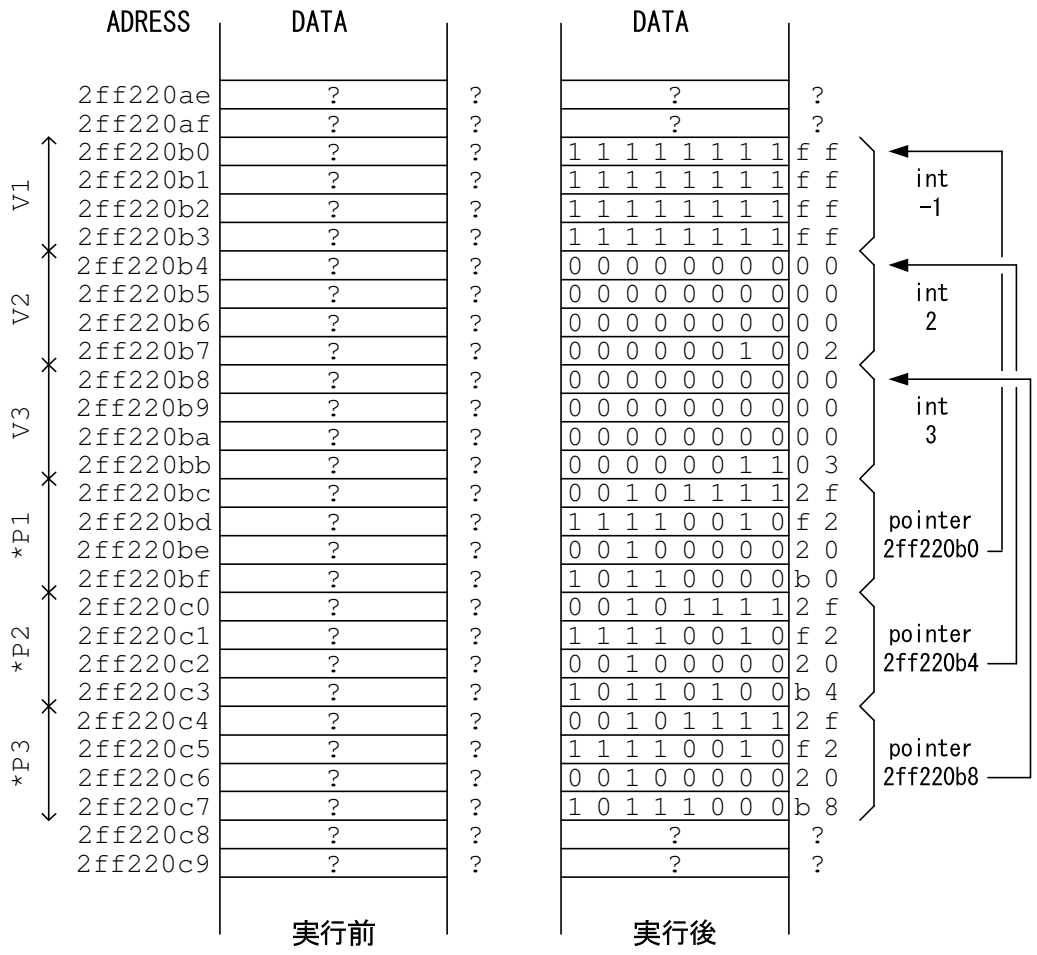

図 1 実行前後のメモリーの内容。実行前は、メモリーのデータは不定です。アドレスは、相対的な場合と絶対的な場合があります。どちらを使うかは処理系できまります。

2.2 ポインターへの数値の代入

ポインターは、アドレスを表すため、ある変数のアドレスを代入して使います。そして、ポインターを通して、その変数の値を変更することができます。

したがって、ポインターが示す正しいアドレスが決まらないうちに、数値を代入するようなプログラムは、問題があります。次のプログラムの場合、実行開始時、ポインターの示すアドレスは不明で、不適切な値です。不適切なアドレスに数値を代入しようとしているため、実行時に暴走するか、エラーを出します。

```
#include <stdio.h>
main()
{
  int v, *p;

  v=1;
  *p=1;

}
```

実際にこのプログラムを、コンパイルして、実行すると、"Segmentation fault"というエラーが出ます。多分、プログラムが許されていないメモリー領域に書き込みを試みたためのエラーです。

では、ポインターに数値を代入するどのようにするかというと、次のようにします。まず、ポインターの示すアドレスを決めてから、それに数値を代入します。そうすると、ポインターの示すアドレスに格納されたデータが書き換わります。

次のサンプルプログラムでは、5 行目の&v で v のアドレスを参照して、そのアドレスをポインターp に代入しています。したがって、8 行目で*p=2 でポインターp が示すアドレス(=変数 v のアドレス)のデータの値を変更しているため、v の値が変化しています。

サンプルプログラム(substitute.c)

```
main()
{
  int v, *p;

  p=&v;

  v=1;
  *p=2;

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

}
```

実行結果

```
v = 2    *p = 2
```

2.3 ポインターへの文字の代入

ポインターへ数値を代入する場合、ポインターの示すアドレスを決めてからでないと、実行時にエラーが出ます。しかし、文字の場合は、別です。文字の場合、それらはメモリーのどこかにかかれます。そして、代入するときにその先頭番地が返されるため、直接代入が出来るように見えます。

次のサンプルプログラムの 6 行目の`abcdef`で、文字を格納する領域を確保して、その先頭のアドレスを戻しています。そのアドレスは、ポインターp へ代入されています。格納される文字は、a〜f までの 6 文字と文字の区切りを表す\0 の 7 文字です。格納された文字を書式付出力の文字列書式 %s で出力しています。%s に対するデータはアドレスを示します。そして、それは\0 まで出力します。

