C言語入門

執筆者：東海林篤、奥田貴志、青木達也　2001 年

このテキストは基本的にコンピュータ言語を生まれてはじめて触れる人向けに書かれている。C言語の入門書は書店に行けば数多くあるが[1]、このテキストでもそれらに書かれている基本的な部分については網羅している。例題もできるだけ簡素かつ内容的に興味のもてるよう努力したつもりである。例題については

```
/usr/local/stex-local/c-sample
```

に置いてあるので、そこから持ってきて欲しい。

また、このテキストはCにある程度造詣のある学生にとっては物足りない内容かも知れない。しかし、「そう言えばこれってどうするんだっけ」とか「○○するにはどうすれば良いのかな?」とかいったことについて、できるだけ手助けできるようにしたつもりである。

§1　C言語を「さわってみる」

> 注意：この章と次章はコンピュータ言語を本当に生まれてはじめて触れる人向けに書いてある。BASIC、Java、Perl 等 (すなわちコンパイル言語ではない言語) しか触ったことがないという人も対象である。自分が該当者ではないと思った読者は先へ進むこと。

まず、用意されたプログラム "xxx.c" を emacs などのエディタを使って見て欲しい。

```
c01:~/c_text> emacs xxx.c &
c01:~/c_text>
```

これがプログラムの中身である。プログラムの中身がこのようなものであるということを覚えて欲しい。中身は理解しなくて良い。

さて、プログラムがあるからといって先に進まない。C言語ではこのプログラムをコンパイルする必要がある。コンパイルとは翻訳の意味、つまり、コンピュータがプログラムに書いてある意味を理解し、それを実行できるように新たなファイル (実行ファイル) を作るように仕向ける必要がある。

```
c01:~/c_text> gcc -o xxx xxx.c -lm
c01:~/c_text>
```

これで OK である。そうすると、新たに "xxx" という実行ファイルができる。エラーが出たら、周りの知ってそうな人に聞いてみて欲しい。これでようやくプログラムの中身を「実行できるようになった」。実行してみよう。ただし注意してほしいのは、この実行ファイルは、データをディスプレイに出力するため、普通に実行すると、

```
c01:~/c_text> ./xxx
 0.000000 0.000000
 3.000000 0.000000
 4.500000 2.598076
…
```

[1] 参考図書 (1) はかなりお勧めである。

というように延々とデータが画面に出力される。そこで、shell のリダイレクションの機能 ”>” を用いてファイルに出力する。

```
c01:~/c_text> ./xxx > out.dat
c01:~/c_text>
```

後は、結果を gnuplot で出力してみる。何が出てくるかは読者のお楽しみである。

```
c01:~/c_text> gnuplot

        G N U P L O T
        Linux version 3.7
        patchlevel 0
        last modified Thu Jan 14 19:34:53 BST 1999

        Copyright(C) 1986 - 1993, 1998, 1999
        Thomas Williams, Colin Kelley and many others

        Type ‘help‘ to access the on-line reference manual
        The gnuplot FAQ is available from
                <http://www.uni-karlsruhe.de/~ig25/gnuplot-faq/>

        Send comments and requests for help to <info-gnuplot@dartmouth.edu>
        Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>


Terminal type set to ’x11’
gnuplot> plot"out.dat" u 1:2 w l
gnuplot>
```

§2　プログラムの作成とコンパイル

「§1」で行った操作について、簡単ではあるが、もう少し詳しく説明する。その後で具体的なプログラミング言語の説明を行う。

プログラムは emacs などのエディタを用いて作成する。エディタの使い方は実験テキストを見て欲しい。プログラムを一通り打ち終え、一応デバッグ (間違いチェック) も完了したら、コンパイルを行う。「§1」の例をもう一度あげると、

```
c01:~/c_text> gcc -o xxx xxx.c -lm
c01:~/c_text>
```

となる。このコマンドの一般的な書式は

gcc（オプション）コンパイルされるファイル名

オプション : -o 実行ファイル名 -- 「ファイル名」という名前の実行ファイルを作る。
-c -- 「ファイル名」に対するオブジェクトファイル（○○.o）を作る（後述）

プログラム内にミスがあればエラーを出力する。

```
c01:~/c_text> gcc -o xxx xxx.c -lm
xxx.c: In function ‘koch’:
xxx.c:27: parse error before ‘theta’
c01:~/c_text>
```

