# 情報処理応用の復習 (前期中間試験に向けて)

山本昌志*

2006 年 6 月 7 日

**概 要**

前期中間試験に向けて，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 前期中間試験の傾向と対策

試験の範囲は，以下の通り．

- 第 1 回の講義から第 9 回の講義に配布したプリント
- 教科書「C 言語によるプログラミング　基礎編」は，以下の通り．
  - 第 9 章「構造体とユーザー定義型」pp.294–323．ただし，共用体 (union) と列挙 (enumeration) は範囲外．
  - 第 10 章「ファイル」pp.326–373
- 教科書「C 言語によるプログラミング　応用編」は，以下の通り．
  - 第 2 章「分割コンパイルとマクロ」pp.20–67．ただし，Makefile については，出題範囲外とする．
  - 第 3 章「様々な関数とライブラリ」pp.70–80.
  - 第 4 章「ポインタ」pp.110–135.
- 最低限，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．このプリントは最低限の内容と考えよ．
- このプリントの他，配布プリントや教科書からも出題する．教科書の方は，課題となっていたので，十分読んでいると信じている．

## 2 構造体とユーザー定義型

### 2.1 宣言と定義，アクセス

関連のあるデータをひとつにまとめることが構造体の役割である．例えば，ある学生の試験成績管理するばあい，(1) 名前 (2) 数学の成績 (3) 英語の成績のようなデータが必要であろう．整数だけであれば，配列を使うことも考えられるが，名前も合わせて管理するとなると不可能である．

*独立行政法人秋田工業高等専門学校電気工学科

構造体の宣言は，次のようする．

```
typedef struct{
  char name[64];
  int math;
  int engl;
}seiseki;
```

いろいろな宣言の方法があるが，typedef を使う方法が一般的である．これは便利なので，諸君はこれを使うように心がけよ．これで，seiseki 型の構造体を決めたことになる．ただ，これは構造体を決めただけ—コンパイラーにこのような型の構造体があると知らせただけ—でメモリーの確保は行われていない．

メモリーの確保を行うためには，次のように型名と変数名を指定して変数の定義を行わなくてはならない．

```
seiseki gakusei;
```

これで，seiseki 型の構造体が格納できるメモリーがひとつ確保でき，変数 gakusei が使用できるようになる．

この構造体には，name と math，engl の 3 つのメンバーがある．これら 3 つのメンバーにアクセスするためには，ドット演算子 (.) を使う．

```
strcpy(gakusei.name, "Yamamda Masashi");
gakusei.math = 92;
gakusei.engl = 78;
```

このようにして，ドット演算子を使い構造体のメンバーを指定して，値を代入することができる．値を取り出すときもドット演算子を使う．

宣言と定義について

これまで**宣言** (declaration) と**定義** (definition) という語句をいい加減に使ってきた．これをはっきりさせておく必要がある．宣言とはオブジェクトの型を指定することである．定義とは，オブジェクト型を指定し必要な記憶領域を確保することである．

## 2.2 応用

構造体の配列を使うと，応用が拡がる．例えば，先ほどの seiseki 型の構造体を使って，秋田高専の 2 年生 4 学科 (各 40 人とする) の成績を管理することを考える．次のように構造体の配列を定義すればよい．

```
seiseki M2[40], E2[40], C2[40], B2[40];
```

配列の場合も同じようにドット演算子を使って，アクセスできる．

```
strcpy(E2[5].name, "Itoh Misaki");
E2[5].math = 89;
E2[5].engl = 96;
```

### 2.3 プログラム例

リスト 1: 構造体をつかったプログラム例.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6
7    typedef struct{
8      char name[64];
9      int math;
10     int engl;
11   }seiseki;
12
13   seiseki gakusei;
14
15   strcpy(gakusei.name, "Yamamda Masashi");
16   gakusei.math = 92;
17   gakusei.engl = 78;
18
19   printf("Name:\t\t%s\n", gakusei.name);
20   printf("Mathmatics:\t%d\n", gakusei.math);
21   printf("English:\t%d\n", gakusei.engl);
22
23   return 0;
24 }
```

実行結果

```
Name:           Yamamda Masashi
Mathmatics:     92
English:        78
```

# 3 ファイル

## 3.1 基本

ファイル—この講義ではハードディスク—を読み書きするためには，プログラムには次の 4 つ項目を記述する.

1. FILE 型のポインター (ファイルポインター) の定義をする.
2. fopen() 関数を使って，ファイルを開く.
3. fscanf() 関数を使って，ファイルの内容を読む. あるいは，fprintf() 関数を使って，ファイルにデータを書き出す.
4. fclose() 関数を使って，ファイルを閉じる.

### 3.1.1 ファイルポインターの定義

ファイルの操作を行うためには，ファイルポインター[1]を定義する必要がある．通常の変数の定義と同じように，型名と変数名を書く．ただし，ポインターなので，変数の前にアスタリスク (*) を忘れないように!