10 行目からは、文字が格納されているアドレスとそのデータを出力しています。最後に文字列区切りとして、\0 (null)が出力されています。スーペースとは異なります。

サンプルプログラム(substitute.c)

```
#include <stdio.h>
main()
{
  char *p, *p2;

  p="abcdef";

  printf("*p = %s\n\n",p);

  printf("address  %x    data  %c\n",p2="xyz",*p2);
  printf("address  %x    data  %c\n",p2+1,*(p2+1));
  printf("address  %x    data  %c\n",p2+2,*(p2+2));
  printf("address  %x    data  %c\n",p2+3,*(p2+3));

}
```

実行結果

```
*p = abcdef

address  200004b8    data  x
address  200004b9    data  y
address  200004ba    data  z
address  200004bb    data
```

3. 配列

3.1 配列とは何か

1 個の文字で複数の数値を表すことがあります。数学では、ベクトルとか行列とか言われる量です。電磁気学で出てくる電場 **E** 等もベクトルの例です。電場 **E** は、ベクトル量で(Ex, Ey, Ez)の 3 個の量から成り立っています。配列は、これと似ています。配列とは、同じ名前で操作される同じ型のデータを集めたものです。

1 次元配列は、図 1 のようなイメージです。データを入れる箱があって、それぞれに数字でアクセスできるように名前があります。

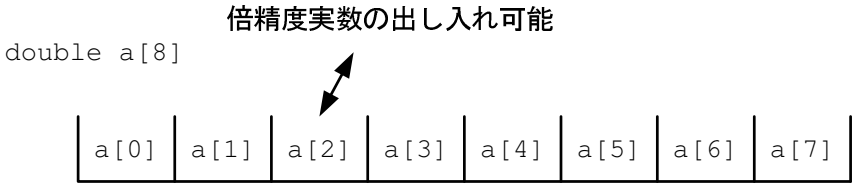

図 2 1 次元配列のイメージ

2 次元配列は、図 2 のように平面的に広がっているイメージです。3 次元だと立体的に、4 次元だと絵に表すことはできませんが、同様です。メモリーの許す限り、多次元の配列は可能です。

double a[5][5]

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]
a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]

図 3 2 次元配列のイメージ

これらの配列がプログラム上重要なことは、数字(添え字)によって、目的の要素にアクセスできることです。線形代数の行列やベクトルと同じです。

数値計算のプログラムでは、大量の数値を使います。それらの数値は、通常、配列に格納されます。それは、要素へのアクセスが非常に簡単だからです。ポインターを上手に使えば、同

様のことは可能ですが、プログラムが分かりにくくなります。普通は、配列を使います。
FORTRAN でも配列はありましたが、C の配列と大きく異なる点があります。異なる点は、

- FORTRAN の配列は、f[1], f[2] ,f[3]のように、添え字は 1 から始まります。したがって、最期の配列要素の添え字は、宣言の大きさと同じです。
- C 言語の配列は、f[0], f[1] ,f[2]のように、添え字は 0 から始まります。したがって、最期の配列要素の添え字は、宣言の大きさ-1 と同じです。

です。最初、よく間違えますので、気をつけてください。
配列の大きさを超えた要素を使っても、プログラムは実行されます。メモリーの内容を、強引に書き換えることになりますので、暴走の可能性があります。プログラマーは、常に配列の終わりを意識しなくてはなりません。
配列名、図 3 の a は、配列の先頭のアドレスです。このアドレスを使って、添え字に分だけ移動して、目的の要素を探します。2 次元配列のアドレスと配列の要素の関係は、

```
int a[m][n]
            要素          アドレス
            a[0][0]       a
            a[0][1]       a+sizeof(int)
            a[0][2]       a+2*sizeof(int)
            a[0][3]       a+3*sizeof(int)
              •
              •
              •
            a[i][j]       a+(n*i+j)*sizeof(int)
              •
              •
              •
```

となります。ここで、sizeof(int)は、int 型のバイト数を計算する演算子です。アドレスが増えるとともに、要素の添え字の後ろ側が先に増えます。3 次元以上の多次元配列も同様です。

3.2 1 次元配列

1 次元配列を使用した例を、次のサンプルプログラムに示します。4 行目は、初期化を含めて配列を定義しています。5 行目は、初期化を実施しないで、配列を定義しています。

サンプルプログラム(onedim.c)

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

   for(i=0; i<=2; i++){
      b[i]=2*a[i];
      printf(" %d   %d  %d\n", i, a[i], b[i]);
   }
}
```

実行結果

```
0   1  2
1   2  4
2   3  6
```

3.3 多次元配列

2 次元配列を使用した例を、次のサンプルプログラムに示します。4 行目は、初期化を含めて配列を定義しています。5 行目は、初期化を実施しないで、配列を定義しています。２次元配列の初期化は、アドレスの先頭から代入されていきます。したがって、サンプルプログラムの場合

```
a[0][0]=0
a[0][1]=1
a[0][2]=2
a[1][0]=3
a[1][1]=4
a[1][2]=5
a[2][0]=6
a[2][1]=7
a[2][2]=8
```

となります。

サンプルプログラム(twodim.c)

```
#include <stdio.h>
main()
{
  int a[3][3]={0,1,2,3,4,5,6,7,8};
  int b[3][3];
  int i, j;


  for(i=0; i<=2; i++){
     for(j=0; j<=2; j++){
        b[i][j] = 2*a[i][j];
        printf("%d %d   %d  %d\n", i, j, a[i][j], b[i][j]);
     }
  }

}
```

実行結果

```
0 0   0  0
0 1   1  2
0 2   2  4
1 0   3  6
1 1   4  8
1 2   5  10
2 0   6  12
2 1   7  14
2 2   8  16
```