これまでの復習 (学年末試験に向けて)

山本昌志*

2006 年 3 月 2 日

概 要

学年末試験に向けて，これまで学習した内容をまとめる．このプリントは試験対策用である．

1 前期末試験の傾向と対策

試験の範囲は，以下の通り．

- 第 25 回の講義から本日の第 33 回の講義に配布したプリント
- 教科書は，p.204–292 が範囲となる．2〜3 回位は読み直した方がよい．教科書の範囲で講義で触れなかった部分は試験には出さない．
- このプリントの内容を十分理解して，試験に臨むこと．分からなければ，私を含めた他の人に聞くこと．

2 配列

2.1 基礎

同じ型のデータが大量にあるとき配列と呼ばれるデータ構造が便利である．配列を使うときの宣言は，

```
int hoge[100], fuga[200][200];
double foo[300], bar[4][4][4];
```

のようにする．すると，次のようなひとつずつデータの格納できる要素が使えるようになる．

- 整数が格納できる要素 `hoge[0]`〜`hoge[99]` が使えるようになる．この場合，配列の要素数は 100 個である．
- 整数が格納できる要素 `fuga[0][0]`〜`fuga[199][199]`，合計 40000 個が使えるようになる．
- 倍精度実数が格納できる要素 `foo[0]`〜`hoge[299]`，合計 300 個が使えるようになる．
- 倍精度実数が格納できる要素 `bar[0][0][0]`〜`bar[3][3][3]`，合計 64 個が使えるようになる．

*独立行政法人秋田工業高等専門学校電気工学科

配列の初期化　配列のサイズが小さい場合，宣言と同時に初期ができる．

```
int hoge[3]={111,222,333};
int fuga[2][2]={{111,222},{333,4444}};
```

配列よりも初期値が少ない場合には，残りはゼロに初期化される．多い場合にはエラーとなる．

配列の要素へのアクセス　配列名と添え字 (インデックス) を指定する—たとえば `i[3]` や `j[25][49]`—ことにより，記憶領域から値 (データ) を入出力できる．

```
i[3]=5;          /* 配列 i[3] に 5 を代入  */
c=j[25][49];     /* 配列 j[25][49] の値を変数 c へ代入 */
```

ほとんど今まで使ってきた変数と同じである．インデックスには自然数が格納された整数型の変数を使うことも可能である．

```
for(i=0; i<=360; i++){
    my_sin[i]=sin(M_PI*i/360.0);
}
```

こうすると，`my_sin[45]` には $0.707107\cdots$ が格納される．

2.2　応用

2.2.1　ファイルからのデータ読み込み

配列は大きなデータを扱うことが多い．格納するデータはファイルから読み込むことが多い．ファイルからデータを取得するには，(1) ファイル情報を格納する変数を用意する，(2) ファイルをオープンする，(3) ファイルからデータを読み込む，(4) ファイルをクローズするという一連の動作が必要である．リスト 1 では次のようにしている．`exercise.txt` というファイルを読み込んでいる．

```
FILE *in_file;

in_file = fopen("exercise.txt", "r");

for(i=0; i<10000; i++){
  fscanf(in_file,"%d%d%d%d",
 &data[i][0],&data[i][1],&data[i][2],&data[i][3]);
}

fclose(in_file);
```

2.2.2　関数へデータを渡す方法

配列格納されたデータは大量な場合が多い．ユーザー定義関数を利用すると分かり安プログラムになる．配列のデータをユーザー定義関数に渡す方法を学習した．

単純型の変数と配列では，ユーザー定義関数にデータを渡す方法は大きく異なる．

- 単純型の変数の場合，呼出側の実引数の値がユーザー定義関数の仮引数にコピーされる．したがって，ユーザー定義関数の変数の値を変化させても，呼出側の変数の値は変わらない．ようするに，別々のメモリーを使っている．
- 配列の場合，呼出側の実引数の配列とユーザー定義関数の配列は，同じ メモリーを使う．したがって，ユーザー定義関数の配列の値を変化させると，呼出側の配列の値が変わる．なぜこのようにするかというと，コンピューターの資源—メモリーと CPU の計算時間—を節約したいためである．大量のデータが格納されている配列のコピーには多大な資源である．

実際に，ユーザー定義関数に配列を渡す例をリスト 1 で見てみよう．このプログラムでは呼び出し側では配列名のみを，ユーザー定義関数では配列名と左端を空欄とした要素数を記述している．こうすることにより，ユーザー定義関数に配列のデータを渡すことができる．

```
  sum_data = cal(data);

    長いので省略


  int cal(int hoge[][4]){

    ここに関数での処理の内容を書く．

  return sum0+sum1+sum2+sum3;
}
```

