これまでの復習 (前期中間試験に向けて)

山本昌志*

2004 年 6 月 9 日

これまでの、学習をまとめる。中間試験には、このプリントに書いてある内容を理解して臨むこと。

1 構造体

1.1 いろいろなデータ構造

これまで 3 つのデータ構造を学習した。単純型と配列、構造体である。それぞれの概略は以下の通り。

1.1.1 単純型

単純型の変数は、次のように変数に一つの数値[1]しか代入できないものを言う。

```
char c, h, moji;
int i, j, seisu;
double x, y, jisu;
```

1.1.2 配列

順序づけられた同じ型のデータが複数ある場合、配列の出番となる。添え字 (これが順序を表す) により、それらにアクセスできるので、データの操作が簡単にできる。配列を使う場合、

```
int i[10], j[100][100];
```

のように宣言を行う。そうすると必要なメモリー領域が確保され、配列が使えるようになる。この配列のデータにアクセスするためには、配列名と添え字を指定する。次のようにである。

```
i[3]=5;
c=i[3];
```

*国立秋田工業高等専門学校　電気情報工学科

[1]一つの文字のみ代入可能なものは文字型の変数である。文字も整数として扱えるので、この範疇と考える。

1.1.3 構造体

配列は同じ型のデータの集まりであったが、構造体は異なる型のデータの集まりである[2]。この構造体を使う場合、

1. 構造体のメンバーを規定する。これにより構造体を定義する。
2. 構造体変数の宣言。これによりメモリーが確保される。

という手順が必要である。このようにして、構造体を定義し、メモリーを確保した後、それを使うことができる。今までは、データの型の内容があらかじめ決まっていたので、最初の手順は不要であった。一方、構造体のメンバー、すなわちデータの型はプログラマーが決めなくてはならないので、最初の手順が必要となる。

最初のメンバーの規定は、つぎのように行う。

```
struct seito{
  char name[128];
  int kokugo;
  int sansu;
};
```

これでは、メモリーがまだ確保されていないことに注意が必要である。これは、プログラマーが新たに変数を定義したのと同じである。

そして、この構造体を実際に使う場合には、次のようにしてメモリーを確保する。

```
struct seito yamamoto, ninensei[50];
```

そうすると構造体変数が使用可能となる。

メモリーに確保された構造体のデータにアクセスするためには、ドット (.) 演算子を使うことになる。次のようにである。

```
yamamoto.sansu = 63;
ninensei[5].kokugo=76;
tensu = ninensei[6].sansu;
```

1.2 構造体

1.2.1 構造体型の定義

構造体は、メンバーと呼ばれる変数の集合体の型である。プログラマーが新たに型を定義するようなものである。定義の方法は、一般的には次のようになる。

[2]同じ型でも良い

```
struct タグ名 {
  型 メンバー１名;
  型 メンバー２名;
  型 メンバー３名;
  ・
  ・
  ・
  型 メンバーＮ名;
} 変数リスト;
```

例として学生の名前と成績、身長、体重を示した構造体を定義してみる。

```
struct gakusei{
  char name[80];
  int mathematics;
  int english;
  int japanese;
  int electrical_eng;
  int info_eng;
  double height;
  double weight;
} sato, tanaka, yamamoto;
```

これで、gakusei 型の構造体変数の sato と tanaka、yamamoto が使えるようになる。さらに、watanabe という変数を追加したい場合は、

```
struct gakusei watanabe;
```

とすればよい。

1.3 構造体型のメンバーの参照

構造体のメンバーの参照には、. 演算子 (ドット演算子) をつかう。次のようにするのである。

```
構造体変数.メンバー                                    /* 通常 */
構造体変数.メンバー.メンバー                            /* メンバーが構造体 */
構造体変数.メンバー.メンバー.メンバー                    /* メンバーのメンバーも構造体*/
構造体変数[配列の添え字].メンバー                       /* 構造体が配列*/
構造体変数.メンバー[配列の添え字]                       /* メンバーが配列*/
構造体変数[配列の添え字].メンバー[配列の添え字]         /*構造体もメンバーも配列*/
```

