C 入門 —gcc の使い方—

浅川伸一

2002 年 11 月 8 日

目 次

1 この文書の目的 **1**

2 gcc の入手 **1**

3 表記上の注意 **1**

4 コンパイル **1**
4.1 コンパイルとリンク 2
4.2 ライブラリ 3

5 gdb **3**

6 初めてのプログラム —文字の出力— **4**
6.1 `Hello, world!` 4
6.2 エスケープキャラクタ 5

7 簡単な数式計算 **5**
7.1 加減乗除 5
7.2 剰余演算子 7

8 変数 **7**
8.1 変数の定義 7
8.2 変数宣言 8
8.3 変数命名規則の制限 9
8.4 `printf()` 関数の書式 9

9 配列 **10**
9.1 C におけるコメント文の書き方 11
9.2 配列の要素の参照と代入 12
9.3 2 次元配列について 14

9.4 #define 15
9.5 型変換キャスト 15

10 繰り返し **16**
10.1 for ループ 16
10.2 while ループ 17
10.3 条件式に使われる演算子 18
10.4 do while ループ 19

11 条件分岐 **20**
11.1 if 文 20
11.2 switch と case による条件分岐 21
11.3 break と continue 23

12 文字列 **24**
12.1 ASCII コードについて 24
12.2 一文字型変数 char 24
12.3 文字の配列，文字列 25
12.4 文字列の初期化が許される場合 27
12.5 シングルクォートとダブルクォート 29

13 関数 **30**
13.1 数学の関数との比較 30
13.2 関数宣言と関数の実装 31
13.3 #include について 33
13.4 関数の再帰呼び出し 34
13.5 main() 関数の引数 35
13.6 文字列を数値に変換する関数 39
13.7 文字種判定関数 42
13.8 乱数系列生成関数 rand() 42

14 ポインタ **47**
14.1 ポインタの定義 47
14.2 デバッグライト 48
14.3 call by value 48
14.4 引数に変数のアドレスを用いる 49
14.5 ポインタ変数宣言 50
14.6 ポインタの有用性 53
14.7 配列とポインタ 53

15 ファイル入出力 **56**
15.1 `FILE` 型ポインタ変数 56
15.2 `fopen(), fclose()` 57
15.3 ファイル入出力関数群 58

16 構造体 **64**
16.1 構造体の宣言 64
16.2 構造体を使ったニューロンの表現 65
16.3 構造体へのメモリの動的割り付け 67
16.4 構造体のメモリイメージ 75

17 演算子の優先順位 **77**

1　この文書の目的

まったく C のことを知らない読者を対象に，初歩的なニューラルネットワークのシミュレータのプログラムが書けることを目的としている．従って標準的な教科書には必ず載っていて，欠くことのできない重要な項目の幾つかも本書では全く言及していない項目が多数ある．例えば低レベルのファイル入出力関数群 `open()`, `read()`, `write()` などは言及しなかったし，文字列操作関数群にもほとんど触れていない．本書を読んで C によるプログラミングに興味を持った読者はしかるべき文献にあたって知識を深められたい．

C を学ぶ場合の教科書としては、やはり C の開発者が執筆したオリジナルで定評のある K & R がよいだろう．K & R とはカーニハン、リッチ著、石田晴久訳、「プログラミング言語 C 第 2 版」共立出版のことである．

2　gcc の入手

Windows 上の C の処理系は cygwin と呼ばれる擬似 UNIX 環境をインストールするのがよいだろう．以下の URI から入手できる．

`http://sources.redhat.com/cygwin/download.html`

ただし，最近の cygwin はデフォルトでは C コンパイラがインストールされない．インストール時に C コンパイラを明示的にチェックしなければインストールされない．全ての項目にチェックを付けてフルインストールすると，遅い回線では数時間かかる場合もあるので注意が必要である．

PC UNIX(Linux, FreeBSD 等) を使っている場合には C の処理系 (gcc or egcs) はデフォルトでインストールされているはずである．

3　表記上の注意

日本語キーボードには￥マークがあるが，これは US キーボードではバックスラッシュ \ と同じコードが割り当てられている．したがって本文中の\は￥と読みかえても構わない．

また `Enter` はエンターキーを押下する意味である．

4　コンパイル

C のプログラムを作るとは,

- テキストエディタなどでソースコードを入力し `.c` という拡張子をつけて保存する．

- コマンド gcc によって実行ファイルを作る

という作業をいう．

コンパイル時には，エラーがない限りメッセージは表示されない．慣れないうちは`-Wall` オプションを指定してコンパイルする習慣を身に着けた方がよい．`-Wall` オプションを付けてコンパイルすると，書いたプログラムの欠点を指摘してくれる． `-Wall` オプションとは警告 (Warning) をすべて (all) 表示しなさいと言う意味である。

C のソースファイルをコンパイルし実行可能なプログラムを作るにはコマンドラインから

`gcc -Wall（ソースファイル名）-o（実行ファイル名）` [Enter]

とタイプする．`-o`（実行ファイル名）を省略すると `a.out` というファイル名で実行ファイルが作成される．Windows では `a.exe` というファイルが作成される．

実行例

例 1 `gcc -o first first.c` [Enter]
`first(first.exe)` という実行ファイルができる

例 2 `gcc first.c` [Enter]
`a.out(a.exe)` という実行ファイルができる

例 3 `gcc -o second first.c` [Enter]
`second(second.exe)` という実行ファイルができる

4.1 コンパイルとリンク

実行ファイルではなくリロケータブルオブジェクトを作るには `-c` オプションを使う．リロケータブルオブジェクトの拡張子は `.o` となる．

この方法は一つのプログラムが複数のファイルによって構成されるときに使われる．リロケータブルオブジェクトから実行ファイルを作るときにはリンクをする必要がある．

実行例

例 1 `gcc -c first.c` [Enter]
`first.o` というオブジェクトファイルができる

例 2 `gcc -c a1.c a2.c a3.c` [Enter]
`a1.o a2.o a3.o` というオブジェクトファイルができる

リロケータブルオブジェクトファイルをリンクして，実行ファイルを作る場合にも gcc を使う。以下の例は `a1.o a2.o a3.o` という 3 つのファイルから f という実行ファイルを作成する例である．

`gcc -o f a1.o a2.o a3.o` [Enter]

4.2　ライブラリ

数学関数を使ったプログラムを作る場合は数学関数ライブラリをリンクする必要がある。-l オプションを使う.

gcc（他の引数）-l(ライブラリ名) Enter

ここで l と ライブラリ名の間にはスペースが入らないことに注意してほしい. sin(), cos(), sqrt() などの数学関数を用いる場合は -lm となる.

実行例

例 1 gcc -o first -c first.c -lm Enter

first.c をコンパイルし数学ライブラリをリンクして first(first.exe) という実行ファイルを作る

5　gdb

プログラムにミスがあるとそのプログラムは強制終了する。このとき Bus error（core dumped）のようなメッセージが表示され、core というファイルにエラーに関する情報が書き出される. core ファイルは可読形式のファイルではないので通常の方法では読み出せない.

上述のような実行できないプログラムを修正することをデバッグするという. デバッグのためには gdb と呼ばれるデバッガを用いる. gdb でデバッグするためには実行ファイルにデバッグ情報を埋め込むオプション -g が必要である.

gcc -g（その他の引数）Enter

gdb は次のようにして起動する。

gdb（実行ファイル名）core Enter

gdb が起動するとプロンプト（gdb）が表示される. プログラムを起動するためには

(gdb) run Enter

とする. プログラムが中断してしまうときには

(gdb) where Enter

とするとどこで停止したかが表示される.。gdb を終了するには

(gdb) quit Enter

とタイプする.

6　初めてのプログラム —文字の出力—

6.1　Hello, world!

カーニハンとリッチーの教科書にしたがって，まずは画面に Hello, world と表示するプログラムを作成する．ファイル名は hello.c などとして，以下のプログラムを入力し gcc でコンパイルしてほしい．

```
#include <stdio.h>

int  main(void)
{
    printf( "Hello, world!\n" );
    return  0;
}
```

このプログラムは，画面に，Hello, world! と表示させるプログラムである．

まず，１行目の#include <stdio.h>であるが，今は C プログラムを作るときのおまじないのようなものと考えて後回しにしておく．

次の int main(void) は,「ここからプログラムが始まりまる」と知らせるためのものである．逆に，このプログラムの最後の return 0; は,「ここでプログラムが終了する」ということを知らせるためのものだと考えてよい．C の文の最後には必ずセミコロン；がつく[1]．

int main(void) でプログラムが始まるとして、次の行の{ は，最後の} と対応関係にあり,「ここで１つのまとまりを示す」ことを明示するためのものである．

その次の行で，画面に文字を出力している．任意の文字列を画面に表示するには printf() 関数を使う．

printf() の使い方は，表示したい文字列を"ダブルクオート"でくくればよい．上の例題のプログラムを見てみると，

```
    printf( "Hello, World!\n" );
```

となっている．この例題のプログラムをコンパイルして実行すると，この部分は,

```
    Hello, World!
```

と表示されるだけで，printf() に指定した最後の \n が表示されない．

---

[1] したがってもっとも短い C のソースコードは
int main(void){return 0;}
となる．このプログラムは開始と終了だけからなるなにもしないプログラムである．

6.2 エスケープキャラクタ

C の \ 記号はエスケープキャラクタとよばれ特別な意味がある．\ の後ろに来る文字によって，色々な働きをする．以下の表にその一部を示す．次の

表 1: エスケープキャラクタ

記号	意味
\n	改行
\t	タブ文字
\r	キャリッジ・リターン (カーソルを行頭に戻す)
\x 数値	数値は 16 進数（ 例 : \xA6 は 10 進数で 166 ）
\0	ヌル文字（ 文字列の終端を表す）
\\	\ そのものを表す.

ように，

```
printf( "Hello, \n\n\nWorld\n" );
```

とすると，3 行改行されるので，

```
Hello,



World!
```

となり，２つの文字列が離れて表示される．

7 簡単な数式計算

7.1 加減乗除

第 6 章で `printf( "???" );` の ??? のところに文字を指定すれば，画面にその文字が出力できることを記した．では，$1 + 1$ を計算し，それを出力するにはどうしたらよいのだろうか．普通に

```
printf( "1 + 1\n" );
```

とやると，画面には，

```
1 + 1
```

と表示されるだけで，答えの 2 は表示してくれない．計算結果を表示するには以下のようにする．

```
printf( "%d\n", 1 + 1 );
```

こうすると，画面に 2 が表示される．また，次のように

```
printf( "1 + 1 = %d.\n", 1 + 1 );
```

とすると，画面には

```
1 + 1 = 2.
```

と表示される．??? の中にある %d が，??? の外にある $1 + 1$ に対応している．
次のように，

```
printf( "1 + 3 = %d, 3 * 2 = %d.\n", 1 + 3, 3 * 2 );
```

（注意：* は，算術積演算子．つまり「かける」を表す.）とすると，

```
1 + 3 = 4, 3 * 2 = 6.
```

と表示される．ダブルクオート " の中にある 1 個目の %d が外の $1 + 3$ に，2 個目の %d がカンマ越しの $3 * 2$ に対応している．これを，視覚的に表すと，

```
printf( "1 + 3 = %d, 3 * 2 = %d . \n", 1 + 3,  3 * 2 );
                 ┬            ┬         ──┬──    ──┬──
                 └────────────┼───────────┘        │
                     対応     └────────────────────┘
                                       対応
```

となる．すなわち，%d の位置に表示したい数式の答えが表示され，その数式をダブルクオート" の外にカンマで区切って記述しておけば，計算式の結果が表示できる．

ただし，%d での出力は答えが整数の場合に限る．これは以下の例で確かめられる．次のように 5 / 2 を表示させようとすると，

```
printf( " 5 / 2 = %d .\n", 5 / 2 );
```

画面には

```
5 / 2 = 2.
```

と表示されてしまう．

小数を表現するには，%d でなく，%f を使う．そして，5 や 2 だけでは，コンピュータの内部では小数でなく，整数として扱われてしまうので，5.0 と 2.0 にしておく．すなわち

```
printf( " 5 / 2 = %f.\n", 5.0 / 2.0 );
```

とする．こうすれば，

```
5 / 2 = 2.500000.
```

と表示される．%d や %f のような数値を出力する部分は表示する桁数を定義することができる．たとえば

```
printf( " 5 / 2 = %3.2f.\n", 5.0 / 2.0 );
```

とすると

```
5 / 2 = 2.50.
```

と表示される．3 が全体の表示桁数であり . 以下が小数点の表示桁数である．

5 や 2 だけでは，コンピュータの内部では小数でなく，整数として扱われてしまうことは既に書いたが，これは C の制限でなく，Fortran, Pascal, BASIC など，整数と小数の違いを明確にしている言語一般に起こる現象である．小数で表現したい場合は，たとえ実世界で .0 は不要であっても，コンピュータの世界では.0 が必要となると覚えけばよいだろう．

7.2 剰余演算子

加減乗除が +-*/ で表現されているのは見てきたとおりである．この他に剰余演算子と呼ばれる演算子があって % を用いる．剰余演算子は割り算の余りを求めるために使用される．例えば，10 / 3 の商と余りを求めるには以下のようにする．

```
int  iResidual. iQuotient;

iQuotient = 10 / 3;
iResidual = 10 % 3;
```

上記のようにすると iQuotient には 3 が，iResidual には 1 が代入される．ただし，剰余演算子 % は，整数演算にしか適用できない．

演習問題 79073 を 283 で割ったときの商と余りを表示するプログラムを書け

8 変数

8.1 変数の定義

プログラムにおいて，そのように演算結果を一時的に記憶させる入れ物，場所のようなものを変数と言う．

変数を使って，3 つの数字 80, 90, 75 の平均と分散を求めるプログラムを書いてみると，

```
#include <stdio.h>
#include <math.h>

int  main(void)
{
    float  mean, variance, sd;

    mean = ( 80.0 + 90.0 + 75.0 ) / 3.0;
    variance = ( ( 80.0 - mean ) * ( 80.0 - mean ) +
                 ( 90.0 - mean ) * ( 90.0 - mean ) +
                 ( 75.0 - mean ) * ( 75.0 - mean ) ) / 3.0;
    sd = sqrt( variance );

    printf( " mean, variance, standard deviation, \n" );
    printf( "%f, ", mean );
    printf( "%f, ", variance );
    printf( "%f.\n", sd );

    return  0;
}
```

平方根を求める `sqrt()` を使うには `#include <math.h>`が必要であり，コンパイル時には 4.2 章で説明した `-lm` オプションが必要である．

8.2 変数宣言

プログラムの解説をすると，まず，`int main(void)` の中で，

```
float  mean, variance, sd;
```

と書かれている．これらは変数宣言と言い，`mean`，`variance`，`sd` という名前を持つ 3 つの変数 — 数値を格納しておく場所を確保するということ — を宣言している．また，それら 3 つの名前の変数には浮動小数点数型 `float` (整数ではなく小数も表現できる変数型) という 型を付け，その 3 つの変数が小数点を含む数を格納する，ということを明示しておく．ちなみに，計算式が整数値演算で用が足りる場合は,

```
int mean, variance, sd;
```

のようにする．`int` は整数 integer の略である．gcc の 32 ビットコンパイラでは，`int` で宣言された変数は，-2,147,483,648 から 2,147,483,647 の範囲でしか演算できない.