エラーの元になっている行と、その原因が出力されるので、それに従って直していく。そうして、最終的にエラーを除去してはじめて実行ファイルが作られる。

§3　基礎事項

プログラムを作成する上での基礎となる事項を簡単に説明する。項目は主に

1. Cプログラミングの基礎の基礎
2. 変数
3. 四則演算
4. 条件文
5. 繰り返し文

がある。順に説明していき、最後にこれらを用いた例を紹介する。

§3.1　Cプログラミングの基礎の基礎

プログラムの中心となる部分をメイン関数という。

```
 int main(){
  文;
}
```

がメイン関数を表す。{} 内に具体的なプログラムの内容を書く。
”;” を付けることにより、コンパイラは 1 つの命令が終了したと見なす。

§3.2　変数

C言語 (に限らず一般的なプログラミング言語) では、数値計算をする時には変数というものを用いることがほとんどである。具体例として、2×5 という計算を、変数 x に 5 を代入して行い、その結果を変数 y に代入したい時には

```
  x = 5;
  y = 2*x;
```

の様に書く。この例では、わざわざ変数 x を用いなくても良いと思うかも知れないが、たとえば、関数 $y = 2x$ の直線を書きたいと思った時には x をある範囲で変化させなければいけない。この時、変数を使わずに数値をいちいち代入していくよりも、変数 x をある範囲で変化するようなプログラムを書いて、y = 2*x; とする方がより一般的に書くことができる。
変数を使うためにははじめに変数の「型」を定義しなければいけない。変数定義の基本的な書式は

基本型 変数名，変数名，… ；

で表される。基本型の種類、意味を以下の表に示す。

`int`	整数型	-2147483648〜2147483647
`float`	単精度実数型	0.0 と $\pm 0.29 \times 10^{-38}$〜$\pm 1.7 \times 10^{38}$、有効数字 7 桁
`double`	倍精度実数型	0.0 と $\pm 0.29 \times 10^{-38}$〜$\pm 1.7 \times 10^{38}$、有効数字 15 桁
`char`	文字列	1 文字

ここで、単精度/倍精度実数型という似たような概念が出てきたが、通常は倍精度実数型のみを扱う。

例えば、整数型の変数として `i` を倍精度実数型の変数として `x,y,z` を定義する時には

```
int i;
double x,y,z;
```

の様に書く。

§3.3 簡単な演算

C言語での計算の手順 (計算の方法、値の代入は方法) は上で説明した。ここでは算術演算の記号を表にまとめる。

`a = b`	右の値、計算式を左の変数に代入する。
`a + b`	加法
`a - b`	減法
`a * b`	乗法
`a / b`	除法
`a % b`	`a` を `b` で割った時の剰余

§3.4 条件文

「もし○○だったら××しなさい、そうでなければ…」などという様な働きをする文を条件文という。条件文の最も基本的なものは `if` 文である。`if` 文の最も単純な書式は

```
if(○○){
  ××;
}
```

である。上の「○○」の部分を条件式という。「××」の部分を実行文という。例えば「xy 平面上の任意の点 (x, y) が半径 1 の円内にあるならば、変数 `n` に 1 を加えなさい」という条件文は

```
if(x*x+y*y <= 1.0){
  n = n+1;
}
```

の様に書く。
if 文は else 文によって更に多様な条件をつけることができる。

```
if(条件式 1){
  条件式 1 を満たした時の実行文;
}else if(条件文 2){
  条件式 2 を満たした時の実行文;
}else if …
  …
}else{
  上の全ての条件を満たさなかった時の実行文;
}
```

条件式は主に大小の比較、等価、論理演算による真偽 (真なら 1、偽なら 0) によって評価される。以下の表ににこれらの演算を行う演算子をまとめる。

a == b	a と b が等しい
a != b	a と b が等しくない
a > b	a は b より大きい
a >= b	a は b より大きいか、等しい
a < b	a は b 小さい
a <= b	a は b より小さいか等しい
a || b	a と b の論理和
a && b	a と b の論理積
!a	a の否定

これらの演算結果を変数 (int 型) に代入することもできる。次の例は先の例と同じ意味である。

```
L = x*x+y*y <= 1.0
if(L){
  S = S+1;
}
```

§3.5 繰り返し文

「○○という条件の間××を繰り返しなさい」という働きをする文を繰り返し文という。繰り返し文の一番単純な文は while 文である。

```
while(○○){
  ××;
}
```

また、良く使う文として for 文をあげる。

```
for(初期条件; ○○; ×× 2){
  ×× 1
}
```

例えば、「§3.2」で示した例を、x の範囲を 0 〜 10 として、 for 文を用いて書くと、

```
for(x = 0; x <= 10; x++){
  y = 2*x;
}
```

となる。ここで出てきた ”x++” は ”x = x+1” つまり、「x+1 という計算をして、その値を再び x に代入する」の省略形である。この ++ をインクリメント演算子といって、良く使うので覚えて欲しい。同様に ”x--” は ”x = x-1” を表し、”--” をデクリメント演算子という。