2.3 プログラム例

それぞれについて，リスト 1 のプログラムを例にして説明する．このプログラムは，つぎに示すデータをファイルから読み込んで，全ての整数の合計値と各列の平均値を計算するプログラムである．データは，10000 行 4 列，合計 4 万個の整数である．

```
-76797  99987   53528   -43172
-34698  44783   -106207 -106631
-83424  31615   18774   -4134
78694   -886    64632   103022
-86516  -99744  -51044  -11396
90058   54995   -39364  -36610
-78969  106494  -5209   -95276
75913   32002   39260   106490
24615   -14585  -44056  97291
-77175  -42889  98034   -53226
96100   9434    50013   67420

   この辺は長いので省略

100747  -2875   37515   4509
77468   12111   76950   82072
-102787 41331   75324   -96228
-53535  -6367   65795   -62947
```

リスト 1: 全ての合計と各列の平均値を計算するプログラム．

```c
#include <stdio.h>

int cal(int hoge[][4]);                                  // プロトタイプ宣言

double ave0, ave1, ave2, ave3;              // 各列の平均値はグローバル変数とする

//=============================================================================
// main 関数
//=============================================================================
int main(void)
{
  FILE *in_file;
  int i,data[10000][4], sum_data;

  in_file = fopen("/home/yamamoto/tmp/program/int_data.txt", "r");

  for(i=0; i<10000; i++){
    fscanf(in_file,"%d%d%d%d",
           &data[i][0],&data[i][1],&data[i][2],&data[i][3]);
  }

  fclose(in_file);

  sum_data = cal(data);

  printf("sum all = %d\n", sum_data);
  printf("average = %f\t%f\t%f\t%f\n", ave0,ave1,ave2,ave3);

  return 0;
}

//=============================================================================
// データ処理のためのユーザー定義関数
//=============================================================================
int cal(int hoge[][4]){
  int i;
  int sum0=0, sum1=0, sum2=0, sum3=0;

  for(i=0; i<10000; i++){
    sum0+=hoge[i][0];
    sum1+=hoge[i][1];
    sum2+=hoge[i][2];
    sum3+=hoge[i][3];
  }

  ave0=sum0/10000.0;
  ave1=sum1/10000.0;
  ave2=sum2/10000.0;
  ave3=sum3/10000.0;

  return sum0+sum1+sum2+sum3;
}
```

実行結果

```
sum all = -15594426
average = -612.222100   377.635000      -430.162000     -894.693500
```

3 文字列

3.1 基本

- コンピューターで文字を扱うときは，それはすべて，整数に置き換えて取り扱う．文字と整数との対応を決めたものをコード表と言う．
- 1 文字を表すためには，英数字では 1 バイト，日本語では 2 バイト必要である．
- 文字を扱うときには文字型の変数を使う．変数宣言の型は,「char」である．これで，宣言された変数は 1 バイトの情報を記憶できる．すなわち，英数字の 1 文字分である．
- 複数の文字が連なったものを文字列と言う．文字列を扱うためには，文字型の配列を使う．
- 文字型の配列に文字列を記憶させた場合，文字列のすぐあとに「\0」が格納される.「\0」が無いと，文字列の終わりが分からない．

3.1.1 文字型変数と代入

つぎに，文字型の変数宣言とそれへの値の代入である．

- 英数字が 1 文字の場合
  - 変数宣言

```
char hoge;              /* 文字型の変数 */
```

  - 代入．シングルクォーテーションで囲めば，代入演算子「=」が使える．

```
hoge='A';               /* 代入 */
```

  - 表示．変換指定子に%c を使う．

```
printf("%c", hoge);     /* 表示 */
```

- 英数字の文字列の場合
  - 変数宣言

```
char hoge[10];              /* 文字型の配列 */
```

  - 文字型の配列に文字を格納する方法はいくつかある．代表的な方法は,「strcpy」と「sprintf」を使う方法である．ただし，前者を使う場合,「string.h」をインクルードする必要がある．

```
strcpy(hoge, "Akita");              /* 文字列の格納 */
sprintf(hoge, "Akita");
```