変数の種類としては，文字型 (char)，整数型 (int)，ロング整数型 (long)，浮動小数点型 (float)，倍精度浮動小数点型 (double) などがある．これらの変数型には unsigned という修飾子をつけて正の値しか取らないようにすることもできる．また，後述する typedef 宣言によって新たな型を定義することもできる．それぞれの変数が何バイトからなっているかを知る方法として sizeof 演算子がある．sizeof 演算子の使い方は 14.7 章で説明している．

8.3 変数命名規則の制限

変数の名前の付け方や文字数には制限がある．

1. 数字で始まらない．つまり，100_mean などはダメ．
2. a から z の英文字の小文字，A から Z までの英文字の大文字，それと _ (アンダーバー) が使える．
   良い例：Standard_Deviation_of_100
   悪い例：Standard Deviation @ "100"(スペースや @ や ” などの特殊文字が入っている．)
3. 原則として，変数に付けられる文字数は 31 文字以下．ただし，gcc コンパイラは，これ以上の文字数でも良いようである．他のコンパイラとの移植性を考える場合，31 文字以下にしておいた方が無難ではある．
4. 変数名の大文字と小文字は区別される．すなわち，Index と index は，違う変数として認識される．

8.4 printf() 関数の書式

printf() 関数内では型と数値の関係は次のようになっている．

表 2: printf() 関数で使われる仮引数

記号	型
%d	int 型
%f	float 型 および double 型
%e	float 型 および double 型 の 10 を底とする科学表記
%c	char 型
%s	文字列 char の配列
%x	16 進数
%p	ポインタ変数の値を 16 進数で表示

そのほかにも，いろいろな記号があるが省略する．

ちなみに，int 型で宣言された変数の範囲は，機種依存あるいはコンパイラ依存である．コンパイラによっては -32,768 から 32,767 の範囲でしか演算できない処理系も存在する．他の環境でも自分のプログラムを動かしたい場合，変数で扱える数値の範囲に注意してプログラムを作成する必要がある．

9 配列

プログラムを作成する過程で数値などを一時的に保存させておく必要があった場合，その変数名と格納させる値の型の 2 つを指定すれば，変数が使えるということを記した．

前章のプログラムで複数の平均や分散などを求めることを考える．以下のようなプログラムが考えられよう．

```
/*
                      テスト 1, テスト 2, テスト 3
    被検者 A の得点  :  80 点,  90 点,  75 点
    被検者 B の得点  :  30 点,  30 点,  25 点
    被検者 C の得点  :  90 点, 100 点,  90 点
*/
#include <math.h>
#include <stdio.h>

int  main(void)
{
    float  fMean_of_1, fVar_of_1, fSD_of_1;
    float  fMean_of_2, fVar_of_2, fSD_of_2;
    float  fMean_of_3, fVar_of_3, fSD_of_3;

    fMean_of_1 = ( 80.0 + 30.0 + 90.0  ) / 3.0;
    fVar_of_1=(( 80.0 - fMean_of_1) * (80.0 - fMean_of_1) +
                (30.0 - fMean_of_1) * (30.0 - fMean_of_1) +
                (90.0 - fMean_of_1) * (90.0 - fMean_of_1) )
                 / 3.0;
    fSD_of_1 = sqrt( fVar_of_1 );

    fMean_of_2 = ( 90.0 + 30.0 100.0 ) / 3.0;
    fVar_of_2=((90.0 - fMean_of_2) * (90.0 - fMean_of_2) +
               (30.0 - fMean_of_2) * (30.0 - fMean_of_2) +
               (100.0 - fMean_of_2) * (100.0 - fMean_of_2))
                / 3.0;
```

```
    fSD_of_2 = sqrt( fVar_of_2 );

    fMean_of_3 = ( 90.0 + 100.0 + 90.0 ) / 3.0;
    fVar_of_3=((75.0 - fMean_of_3) * (75.0 - fMean_of_3) +
               (25.0 - fMean_of_3) * (25.0 - fMean_of_3) +
               (90.0 - fMean_of_3) * (90.0 - fMean_of_3))
               / 3.0;
    fSD_of_3 = sqrt( fVar_of_3 );

    printf( " Mean, variance, and S.D of test 1 are \n" );
    printf( "%f, ", fMean_of_1 );
    printf( "%f, ", fVar_of_1 );
    printf( "%f,\n", fSD_of_1 );

    printf( " Mean, variance, and S.D of test 2 are \n" );
    printf( "%f, ", fMean_of_2 );
    printf( "%f, ", fVar_of_2 );
    printf( "%f,\n", fSD_of_2 );

    printf( " Mean, variance, and S.D of 3 are \n" );
    printf( "%f, ", fMean_of_3 );
    printf( "%f, ", fVar_of_3 );
    printf( "%f.\n", fSD_of_3 );

    return 0;
}
```

9.1 C におけるコメント文の書き方

まず，C におけるコメントの付け方を記しておく．/* と */ で囲まれた部分は，コメントと見なされ，コンパイラはこの部分を無視する．すなわちコメント部分はプログラムの実行には影響を与えない．コメントはプログラマが処理の説明をしたり，注意書きを書いたり，そのときの記録を残しておくために使われたりする．

コメントは入れ子構造（ネスト構造）にはできない．例えば

```
    /*  /* 不正なコメント文 */  */
```

のようにして使うことができない．最初の */ でコメントが終了したと見なされ，あとの */ が，掛ける，割るの演算子として見なされてしまう．

9.2 配列の要素の参照と代入

上記のソースコードをコンパイルすれば正しい答えが出力されるが，ソースコードには無駄が多い．その理由は

- 平均，分散，標準偏差の値を格納するのに，テスト数と同じだけ変数宣言を行なっている．
- 点数から，平均，分散，標準偏差を求めるという演算は同じなのに，それをわざわざ 3 つのテストに同じことを繰り返している．

だからである．このような場合同じ属性を持つ変数を 1 つにまとめて表現してやればよい．同じ処理を繰り返すための変数を配列と言う．

例えば 3 つの平均値をひとまとめにしたければ

```
float Means[3];
```

などと表現すればよい．これを配列宣言という．あらかじめ配列の要素が決まっているなら，

```
float Means[3] = { 80, 30, 90 };
```

のようにすれば `Means[0]` には 80 が，`Means[1]` には 30 が，`Meand[2]` には 90 がそれぞれ代入される．このような方法を配列の初期化という．

配列の中の個々の要素の値を参照するには，`[ ]` でくくって参照したい番号を指定し，

```
printf( "Means[%d]=%f\n", 2, Means[ 2 ] );
```

のようにすればよい．配列の要素に値を代入するには

```
Means[2]=90;
```

とする．

C の場合には配列の最初が 0 であることに注意してほしい．これを 0 から始まるという意味で，ゼロオリジンと言ったりする．上記の例では

```
Means[3]=80;
```

などとするとエラーになる場合もあるが，エラーにならない場合がある．一般に N 個の要素からなる配列を宣言した場合，0 から N-1 番目までの N 個の要素が利用可能で，N+1 番目を参照したり，代入したりすると結果は保証されない．

配列を使ったソースコードを以下に示す．

```
#include <stdio.h>
#include <math.h>


#define STUDENT_MAX (3)
```

```
#define TEST_MAX    (3)

int main (void)
{
    int  i, j;
    int  iTest[ STUDENT_MAX ][ TEST_MAX ] =
             {{ 80, 90, 75 },
              { 30, 30, 25 },
              { 90,100, 90 }};
    float  fMean[ TEST_MAX ], fVar[ TEST_MAX ];
    float  fSD[ TEST_MAX ];

    for ( i = 0; i < TEST_MAX; i++ ) {
        fMean[i] = 0.0;
        fVar[i] = 0.0;
        fSD[i] = 0.0;
        for ( j=0; j < STUDENT_MAX; j++ ) {
            fMean[i] += ( (float)( iTest[j][i] )
                          /(float)STUDENT_MAX);
        }

        for ( j=0; j < STUDENT_MAX; j++ ) {
            fVar[i] += ( (float)( iTest[j][i] - fMean[i] ) *
                                ( iTest[j][i] - fMean[i] ));
        }
        fVar[i] /= (float)STUDENT_MAX;
        fSD[i] = sqrt( fVar[i] );
    }

    for( i = 0; i < TEST_MAX; i++ ){
        printf( "Mean,variance,S.D. of Test[%d] \n",i+1);
        printf( "%f, ", fMean[ i ] );
        printf( "%f, ", fVar[ i ] );
        printf( "%f.\n\n", fSD[ i ] );
    }
    return  0;
}
```

9.3 2 次元配列について

プログラムの中では 2 次元配列 iTest[ ][ ] が使われている．2 次元配列とは配列の配列の意味である．プログラム中では各行に各被検者を，各列に各テストの点数を割り当てた．

例えば 2 次元平面上の 3 点 (-1, 2), (1, 3), (2, 5) をまとめて 2 次元配列を定義するなら

```
int  iPoint[ 3 ][ 2 ] = {
    { -1, 2 },
    {  1, 3 },
    {  2, 5 }
};
```

のようにする．2 次元配列への参照は 1 次元配列と同じで

```
    printf("[%d,%d]=%d\n", i, j, iPoint[i][j]):
```

などとする．2 次元配列も 1 次元配列と同じくゼロオリジンである．

また，char 型の 2 次元配列もよく使われる．詳しくは 12.3 章 (p.25) で説明するが, char 型の配列は文字列を意味するのだから，文字列の配列が char 型 2 次元配列の意味である．

```
    char  strings[ 3 ][ 40 ] ={
        "Tokyo woman's christian university.",
        "2-6-1 Zempukuji, Suginami,",
        "zip code: 167-8585"
    };
```

などとなる．strcpy() 関数を使って文字列を代入するには,

```
    strcpy(strings[0], "Tokyo woman's christian university.");
    strcpy(strings[1], "2-6-1 Zempukuji, Suginami," );
    strcpy(strings[2], "zip code: 167-8585" );
```

とする．また，strings[2][0] = 'Z' とすれば strings[2] は "Zip code: 167-8585" となって先頭の z が大文字になる．

ただし，2 次元配列はあまり使われない．2 次元配列が使われない理由は,

- 汎用性がない．構造体の方が取扱いが便利である
- 配列なので，要素が固定長となり可変長のデータを格納するの難しい構造体を使った方が効率良い
- 動的なメモリ確保や解放が困難である．

などである．

9.4 #define

プログラムには，新しく #define というのが入っている．さらにキャストと呼ばれる変数型の変換が用いられている．

#define というのは，定数を宣言する方法の 1 つものだと理解してほしい．実際は別の用途もある．#define の定数宣言としての使い方は以下のとおりである．

```
#define  定数名  ( 値 )
```

コンパイル時に定数名は値に置換される．注意すべきことは，#define の文には終端にセミコロン ; を付けないことである．セミコロンを付けると，コンパイルエラーになる．それから，通常 #define で付ける定数名には大文字を用いるというプログラマが多い．

#define 文による定数宣言によってプログラムの汎用性，柔軟性が増すという利点がある．もし仕様変更があった場合，例えば，テスト数や被検者数を変更する必要がある場合，その都度 [ ] の中の 3 を 6 などに変えなければならない.「どうせ，その増えた 3 人の得点も記入するんだから，ついでだし，そんなの別に... 」と思うかも知れない．しかし，プログラムをよく見てみると，この数値は，配列の最大数の指定以外にも，for 文でループ制御を行なうのにも使っている．これはテストの回数と被検者数分の出力をせねばならないのだから当然である．仕様変更に柔軟に対処できるように，できるだけこのような場合には定数宣言を用いた方がよい．そうすれば，被検者数 3 人が 50 人になろうが 600 人になろうが，

```
#define  STUDENT_MAX  ( 600 )
```

に書き替えるだけで良い．

9.5 型変換キャスト

プログラムの平均と分散の計算式のところを見ると，以下のようなキャストが使われている．

```
( float )(iTest[j][i])/(float)STUDENT_MAX;
```

上記の (float) のような書式をキャストという．キャストを使って int 型で宣言された変数を float 型に変換することができる．整数と浮動小数点数の演算結果は，原則として整数となる．つまり，小数点以下の値が切り取られて演算されてしまう．そのため，数値に対しては .0 を付けた．既に宣言されている整数に対しては (float) を付ける．このことを，キャストする，あるいは型変換すると言う．

10　繰り返し

C で繰り返しを制御する命令は 3 種類あり，while 文，for 文，そして do{ }while(); 文である．

10.1　for ループ

for を使って繰り返し文を作るには，以下のようにする．

```
for ( 初期値 ; 繰り返し継続条件 ; 各繰り返しごとにする処理 ) {
    繰り返ししたい処理;
}
```

for の ( ) の中にはセミコロン ; で区切られた 3 つの項がある．これら 3 つをを指定することで，繰り返し回数や，繰り返される処理内容を制御することができる．3 つの項で繰り返し回数などの制御できることから，これらの項目を繰り返し制御変数と言ったりする．

最初の項である初期値では繰り返し制御変数の初期値を与える．例えば

```
for (i=0; i < TEST_MAX; i++)
```

において，i が 0 から TEST_MAX -1 まで TEST_MAX 回繰り返される．複数の変数を初期化しておきたければカンマ，で区切って書けばよい．なお，繰り返し制御変数は省略できる．for(;;) として無限に繰り返すようにすることもできる．

2 番目の項である繰り返し継続条件には，論理式，すなわち等号や不等号を指定する．この論理式が成立しているのならば繰り返しを継続することになる．上の例では，i が 0 から TEST_MAX-1 まで繰り返される．

最後の繰り返しごとにする処理では，繰り返し制御変数の値を変える．ここに指定されている i++ というのは，i の値を 1 増加させると言う意味である．またこれは i=i+1 と書いてもよい．もちろん i=i+1 は数学的には正しくない．C における = は代入演算子である．すなわち左辺の値 i に右辺の計算結果を代入せよという命令になる．C の等位演算子は == である．これは C の使用で仕方が無いのだが，この代入演算子 =と，比較または等位演算子 == の間違えによって引き起こされる問題は，例えば 11.1 で説明する if 文で，

```
if ( a = b ){
```

という書き方は混乱を招くし，コンパイラが誤りを検出してくれない．この if 文はおそらく a と b とが等しければという意味で書かれたものであろう．ところが = は代入演算子なので a に b の値を代入し，代入した結果が 0 でなれば{ } の内容が実行されてしまう．プログラマの中には，この種の誤りをふせぐために

```
if ( a == 1000 )
```

と書く代わりに

```
if ( 1000 == a )
```

と書くべきだと主張する人もいる．左辺値が定数になっているので，定数にはどんな値も入力されないからである．このようにしておけば，代入演算子と等位演算子との混乱をコンパイラが見つけてくれるからである．

またCでは

```
i += 2;
```

という書式が許されている．これは変数 i に 2 を加えよ，という命令であり，

```
i = i + 2;
```

と同じ意味である．同様に -=, *=, /= という演算子も用意されている．

演習問題 1 から 100 までのすべての奇数を表示するプログラムを書け．

演習問題 次のような掛算九九の表を表示するプログラムを書け

```
       1    2    3    4    5    6    7    8    9
-----------------------------------------
  2 :  2    4    6    8   10   12   14   16   18
  3 :  3    6    9   12   15   18   21   24   27
  4 :  4    8   12   16   20   24   28   32   36
  5 :  5   10   15   20   25   30   35   40   45
  6 :  6   12   18   24   30   36   42   48   54
  7 :  7   14   21   28   35   42   49   56   63
  8 :  8   16   24   32   40   48   56   64   72
  9 :  9   18   27   36   45   54   63   72   81
```

10.2 while ループ

while による繰り返しは

```
while ( 条件式 ) {
    処理;
}
```

のような形で用いる。条件式が真である限り処理を繰り返すことになる。すなわち

```
for (i=0; i < TEST_MAX; i++) {
    処理;
}
```

という for 文による繰り返しは

```
i=0;
while ( i < TEST_MAX ) {
    処理;
    i++;
}
```

と等価である。

for 文と while 文が異なるのは，for 文の場合，第２項は書かないと無限ループになる．一方 while 文では，必ず条件式を書かねばならないことである．while 文で無限ループにしたい場合は，while(0 以外の値) と書く必要がある．つまり

```
for( ; ; )
```

は,

```
while( 1 )
```

と等価である.