```
FILE *hoge;
```

ここで，hoge がファイルポインターで，これを使ってアクセスするファイルを指定する．

### 3.1.2 ファイルポインターのオープン

読み書きするファイルを開くためには，fopen() 関数を使う．例えば，ファイル「fuga.txt」からデータを読み込む場合，

```
hoge=fopen("fuga.txt", "r");
```

と記述する．最初にファイル名，そしてオープンモードを指定する．この間数の戻り値の型は，FILE 型のポインターである．オープンモードについては，いろいろ用意されているが，諸君はとりあえず次の二つが分かればよい．

- ファイルからデータを読み込む場合は，オープンモードの部分は"r"
- ファイルへデータを書き込む場合は，オープンモードの部分は"w"

### 3.1.3 ファイルの読み書き

ファイルからデータを読み込むプログラムは，キーボードからデータを読み込むプログラムと似ている．また，ファイルへデータを書き込むプログラムは，ディスプレイに出力するプログラムと似ている．

ファイルからデータの読み込みとキーボードからのデータ読み込みを並べて書くと

**ファイル入力**　int fscanf(ファイルポインター, 書式指定, 引数並び)
**キーボード入力**　int scanf(書式指定, 引数並び)

となる．ファイルポインターを指定する以外，すべてキーボード入力の場合と同じである．例えば，

```
fscanf(hoge, "%d%lf", &i, &d);
```

と書く．キーボードからデータを入力するのも，ファイルから読み込むのも同じイメージで取り扱える．戻り値の整数は入力したデータの個数である．もし，ファイルの最後あるいは読み込みに失敗すると EOF を返す．

出力もまったく同じである．

**ファイル出力**　int fprintf(ファイルポインター, 書式指定, 引数並び)
**ディスプレイ出力**　int printf(書式指定, 引数並び)

[1] ファイルポインターは，FILE 型の構造体を示すポインターである．この構造体のメンバーは，ファイルを読み書きするために必要な情報により構成されている．例えば,「次のファイルの読み書きの位置」「残っている文字数」「ファイルの番号」等である．こんなことは諸君は知る必要はない．ただし，ファイルポインターを使って，ファイルを指定していると考える．ファイル名だけではファイルを読み書きできない!!.

例えば，プログラムは，次のように書く．

```
fprintf(hoge, "%d\t%f\n", i, d);
```

ハードディスクのファイルにデータを書き込むのは，ディスプレイにデータを出力するのと全く同じイメージである．実際，ファイルの内容を見ると，ディスプレイと同じであることが分かる．戻り値の整数は出力した文字数である．書き込みに失敗すると負の値を返す．

### 3.1.4 ファイルのクローズ

使い終わったファイルは，fclose() 関数を使って，クローズしなくてはならない．

```
fclose(hoge);
```

## 3.2 プログラム例

### 3.2.1 ファイル出力

ファイルにデータを書き出すプログラムの例として，三角関数の値を出力するプログラムをリスト 2 に示す．

リスト 2: 三角関数の値のファイル出力プログラム

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6    FILE *out_file;           // FILE型のポインターの宣言
7    double x, y1, y2, y3;
8    double dphi;
9    int i, n;
10
11   n = 360;
12
13   dphi = 2*M_PI/n;
14
15   out_file = fopen("trifunc.txt","w");      //ファイルのオープン
16
17   for(i=0; i<n; i++){
18     x=i*dphi-M_PI;
19     y1 = sin(x);
20     y2 = cos(x);
21     y3 = tan(x);
22     fprintf(out_file, "%e\t%e\t%e\t%e\n", x, y1, y2, y3);  //書き込み
23   }
24
25   fclose(out_file);                         //ファイルのクローズ
26
27   return 0;
28 }
```