  - 表示．変換指定子に%s を使う．

```
printf("%s", hoge);     /* 表示 */
```

- 日本語の場合は，文字型の配列を使わなくてはならない．必要な配列のサイズは，2× 文字数 +1 である．なぜならば，日本語の 1 文字は 2 バイトで，最後に「\0」を付加するためである．

  – 変数宣言

```
char hoge[10];              /* 文字型の配列 */
```

  – 文字型の配列に日本語の格納は，英数字と同じで，ダブルクォーテーションで囲む．

```
strcpy(hoge, "秋田");              /* 文字列の格納 */
sprintf(hoge, "秋田");
```

  – 表示．変換指定子に%c を使う．

```
printf("%s", hoge);    /* 表示 */
```

3.2 標準ライブラリー関数

文字や文字列を扱うために，表 1 や表 2 に示すライブラリー関数が容易されている．学年末試験の時には，この表は参考資料として添付するのでこれを憶える必要はない．この表の関数を使うためには，<ctype.h>や<string.h>をインクルードする必要がある—ということは憶えよ．

3.2.1 文字処理関数

表 1: 文字処理関関数．#include <ctype.h>が必要．変数は，int c;.

関数名	動作	戻り値
isalnum(c)	英数字なら真	真/偽 (整数型)
isalpha(c)	英文字なら真	真/偽 (整数型)
iscntrl(c)	制御文字なら真	真/偽 (整数型)
isdigit(c)	数字なら真	真/偽 (整数型)
isgraph(c)	印字可能文字なら真	真/偽 (整数型)
islower(c)	小文字なら真	真/偽 (整数型)
isprint(c)	空白以外の印字可能文字なら真	真/偽 (整数型)
ispunct(c)	区切り文字なら真	真/偽 (整数型)
isspace(c)	空白類文字なら真	真/偽 (整数型)
isupper(c)	大文字なら真	真/偽 (整数型)
isxdigit(c)	16 進表示文字なら真	真/偽 (整数型)
tolower(c)	文字 c を小文字に変換	小文字 (文字型)
toupper(c)	文字 c を大文字に変換	大文字 (文字型)

3.3 文字列処理関数

表 2 を使うためには，#include <string.h>が必要である．変数は，char s1[256],s2[256]; のように文字型の配列．そのサイズは，処理に必要なサイズよりも大きいこと (256 とは限らない)．後の学習範囲

であるが，s1 や s2 は文字型のポインターでも良い．ま た，ダブルクォーテーションで囲んだリテラル表現も可能な部分もある．c は文字型の変数，char c; である．

表 2: 文字列処理関関数.

関数名	動作	戻り値
strlen(s1)	文字列 s1 の長さ，すなわち文字数を整数値返す.	文字列長 (整数型)
strcpy(s1,s2)	s1 に，文字列 s2 をコピーする.	ポインター s1 の値
strcat(s1,s2)	文字列 s1 の後に，文字列 s2 をコピーする.	ポインター s1 の値
strcmp(s1,s2)	文字列 s1 と s2 を比較する. s1 > s2 の場合，戻り値は正 s1 == s2 の場合，戻り値は 0 s1 < s2 の場合，戻り値は負	整数値
strncpy(s1,s2,n)	s1 に文字列 s2 の先頭から n 文字をコピーする.	ポインター s1 の値
strncat(s1,s2,n)	文字列 s1 の後にと文字列 s2 の先頭から n 文字を連結する.	ポインター s1 の値
strncmp(s1,s2,n)	文字列 s1 と文字列 s2 の先頭から n 文字を比較する．比較の結果は，srcmp と同じ.	整数値
strchr(s1,c)	文字列 s1 の中の文字 c の位置を返す．文字がないときは，NULL を返す.	ポインター
strstr(s1,s2)	文字列 s1 の中にある文字列 s1 の位置を返す．もし，文字列がない場合，NUL を返す.	ポインター

4　位取り記数法と 2 進数，10 進数，16 進数

2 進数と 10 進数，16 進数の相互の変換は，必ずできるように練習しなくてはならない．

- 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．

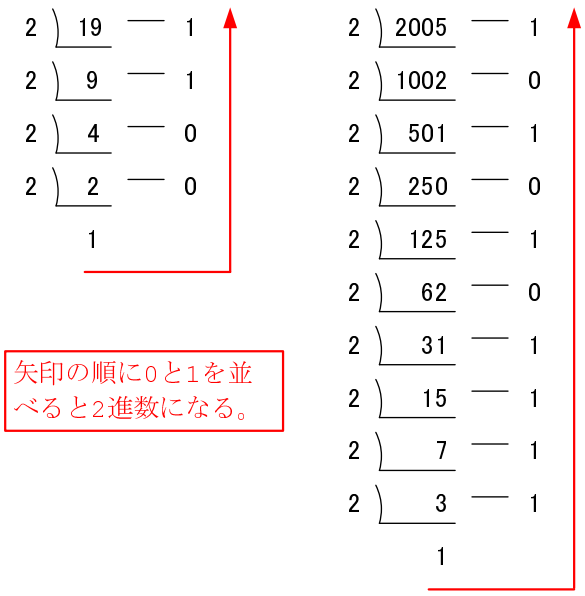

図 1: 10 進数から 2 進数への変換方法.