10.3 条件式に使われる演算子

条件式には次のような記号が使える

表 3: 条件式に用いられる演算子

記号	意味
==	比較等位演算子
!=	比較否定演算子
<=, >=, >, <	比較不等号演算子
&&	論理積
||	論理和
!	否定

次のようなプログラムを見てみよう.

```
#include <stdio.h>

int main (void)
{
    int i, x;
```

```
    for (i=1,x=1;x>0;i++) {
        x *= 2;
        printf("%d: %d\n",i,x);
    }
    x--;
    printf("%d\n",x);
    return 0;
}
```

このプログラムは 2 のべき乗を計算して表示するプログラムである．`x *= 2;` とは `x = x * 2;` と等価である．x の元の数を 2 倍せよという文である．これは 2 のべき乗を計算してることになる．一見，このプログラムは終了しないと思うかも知れない．2 のべき乗は無限個存在するのだから終らないはずである．ところが上のプログラムは (int 型が 32 ビットの処理系ならば) 31 回目で -2147483648 となって終了してしまう．さらに奇妙なことに，この繰り返しが終った後で `x--;` と変数 x から 1 を引いている．いったいどんな数字が表示されるのだろうか，答えがどうなるか予想できるだろうか．実は，このことからコンピュータ内部では整数がどのように表現されているのかが分かるのだが，ここではこれ以上立ち入らない．

演習問題 上記のプログラムを while() 文を使って書き直せ．

演習問題 while() 文を使って 1 から 100 までのすべての奇数の和を求める関数を書け．

10.4 do while ループ

3 番目の繰り返し制御文は `do{ }while();` 文である．書式は

```
do {
    繰り返して処理する演算
} while( 繰り返し継続条件 );
```

である．`while(繰り返し脱出条件);` の後にはセミコロンが必要である．
`do{ } while();` 文は while 文と似ているが，繰り返し継続条件の位置が処理の終りにある．すなわち `do{ }while();` 文では，繰り返したい処理を行なったあとで，繰り返し継続条件を満たしているか否かの条件判断がなされる．換言すれば `do {}` の内部の演算が最低 1 度は実行されることを意味している．一方 while() 文では繰り返し継続条件が偽であれば，繰り返される演算は一度も実行されないことが起こりうる．

11 条件分岐

11.1 if 文

int 型変数はおよそ -20 億から +20 億までしか演算できないことは既に述べた．ここではこのことを実習してみる．以下のプログラムを見て欲しい．

```
#include <stdio.h>

int main (void)
{
    int i, x;

    i=1; x=1;
    while ( 1 ) {
        x *= 2;
        printf("%d: %d\n",i,x);
        if ( x < 0 )
            break;
        i++;
    }
    return 0;
}
```

このプログラムには while (1) という無限回の繰り返しを含まれている．C の論理式の標記では 0 が偽で 0 以外が真である．この無限回の繰り返しを脱出する手段として if 文が使われている．break; とは繰り返しから脱出する命令である．上の例では変数 x が負になったら繰り返しを終了するという意味になる．if 文の一般的な書式は以下のようになる．

```
if ( 条件式 ) {
    条件式が真のときの処理;
} else {
    条件式が偽のときの処理;
}
```

となる．else 以下は省略が可能である．条件が複数あるときには

```
if ( 条件式 1 ) {
    条件式 1 が真のときの処理；
} else if ( 条件式 2 ) {
    条件式 2 が真のときの処理；
} else if ( 条件式 3 ) {
```

```
    条件式 3 が真のときの処理；
} else if ( 条件式 4 ) {
    ....以下繰り返し....
} else {
    すべての条件に当てはまらない場合の処理
}
```

という書式が許される．この場合条件判断は上から順になされるので，例えば条件式 1 と条件式 3 の両方で条件式が真になる場合でも条件式 1 が真のときの処理しか実行されない．従ってどのような順番で条件判断をするのかが問題になる場合がある．

演習問題： 次のうち処理 1 が実行されるのはどれか

1.
```
if( 0 ) {
    処理１;
 }
```

2.
```
if( 1 ){
    処理１;
}
```

3.
```
if( 10 * 8 < 80 ) {
    処理１;
}
```

4.
```
if( -1 ) {
    処理１;
}
```

5.
```
if( !( 10 == 9 ) ) {
    処理１;
}
```

11.2 switch と case による条件分岐

if ... else if ... else では条件判断の順番が問題になることを述べた．これ以外にもソースコードが読みにくくなるという欠点もある．

この問題を回避する方法として，switch, case 文を紹介する．switch case の文法は,

```
switch( 評価したい値 ) {
case  定数１ :
    評価したい値が定数１だったときの処理;
```

```
        break;

    case  定数２ :
        評価したい値が定数２だったときの処理;
        break;

    case  定数３ :
        評価したい値が定数３だったときの処理;
        break;
        ....
    default :
        評価したい値が，定数１でも定数２でも定数３でも，
        すなわちどれにも当てはまらなかったときの処理;
        break;
    }
```

である．break は繰り返しからの脱出だけでなく switch の脱出にも使われる．

定数 1 と定数 2 のどちらかを満たした場合に処理を行ないたい場合には，

```
switch( 評価したい値 ) {
case 定数 1 :
case 定数 2 :
    評価したい値が定数 1 あるいは定数 2 だったときの処理;
    break;
}
```

とする．switch 文は break に出会うまでは逐次命令が実行される．case 定数 1 : のあとに break がないので，switch の脱出をせずに，次の case 定数 2 : のあとの処理を実行する．そのあと，break があるので，switch の脱出をしている．これにより，if 文で

```
if( 評価したい値 == 定数１ || 評価したい値 == 定数２ ) {
    評価したい値が定数１あるいは定数２だったときの処理;
}
```

としていた動作と全く同じ動作が switch で実現できる．

演習問題 次のプログラムで，i が 1 のとき a はいくつになるか．i が 4 のとき a はいくつになるか．

```
switch( i ) {
case  1 :
    a = 1;
```

```
case  2 :
    a = 2;
    break;
case  3 :
    a = 3;
    break;
case  4 :
case  5 :
    a = 5;
}
```

11.3 break と continue

break; 文の他に繰り返しから脱出する文に continue; 文がある．break; 文は 繰り返しからの脱出を行なったのに対し，continue; 文は繰り返しの残りの部分をスキップするという意味がある．

break; 文と continue; 文の違いを図示してみると以下のようになる．まず，break; 文は繰り返しからの脱出であるから処理の流れは，

```
    for( ; ; ) {
        ....

        break;
┌─────────┘
|       ....
|   }
↓
（ break; は繰り返し脱出なので，次の処理に移る）
```

一方，continue; 文は繰り返しの残りの処理をスキップするのだから，プログラムの流れは，以下の図のようになる．

```
    for( ; ; ) {
┌→（ continue は繰り返しの残りの処理をスキップするので，
|       ここにたどり着く.）
|       ....
|
|       continue;
└────────────┘
        ....
    }
```

ただし，for 文の第 3 項で指定された繰り返しごとに行なわれる処理は実行される．

12　文字列

12.1　ASCII コードについて

ASCII コードとは，画面に表示されている英数字や，@ や改行コードなどの特殊記号を表すために割り当てられているキー識別番号である．大文字の 'A' に対応する ASCII コードは 0x41 である．小文字の 'a' に対応する ASCII コードは 0x61 である．C では数値の前に 0x をつけると 16 進数を意味することになる．従って

```
char  chA = 'A';
chA += 0x20;
printf("%c\n", chA);
```

などとすると画面には小文字の 'a' が表示される．すなわち大文字を小文字にするには 0x20 を足し，反対に小文字を大文字にするには 0x20 を引けばよい．

12.2　一文字型変数 char

一文字を表す変数型は char という．文字を宣言するには，char という型を使い，次のように宣言したり，参照したりする．

```
char  chChr;    /* char 型 (文字型) の変数宣言 */

chChr = 'A';    /* 文字型変数に値を格納． */
printf( " chChr is %c. \n", chChr );
```

文字型の変数に値を格納するには，格納したい文字を' シングルクォート' でくくる．また，2 行目の chChr = 'A'; は，ASCII コードを知っていれば chChr = 0x41; と書いても同じことである．0x??? は，??? が 16 進数であることを示すものである．'A' は ASCII コードの 16 進数で 41 番目である．3 行目で，printf で出力させている．%c は，文字型変数の文字を表示するための指定である．アスキーコードと文字の対応関係を表示するプログラムを作ってみると以下のようになる．

```
/*
 * ASCII コードの可視部分を表示する
*/
```

```
#include <stdio.h>

int  main(void)
{
    char  chASCIICode;

    for(chASCIICode = '!';chASCIICode <= '~'; chASCIICode++){
        printf( "0x%x is %c\n", chASCIICode, chASCIICode );
    }
    return  0;
}
```

上記のプログラムを解説すると，まず 1 行目の char chASCIICode; で，文字型変数 chASCIICode の宣言している．次の for 文の中の 3 つの繰り返し制御変数文を一つ一つ見ていく．最初の chASCIICode ='!'; 文で初期値を与えている．ASCII コードの可視領域は，'!' で始まり，'~' で終わっている．それゆえ初期値 は '!' で，終了値は'~' としている．文字に順番が割り当てられていて不思議に思うかも知れないが，コンピュータの内部ではどのような情報もこうした数値として表現されているのである．したがって，第 2 番目の繰り返し継続条件の項が chASCIICode <= '~' となる．そして，第 3 の各繰り返しごとにする処理の項で，文字型変数の値を 1 づつ足して，繰り返し内部でそれを表示させ，すべての ASCII 可視コードを表示するようになっている．

上のプログラムの for 文は，次のようにも書き替えることが可能である．

```
for( chASCIICode = 0x21; chASCIICode <= 0x7E; chASCIICode++ ){
    printf( "0x%x is %c", chASCIICode );
}
```

上は，文字を文字コードそのものに直しただけである．

12.3　文字の配列，文字列

C には文字列型などという型は存在しない．C では文字列は文字の配列として扱われる．すなわち，以下のようにすれば文字列となる．

```
#include <stdio.h>

int main(void)
{
    char str[14] = {'C', ' ', 'i', 's', ' ',
```

```
                    's', 'i', 'm', 'p', 'l', 'e', '.', '\0' };
    printf("String is \"%s\"\n", str);
    return 0;
}
```

上の実行結果は,

```
String is "C is simple."
```

となる.

プログラムの解説をすると，宣言の部分で 14 文字入る分だけの文字の配列を確保している．また，宣言と同時に初期化も行なっている．最後の '\0' は NULL 文字といって文字列の終端を表すために使われている．文字列の初期化は,

```
    char  str[ 14 ];

    str[ 0 ] = 'C';
    str[ 1 ] = ' ';
    str[ 2 ] = 'i';
    str[ 3 ] = 's';
    /* 以下省略. */
```

としても同じである．文字列の出力には，printf() 関数の中で%s という指定を使う.

このように文字列の１つ１つの文字に対して''で文字列を作るという作業は面倒である．これは C が文字列型という変数型をサポートしていないからであり

```
    str = "C is simple.";
```

のようにはできない．そのかわりとして C には strcpy() という関数が用意されており,

```
#include <string.h>

    char str[14];
    strcpy( str, "C is simple." );
```

とすれば，わざわざ上のように 1 文字ずつ代入するという手間が省ける．strcpy() を使うには，プログラムの冒頭部分で #include <string.h> という 1 行が必要である.