実行結果

ファイル「trifunc.txt」の内容は，次の通り．

```
-3.141593e+00 -1.224606e-16 -1.000000e+00 1.224606e-16
-3.124139e+00 -1.745241e-02 -9.998477e-01 1.745506e-02
-3.106686e+00 -3.489950e-02 -9.993908e-01 3.492077e-02
-3.089233e+00 -5.233596e-02 -9.986295e-01 5.240778e-02
-3.071779e+00 -6.975647e-02 -9.975641e-01 6.992681e-02
-3.054326e+00 -8.715574e-02 -9.961947e-01 8.748866e-02
-3.036873e+00 -1.045285e-01 -9.945219e-01 1.051042e-01

   この辺は長いので省略

3.089233e+00 5.233596e-02 -9.986295e-01 -5.240778e-02
3.106686e+00 3.489950e-02 -9.993908e-01 -3.492077e-02
3.124139e+00 1.745241e-02 -9.998477e-01 -1.745506e-02
```

### 3.2.2 ファイル入力

リスト 3 に先ほど作成したファイル内容を読み込み，各列の 2 乗平均値を計算する．

リスト 3: ファイルの内容を読み込むプログラム例

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    FILE *in_file;
7    double x[500], y1[500], y2[500], y3[500];
8    double sum2_x=0, sum2_y1=0, sum2_y2=0, sum2_y3=0;
9    int i=0;
10
11
12   in_file = fopen("trifunc.txt","r");
13
14   while(fscanf(in_file, "%lf%lf%lf%lf",&x[i], &y1[i], &y2[i], &y3[i])!=EOF){
15     sum2_x += x[i]*x[i];
16     sum2_y1 += y1[i]*y1[i];
17     sum2_y2 += y2[i]*y2[i];
18     sum2_y3 += y3[i]*y3[i];
19     i++;
20   }
21
22   fclose(in_file);
23   printf("ave x^2=%f\n",sum2_x/i);
24   printf("ave y1^2=%f\n",sum2_y1/i);
25   printf("ave y2^2=%f\n",sum2_y2/i);
26   printf("ave y3^2=%f\n",sum2_y3);
27
28
29   return 0;
30 }
```

実行結果

```
ave x^2=3.289919
ave y1^2=0.500000
```

```
ave y2^2=0.500000
ave y3^2=533454075936799969852241897062400.000000
```

# 4 さまざまな関数とライブラリー

## 4.1 再帰関数とは

関数の定義において，自分自身を呼び出す関数を再帰関数という．具体的には，リスト 4 のような階乗を計算する関数である．階乗は，

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 2 \times 1 \tag{1}$$

であるが，漸化式を使って

$$n! = n \times (n-1)! \qquad (\text{ただし}, 1 \leqq n) \tag{2}$$

$$0! = 1 \tag{3}$$

のように定義することもできる．この漸化式そのものをプログラムにすると，リスト 4 のように再帰関数を使うことになる．

リスト 4: 繰り返し文を使った階乗の計算.

```
1  #include <stdio.h>
2
3  int kaijyo(int n);                    // プロトタイプ宣言
4
5  //========== メイン関数 ================================
6  int main(void)
7  {
8    int nx, result;
9
10   scanf("%d", &nx);                   // 整数入力
11   result=kaijyo(nx);                  // 関数呼出し
12   printf("%d!=%d\n", nx, result);     // 計算結果表示
13
14   return 0;
15 }
16
17 //========== 階乗を計算する関数(再帰呼出し)===============
18 int kaijyo(int n)
19 {
20
21   if(n==1){
22     return 1;
23   }else{
24     return n*kaijyo(n-1);
25   }
26
27 }
```

実行結果

```
12
12!=479001600
```

次の二つのことをおさえれば，再帰関数を書くことができる．

- 再帰関数の終了条件を書く．
- 漸化式のように問題を分割し，その通りにプログラムを書く．ただし，問題の分割はつぎのようにする．
  - 分割したものを合わせることにより，元の問題となる．
  - 分割したひとつの問題は，元の問題よりも小さい．
  - 分割したひとつの問題は，元の問題と同じ方法で解ける．

  このように分割することにより，ある自明な解—再帰関数の終了条件—まで分割できる．そうすると問題が解けるのである．

## 4.2 プログラム例

## 4.3 2進数への変換

入力された正の整数を 2 進数で表示するプログラムを書く．これは再帰関数を使って，記述することができる．10 進数から 2 進数に変換するとき，2 で割ってあまりを書く—という作業を繰り返したことを思い出してほしい．同じことを繰り返したはずである．

このプログラムを書くためには，2 で割った商とあまりの演算が必要である．商の演算には「/」を，あまりの演算には「%」をつかう．これは整数同士の演算でしばしば使われるので憶えておく必要がある．

準備はできた．再帰関数を使った 2 進数への変換プログラムは，リスト 5 のようになる．

リスト 5: 正の 10 進数を 2 進数になおすプログラム．