- 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．

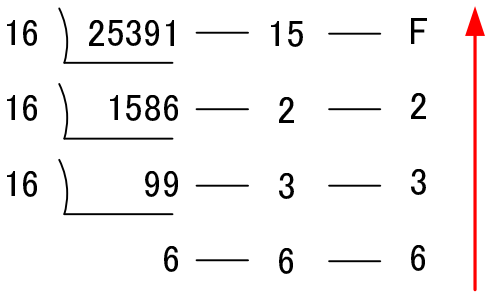

図 2: 10 進数から 16 進数への変換方法.

- 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．

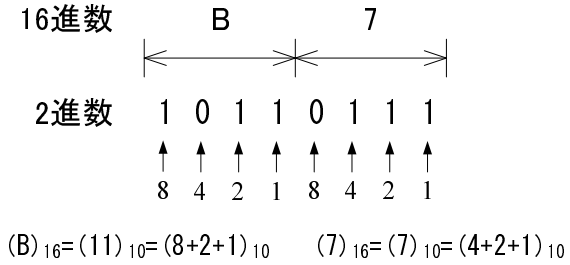

図 3: 2 進数と 16 進数の相互変換

- 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．

5　ポインター

5.1　コンピューターの構造

ポインターを学習する前に，ハードウェアーについて説明した．

- 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．

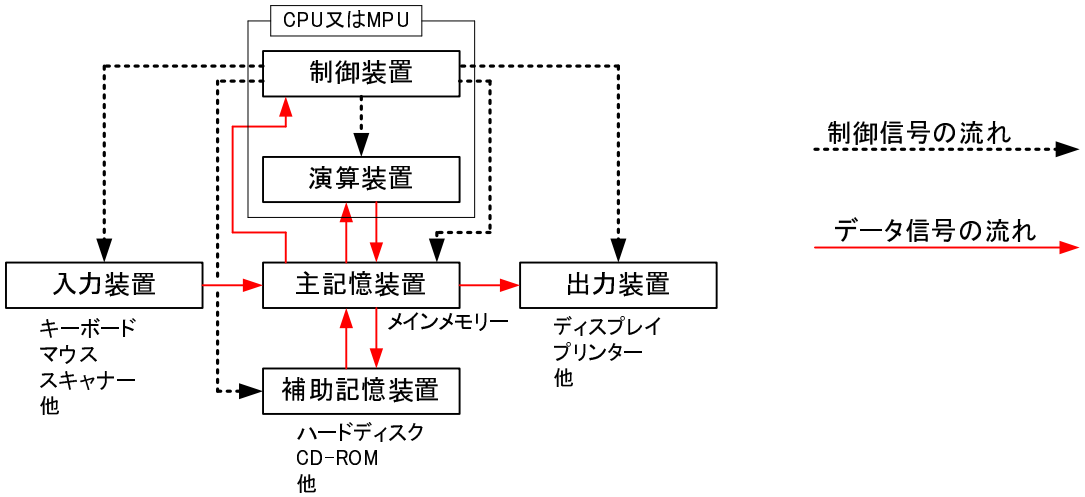

図 4: コンピューターの基本構成

5.2 メモリーとデータ

- 2 進数の 1 桁が 1 ビットである．
- 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．したがって，アドレスを記憶するためには 4 バイト—メモリーの 4 番地分—必要である．
- 諸君が使っているコンパイラーでは，型は表 3 に示しているバイト数で表現されている．

表 3: 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．
- 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．

5.3 ポインターと演算子

- アドレスを格納するための変数を**ポインター変数**[1]という．それは，

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

と宣言する．アスタリスク (*) をつければ，ポインター変数の宣言になる．

- 変数のアドレスは，**アドレス演算子** (&) により取り出すことができる．たとえば，整数型変数 i と実数型変数 x のアドレスは，&i と&x とすると取り出すことができる．ここで，アドレス演算子&は，それに引き続く変数の先頭アドレスを取り出す演算子である．取り出したアドレスは，ポインターに