1.4 プログラム例

構造体を用いたプログラム例としてリスト 1 を示す。この内容をよく理解する必要がある。

リスト 1: 構造体を使ったプログラム例

```c
#include <stdio.h>
#include <string.h>

struct seiseki{
  int mathematics;
  int english;
};

struct gakusei{
  char name[80];
  struct seiseki test;
  double height;
};

int main(void){

  struct gakusei e2[10];

  strcpy(e2[0].name,"yamamoto");
  e2[0].test.mathematics = 95;
  e2[0].test.english = 65;
  e2[0].height = 174.8;

  printf("%s\n", e2[0].name);
  printf("  Mathematics : %d\n", e2[0].test.mathematics);
  printf("  English     : %d\n", e2[0].test.english);
  printf("  Height      : %f [cm]\n", e2[0].height);

  return 0;
}
```

実行結果

```
yamamoto
  Mathematics : 95
  English     : 65
  Height      : 174.800000 [cm]
```

2　2 進数と 16 進数

- いろいろな数の表記方法がある。N 進数の場合、次のように N 個の底で数を表現する。

  2 進数　$0, 1$
  10 進数　$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
  16 進数　$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}$

- 桁上がりは、2 進数の場合 1 の次で 10 に、10 進数の場合 9 の次で 10 に、16 進数の場合 F の次で 10 になる。

- 我々が通常用いている数の表現の意味は、次の通りである。数字の並ぶ順序が重要で、これを「位取り記数法」と言う。

$$(1905)_{10} = (1 \times 10^3 + 9 \times 10^2 + 0 \times 10^1 + 5 \times 10^0)_{10}$$

- 基数の変換 (2 → 10 進数)。通常の位取り記数法が理解できれば、簡単である。

$$\begin{aligned}(1101)_2 &= (1 \times 10^{11} + 1 \times 10^{10} + 0 \times 10^1 + 1 \times 10^0)_2 \\ &= (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)_{10} \qquad \leftarrow \text{普通はここから計算} \\ &= (8 + 4 + 0 + 1)_{10} \qquad \leftarrow \text{ここから計算しても良い} \\ &= (13)_{10}\end{aligned}$$

- 2 進数の各桁の 10 進数の値 (重み) を覚えておくと便利である。

$$1,\ 2,\ 4,\ 8,\ 16,\ 32,\ 64,\ 128,\ 256,\ 512,\ 1024,\ 2048,\ 4096$$

- 基数の変換 (10→2 進数)。2 で割った余りを並べればよい。変換方法の例を、以下に示す。
  $(19)_{10} = (10011)_2$,　$(2003)_{10} = (11111010011)_2$ である。

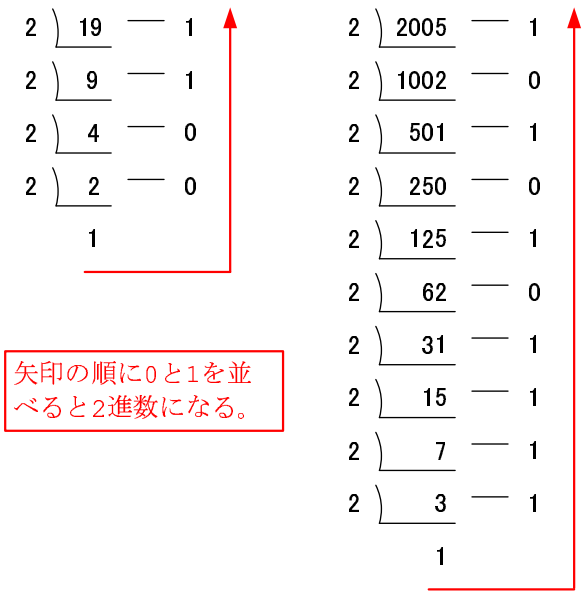

図 1: 10 進数から 2 進数への変換方法。

- 基数の変換 (16→10 進数)。これも、2 進数と同じ。

$$\begin{aligned}(376)_{16} &= (3\times 10^2 + 7\times 10^1 + 6\times 10^0)_{16}\\ &= (3\times 16^2 + 7\times 16^1 + 6\times 16^0)_{10}\\ &= (3\times 256 + 7\times 16 + 6\times 1)_{10}\\ &= (886)_{10}\end{aligned}$$