```
1  #include <stdio.h>
2
3  void print_binary(int n);                    // プロトタイプ宣言
4
5  //========== メイン関数 ================================
6  int main(void)
7  {
8    int nx;
9
10   scanf("%d", &nx);                    // 整数入力
11   print_binary(nx);                    // 関数呼出し
12   printf("\n");
13
14   return 0;
15 }
16
17 //========== 2進数を表示する関数(再帰呼出し)===============
18 void print_binary(int n)
19 {
20
21   if(n==0){
```

```
22      return;
23    }else{
24      print_binary(n/2);
25      printf("%d",n%2);
26    }
27
28  }
```

実行結果

```
1234
10011010010
```

## 4.4 フィボナッチ数列

フィボナッチ数列も再帰関数の代表的な問題である．この数列は次のようなものである．

> 成熟した 1 つがいのウサギは，1ヶ月後に 1 つがいのウサギを生むとする．そして，生まれたウサギは 1ヶ月かけて成熟して次の月から毎月 1 つがいのウサギを生む．全てのウサギはこの規則に従うとし，死ぬことは無いとする．$n$ヶ月後にウサギはつがいの数は?

この数列は，

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots \tag{4}$$

となる．漸化式で表すと

$$F_n = F_{n-1} + F_{n-2} \tag{5}$$

$$F_0 = 1 \tag{6}$$

$$F_1 = 1 \tag{7}$$

である．リスト 6 にこの数列の計算プログラムを示す．

リスト 6: フィボナッチ数列 (ウサギの問題).

```
1  #include <stdio.h>
2
3  int fibonacci(int n);                    // プロトタイプ宣言
4
5  //========== メイン関数 ================================
6  int main(void)
7  {
8    int month;
9
10   printf("何ヶ月後?\t");
11   scanf("%d", &month);                          // 整数入力
12   printf("つがいの数 = %d\n", fibonacci(month));  // 関数呼出し
13
14   return 0;
15 }
```

```
16
17 //========== 2進数を表示する関数(再帰呼出し)===============
18 int fibonacci(int n)
19 {
20
21   if(n==0 || n==1){
22     return 1;
23   }else{
24     return fibonacci(n-1)+fibonacci(n-2);
25   }
26
27 }
```

実行結果

```
何ヶ月後?        8
つがいの数 = 34
```

# 5 ポインター

## 5.1 基本

ポインターとは何か?—と問われると，最も適切な答えは「ポインターはオブジェクトのアドレスである」であろう．そうするとオブジェクトとは何か?—ということになる．オブジェクトとはメモリー上のデータ領域のことでアドレスがあり，型を持つのでサイズもある．ポインターはオブジェクトの先頭アドレスを示すことにより，オブジェクトを指し示す．実際には，ポインター変数にオブジェクトの先頭アドレスを格納しているのである．先頭アドレスのみならず，オブジェクトの大きさもどこかに情報として持っていることを忘れてはならない．

このことを具体例をつかって，説明しよう．リスト 7 のプログラムでは，p がポインター変数で hoge がオブジェクトである．この関係は図 1 のように表すことができる．

リスト 7: ポインターを使った簡単なプログラム．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int hoge=3;
6    int *p;
7
8    p=&hoge;
9
10   printf("hoge=%d\n",*p);
11
12   return 0;
13 }
```

実行結果

```
hoge=3
```

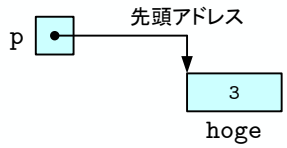


図 1: ポインター p とオブジェクト hoge の関係.