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

のようにして代入できる．アドレス演算子 (&) により変数の先頭アドレスを取り出して，代入演算子 (=) を用いて，ポインター変数に代入している．アドレスを表示するときには，変換指定子%p を使う．

```
printf("%p",&x);
```

- ポインター機能は，アドレスの格納のみに止まらず，そのアドレスが示しているデータの内容も表すことができる．今までの例の通り，ポインター変数には変数の先頭アドレスが格納されている．そして，ポインターの宣言の型から，そのポインターが指しているデータの内容までたぐり寄せることができる．ポインター pi と px が示しているデータの値を，整数型変数 j と実数型変数 y に代入する場合

[1]たんに,「ポインター」と呼ぶことの方が多い.

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

と書く．ここで，アスタリスク (*) は**間接参照演算子**[2]で，ポインターが示しているアドレスのデータを取り出す演算子である．このようにアドレスのみならず，そのアドレスのデータの型までポインターは持っているから，これが可能なのである．このことから，アドレスとは言わずにポインター (pointer 指し示すもの) と言うのであろう．

- 紛らわしいことに，間接参照演算子と積の演算子は同じアスタリスク (*) をつかう．C 言語の悪いところだが，そうなってしまっているので仕方ない．コンパイラーは前後の式からどちらなのか判断している．

ポインターに関係する演算子を表 4 にまとめておく．ただし，各変数は

```
double x, *xp;
```

と宣言したとする．

表 4: 普通の変数とポインター変数に演算子を作用させた場合に取り出せる値.

演算子	通常の変数 (x)	ポインター変数 (xp)	例
無し	格納されている値	格納されているアドレス	x,xp
&	変数のアドレス	ポインター変数のアドレス	&x,&xp
*	コンパイルエラーのため不可	ポインターが示すアドレスに格納されている値	*xp

リスト 2 のプログラムをよく理解すること．アドレスとポインターの関係や演算子の使い方を分からなくてはならない．

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．

[2]教科書では,「間接演算子」と記述している．どちらでも同じである．ただし，間接参照演算子と言う方が詳しく意味を説明しているので，ここではこちらを使う．

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．

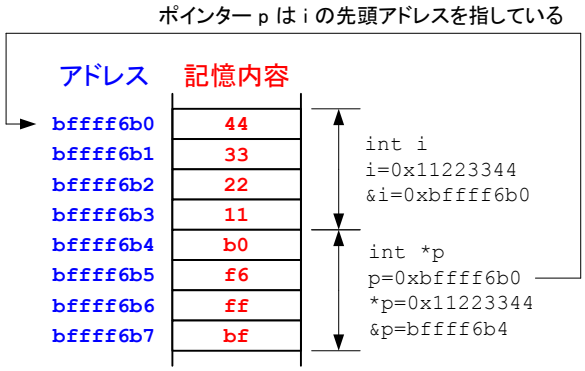

図 5: リスト 2 のプログラム実行後のメモリーの内容

5.4 ポインターの演算

ポインターは，整数型の値の和と差の演算ができる．しかし，積と商の演算はできない．ポインターの和や差の演算には，どういう意味があるのだろうか? リスト 3 がそれに対する答えである．ポインターの演算—整数の加算と減算—は，データの型のバイト数分，ポインターがシフト (移動) する．プログラムの実行結果から分かるように，

- 文字型のポインターでは，整数の 1 加算はアドレスの値が 1 増える．
- 整数型のポインターでは，整数の 1 加算はアドレスの値が 4 増える．
- 倍精度実数型のポインターでは，整数の 1 加算はアドレスの値が 8 増える．

となっている．

ポインターに加算される整数は，ポインターが指し示すデータの移動量を表す．ひとつのアドレスには 1 バイトのデータが格納できる．そして，文字型のデータでは 1 バイト，整数型では 4 バイト，実数型では 8 バイトのメモリーである．これが，型に依存して，アドレスの変化の仕方が異なった理由である．

リスト 3: ポインターに整数を加算した場合のアドレスの変化

```
1 #include <stdio.h>
2
```

```
3  int main(void)
4  {
5    char *cp;
6    int *ip;
7    double *dp;
8    int i;
9
10   for(i=0; i<4; i++){
11     printf("%d %p\t%p\t%p\n",i, cp+i, ip+i, dp+i);
12   }
13
14   return 0;
15 }
```