12.4 文字列の初期化が許される場合

ただし，文字の配列を宣言するときに任意の文字列で初期化することは許されている．すなわち

```
char string[] = "Hello, world.";
```

という書式は許されている．このとき string は 13 個の要素からなる文字の配列となる．次のプログラムはユーザに文字列を入力させ，その文字列が Hello, world. と一致していれば Hit! と表示し，一致していなければ正解を表示して終るプログラムである．

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

int main(void)
{
    char sTarget[] = "Hello, world.";
    char sInput[128];
    char *pch;

    printf("Input %d charcters : ", strlen(sTarget));
    if ( fgets(sInput, strlen(sTarget) + 1, stdin) == NULL ) {
        printf("Invalid input.\n");
        exit (EXIT_FAILURE);
    }

    pch = strchr( Sinput, '\n');
    if ( pch != NULL ) {
        *pch = '\0';
    }

    printf("Your input string is [%s]\n", sInput);
    if ( strcmp( sTarget, sInput ) == 0 ) {
        printf("Hit. \n");
    } else {
        printf("Miss.\n");
        printf("The answer is [%s]\n", sTarget);
    }
    return 0;
}
```

このプログラムには，まだ紹介していない strlen(), fgets(), strchr(), strcmp() などの入出力関数，文字列操作関数が含まれている．簡単にこれらの関数を紹介しておく．

まず strlen() 関数は引数で指定された文字列の長さを返す関数である．

fgets() は第二引数で指定された数より 1 だけ少ない文字列を第 3 引数で指定された入力から読みこんで第一引数で指定された文字列 (文字の配列) へ代入する関数である．上のプログラムの第 3 引数では標準入力を意味する stdin が指定されている．stdin は何も指定しなければキーボードからタイプされる文字列になる．C の初心者向けの教科書では，キーボードから文字を入力するときに scanf() 関数を使うように書かれている本があるが，scanf() 関数は危険であると言う理由で使われることはほとんど無いと言って良い．fgets() 関数の読みこみは改行文字，または EOF (End-Of-File の略，通常は -1 と定義されている) を読みこんだときに終了する．fgets() 関数は成功すると第一引数で指定された文字へのポインタを返し，失敗すると NULL ポインタとよばれる値を返す．if 文の中で fgets() 関数を使い，その戻り値が NULL であればプログラムを終了するようにしている．fgets() 関数については，15 章 p.56 にも説明がある．

strchr() 関数は第一引数で指定された文字列の中で第二引数で指定された文字が最初に見つかる場所を char 型変数のポインタ (14 章 p. 47 で取り上げる) として返す．文字列が見つからなかったときは NULL ポインタを返す．プログラム中では改行文字を文字列の終端記号である '\0' で置き換えるようにしてある．キーボードから入力されることを仮定して良いのなら，このような改行文字を'\0' で置き換える必要は無い．しかし，標準入力 stdin がパイプ処理などによってリダイレクトされている場合にはこの処理が必要となる．標準入力とは何か，パイプやリダイレクトとは何かについてはおぼえる必要はないが，操作例のみを示すと，上のプログラムは次のように使うこともできる．

```
echo "Hello, world." | program
```

echo は標準出力 (通常は画面) にダブルクォートで囲まれた文字列を表示するものである．この操作例では echo の標準出力を program の標準入力に結びつける働きをする | 記号を使って program への入力としている．これをパイプ処理などという．また，任意の文字列が書きこまれっているファイルがあったとして

```
program < file
```

としても動作する．このように標準入力をファイルに切り替えることをリダイレクションと言う．入力を標準入力 stdio にしたのだから出力も標準出力へ，エラーメッセージは標準エラー出力 stderr へ出力した方が統一が取れている．それぞれ

```
fprintf(stdout,"書式つき文字列", その他の引数, ...);,
fprintf(stderr,"書式つき文字列", その他の引数, ...);
```

のように書く.

最後の strcmp() 関数は第一引数で与えられる文字列と第二引数で与えられる文字列が等しいかどうかを判定する関数である．strcmp() 関数の戻り値は，二つの文字列が同じときには 0 を，そうでなければ 0 以外の値を返す.

12.5 シングルクォートとダブルクォート

初心者がつまずきやすいことを書いておく．それは ’(シングルクォーテーション) と "(ダブルクオーテーション) の違いを混同することである.

まず，シングルクォートは文字 1 文字を示す．すなわち，以下は C の正しい文である.

```
char  ch = 'A';

printf( " ch is %c, B is %c\n", ch, 'B' );
```

以下は誤った文である．シングルクォートは 1 文字を指すからである.

```
char  ch = 'ABCDE';

printf( " ch is %c, BCD is %c\n", chChr, 'BCD' );
```

一方，ダブルクォートは文字列を指す．したがって以下の文は C の文法に従った書式である.

```
char  strName[ 8 ];

strcpy( strName, "Asakawa" );
printf( "My name is %s %s\n", strName, "Shinichi" );
```

以下は C の文法に従わない誤った文である.

```
char  strName[ 8 ];

strcpy( strName, 'Asakawa');
printf( 'My name is %s s\n', strName, 'Shinichi' );
```

例題：

```
char  str[ 128 ];
```

という宣言があった場合，次の文は正しいか？

(1) strcpy(str, ’A’);
(2) strcpy(str, "A");
(3) printf("%s", ’A’);
(4) printf("%s", "A");
(5) printf("%s", "ABC");
(6) printf("%s", ’ABC’);

正解は，(2), (4), (5) が正しい文である．ただ，(3) は，コンパイルエラーにならない場合がある．(1) は初心者がよく間違う大代表のようなものなのであり，1 文字ならシングルクォートと安易に決めてしまった例である．str は char の配列，すなわち文字列として宣言されているので，文字列のアクセスは" " でせねばならないことから，’A’ でなく，(2) のように"A" として strcpy() を用いる．どうしても ’’ を使う場合は,

```
str[ 0 ] = ’A’;
```

としなければならない．文字列は文字型の配列であるから，文字列の個々の要素は，カッコ [ ] でくくらなければならない.

13 関数

C でプログラムを作るということは関数を作ることに他ならない．数学で使う関数よりも C の関数は汎用性がある.

13.1 数学の関数との比較

数学の関数では以下のように関数が定義されているとすると

$$f(x) = 3x + 2$$

x に任意の値を代入して得られる値のことを関数の値といった．x に 2 を代入し $x = 2$ とすると

$$\begin{aligned} f(2) &= 3 \times 2 + 2 \\ &= 8 \end{aligned}$$

などとなる．これを C で表現すると

```
#include <stdio.h>

/* 関数の宣言．*/
int  f( int  x );
```

```
int  main(void)
{
    int  fx;

    fx = f( 2 );
    printf( "f(2) = %d\n", fx );

    return  0;
}

int  f ( int  x )
{
    int ret;
    ret = 3 * x + 2;
    return  ret;
}
```

のようになる．

13.2 関数宣言と関数の実装

まず，プログラムの冒頭で，変数と同様に，

```
    int  f( int  x );
```

として，プログラムで用いられる関数を宣言する．C++ では関数の宣言（プロトタイプ宣言）は必須で，無いとコンパイルエラーとなるが，C では関数を呼び出す箇所の前に関数のインプリメンテーション (実装＝プロトタイプの反対で，中身のあるもの) があればコンパイルエラーとならない．例えば，

```
int  f(void){
     /* 中身は，省略 */
}

int  main(void)
{
    f();
    return  0;
}
```

は，プロトタイプ宣言が無いが，main() 関数の中で f() が呼び出されるより前に，関数 f() のインプリメンテーションがあるのでエラーとはならない．

この main() 関数と f() 関数の順番を逆にして，プロトタイプ宣言を書かないと警告が出る．関数の戻り値はデフォルトで int 型とみなされるのでコンパイラーによっては警告がでるだけでコンパイルができてしまう場合もある．すなわち，関数はその関数が呼び出される以前にプロトタイプ宣言があるか，または実装がされているかする必要がある．プロトタイプ宣言の最後にはセミコロンをつけなければならない．

関数の宣言方法は，

戻り値の型　関数名　(関数に渡す引数の型 引数名，...);

とする．戻り値の型とは例えば

$$3 * x + 2$$

を計算した結果を，どのような型に格納するかを定める．ここでは，結果が整数と仮定してもよいので int 型になっている．

関数名は，自分で付けたい名前を決めればよい．

ちなみに，古い関数宣言と実装というのが存在する．旧式の関数宣言では

```
int  f( int  x );
```

とするかわりに，

```
int  f( int );
```

のように変数名を書かずに型だけをプロトタイプで宣言する．古い実装は，

```
int  f ( int  x )
{
    int ret;
    ret = 3 * x + 2;
    return  ret;
}
```

と書くかわりに

```
int  f ( x )
    int x;
{
    int ret;
    ret = 3 * x + 2;
    return  ret;
}
```

と書く．つまり，先に変数の名前だけを書き，そのあとで変数の型と名前を書きセミコロンを書くような書式になる．古い書式で書かれた実装はコンパイラによっては警告 warninig を出すものがある．ここで紹介した古い宣言や書式は ANSI の C の規格にも，時代遅れであると書かれている．

最後に，関数に渡す引数の型と引数名で，上の例で言えば，関数 f() には，x という変数が必要になる．この変数は，int 型で宣言してある．小数点の値を求めたいときや，y, z などの変数も必要になれば，関数 f() は，

```
    float  f( float  x, float  y, float  z );
```

とカンマで区切って宣言する．以上が関数のプロトタイプが宣言である．

関数 f() の実装は

```
int  f ( int  x )
{
    int ret;
    ret = 3 * x + 2;
    return  ret;
}
```

のように最初に，関数の宣言と同じものを書き（ ただしセミコロンを除く），main() と同様に，{ と } で囲んだ部分が関数の本体となる．return 文の末尾に，呼び出し先に返す値を書く．このとき return 文の末尾の型と関数のプロトタイプで宣言した戻り値の型が一致していなければならない．

演習問題 1 から 100 までの全ての素数を表示するプログラムを書け

13.3 #include について

今まで使ってきた main() も printf() も strcpy() も全て関数である．printf() や strcpy() がプロトタイプ宣言や実装が無いのに使えるのは

```
    #include < ... >
```

の拡張子が .h で終るファイルに書かれている．.h で終るファイルをヘッダファイルと言う．gcc をインストールしたディレクトリに/include というディレクトリがある．このディレクトリこ stdio.h という名前のファイルがある．ヘッダファイルはテキストファイルであるから，エディタなどで見てみることができる．stdio.h には printf() のプロトタイプ宣言がなされている．

コンパイル時にプリプロセッサとよばれるプログラムが起動され，この #include で指定されたファイルが展開される．これにより，printf() などのプロトタイプ宣言が要らなくなる．

printf() などの実装は gcc のインストールされたディレクトリの/lib ディレクトリの中の拡張子 .a のファイルや.so ファイルに入っている．これらのファイルをライブラリと言う．ライブラリファイルはコンパイル済みなので，エディタで見ることはできない．printf() などは，この内の libc.a に入っている．gcc のコンパイラはこの libc.a をデフォルトで組み込むようになっている．一方，libm.a は初期設定では組み込まれていないので，コンパイル時に明示的に指定してやる必要がある．いままで sqrt() などを使う場合は，

```
gcc hoge.c -lm
```

として明示的にライブラリを指定する理由がこれである．

最後に sqrt() は戻り値を左辺に代入することができたが，実は printf() も戻り値がある．printf() の戻り値の型は int であり書き込まれた文字数が返ってくる．今まで行なってきた print() はその戻り値を利用していないだけの話なのである．

値を返さない関数を作るには戻り値の型を void にする．

```
void  NonReturnFunction();
```

13.4 関数の再帰呼び出し

以下のプログラムは 5 の階乗を求めるプログラムである．

```
#include <stdio.h>

int Fact( int n );

int main(void)
{
    int n;

    n = 5;
    printf("Factorial of %d is %d.\n", n, Fact(n));
    return 0;
}

int Fact( int n )
{
    if ( n == 1 )
        return 1;
    else
```

```
        return n * Fact( n - 1 );
}
```

ここでのポイントは Fact() 関数の中で自分自身である Fact() が呼び出されていることである．関数 Fact() は呼び出されるごとに 1 だけ少ない値を引数として自分自身を呼び出す．引数 n が 1 の場合だけ 1 を返す．このような関数を再帰的な関数と呼ぶ．呼び出された関数と呼び出した側の関数で引数 n が異なる値になっていることを理解して欲しい．また，main() の中にある変数 n と Fact() 側の引数 n は全く別物であることにも注意が必要である．

演習問題 上の階乗を求めるプログラムを再帰関数を使わずに，繰り返し for() 文を用いて書き換えよ．

13.5 main() 関数の引数

main() も C の関数であり，引数を取ることができる．慣例では

```
int main( int argc, char **argv )
```

などとする．main() の引数を使って先の階乗を求めるプログラムを汎用にしてみよう．main() の第一引数は整数型であり，第２引数は文字型のポインタのポインタである．ポインタについては後述する．

```
#include <stdio.h>
#include <stdlib.h>

int Fact( int n );

int main(int argc, char **argv)
{
    int n;
    if ( argc == 2 ) {
        n = atoi(argv[1]);
    } else {
        n = 1;
    }
    printf("Factorial of %d is %d.\n", n, Fact(n));
    return 0;
}

int Fact( int n )
{
```

```
    if ( n == 1 )
        return 1;
    else
        return n * Fact( n - 1 );
}
```

このプログラムは 1 つの引数を取る．引数は整数であることが仮定され，文字列を整数に変換する関数 atoi() が使われている．使い方は上記のプログラムを fact.c という名前で保存してあるとして，

```
gcc -Wall fact.c -o fact
```

とコンパイルしてから，

```
./fact 3
Factorial of 3 is 6.
```

となる．階乗の計算は引数の数が大きくなると答えは急速に大きくなる．int 型の変数が 32 ビットの処理系では 12 程度までしか正しい答えを表示できないので注意が必要である．

今回の main() 関数には 2 つの引数が用いられている．1 番目の引数は argc という引数名であって，プログラムが実行されるときの引数の数を参照する int 型の変数である．2 番目の引数は文字の配列へのポインタであり引数の具体的な内容が文字列として格納されている．

プログラム中の

```
if ( argc == 2 ) {
    n = atoi(argv[1]);
else {
    n = 1;
}
```

は引数の数が 2 のときには，文字列 argv[1] の内容を整数に変換して n に代入している．すなわち

./Fact Enter

と引数なしで実行した場合には argc には 1 が代入され，argv[0] には ./Fact が代入されてプログラムが開始される．このとき if 文の argc == 2 が偽となるので，変数 n には 1 が代入された状態でプログラムが起動される．もし，

./Fact 6 Enter

と実行したのなら，argc 変数には 2 が代入され，argv[0] には./Fact が argv[1] には 6 が代入されて main() 関数が起動される．このとき argv[1] は文字型のポインター (すなわち文字列) であるから，文字列を int 型の変数に変換するための関数 atoi() が用いられている．関数 atoi() のプロトタイプ宣

言は stdlib.h に入っているのでプログラムの初めに #include <stdlib.h> が必要である．

main() 関数の引数についての理解を深めるために以下のようなプログラムをつくって実行してみよう．

```
#include <stdio.h>

int main( int argc, char **argv)
{
    int i;

    for ( i = 0; i < argc; i++){
        printf("The content of arg[%d] is [%s]\n",i,argv[i]);
    }
    return 0;
}
```

このプログラムを例えば argcheck.c などという名前で保存してコンパイルし，実行してみると．以下のような動作をする．

```
./argcheck Enter
The contenet of argv[0] is [./argcheck]

./argcheck a b c d Enter
The contenet of argv[0] is [./argcheck]
The contenet of argv[1] is [a]
The contenet of argv[2] is [b]
The contenet of argv[3] is [c]
The contenet of argv[4] is [d]

./argcheck 私の 名前は 浅川 伸一 です． Enter
The contenet of argv[0] is [./argcheck]
The contenet of argv[1] is [私の]
The contenet of argv[2] is [名前は]
The contenet of argv[3] is [浅川]
The contenet of argv[4] is [伸一]
The contenet of argv[5] is [です．]
```

プログラムの引数はスペースで区切っていくつでも書くことができそれぞれ argv[] という文字列に格納されている．

演習問題： 上の引数を順に表示するプログラムを，逆順に表示するように書き換えよ．

これまでの C の知識を使ってハノイの塔を解くプログラムを作ることができる．main() 関数の引数と再帰呼び出しを用いている．以下のソースコードを hanoi.c などの名前で保存し実行してほしい．プログラムの解説はあえてしない．自分で考えてみてほしい．