このような結果が得られるのは，リスト 7 の 10 行目でポインター p が hoge を指し示すからである．オブジェクトとポインターの関係は 8 行目で hoge で決められている; オブジェクトの先頭アドレス[2] をポインター変数に代入している．

## 5.2 さまざまなポインター

### 5.2.1 構造体へのポインター

構造体のポインターの例をリスト 8 にしめす．図 2 にポインターと構造体の関係を示す．リストを見て分かるように，構造体へのポインターは次のようにして使う．

- アドレス演算子&を使い先頭アドレスを取り出し[3]，ポインター変数へ代入する．
- ポインターが指し示す構造体のメンバーへのアクセスは，アロー演算子 (->) を使う．

リスト 8: 構造体へのポインターを使ったプログラム例.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    typedef struct{                          // 構造体の定義
6      char name[16];
7      int math;
8      int info;
9    }student;
10
11   student yama={"yamamoto",72,83};         // 宣言と初期化
12   student *p;                              // 構造体 student 型へのポインター
13
14   p=&yama;                                 // 先頭アドレスの代入
15
16   printf("name = %s\n", p->name);          // メンバーへのアクセスは，アロー演算子
17   printf("math = %d\n", p->math);
```

[2] int は 4 バイトなので 4 つのアドレスがあるが，アドレス演算子&で先頭アドレスを取り出している．そして，それをポインター変数に代入している

[3] ポインターを取り出している—と表現した方がよい．しかし，混乱するだろう．ほとんどの場合，ポインターはオブジェクトの先頭アドレスである．

```
18    printf("info = %d\n", p->info);
19
20    return 0;
21  }
```

実行結果

```
name = yamamoto
math = 72
info = 83
```

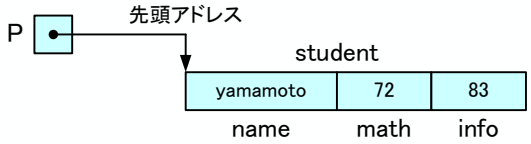


図 2: ポインター p と構造体のオブジェクト student の関係.

## 5.3 関数へのポインター

諸君にとって，最もけったいに感じるのが関数へのポインターであろう．実行すべき命令が書かれている関数もメモリーにロード—読み込んで格納—-される．したがって，先頭アドレス—関数のエントリー—をポインターに代入することができる．

リスト 9: 関数へのポインターを使ったプログラム例.

```
1  #include <stdio.h>
2
3  int add(int i, int j);              // プロトタイプ選言
4
5  //============ メイン関数================================
6  int main(void)
7  {
8    int (*fp)(int, int);              // 関数へのポインター
9
10   fp=add;                           // 関数のアドレスを代入
11
12   printf("%d\n",fp(5,9));
13
14
15   return 0;
16 }
17
18 //============ ユーザー定義関数==========================
19 int add(int i, int j)
20 {
21   return i+j;
22 }
```

実行結果

```
14
```

これは使い方によってはかなり便利である．サブルーチンへ関数を渡すことが可能となる．リスト 10 がそれを使った例である．ここで，関数定義の仮引数の `double (*f)(double)` が関数へのポインターの宣言で，

戻り値の型 (*関数へのポインター変数)(引数の型)

と書く．この関数へのポインター変数に関数のポインター—先頭アドレス—を代入すると，ポインター変数が関数のように使える．関数のポインターへの代入は関数名を右辺値として，代入するだけである．

リスト 10: 関数を引数にしたプログラム例.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  void print_func(double (*f)(double));  // プロトタイプ選言
5
6  //======== メイン関数 ========================================
7  int main(void)
8  {
9
10   print_func(sin);
11   print_func(cos);
12
13   return 0;
14 }
15
16 //========== 関数の値を表示する関数 ============================
17 void print_func(double (*f)(double))
18 {
19   int i;
20   double dx=0.1;
21
22   printf("--------------------------\n");
23   for(i=0; i<=5; i++){
24     printf("%f\t%f\n", i*dx, f(i*dx));
25   }
26 }
```

実行結果

```
--------------------------
0.000000        0.000000
0.100000        0.099833
0.200000        0.198669
0.300000        0.295520
0.400000        0.389418
0.500000        0.479426
--------------------------
0.000000        1.000000
0.100000        0.995004
0.200000        0.980067
0.300000        0.955336
```

```
0.400000        0.921061
0.500000        0.877583
```

## 5.4 ポインターの配列

複数組の文字列を扱う場合，ポインターの配列は便利である．リスト 11 にプログラム例を，図 3 にポインターと文字列定数の関係を示す．ここでの動作は次のようになる．