- 基数の変換 (10→16 進数)。16 で割って、その余りが各桁になる。

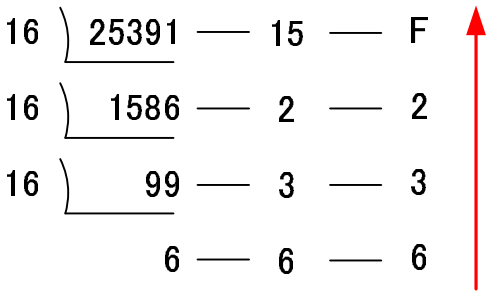

図 2: 10 進数から 16 進数への変換方法。

- 基数の変換 (2↔16 進数)。2 進数の 4 桁が、16 進数の 1 桁に等しいことを利用する。

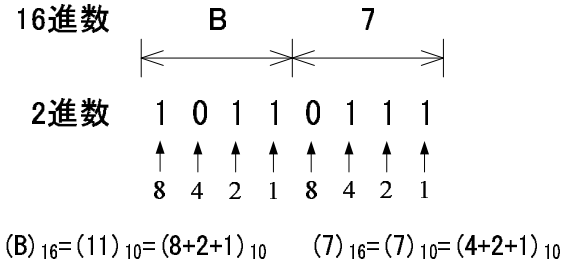

図 3: 2 進数と 16 進数の相互変換

- 桁数が合わない場合は、先頭に必要なだけゼロを書き足して考える。例えば、$(101100)_2 = (00101100)_2 = (2C)_{16}$ となる。

3　ポインター

3.1　コンピューターの構造

- 図 4 がコンピューターの基本構成である。
- **制御装置**と**演算装置**、**記憶装置**、**入力装置**、**出力装置**をコンピューターの五大装置と呼び、それぞれには以下の働きがある。

  制御装置　主記憶装置 (メインメモリー) に格納されているプログラムを受け取り、各装置に指令を出す。

  演算装置　主記憶装置の中にあるデータを受け取り、制御装置の指令に従い、それを処理する。

  記憶装置　主記憶装置と補助記憶装置がある。

  主記憶装置　命令とデータからなるプログラムを格納する。

  補助記憶装置　大容量、あるいは半永久的に残したいデータを補助記憶装置に格納する。

  入力装置　コンピューターの外部からデータを取り込む装置である。

  出力装置　主記憶装置に格納されているデータをコンピューター外部に出力する装置である。

- 制御装置と演算装置をまとめて中央制御装置 (CPU:Central Processing Unit) あるいは、 MPU(Micro Processing Unit) と言う。

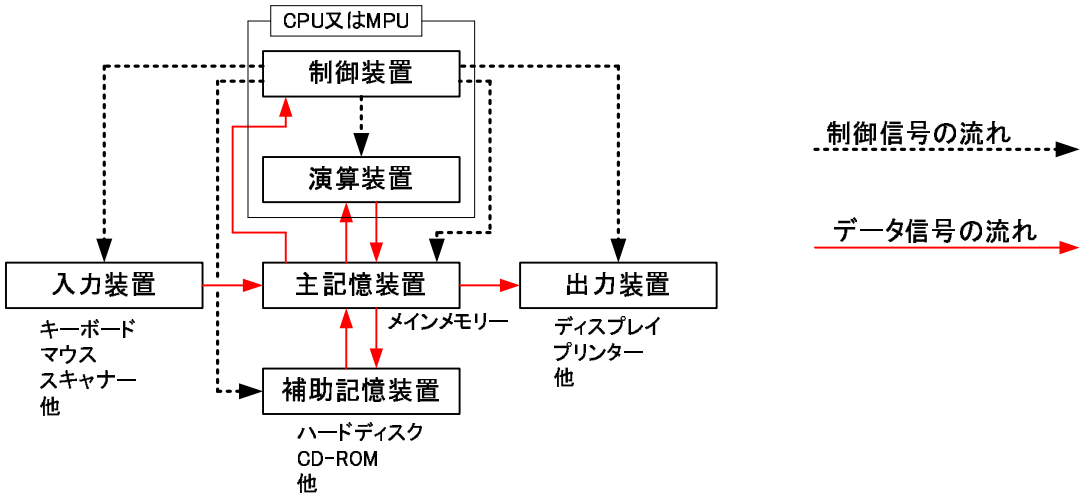

図 4: コンピューターの基本構成

3.2 メモリーとデータ

- 2 進数の 1 桁が 1 ビットである。
- 諸君が使っているパソコンのアドレスは 32 ビットで表現されている。これは 16 進数では 8 桁である。
- 一つのアドレスに 8 ビット (1 バイト) 記憶できる。これは、16 進数 2 桁で表現できる。
- 8 ビットを 1 バイトと言う。
- 諸君が使っているコンパイラーでは、変数は表 1 に示しているバイト数で表現されている。