```
/*
 * Tower of Hanoi
 * written by ASAKAWA Shinichi <asakawa@twcu.ac.jp>
 *
 * How to compile this source code
 *    gcc -o hanoi hanoi.c
 */
#include <stdio.h>
#include <stdlib.h>

int hanoi(int n, char x, char y, char z)
{
    if (n==1) {
        printf("move disk %d from %c to %c\n",n, x, y);
    } else {
        hanoi( n-1, x, z, y );
        printf("move disk %d from %c to %c\n",n, x, y);
        hanoi( n-1, z, y, x );
    }
    return EXIT_SUCCESS;
}

int main(int argc, char **argv)
{
    int disks = 3; /* default number of disks */

    if (argc == 2) {
        if ( (disks=atoi(argv[1]))<= 0 ) {
            fprintf(stderr, "Invalid argument %s\n", argv[1]);
            exit (EXIT_FAILURE);
        }
    }
    hanoi(disks, 'a', 'b', 'c');
    return EXIT_SUCCESS;
}
```

13.6 文字列を数値に変換する関数

入力した文字列を数値に変換する方法をまとめておく．文字列を数値に変換するためには数値の型によって別々の関数が用意されている．これらの関数を使うには，プログラムの冒頭で `#include <stdlib.h>` と書かねばならない．

表 4: 文字列から数値型への変換関数

変数型	関数名
int 型	atoi()
long 型	atol()
float 型	atof()

引数を 2 つとって，2 つの引数の積を表示するプログラム times.c を作ってみよう．

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    float a, b;

    if ( argc != 3 ) {
        printf("You need 2 argments.\n");
        exit (EXIT_FAILURE);
    } else {
        a = atof(argv[1]);
        b = atof(argv[2]);
    }
    printf("%s times %s is %f.\n",argv[1],argv[2], a * b);
    return EXIT_SUCCESS;
}
```

上記のプログラムは引数が 2 つでないときには，You need 2 argments. と表示し終了してしまう．引数が 2 つのときは atof() 関数を使って，それぞれの引数を float 型の変数 a と b とに代入し計算結果を表示する．なお stdlib.h には

```
#define EXIT_FAILURE    1       /* Failing exit status.  */
#define EXIT_SUCCESS    0       /* Successful exit status.  */
```

が定義されているの main() 関数の戻り値にこの値を用いている．

さて上記のプログラムには欠陥 (バグ) がある．それはユーザが引数に文字をしてしてしまったとき，atof() 関数はどのような値を返すのだろうかという点である．

```
    int a, b;

    a = atoi("abc");
    b = atoi("12abc");
    printf("a=%d, b=%d\n");
```

のようなプログラムがあった場合，a には 0 が，b には 12 が代入される．すなわち atoi() や atof() は途中に文字があるとその直前の数字までしか数値に変換してくれないのである．

ユーザが不正な引数でプログラムを起動した場合の処理を組み込む必要がある．

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main( int argc, char **argv )
{
    int i,j;
    float a,b;

    if ( argc != 3 ) {
        printf("You need 2 argments.\n");
        return EXIT_FAILURE;
    }

    for (i=1; i<3; i++) {
        for( j = 0; argv[i][j] != '\0'; j++ ) {
            if ( (isdigit( argv[i][j] ) == 0 )
                && (argv[i][j] != '-')
                && (argv[i][j] != '.') ) {
                printf( "Illegal charctar was detected");
                printf(" at %d-th charcter ",j);
                printf(" in arg[%d]=[%s].\n",
                        i, argv[i]);
                return  EXIT_FAILURE;
```

```
            }
        }
    }
    a = atof(argv[1]);
    b = atof(argv[2]);
    printf("%f times %f is %f.\n",a, b, a * b);
    return EXIT_SUCCESS;
}
```

このプログラムには 2 重ループ (繰り返し) が用いられている．
最初の　`for (i=1; i<3; i++) {`
の繰り返しでは全ての引数について繰り返すことを意味している．2 番目の繰り返し `for( j=0; argv[i][j] != '\0'; j++ ) {` では，文字列の最後が `'\0'` で終わることを利用して，繰り返しの中断を判断している．この繰り返しの中にある `if` 文では `isdigit()` 関数が使われている．`isdigit()` 関数は引数が 0 から 9 までの数値であれば 0 以外の値を返し，数値以外であれば 0 を返す関数である．この関数と小数点を調べる `argv[i][j] != '.'` と負の値を表す `argv[i][j] !='-'` とを使い，論理積 `&&` を使って，これらの条件が真ならば，数値ではないと判断して処理を中断する．この `if` 文は次のようにも書ける

```
    if ( (!isdigit(argv[i][j]) ) && ( argv[i][j] != '.' ) ) {
```

ここで `!` は否定を表す．

演習問題 任意の数の引数を取り，その最大値を表示するプログラムを書け．

次のプログラムはユークリッドの互除法を用いて 2 つの引数の最大公約数を求めるプログラムである．

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv )
{
    int x, y, w, q;

    if ( argc != 3 ) {
        printf("### Usage: %s <int> <int>\n", argv[0]);
        exit (EXIT_FAILURE);
    }
    if ( (x = atoi(argv[1])) <= 0 ) {
        printf("### Invalid argument [%s]\n", argv[1]);
        exit (EXIT_FAILURE);
```

```
    }
    if ( (y = atoi(argv[2])) <= 0 ) {
        printf("### Invalid argument [%s]\n", argv[2]);
        exit (EXIT_FAILURE);
    }
    if ( x < y ) {
        w = x;
        x = y;
        y = w;
    }

    for ( q = 1; q != 0 ; ) {
        q = x % y;
        w = x / y;
        printf("x=%d, y=%d, quotient=%d, residual=%d\n",
                x,      y,          w,          q);
        x = y;
        y = q;
    }
    printf("G.C.D of (%d,%d) is %d\n",
               atoi(argv[1]), atoi(argv[2]), x );
    return EXIT_SUCCESS;
}
```

13.7　文字種判定関数

C の文字種判定関数には表 13.7 のような関数がある．これらの中には厳密には関数ではなくマクロとして定義されているものもある．関数とマクロの違いについては後述する．`ctype.h` には，`isdigit` などの文字種判定に関する関数やマクロが宣言されている．使い方はいずれも同じで，引数が条件に当てはまっていれば 0 でない値を返し，条件に当てはまっていなければ 0 を返す．

13.8　乱数系列生成関数 rand()

数値シミュレーションなどを実行するときのために出鱈目な数，乱数を発生させる関数について解説しておく．

`rand()` 関数は，0 から `RAND_MAX` の間の疑似乱数整数を返す関数である．`rand()` 関数のプロトタイプ宣言も `RAND_MAX` の定義も `stdlib.h` で定義さ

表 5: 文字種判定に用いられる関数またはマクロ

関数名	説明
isalpha()	アルファベット（ 'A' to 'Z' or 'a' to 'z' ）か？
isupper()	アルファベットの大文字（ 'A' to 'Z' ）か？
islower()	アルファベットの小文字（ 'a' to 'z' ）か？
isdigit()	数字（ '0' to '9' ）か？
isspace()	空白文字（ 0x20, '\t', '\n', —, '\f', '\v' ）か？
isalnum()	isdigit と isalpha を合わせたもの.
iscntrl()	コントロール文字 (0x00 to 0x1f or 0x7f(=DEL)) か
isprint()	印字できる文字か？
isascii()	アスキー文字（ 0x00 to 0x7F ）か？

れているので rand() 関数を使う場合には必ず#include <stdlib.h> が必要である．rand() 関数は呼び出されるたびに異なる整数を返す．10 個の乱数を発生させるサンプルプログラムを以下に示す.

```
#include <stdio.h>
#include <stdlib.h>

int  main(void)
{
    int  i;

    for( i = 0; i < 10; i++ ) {
        printf( "%d\n", rand() );
    }
    return 0;
}
```

このプログラムの実行結果は例えば以下のようになる.

```
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
```

```
    1189641421
```

結果は何度実行しても同じ数値となる．

同じ数値になるのではシミュレーションの際に不都合である．C に組み込まれている乱数発生関数 rand() は線形合同法というアルゴリズムに従って乱数を発生させる実装がなされている．この線形合同法では，乱数の種を変えない限り常に同じ乱数系列が発生する．srand(unsigned int seed) 関数は引数で与えられた数値によって乱数発生器の種を設定する．従って実行するたびに別の乱数系列を生成するためには srand() 関数を使って乱数の種を変えなければならない．同じ種から生成される乱数系列は常に同じ乱数系列となる．

乱数の種を与える方法としてはそのときの時刻を種にする方法がある．まず，プログラムの冒頭で time.h を include し，

```
#include <stdlib.h>
#include <time.h>

    long loadingtime;
    long seed;

    seed = time(&loadingtime);
    srand((unsigned)seed);
```

としてから rand() を用いればよい．time() 関数は 1970 年 1 月 1 日午前 0 時 0 分 0 秒からの経過時間を秒単位で返す関数である．乱数の種を時刻で初期化した乱数発生プログラムを以下に示す.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int  main(void)
{
    int i;
    long loadingtime;
    long seed;

    seed = time(&loadingtime);
    srand((unsigned)seed);

    for( i = 0; i < 10; i++ ) {
        printf( "%d\n", rand());
```

```
    }
    return 0;
}
```

rand() 関数を用いて 1 から 10 までの乱数を作りたければ以下のような関数を作ればよい．

```
/*
 *  from から to までの間の乱数を生成する関数
 */
int irand ( int from, int to )
{
    return ( from + (rand() % ( to - from + 1 )) );
}
```

ここでは 7.2 章 (p.7) で紹介した剰余演算子 % を使っている．

$[0, 1)$ の区間の一様乱数系列を発生させる関数は

```
#include <values.h>

double frand(void)
{
    return ( (double)rand() / (double)DBL_MAX );
```

とすればよい．なお rand() 関数が生成する乱数系列は周期が短い，発生する乱数系列に相関が存在する，などの乱数としてはできが悪い実装であるといわれる．自作の乱数生成関数を作っているプログラマも多い．

演習問題 上記の関数を使って乱数を 100 個発生させ，その平均と分散を求めるプログラム作成せよ．

平均 0.0 分散 1.0 の正規分布に従う乱数を発生させるには単に rand() の値を 12 加え 6.0 を引けば近似的に正規乱数が得られる．

```
double gauss_rand(void)
{
    return rand() + rand() + rand() + rand()
         + rand() + rand() + rand() + rand()
         + rand() + rand() + rand() + rand() - 6.0;
}
```

これは大数の法則を使って一様乱数を正規乱数に変換している例である．また，多くの教科書で採用されている Box と Muller の方法がある．

```
#include <stdlib.h>
#include <math.h>
```

```
double gauss_rand(void)
{
    static double V1, V2, S;
    static int phase = 0;
    double ret;

    if ( phase == 0 ) {
        do {
            double U1 = (double)rand() / RAND_MAX;
            double U2 = (double)rand() / RAND_MAX;

            V1 = 2.0 * U1 - 1.0;
            V2 = 2.0 * U2 - 1.0;
            S = V1 * V1 + V2 * V2;
        } while ( S >=1 || S == 0 );
            ret = V1 * sqrt( -2.0 * log(S) / S);
        } else {
            ret = V2 * sqrt( -2.0 * log(S) / S);
        }
        phase = 1 - phase;

        return ret;
}
```

Box と Muller の方法のなかで変数 `V1`，`V2`，`S` と `phase` には `static` 修飾子がついている．`static` 修飾子の意味は以下のようになる．通常，関数内部で宣言された変数はその関数内部でのみ有効で，関数を抜け出したときにその内容は廃棄されるか不定になってしまう．次に同じ関数が呼び出されたときには，以前の結果がそのまま使えると考えてはならない．`static` 修飾子のついた変数は，その関数が呼び出されるたびに，以前の値を保持している．例えば

```
    static int phase = 0;
    pahse = 1 - phase;
```

の変数 `phase` は呼び出されるたびに 1, 0 の値を交互に繰り返すことになる．

どのような方法を使っても，`srand()` を使って乱数の種を与えた場合，同じ種であれば同じ乱数系列が発生されることに注意してほしい．

演習問題 上の `gauss_rand()` 関数を使って平均 m 分散 s^2 従う正規乱数を 100, 1000, 10000 個発生させるプログラムを書きの出力結果をヒストグラムにして比較せよ．

演習問題 正規乱数を 100, 1000, 10000 個発生させるプログラムの出力結果 2 つの相関係数を計算せよ．相関係数が 0.0 と見なせるかどうか検討せよ.

14　ポインタ

14.1　ポインタの定義

ポインタとは，メモリのアドレスを記録しておくための変数である．次のプログラムで main() 関数の最後の printf() によって何が表示されるかを考えてほしい.

```
#include <stdio.h>

void  swap( int a, int b );

int  main(void)
{
    int  a, b;

    a = 3;
    b = 5;
    swap( a, b );

    printf( "a = %d, b = %d\n", a, b );
    return  0;
}

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

    c = a;
    a = b;
    b = c;
}
```

main() 関数では a を 3 にし，b に 5 を代入してから関数 swap() に渡して計算を行なっている．結果は

```
    a = 3, b = 5
```

と表示され swap() 関数内部で交換されたはずの a,b が反映されていないように見える．

14.2 デバッグライト

本当に計算されていないのかを検証するため，以下のように swap() を書き替えてみよう．

```
void  swap( int a, int b )
{
    int c;

    c = a;
    a = b;
    b = c;
    printf( "### Debug : a = %d, b = %d\n", a, b );
}
```

swap() 関数内 では変数の値を入れ換えた後に

```
    printf( "### Debug : a = %d, b = %d\n", a, b );
```

として計算結果を表示するようにした．このように，プログラムの動作が自分の意図しないような振る舞いをした場合 printf() などによって変数の値を表示することをデバッグライトなどと言う．

デバッグライトを付けてプログラムを実行すると，

```
    ### Debug : a = 5, b = 3
    a = 3, b = 5
```

と表示され，確かに swap() 関数内部では変数の値が交換されていることがわかる．すなわち swap() 関数内で計算された結果が main() 関数の方の変数 a, b に反映されていないことがわかる．

14.3 call by value

C の関数呼び出しは call by value といって関数を呼び出したときの引数のコピーがメモリ上に確保され，呼び出された側の関数ではこのコピーされた引数をもちいて演算を行なうのである．言い替えれば main() 関数内の変数 a と変数 b とは，swap() 関数内の引数 a, b とは異なる実体である．関数を呼び出された先での演算結果は，呼び出した側の関数内の変数には影響を与えないのである．

14.4 引数に変数のアドレスを用いる

このようなときには，呼び出す側の関数では変更したい変数のアドレスを引数として与え，呼び出された側の関数は変更すべき変数をポインタとして引数を扱うことにすればよい．

ポインタとはメモリのアドレスを記録しておくための変数である．先の例をポインタを用いて書き変えたプログラムを以下に示す．

```
#include <stdio.h>

void  swap( int *a, int *b );

int  main(void)
{
    int  a, b;

    a = 3;
    b = 5;
    swap( &a, &b );

    printf( "a = %d, b = %d\n", a, b );
    return  0;
}

void  swap( int *a, int *b )
{
    int c;

    c = *a;
    *a = *b;
    *b = c;
}
```

main() 関数の中の swap() 関数の呼び出す際に，引数に実際の変数を書くのではなく，変数の前に& をつける操作によってその変数のアドレスを渡している．そして呼び出された関数 swap() では引数を int 型のポインタとして受け取っている．

14.5 ポインタ変数宣言

ポインタはメモリアドレスを記録する変数なのだからポインタを使うには宣言が必要である．ポインタの宣言方法は