- プログラムの実行に先だって，プログラムがロードされたときにメモリーのどこかに文字列定数の文字列が格納される．
- そして，文字列のポインター—先頭アドレス—がポインターの配列に格納される．
- これらの処理が完了した後に，プログラムが動作する．

リスト 11: ポインターの配列を使ったプログラム例．

```c
#include <stdio.h>

int main(void)
{
  char *animal[]={"cat", "dog", "rabbit", "horse"};

  printf("1st : %s\n", animal[0]);
  printf("2nd : %s\n", animal[1]);
  printf("3rd : %s\n", animal[2]);
  printf("4th : %s\n", animal[3]);

  return 0;
}
```

実行結果

```
1st : cat
2nd : dog
3rd : rabbit
4th : horse
```

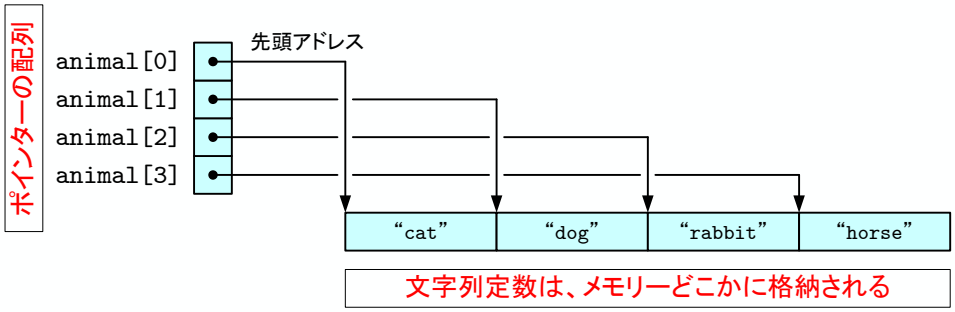


図 3: 文字列定数とポインターの配列 animal の関係.

## 5.5 メモリー配置

プログラムを実行するとき，メモリーはいろいろなデータがある．プログラム実行に必要なデータは 4 つのメモリー領域に分けて格納される．

- プログラムの関数は，**コード 領域**に格納される．
- グローバル変数や文字列定数は，**データ領域**に格納される．
- 次節で述べる malloc() では，**ヒープ領域**が確保される．
- ローカル変数のメモリー割り当てにはスタックと呼ばれる仕組[4]みが使われている．このようなことから，ローカル変数は，**スタック領域**に格納される．

C 言語では大雑把に言って，コード (code)、データ (data)、ヒープ (heap)、スタック (stack) の 4 つの領域にメモリーを分けて，管理する．これらの使い分けに，プログラマーはほとんど気にする必要はない．ただし，変数—配列や構造体を含む—を使う場合, メモリーは次のような使い方があると，プログラマーは認識しておくべきである．

- 静的変数 (static) やグローバル変数，文字列定数などは，メモリーの確保も解放もしない変数である．これらの変数はデータ領域に格納される．
- 自動変数 (auto) の場合，メモリー確保と解放は自動に行われる．自動変数は，スタック領域に格納される．
- プログラム中で malloc 関数を使い，ヒープ領域にメモリーの確保ができる．確保したメモリーを開放する場合には，free() 関数をつかう．

[4] 今は，分からなくてよい．

## 5.6 動的メモリーの確保

大量のデータを処理するために，大きな配列をローカル変数で宣言するとコンパイルはできても，実行時にエラーとなることがある．大きなメモリー使えない理由は，スタック領域が狭いことによる．先に述べたようにローカル変数はスタック領域を使うため，それに制限されるのである．

大きな配列を使いたい場合，ヒープ領域をつかう．malloc() 関数によりメモリーを確保して，free() 関数によりメモリーを開放する．その例をリスト 12 に示す．

リスト 12: 文字定数や静的変数やユーザー定義関数のアドレス．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    int *a;
7
8    a=malloc(sizeof(int)*1024*1024*10);
9
10   a[0]=1;
11   printf("a[0]=%d\n",a[0]);
12
13   free(a);
14
15   return 0;
16 }
```

実行結果

```
a[0]=1
```

malloc() 関数を使って，メモリーを確保している．ただし，この関数でいつも思い通りのメモリーを確保できるとは限らない．この関数は，メモリー確保に失敗した場合，NULL ポインターを返す．