実行結果

```
0 0xbff0eb9c    0xbff0eb08      0x8048416
1 0xbff0eb9d    0xbff0eb0c      0x804841e
2 0xbff0eb9e    0xbff0eb10      0x8048426
3 0xbff0eb9f    0xbff0eb14      0x804842e
```

5.5 関数の引数にポインターを使う

5.5.1 参照渡しと値渡しの違い

関数呼出しの時，次のふたつの方法で値—処理すべきデータ—を渡すことができる．

- 値そのものを渡す方法を**値渡し** (call by value) と言う．呼出側の実引数の値は，呼び出された関数の仮引数 (変数) にコピーされる．この方法では，呼出側の関数と呼び出された側の関数のそれぞれが，記憶領域—変数—を持つので，関数の独立性が高くなる．すなわち，呼び出された側で，呼出側の変数の値を変えることができない．
- アドレスを渡す方法を**参照渡し** (call by reference) と言う．呼出側の引数はアドレスである．呼び出された関数の仮引数のポインターにそのアドレスがコピーされる．この方法では，呼び出された側の関数で呼出側の関数のデータ領域の操作ができる．そのため，関数の独立性が失われる．すなわち，呼び出された側で，呼出側の変数の値を変えることができる．

参照渡しを使うと，グローバル変数を使わないで複数の計算結果を呼出元へ知らせることができる．リスト 4 の関数 `cal()` は，和と差，積，商を一度に計算し，参照渡しを使い 4 つの値を一度に呼出元へ知らせている．実際には，呼出し元から指定されたアドレスに，呼び出された関数 `cal()` が計算結果を書き込んでいるのである．

リスト 4: 参照渡しを使い，複数の計算結果を戻す．

```
1  #include <stdio.h>
2
3  void cal(int *wa, int *sa, int *seki, int *sho, int a, int b);
4
```

```
5  //-------------------------------------------------------
6  int main(void){
7    int add, sub, mul, div;
8
9    cal(&add, &sub, &mul, &div, 33, 3);
10   printf("add = %d\n", add);
11   printf("sub = %d\n", sub);
12   printf("mul = %d\n", mul);
13   printf("div = %d\n", div);
14
15   return 0;
16 }
17
18 //-------------------------------------------------------
19 void cal(int *wa, int *sa, int *seki, int *sho, int a, int b){
20
21   *wa = a+b;
22   *sa = a-b;
23   *seki = a*b;
24   *sho = a/b;
25 }
```

実行結果

```
add = 36
sub = 30
mul = 99
div = 11
```

5.5.2 配列をユーザー定義関数に渡す方法

関数の引数に配列名を使う場合，参照渡しとなる．配列名は配列の先頭アドレスを表すポインターとなっているからである．すなわち，配列名という変数 (のようなもの) には，その配列の先頭アドレスが格納されている．その配列名—アドレスの値—を渡すのであるから，参照渡しである．

リスト 5 に配列を関数に渡す例を示す．この例を見て分かるように，配列を渡す場合は次のようにする．

- 呼出側の実引数は配列名のみを書く．配列のサイズを書いてはならない．配列名は配列の先頭アドレスが格納されたポインターなので，これにより呼び出された関数へアドレスを渡すことができる．
- 呼び出された関数の仮引数は，配列名にかぎかっこ [] をつける．
- 配列のサイズが呼び出された関数で必要であれば，別の引数を使う．

リスト 5 の関数 reverse() は，配列に格納されている順序を逆にする関数である．

リスト 5: 一次元配列が関数の引数となる場合．

```
1  #include <stdio.h>
2  void reverse(int n, int a[]);
3
4  //-----------------------------------------
5  int main(void){
6    int hoge[3]={1,2,3};
7
```

```
8     printf("hoge=\t%d\t%d\t%d\n",hoge[0],hoge[1],hoge[2]);
9     reverse(3, hoge);
10    printf("hoge=\t%d\t%d\t%d\n",hoge[0],hoge[1],hoge[2]);
11
12    return 0;
13  }
14
15  //-----------------------------------------
16  void reverse(int n, int a[]){
17    int i, i_max, temp;
18
19    i_max=n/2;
20
21    for(i=0; i<i_max; i++){
22      temp=a[i];
23      a[i]=a[n-1-i];
24      a[n-1-i]=temp;
25    }
26  }
```

実行結果

```
hoge=   1       2       3
hoge=   3       2       1
```