```
型名  *変数名;
例 : int  *iptr;
```

のようにする．普通の変数の宣言に * を付けるだけである．C における * には 3 つの役割がある．

- 乗算を行なうための演算子
- ポインタを宣言するための演算子
- ポインタに記録されたメモリアドレスに書かれている内容を取り出すために使われる演算子

初心者は 3 番目の意味は理解しにくいかも知れない．よく言われることだが，C における最大の難関はポインタである．例えば，つぎのプログラムを見てほしい．

```
#include <stdio.h>

int main(void)
{
    char Str[24];
    char *pstr;

    strcpy( Str, "Pointer is a point of C.");
    printf("%s\n", Str);
    pstr = Str;
    *pstr = 'p';
    printf("%s\n", Str);

    return 0;
}
```

このプログラムの実行結果は

```
Pointer is a point of C.
pointer is a point of C.
```

となる．ここで char 型のポインタ宣言は main() の 2 行目の char *pstr; である．そして pstr = Str; によって文字の配列 (文字列) のアドレスを pstr

に代入している．このとき *pstr は文字配列 Str の先頭の要素 Str[0] を指している．従って*pstr に文字 'p' を代入する (すなわち *pstr = 'p'; という文) ことで文字列 Str の先頭の要素 'P' が小文字の'p' に置き換わっている．pstr は char 型へのポインタとして宣言したので Str に格納されている値を char 型の変数であると見なすのである．変数 pstr は char 型の変数のメモリアドレスを指すのであるから，pstr = "Pointer is a point of C." などとするのは間違いである．ちなみに pstr = Str という文は pstr = &Str[0] という文と等価である．すなわち char 型の配列の先頭の要素のアドレスを Str と省略して記述することができるのである．

初心者の間違える典型例は

```
int i, *iptr;

iptr = 1000;
i = *iptr;
```

などとすることである．この文はメモリ上の 1000 番地というアドレスを iptr に代入している．そして 1000 番地に書かれている情報を int 型の変数だと解釈して int 型の変数に 1000 番地の情報を代入している．このような文はコンンパイル時にはエラーとして表示されないので注意が必要である．8 章では，演算結果を一時的に記憶させる入れ物，場所のようなものを変数と言う，と定義した．本章で定義したポインタはアドレスを格納するための変数なのだから iptr = 1000; という文は 1000 番地というのアドレスをポインタ変数に代入することになるので C の構文としてはまったく正しい．そのため，コンパイル時にエラーが出ないことがある．また次のプログラムも誤りである．

```
int i, *iptr;

*iptr = 1000;
i = *iptr;
```

ポインタ変数として宣言された *iptr にたいして実体が何もないのにいきなり 1000 という数値を代入しようとしているからである．一方，次のプログラムは正しい．

```
int i, *iptr;

i = 1000;
iptr = &i;
printf("%d\n", *iptr);
```

この文では i というあらかじめ宣言された (すなわち記憶領域が確保された) int 型の変数のアドレスを iptr に代入しているからである．この操作により，

*iptr の * はポインタに記録されたメモリアドレスに書かれている内容を取り出すために使われる演算子としての役割を果たすことになり，printf() 関数によって 1000 という値が印字されることになる．

ポイントはポインタ変数を宣言するときの * と，ポインタの参照先のデータにアクセスするための演算子 * とは異なったものであるという点にある．

次のプログラムが理解を助けるかも知れない．

```
#include <stdio.h>

int  main()
{
    int  i, *iptr;

    printf("*iptr=%d, address of iptr = %p\n",*iptr,iptr);

    iptr = &i;
    i = 5;

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

    printf("*iptr=%d, address of iptr=%p\n",*iptr,iptr);
    return  0;
}
```

このプログラムの実行結果はそれぞれの環境や起動しているプログラムによっても異なるが，

```
    *iptr=75979098, address of iptr = 4000bcd0
    i=5, address of i = bffff2e8
    *iptr=5, address of iptr = bffff2e8
```

などとなる．最初の printf() 関数では初期化されていない*iptr の内容やアドレスを参照しているので出鱈目な数値が印字されている．次に iptr = &i と実体のある変数 i のアドレスを iptr に代入している．2 番目の printf() 関数では変数 i の値とそのアドレスを印字している．最後の printf() 関数では，iptr 変数によって指定されたアドレスのデータとそのアドレスを印字していることになるので，結果は変数 i と同じ値，同じアドレスとなっている．

ポインタ変数は * で値を表し，何も付けないとアドレスを指し示す．そして，一般変数は逆に何も付けないと値を示し & でアドレスを示すということになる．よって，以下のプログラムはトリッキーなものだが正しく動作する．

```
#include <stdio.h>

int  main(void)
{
    int  i, j;

    i = 5;
    j = *&i;

    printf( "i = %d, j = %d\n", i, j );
    return  0;
}
```

このプログラムの中では一般変数 i のアドレスを参照し，i のアドレスに格納されている情報をポインタの指し示す値として取りだして変数 j に代入している．結果は i も j も 5 となる．

14.6　ポインタの有用性

なぜポインタが使われるかについては

- ポインタを関数の引数に指定して，呼び出し元の 変数の値を変更することができる
- 配列の代わりにポインタを使うと便利がことがある．(ポインタは変数なので，配列を使うよりも便利である.)
- 構造体にポインタを指定することで，可変のサイズを持つメモリイメージ，すなわち「リスト」を実現できる．

などが挙げられる．

14.7　配列とポインタ

前章の最後に配列の代わりにポインタを使うことができるということを述べた．このことを実感するために以下のプログラムを見てみよう．

```
#include <stdio.h>

int main(void)
{
    char str[128], *chptr;
```

```
    strcpy(str, "C is fun.");
    chptr = str;
    while ( *chptr != '\0' ) {
        printf("%c : address of chptr=%p\n", *chptr, chptr);
        *chptr++;
    }
    return 0;
}
```

このプログラムの実行結果は例えば次のようになる．

```
C : address of chptr=bffff26c
  : address of chptr=bffff26d
i : address of chptr=bffff26e
s : address of chptr=bffff26f
  : address of chptr=bffff270
f : address of chptr=bffff271
u : address of chptr=bffff272
n : address of chptr=bffff273
. : address of chptr=bffff274
```

アドレスの値は同じようになるとは限らないのは先にも述べた．ここでのポイントは `str[0]`, `str[1]`,..., `str[128]` がメモリ上で連続した番地を占めていることである．上の例では `str[0]` は 0xbffff26c, `str[1]` は 0xbffff26d となっている．このように，配列の個々の要素は連続してメモリに展開される．同じことを `float` 型の配列で行なっても同様の結果が得られる．サンプルプログラムを以下に示す．

```
#include <stdio.h>

int main(void)
{
    int i;
    float fArray[5] = { 1, 10, 100, 1000, 10000}, *fptr;

    fptr = fArray;
    for (i=0; i < 5; i++ ) {
        printf("fArray[%d]=%8.2f : address of chptr=%p\n",
            i, *fptr, fptr);
        *fptr++;
    }
```

```
    return 0;
}
```

このプログラムの実行結果は次のようになる．

```
fArray[0]=    1.00 : address of chptr=bffff2d4
fArray[1]=   10.00 : address of chptr=bffff2d8
fArray[2]=  100.00 : address of chptr=bffff2dc
fArray[3]= 1000.00 : address of chptr=bffff2e0
fArray[4]=10000.00 : address of chptr=bffff2e4
```

今度は文字列型の変数と違って文字列の最後の '\0' を仮定することができきなので for 文の繰り返しを使っている．上記の結果では float 型は 4 バイトで 1 つの数字を表すのでアドレスは 4 つずつ増えて行く．float 型が 4 バイトではない処理系も存在する．それぞれの変数型が何バイトになるかは処理系に依存するのである．

なお，ある型の変数が何バイトであるかを知るには sizeof() 演算子がある．

```
    int Array[100];

    printf("Size of char   is %d byte\n", sizeof(char));
    printf("Size of int    is %d byte\n", sizeof(int));
    printf("Size of float  is %d byte\n", sizeof(float));
    printf("Size of double is %d byte\n", sizeof(double));
    printf("Size of Array  is %d byte\n", sizeof(Array));
```

などとして，各自結果を確認されたい．

さて，プログラムの先頭の方で

```
    fptr = fArray;
```

として float 型変数の配列の先頭のアドレスを fptr に代入している．この文は fptr = &fArray[0] と等価であることは既に述べた．for 文の繰り返しの中で *fptr++; という文がある．これはポインタの参照先を 1 つ増やすことを意味し，結果として fArray[1] を参照することになる．このとき参照先のアドレスが 4 つずつ増えて行くのは最初に float 型のポインタ変数として fptr を定義してあるからである．

これを発展させると，配列の先頭の要素から 100 番目の要素にアクセスしたければ *(fptr+100) などと書くことができる．すなわち

```
    float fArray[128], *fptr;
    fptr = fArray;
```

などとした場合には *(fptr+100) と fArray[100] は等価である．すなわち配列は C の処理系内部ではポインタとして扱われる。従って*(100+fptr) と書いても良いし 100[fArray] と書くこともできる。あまりお勧めできる書式ではないのだが、以下のプログラムは完全に正しく動作するし、コンパイル時に -Wall オプションを指定しても警告もでない。

```
#include <stdio.h>

int main(void)
{
    int a[10];

    a[1]   = 1;
    *(a+2) = 2;
    3[a]   = 3;

    printf("%d %d %d\n", *(1+a), a[2], *(1+2+a));
    return 0;
}
```

このプログラムはまったくお遊びであり、通常のプログラムにこのようなコードを書いたら混乱を招くだけであるからしないでほしい。ここでは配列とポインタの関係について理解を助ける意味でしかない。

15　ファイル入出力

C のファイル操作の方法には

- 低レベルのファイル入出力関数群
- 高レベルのファイル入出力関数群

の 2 種類の関数が用意されている．ここでは高レベルのファイル操作関数のみを紹介し，低レベルのファイル操作関数の説明は省略した．

15.1　FILE 型ポインタ変数

ファイルを操作するには FILE 構造体へのポインタを介して行なわれる。従ってファイルの操作をするには，必ずこの変数を宣言しておく必要がある。

```
FILE  *FP;
```

ファイルからのデータの読みこみや，ファイルへのデータの書き出しなどはすべてこの FILE 型のポインタ変数を介して行なわれる．

15.2 fopen(), fclose()

ファイル操作を始めるには最初に fopen() 関数でファイルを開き，処理が終ったら fclose() 関数でファイルを閉じなければならない．fopen() 関数のプロトタイプ宣言は以下のようになっている．

```
#include <stdio.h>
    FILE *fopen(const char *path, const char *mode);
```

fopen() 関数の第一引数にはファイル名を指定する．第二引数の*mode には文字列を指定し，読み出しならば "r" を，書き込みならば "w" を，追記書き込みならば "a" を指定する．細かいアクセス制御のために，r,w,a のあとに + や b などを指定することができるが，ここでは取り上げない．ファイル操作の基本は r, w, a だけで十分だからである．*mode を"w" にしたときに，第一引数に指定したファイルが存在しない場合は，新たにそのファイルを作成する．ファイルが既に存在する場合は，そのファイルの内容を破棄し，新しくファイルを作りなおす．

fopen() 関数は成功すると FILE 型のポインタを返す．ファイルが正常に開けなかったり，許可が無かったりした場合には fopen() 関数は NULL を返す．*mode が "r" のとき，存在しないファイルを読み出そうとした場合などにも NULL が返される．

fopen() 関数で開いたファイルは fclose() 関数で閉じなければならない．fclose() 関数のプロトタイプ宣言は以下のようになっている．

```
#include <stdio.h>
    int fclose(FILE *stream);
```

fclose() 関数が正常に終了すると 0 が返される．正常に終了しなかった場合には EOF が返される．EOF とは stdio.h の中で -1 と定義されている．

次のプログラムは第一引数で指定したファイル名をオープンし

```
    Neural networks are interesting.
```

という内容の文字列を書き込むプログラムである．ここでは fprintf() 関数を使って文字列を印刷している．

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char **argv )
{
    FILE  *fp;

    if ( argc != 2 ) {
        printf("### You need a file name.\n");
```

```
        exit (EXIT_FAILURE);
    }

    fp = fopen(argv[1], "w");
    if ( fp == NULL ) {
        printf( "### Could not write a file [%s].\n",
                argv[1]);
        exit (EXIT_FAILURE);
    }

    fprintf( fp, "%s\n", "Neural networks are interesting.");
    if ( 0 != fclose( fp ) ) {
    printf("### Could not close a file [%s].\n", argv[1]);
        exit (EXIT_FAILURE);
    }
    return  EXIT_SUCCESS;
}
```

ファイルを識別する変数を FILE 型のポインタ変数で宣言し，その名前を `fp` としている．この変数に `fopen()` 関数の戻り値を代入している．`fopen()` 関数では `main()` 関数の引数である `argv[1]` の内容であるファイル名を書き込みの新規作成モードでオープンするように指定している．もし，書き込み許可が無かったり，ディスクの容量が不足していたりしてファイルが開けなかった場合には `fopen()` 関数は `NULL` を返すので `fp` は `NULL` となる．`fopen()` 関数の次の `if` 文でファイルが開けたか否かを判定し，開けなかった場合はユーザにその旨を表示してプログラムを終了している．

実際の書き込みは `fprintf()` 関数で行なっている．`fprintf()` 関数は `printf()` 関数のファイル出力版で，第一引数に FILE ポインタ型の変数を指定する以外，使用方法は `printf()` 関数と全く同じである．最後に `fclose()` 文でそのファイルを閉じて終了している．

15.3 ファイル入出力関数群

ファイルを操作する関数としては，
一文字単位の読み書き (`fgetc()`, `fputc()`),
一行単位の読み書き (`fgets()`, `fputs()`),
バイト数指定の読み書き (`fread()`, `fwrite()`),
書式付き読み書き (`fscanf()`, `fprintf()`),
がある．それぞれの関数のプロトタイプ宣言を以下に示す.