表 1: 変数の型とバイト数

型名	データ型	バイト数	ビット数
文字型	char	1	8
整数型	int	4	32
倍精度実数	double	8	64

- プログラムは命令とデータから構成され、いずれもメモリーの中に格納される。
- プログラムの関数 (これが命令) が格納されるアドレスは、関数名で参照できる。
- データが格納されるアドレスは、データ名の前に`&`を付けることで参照できる。`&` はアドレス演算子である。
- アドレスの表示には変換指定子 `%p` を使う。
- ローカル変数は名前が同じでも、メモリーの配置場所は異なる。正確言うと、その関数が呼び出されたときのみ、ローカル変数はメモリーに割り当てられる。

3.3 ポインター

- ポインターとは、アドレスを格納する変数のことである。
- ポインターの宣言には、型名とアスタリスク (`*`) を付ける。

```
int *pi;
double *px;
```

- 変数のアドレスを取り出すには、変数名の前にアンパサンド (`&`) をつける。`&`はアドレス演算子である。以下の例では、`pi` と `px` がポインターで、変数 `i` と `x` のアドレスを、それに代入している。

```
pi=&i;
px=&x;
```

- ポインターが示しているデータの値を取り出すためには、ポインター変数の前にアスタリスク (*) をつける。*は間接参照演算子である。つぎの例は、ポインター pi と px が指し示しているアドレスにある値を取り出して、変数 j と y に代入している。

```
j=*pi;
y=*px;
```

- リスト 2 のプログラムをよく理解すること。

  4 行 整数型のポインター p を宣言している。p に整数型のデータの先頭アドレスを格納する。

  5 行 整数型の変数 i を宣言し、$(11223344)_{16}$ を代入している。

  7 行 変数 i の先頭アドレスをアドレス演算子 &により取り出し、ポインター p に代入している。

  9 行 整数変数 i の先頭アドレスを変換指定子 %p により表示している。

  10 行 ポインター p の先頭アドレスを変換指定子 %p により表示している。

  12 行 整数変数 i の値を 16 進数表示の変換指定子 %0x により表示している。

  13 行 ポインター p の値を 16 進数表示の変換指定子 %0x により表示している。ただし、ポインターはアドレスなので、強制型変換 (キャスト) により、符号なし整数にしている。

  15 行 ポインターが指し示すアドレスに格納されているデータを表示している。

リスト 2: 関数の書き方

```
1  #include <stdio.h>
2
3  int main(void){
4    int *p;
5    int i=0x11223344;
6
7    p=&i;
8
9    printf("address i %p\n", &i);
10   printf("address p %p\n", &p);
11
12   printf("value i %0x\n", i);
13   printf("value p %0x\n", (unsigned int)p);
14
15   printf("value *p %0x\n", *p);
16
17   return 0;
18 }
```

実行結果

```
address i 0xbffff6b0
address p 0xbffff6b4
value i 11223344
value p bffff6b0
value *p 11223344
```

この実行結果から、メモリーは図 5 のようになっていることが分かる。

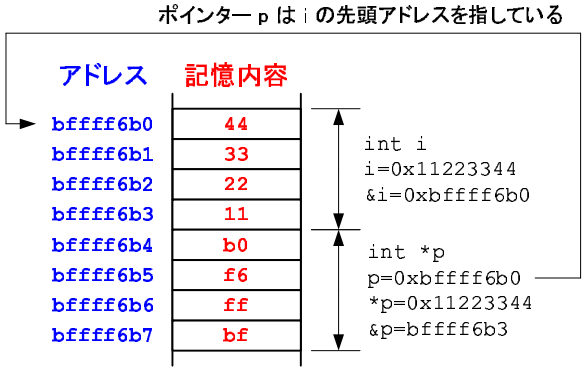

図 5: リスト 2 のプログラム実行後のメモリーの内容