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

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
size_t fread(void  *ptr,  size_t size, size_t nmemb,
             FILE *stream);
size_t fwrite(const void *ptr, size_t size,
              size_t  nmemb, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

ファイルから一文字読みだす関数を fgetc() という．
fgetc(FILE *stream) 関数は stream から次の文字を unsigned char として読み int にキャストして返す．ファイルの終りやエラーとなった場合は EOF を返す．EOF が -1 であることは既に述べた．これには理由がある．それは，もし fgetc() の戻り値が char 型だった場合，char は 0 から 255 までの値しか確保できないので -1 は格納できず，無理矢理格納しようした結果 -1 は 255，つまり 16 進数で言えば 0xFF の値が格納されるのである．0xFF は可読文字コードではないが，実行可能ファイルをエディタなどで無理矢理オープンすると 0xFF は各所に存在する．もし fgetc() 関数のの戻り値が char 型だと，ファイルの終端まで読まず，0xFF を読んだ時点で終端と見なされて，正常なファイル終端か否かの区別がつかなくなってしまう．ゆえに fgetc() の戻り値は int 型となっているのである．int 型なら -1 も格納できるし，char 型の値もすべて格納できる．従って fgetc() の戻り値は int 型の変数に格納し，その後 -1 の判断を行なってから char 型の変数に格納する，という段階を踏むのである．

ファイルに一文字書き出す関数を fputc() という．
fputc(int c, FILE *steam) 関数は文字 c を unsigned char にキャストして stream に書き出す．戻り値は unsigned char として書き込まれた文字を int にキャストして返す．エラーが発生した場合には EOF を返す．

ファイルから文字列を一行読みこむ関数を fgets() という．
fgets(char *s, int size, FILE *stream) 関数は size よりも一文字以上少ない文字を stream から読みこみ，s で示されるバッファに書き込む．読み込みは EOF または改行文字を読み込んだ後終わる。改行文字は読まれるとバッファに書き込まれる。'\0' 文字がバッファの中の最後の文字の後に 1 文字書き込まれる。fgets() 関数は成功すると 第一引数のポインタと同じ位置を示すポインタ値である s を返し，ファイルの終りあるいはエラーの場合には NULL を返す．

ファイルに一行書き出す関数を fputs() という．
int fputs(const char *s, FILE *stream) 関数は文字列 s を stream に

書き込む．文字列の終端記号である'\0'は出力されない．fputs() は成功すると負ではない値を返し，エラーが発生した場合は EOF を返す.

以下のプログラムは fgets() と fputs() を使って第一引数で指定されたファイルの内容を第二引数で指定したファイルに書き出すものである.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_LEN 4096

int main ( int argc, char **argv )
{
    FILE *fin, *fout;
    char str[MAX_LEN];

    if ( argc != 3 ) {
        printf("### You need two files.\n");
        exit (EXIT_FAILURE);
    }
    if ((fin = fopen(argv[1],"r")) == NULL ) {
        printf("### fopen() failed, file [%s].\n",
               argv[1]);
        exit (EXIT_FAILURE);
    }
    if ((fout = fopen(argv[2],"w")) == NULL ) {
        printf("### fopen() failed, file [%s].\n",
                argv[2]);
        exit (EXIT_FAILURE);
    }


    while ( fgets(str, MAX_LEN-1, fin) != NULL ) {
        if ( fputs(str, fout) < 0 ) {
            printf("### fputs() failed, [%s].\n",
                   argv[2]);
        }
    }

    fclose( fin );    fclose( fout );
```

```
    return EXIT_SUCCESS;
}
```

まず最初に引数が 2 つであるかどうかをチェックし，引数が 2 つでなければ終了するようにしている．次に，第一引数を読みこみモードでオープンし，正しくオープンできたかどうかを判定している．C では条件式の中に文を埋めこむことができる．すなわち

```
    fin = fopen(argv[1],"r");
    if ( fin == NULL ) {
```

を 1 つにまとめたのが上のプログラムである．上のプログラムではこれを利用して fopen() でオープンした結果がエラーか否かを判定している．次の if 文は第二引数を書き込みモードでオープンし結果がエラーか否かの判定を上と同じ要領で行なっている．その次の while() 文では fgets() 関数で一行読みこんで，文字列 str に格納し，それがファイルの終りか否かの判定も同時に行なっている．fgets() で str に文字列が読みこまれたら，その結果を次の fputs() 文でファイルに書き出している．

fgets() 関数は第二引数に与えた文字数を読み取る以前に改行文字が現れた場合改行文字の直後に文字列の終端記号である '\0' をつけくわえた文字列を返すので，そのまま fputs() 関数で文字列 str を書き出すことができる．

最後に開いてあった 2 つのファイルを fclose() 文で閉じて終っている．

なお，このプログラムには一行の文字列が 4096 文字を越えた場合動作が保証されないというバグが存在する．入力されたファイルの長さに応じて文字列 str のサイズを可変にできるようなプログラムではないことに注意されたい．入力文字列の長さに応じて可変長の文字列を格納するためには動的メモリ割り付け関数 malloc() を使わねばならない．

実際に 4096 文字以上の文字列を読みこんだ場合 gets() 関数は 4096 文字目に文字列の終端記号である '\0' を代入して 4097 文字以降を無視してしまう．

さらに第一引数で指定したファイル名と第二引数で指定したファイル名が同じであった場合，そのファイルが破壊されてしまうというバグも存在する．

演習問題 第一引数で指定したファイルの内容を第二引数で指定したファイルに書き出す上記のプログラムを fgets(), fputs() を使う代わりに fgetc(), fputc() を使って書き直せ．

fread() 関数と fwrite() 関数とはバイトストリームの入出力に用いられる．本ドキュメントでは説明は省略する．

ファイルから書式付き読みこみする関数を fscanf() という．
int fscanf(FILE *stream, const char *format, ...) 関数は format に従って stream から入力を読みこむ．format については printf() 関

数で説明した % とその後に続く文字で指定された型式で指定する．fscanf() は代入された入力要素の個数を返す．入力が失敗した場合には EOF が返される．

例えば次のプログラムは第一引数で指定されたファイルから文字列を読みこんで表示するプログラムである．

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_LEN 4096

int main(int argc, char **argv)
{
    FILE *fp;
    char str[MAX_LEN];

    if ( argc != 2 ) {
        printf("### You need a file name.\n");
        exit (EXIT_FAILURE);
    }
    if ( (fp = fopen(argv[1],"r")) == NULL )  {
        printf("### Could not open a file [%s].\n",argv[1]);
        exit (EXIT_FAILURE);
    }
    while ( (fscanf(fp, "%s", str)) != EOF){
        printf("[%s]\n", str);
    }
    fclose(fp);
    return EXIT_SUCCESS;
}
```

fscanf() 関数は空白や TAB などを読み飛ばしてしまう．fscanf() 関数で空白や TAB を読みこむ方法はない．fgets() 関数を使って処理する必要がある．なお，このプログラムにも 1 つの入力文字列が 4096 文字を越えた場合動作が保証されないというバグが存在することに注意されたい．

次のプログラムは数値だけからなるファイルを読みこんで，読みこんだ数，平均値，分散，標準偏差 (いずれも不偏推定量ではなく標本統計量である) を表示するプログラムである．

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>

int main(int argc, char **argv)
{
    FILE *fp;
    double x=0.0, mean=0.0, s1=0.0, ss=0.0, n=0.0;

    if ( argc != 2 ) {
        printf("### You need a file name.\n");
        exit (EXIT_FAILURE);
    }
    if ( (fp = fopen(argv[1],"r")) == NULL )  {
        printf("### Could not open a file [%s].\n",argv[1]);
        exit (EXIT_FAILURE);
    }
    while ( fscanf(fp, "%lf", &x) != EOF) {
        n += 1.0;
        s1 = x - mean;
        mean += s1 / n;
        ss += (n-1.0)/ n * s1 * s1;
    }
    fclose(fp);

    printf("n=%d, mean=%f, variance=%f, s.d.=%f\n",
           (int)n, mean, ss/n, sqrt(ss/n));

    return EXIT_SUCCESS;
}
```

このプログラムでは入力値に数字以外の値があると暴走するというバグがある．まともに動くプログラムを作るには入力されたデータが数値であるか文字列であるかを判別する処理を加えなければならない．

なお，上記のプログラムでは平均と偏差平方和とを求めるときに漸化式を用いている．良くいわれるとおり分散の定義式は下記のとおりなのだが

$$\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \bar{X}\right)^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{X}^2 \tag{1}$$

左辺の式は数値計算では使ってはならない．２乗の平均から平均の２乗を引いて分散を求めてはいけないのである．理由は大きな数から大きな数を引いて小さな数を求めているため数値計算による演算誤差が含まれてしまうから

である．上のプログラムでは代わりに，平均と偏差平方和を求める漸化式

$$\bar{X}_n = \bar{X}_{n-1} + \frac{x_n - \bar{X}_{n-1}}{n} \tag{2}$$

$$SS_n = SS_{n-1} + \frac{n-1}{n}\left(x_n - \bar{X}_{n-1}\right)^2 \tag{3}$$

を用いて繰り返しを一度で済ませている．漸化式を用いた計算法を用いれば誤差を着にせず一度の繰り返しで平均と偏差平方和を計算することができる．

ファイルに対して書式付き書き出す関数を `fprintf()` という．

```
int fprintf(FILE *stream, const char *format, ...)
```

関数は `printf()` 関数と同じく書式付きの出力を `stream` に対して行なうものである．`fprintf()` のサンプルプログラムは既に記したのでここでは繰り返さない．

演習問題 5 つの数値が書かれているファイルを読みこんで，その数値の平均と分散を求めるプログラムを書け．あらかじめ `float data[5];` と宣言しておいて，この配列にデータを読みこむようにせよ．

16 構造体

16.1 構造体の宣言

複数の変数を一つにまとめて新しい変数の型を定義することを構造体を作るという．構造体を定義する書式は，

```
typedef  struct 構造体タグ {
    まとめたい変数を列挙;
} 新しく定義した構造体の名前;
```

である．このような書式を構造体の型宣言という．

例えば 2 次元平面上の 1 点を直交座標系 XY 座標で定義し構造体 `Vector2D_t` を作るには

```
typedef  struct Vector2D_t {
    int  X;
    int  Y;
} Vector2D_t;
```

とする．このように宣言すると `Vector2D_t` が `int` や `char` と同じような変数と見なすことができる．構造体タグ名と構造体名は別々の名前空間で監理されるので同じであっても問題は生じない．

プログラムの始めに上記の構造体の宣言があれば，

```
Vector2D_t  a_point;
```

と宣言すれば XY 座標が格納できる a_point という新たな変数 (構造体) が使えるようになる．この変数へ値を参照したり，代入するには

```
a_point.X = 3;
a_point.Y = 4;

printf("(%d,%d)\n", a_point.X, a_point.Y);
```

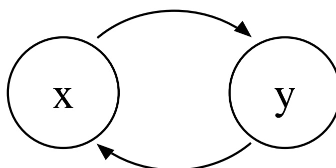
などのように 構造体変数 a_point に . 演算子をつけて後に構造体内部の変数名を書けば良い．このように構造体の中にまとめられた X, Y などの変数のことを構造体のメンバ，またはメンバ変数と言う．

```
a_point.X = 3;
```

は,「Vector2D_t 型の変数 a_point のメンバ X に 3 を代入するなどと言う．

16.2 構造体を使ったニューロンの表現

ここで，2 つのニューロンが互いに結合している簡単なニューロンのモデルを考えよう．それぞれのニューロンは活性値を持っていて，お互いにシナプス結合していると考える．このようなときに，それぞれのニューロンを 1



つの構造体として定義してみると

```
typedef struct NEURON {
    double output;
    struct NEURON *input_neuron;
} neuron;
```

のような構造体が定義できる．この例では自分自身と同じ内容の NEURON という構造体へのポインタが入っている．ニューロン x と ニューロン y とが定義されお互いの入力になるのであれば

```
neuron x, y;
x.input_neuron = &y;
y.input_neuron = &x;
```

とすればお互いに結合されたニューロンと見なすことができる．以下に示すプログラムは完全に動作する．

```
#include <stdio.h>

typedef struct NEURON {
    double output;
    struct NEURON *input_neuron;
} neuron;

int main(int argc, char **argv)
{
    neuron x, y;

    x.output =  1.0; y.output = 10.0;

    x.input_neuron = &y;
    y.input_neuron = &x;

    printf("x.output=%f\n", x.output);
    printf("y.output=%f\n", y.output);
    printf("x.output=%f\n", x.input_neuron->input_neuron->
                      output);
    printf("y.output=%f\n", y.input_neuron->input_neuron->
                      output);
    printf("x.output=%f\n", x.input_neuron->input_neuron->
                      input_neuron->input_neuron->output);
    printf("y.output=%f\n", y.input_neuron->input_neuron->
                      input_neuron->input_neuron->output);

    return 0;
}
```

-> 演算子は，構造体へのポインタデータにアクセスするための演算子である．この演算子のことをアロー演算子と言ったりもする．

最初の 2 つの `printf()` 文は自明だろう．自分の出力値を印字しているだけである．3 つめの `printf()` 文は x の `input_neuron` が指しているポインタのニューロン (すなわち y) の指している `input_neuron` (すなわち x 自身) の出力値を印字している．4 つめの `printf()` 文も同様で自分への入力ニューロンの入力ニューロンである自分自身の出力値を印字してる．同様にして 5 つめと 6 つめの `printf()` 文は参照の参照の参照の参照の参照をして結局自分自身の出力を印字している．

16.3 構造体へのメモリの動的割り付け

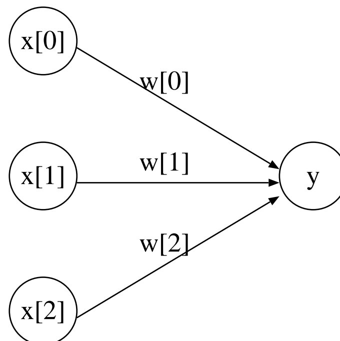
次に x[0], x[1], x[2] という 3 つのニューロンからのシナプス結合がある y というニューロンがあると仮定しよう．今度は構造体を以下のようにする．

```
typedef struct NEURON {
        double output;
        int Ninp;
        double *weights;
        struct NEURON **input_neuron;
} neuron;
```

先の例と異なるのは，neuron 構造体へのポインタのポインタ，すなわち neuron 構造体へのポインタの配列を使わなければならないことである．さらにシナプス結合の程度を表す倍精度浮動小数点の配列を用意している．ニューラルネットワークにおける学習とは，このシナプス結合係数の変化のことである．ここでは単純に全ての結合係数が 1.0 であることを仮定しよう．そして x[0], x[1], x[2] の 3 つのニューロンの出力値がそれぞれ，1.0, 10.0, 100.0 であるとする．ニューロン y の出力は各入力ニューロンの和

$$y = \sum_{i=0}^{2} w_i x_i.$$

に従うとする．このときのプログラムは次のようになる．




```
#include <stdio.h>
#include <stdlib.h>

typedef struct NEURON {
    double output;
    int Ninp;
    double *weights;
```

```
    struct NEURON **input_neuron;
} neuron;

#define N_OF_NEURON 3

int main(int argc, char **argv)
{
    int i;
    neuron y, x[N_OF_NEURON];

    /* 結合係数の初期化，すべて 1.0 とする */
    y.weights=(double *)malloc(N_OF_NEURON * sizeof(double));
    for ( i=0; i < N_OF_NEURON; i++ ) {
        y.weights[i] = 1.0;
    }

    /* neuron 構造体へのポインタの配列の動的メモリ割り付け */
    y.input_neuron
        = (neuron **)malloc(N_OF_NEURON * sizeof(neuron *));
    for ( i=0; i < N_OF_NEURON; i++) {
        y.input_neuron[i] = &x[i]; /* 各結合を表す */
    }

    /* 各入力ニューロン x[0], x[1], x[2] の出力値を設定 */
    /* 下の 3 つの書式は C の文法上問題がない */
    x[0].output = 1.0;
    (x+1)->output = 10.0;
    (*(x+2)).output = 100;

    /* 正しくポインタが動作するかの確認 */
    for ( i=0; i < N_OF_NEURON; i++) {
        printf("y.input_neuron[%d]->output=%f\n", i,
            y.input_neuron[i]->output);
    }

    /* y の出力値の計算 */
    y.output = 0.0;
    for ( i=0; i < N_OF_NEURON; i++ ) {
        y.output += y.weights[i] * y.input_neuron[i]->output;
```

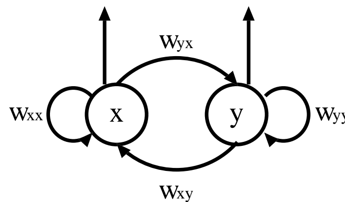
```
    }
    printf("The output of neuron y is %f.\n", y.output);

    return 0;
}
```

このプログラムのポイントは動的メモリ割り付け関数 malloc() を使っていることである．ポインタは定義しただけでは実体がない (14, p.47) ので，その実体をメモリ上に確保する必要がある．このメモリの確保をするのが malloc() 関数である．malloc() 関数は引数で与えられたサイズのメモリを確保し，それを void へのポインタとして返す関数である．従って malloc() 関数を使ってメモリを確保したときにはその用途によって適切にキャストしなければならない．malloc() 関数はメモリの割り当てに失敗したときには NULL ポインタを返すのだが，上のプログラムではチェックしていない．当然実用的なプログラムを作るときには，malloc() 関数の戻り値を参照して要求したメモリが確保できたか否かを確認すべきである．

上のプログラムでは結合係数 y.weights と y.input_neuorn のメモリを確保するために malloc() 関数を使っている．

次に 2 つのニューロンが互いに結合しあっている場合を考える．このとき



の動作方程式は

$$\begin{aligned} \tau \frac{dx}{dt} &= w_{xx}x + w_{xy}y \\ \tau \frac{dy}{dt} &= w_{yx}x + w_{yy}y \end{aligned} \tag{4}$$

に従うものとする．このときプログラムは以下のようになるだろう．

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FROM   0.0
#define TO   100.0
```

```
#define TAU   0.001

typedef struct NEURON {
    double output, old_output;
    int    Ninp;
    double *inp_wgt;
    struct NEURON **inp_neuron;
} neuron;

void initialize_neuron(neuron *a, int N)
{
    int i;

    a->output = 0.0;
    a->Ninp = N;
    a->inp_wgt = (double *)malloc(sizeof(double) * (N));
    if ( a->inp_wgt == NULL ) {
        fprintf(stderr,"### malloc() faild.\n");
        exit (EXIT_FAILURE);
    }
    for (i=0; i< a->Ninp; i++) {
        a->inp_wgt[i] = 0.0;
    }
    a->inp_neuron = (neuron **)malloc(sizeof(neuron *) * (N));
    if ( a->inp_neuron == NULL ) {
        fprintf(stderr,"### malloc() faild.\n");
        exit (EXIT_FAILURE);
    }
}

double output_f(double value)
{
    return value;
}

void calc_neuron(neuron *a)
{
    double wrk;
    int i;
```

```
    wrk = 0.0;
    for(i=0; i< a->Ninp; i++){
        wrk += a->inp_wgt[i] * a->inp_neuron[i]->old_output;
    }
    a->output += output_f(wrk) * TAU;
}

void update_neuron(neuron *a)
{
    a->old_output = a->output;
}

int main_loop( neuron *a, neuron *b)
{
    double t = FROM;
    while ( t <= TO ){
        printf("%6.3f %7.4f %7.4f\n", t, a->output, b->output);
        calc_neuron(a);     calc_neuron(b);
        update_neuron(a);     update_neuron(b);
        t += TAU;
    }
    return 0;
}

int main(int argc, char **argv)
{
    neuron x, y;

    if ( argc != 3 ) {
        printf("### Usage: %s <double> <double>\n", argv[0]);
        exit (EXIT_FAILURE);
    }

    initialize_neuron(&x, 2);
    initialize_neuron(&y, 2);

    x.output = x.old_output = atof(argv[1]);
    y.output = y.old_output = atof(argv[2]);
```

```
    x.inp_neuron[0] = &x;
    x.inp_neuron[1] = &y;
    y.inp_neuron[0] = &y;
    y.inp_neuron[1] = &x;

    x.inp_wgt[0] = 0.0; /* w_{xx} */
    x.inp_wgt[1] = 1.0; /* w_{xy} */
    y.inp_wgt[1] = -1.0; /* w_{yx} */
    y.inp_wgt[0] = 0.0; /* w_{yy} */

    return main_loop(&x, &y);
}
```

このプログラムでは $\tau = 0.001$ とし，結合係数は $w_{xx} = 0$, $w_{xy} = -1$, $w_{yx} = -1.0$, $w_{yy} = 0.0$ として，時刻 $t = 0$ から $t = 100$ までの時間経過に伴うニューロンの変化を出力している．2 つのニューロンへの初期値はプログラムに渡す 2 つの引数としている．

演習問題 初期値を変化させたとき，上のプログラムの動作はどうなるか実行してグラフにしてみよ．

演習問題 上のプログラムの結合係数を変化させて動作をグラフに表せ

演習問題 上のプログラムと定性的に同じふるまいをする結合係数の組をさがせ．(ヒント：行列の固有値問題であることに気がつけばよい．固有値が虚数になる場合とはどういう条件か)

構造体を使ってニューロンの振舞いを記述する方法を説明してきた．次に，ニューロンの出力が非線型のさまざま関数によって記述されることを C の構造体の中に取り入れる方法を紹介しよう．次のプログラムは output というユニットの出力にシグモイド関数を使った例である．

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FROM -10.0
#define TO    10.0
#define STEP  0.01

typedef struct NEURON {
    double output, old_output;
    int N;
    double *weights;
```

```
    struct NEURON **input_neuron;
    double (*outputf)();
} neuron;

double sigmoid( double x )
{
    return 1.0 / ( 1.0 + exp ( -x ) );
}

int initialize_neuron( neuron *a, int N )
{
    int i;

    a->output = 0.0;
    a->old_output = 0.0;
    a->N = N;
    a->weights = (double *)malloc(sizeof(double) * N);
    if ( a->weights == NULL ) {
        fprintf(stderr, "### malloc() failed.\n");
        exit (EXIT_FAILURE);
    }
    for ( i=0; i<a->N; i++ ) {
        a->weights[i] = 0.0;
    }
    a->input_neuron = (neuron **)malloc(sizeof (neuron *) * N);
    if ( a->input_neuron == NULL ) {
        fprintf(stderr, "### malloc() failed.\n");
        exit (EXIT_FAILURE);
    }
    a->outputf = sigmoid;
    return EXIT_SUCCESS;
}

void calc_output ( neuron *a )
{
    int i;
    double wrk = 0.0;

    for ( i=0; i<a->N; i++ ) {
```

```
        wrk += a->weights[i] * a->input_neuron[i]->output;
    }
    a->output = a->outputf( wrk );
}

int main(void)
{
    neuron output, input;

    initialize_neuron( &output, 1 );
    initialize_neuron( &input, 0 );

    output.input_neuron[0] = &input;
    output.weights[0] = 1.0;

    for (input.output=FROM; input.output<=TO;input.output += STEP){
        calc_output( &output );
        fprintf(stdout, "%5.3f %7.3f\n",
                          input.output, output.output);
    }

    return 0;
}
```

このプログラムでは１つのニューロン input から入力を受け取り，その活性値に基づいて自分の活性値を決めるニューロン output が定義されている．input ユニットから output ユニットへの結合は main() 関数の中で

```
    output.input_neuron[0] = &input;
```

という文で行なわれている．さらにこの 2 つのニューロンの結合係数は 1.0 としてある．上のプログラムでは input ユニットの活性値が FROM から TO まで STEP 刻みで変化したときの output ユニットの出力値を表示している．

構造体宣言の中に double (*outputf)(); というメンバが加えられていることに注目してほしい．これは double 型の値を返す関数へのポインタの定義である。

initialize_neuron() 関数の中で a->outputf = sigmoid; として，このユニットの出力関数を sigmoid() 関数に結びつけている。言い替えれば，output ユニットはシグモイド関数 (0 から 1 までの値を返す) を出力関数として割り当てられている．

このように出力関数を構造体の中に埋めこむのは，それぞれのニューロンに対して別の出力関数を定義したい場合などに有効である．例えば入力層のユニットには線形の出力関数を仮定し，中間層のユニットの出力関数にはシグモイド関数用い，出力層のユニットの出力関数には入力が負の値であれば 0 を，そうでなければ 1 を出力するようなステップ関数としたい場合などである．構造体のメンバとして出力関数へのポインタを定義しておくことによって，すべてのニューロンの出力値を計算する場合，たった 1 つの関数 `calc_output()` 関数で済んでしまう．上のプログラムでは `calc_output()` 関数の中で `a->output = a->outputf( wrk );` と書いてある部分で，`sigmoid()` 関数が実行される．

演習問題 上のプログラムを実行し，結果をグラフ化せよ．

演習問題 上のプログラムでシグモイド関数のパラメータを変えて実行し，結果をグラフ化せよ．

16.4 構造体のメモリイメージ

構造体変数を作ったとき，どのようにメモリ上にそれが展開されているかを知っておかないと，ファイルのバイナリアクセスするときに困ったバグに出くわすことになる．例えば次のような構造体を作ったとする．

```
typedef  struct  Chr_Num_t {
    char  chr;
    int  iNum;
} Chr_Num_t;
```

上のような構造体を宣言し，

```
    Chr_Num_t  a_CN;
```

として，変数 `a_CN` を宣言する．ここで，

```
    printf( "%d\n", sizeof( a_CN ) );
```

として `a_CN` のサイズを調べてみてほしい．`sizeof` 演算子は指定した型または変数が，どれだけメモリを占有しているかを示すものである．通常なら，`char` 型が 1 バイトで `int` 型が 4 バイト (gcc の 32 ビットコンパイラの場合) なので，上の `printf()` 関数は

```
    5
```

と表示されるはずである．しかし結果は

```
    8
```

と表示されてしまうことがある．

構造体変数がメモリ上に展開されているのかを見ることにしよう．printf() 関数に構造体メンバのアドレスを表示させるようにすればよい．

```
printf( "&a_CN = %p\n", &a_CN );
printf( "&a_CN.chr = %p\n", &a_CN.chr );
printf( "&a_CN.iNum = %p\n", &a_CN.iNumr );
```

上記をを実行すると，例えば

```
&a_CN = 0xbffff2d4
&a_CN.chr = 0xbffff2d4
&a_CN.iNum = 0xbffff2d8
```

と表示される．

a_CN 自体のアドレスと，そのメンバ chr のアドレスが同じで，あることは当然だが，chr と iNum のアドレスが 4 だけ離れている．

つまり，この表示されたアドレスを図解すると，以下のようになる．

```
              chr      ????                iNum
               ↓        ↓                   ↓
 ┌─────────┬─────────┬─────────┬───────────────────┬─────────┐
 │.........│         │         │                   │.........│
 └─────────┴─────────┴─────────┴───────────────────┴─────────┘
  bffff2d3  bffff2d4           bffff2d8...bffff2dc
               ↑                        ↑
              &chr==&a_CN             &iNum
```

0xbffff2d4 番地には構造体のメンバ chr が入っており，0xbffff2d8 番地から 0xbffff2dc 番地までには iNum が入っているのだが，その間の 0xbffff2d5 番地から 0xbffff2d7 番地までが使用されずにいる．この 3 バイトがあるために sizeof(a_CN) が 8 になってしまっているのである．

このようなメモリの隙間が入ることがある．Intel の現在の 32bit CPU は，32 bit のメモリ領域に対して最も高速な代入操作・演算が行なわれるように設計されている．つまり CPU は 32bit，すなわち 4 バイトのメモリ領域に対して最も高速にアクセスできるのだから，例えば，

```
b_CN = a_CN;
```

などのように構造体変数の代入操作をしたときは，a_CN のメンバ全体が 4 の倍数バイトになっていた方が代入が速くできるようになっている．

```
typedef  struct Chr_Num_t {
    char  ch1;
```

```
        char  ch2;
        char  ch3;
        char  ch4;
        int  iNum;
    } Chr_Num_t;
```

としたときには a_CN が sizeof 演算子で何バイトになっているかを見てみれば 8 バイトとなっているはずである．

このように，コンパイラは CPU の動作特性に合わせて構造体の中のメモリ領域に「空の領域」を生成する場合がある．この「空の領域」のことを「バウンダリ」と呼び，バウンダリが発生する現象のことを「構造体のアラインメント」と言う．

gcc で構造体のアラインメントをなくすには，構造体の宣言のときに，それぞれのメンバ変数の宣言の最後に __attribute__((packed)) という修飾子を付ける．

```
    typedef struct Chr_Num_t {
        char   ch   __attribute__((packed));
        int  iNum   __attribute__((packed));
    } Chr_Num_t;
```

のようにすると，先ほどの sizeof(a_CN) は 5 となる．

構造体のアラインメントが発生するか否かはコンパイラ依存である．自分の使っているコンパイラがアラインメントを発生するかどうかを確認しておかないと，別の環境に移植したとき思わぬバグに悩まされることがある．

17　演算子の優先順位

数学で用いられる数式にも優先順位がある．例えば，$1 + 2 * 3 - (4 + 5)/6$ という式では，掛算と割り算が足し算や引き算よりも優先される．カッコがあれば先にカッコの中の演算を優先させる，などの演算規則がある．同様にして C にも優先順位があり数式の計算順序は，カッコの扱いなどを含めて数学のそれとほぼ同じである．C には数学に用いられる演算子以外にも多くの演算子が存在する．ここでは演算子の優先順位が高い順に並べた表を示す．以下の表は優先順位の高い順に並んでいる．この優先順位を変更するためにカッコ ( ) が用いられる点は数学と同様である．C における演算子の優先順位を p.78 の表 17 に示した．

表 6: 演算子の優先順位

演算子	名前または意味	結合方向
++, --	後置インクリメント，デクリメント	左から右
( )	関数呼び出し	
[ ]	配列要素	
->	構造体，共用体メンバの間接参照	
.	構造体，共用体メンバの直接参照	
++,--	前置インクリメント，デクリメント	右から左
!	否定 (NOT) 演算	
~	1 の補数	
-, +	単項マイナス，単項プラス	
&	アドレス	
*	アドレスに格納させているデータ	
sizeof	サイズ (バイト数)	
(type)	型キャスト	
*, /,%	乗算，除算，剰余	左から右
+, -	加算，減算	左から右
<<, >>	左シフト，右シフト	左から右
<, <=, >=, >	比較演算子	左から右
==,!=	等位演算子	左から右
&	ビットごとの AND	左から右
^	ビットごとの排他的 OR	左から右
|	ビットごとの OR	左から右
&&	論理積	左から右
||	論理和	左から右
? :	３項条件式	左から右
=	代入	左から右
*=	複合代入	
/=		
%=		
+=		
-=		
<<=,>>=, &=,^=,|=	複合代入	
,	カンマ	左から右