§3.6 例：円周率の計算

ここまで学んだことを基に例題を見てみる。ここでは疑似乱数を用いて円周率を求めるルーチンを作成する。

今、半径 1 の円と、それに外接する正方形を考える。それぞれの面積は

$$S_{\text{正方形}} = (2 \times 1)^2 = 4$$
$$S_{\text{円}} = \pi \times 1^2 = \pi$$

である。さて、この正方形の中にランダムに点を打っていくと、正方形内に打ち込まれた点の数 $N_{\text{正方形}}$ と円内に打ち込まれた点の数 $N_{\text{円}}$ の比はそれぞれの面積の比に等しくなる。

$$N_{\text{円}}/N_{\text{正方形}} = S_{\text{円}}/S\ \text{正方形}$$

これと先の式を合わせると、

$$\pi = 4N_{\text{円}}/N_{\text{正方形}}$$

となり、π の値を求めることができる。

さて、プログラムを見てみよう。[2]

ludolph.c ----------

```
#include<stdio.h>

int main(){

  int NEVENT = 10000;                    /* 正方形に入る点の数 */
  int IA = 421,IC = 54773,IM = 259200;   /* 乱数のパラメタ */
  int iseed = 12345;                     /* 乱数のたね */
  int n = 0;                             /* 円内に入った点の数。初期値は 0 */
  int i,j;                               /* for 文のカウンタ */
  double x,r, pi;                        /* x(y) 座標，動径の大きさ，円周率 */
```

[2] このプログラム名についている “ludolph” はドイツで使われている円周率の別称、ルドルフ数 (die Ludolphsche Zahl) からきている。ドイツの数学者 Ludolph Van Ceulen(1540 − 1610) は生涯をかけて円周率の計算した。その精度は小数点以下 35 桁 (3.14159265358979323846264338327950288) である。

```
  for(i = 0; i < NEVENT; i++){           /* NEVENT だけループを回す */
    r = 0.0;
    for(j = 0; j < 2; j++){
                    /* 2 回まわすことで、変数 x に x、y 両方の役割を負わせる */
      iseed = (iseed*IA+IC)%IM;          /* 乱数発生ルーチン */
      x = (double)iseed/(double)IM;
      r += x*x;                          /* r = r+x*x と同じ */
    }
    if(r <= 1.0){               /* r が 1 より小さいか同じなら n に 1 を加える */
      n++;
    }
  }
  pi = 4.0*(double)n/(double)NEVENT;     /* n : NEVENT = pi:4.0 */
  printf("pi = %f \n",pi);               /* 結果の表示 */
}
```

上から順に見ていく。

```
int main(){
    …
  }
```

メイン関数全体を表している。メイン関数の上にある、

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

については後で説明する。

```
  int NEVENT = 10000;                        /* 正方形に入る点の数 */
  int IA = 421,IC = 54773,IM = 259200;       /* 乱数のパラメタ */
  int iseed = 12345;                         /* 乱数のたね */
  int n = 0;                                 /* 円内に入った点の数。初期値は 0 */
  int i,j;                                   /* for 文のカウンタ */
  double x,r, pi;                            /* x(y) 座標, 動径の大きさ, 円周
率 */
```

変数の定義を行っている。変数のプログラム内での意味は後で説明する。ここでは先程述べなかった点が 2 つ含まれている。

- 初期値の代入：変数に初期値を代入したい時は、変数名と代入する値を "=" で結ぶ。
- 変数名の規則：と言うと大袈裟だが、大文字と小文字は区別される。"a" と "A" は別の変数と見なされる。通常、全て大文字で書いた場合は定数とする (#define 参照)。

```
   for(i = 0; i < NEVENT; i++){           /* NEVENT だけループを回
す */
    …
  }
```

for 文を用いて正方形内に入る点の数だけループをまわす。つまり、変数 i が 0 〜 NEVENT-1 までの計 NEVENT 回中の実行文を行う。この中で、点が円の中に入ったか判定し、入ったらその数を数えるという行為を繰り返す。

```
    r = 0.0;
    for(j = 0; j < 2; j++){
                    /* 2 回まわすことで、変数 x に x、y 両方の役割を負わせる */
      iseed = (iseed*IA+IC)%IM;          /* 乱数発生ルーチン */
      x = (double)iseed/(double)IM;
      r += x*x;                          /* r = r+x*x と同じ */
    }
```

ここでは (x, y) の組を決めるのだが、ここでは x と y をそれぞれ別の変数に代入するのではなく、一つの変数 x に 2 つの値の役割を負わせ、その自乗和を計算することにすることにした。

```
    r = 0.0;
    for(j = 0; j < 2; j++){
                    /* 2 回まわすことで、変数 x に x、y 両方の役割を負わせる */
     …
     r += x*x;                           /* r = r+x*x と同じ */
    }
```

まずこの部分から見ていく。ここで新たに出てきた記号 "+=" は、"r = r+x*x" 、すなわち「r+x*x を計算し、それを再び r に代入する」ことを意味する。このような使い方は四則演算の記号全てに可能である。

```
a += b --> a = a + b
a -= b --> a = a - b
a *= b --> a = a * b
a /= b --> a = a / b
a %= b --> a = a % b
```

そうしてみると、まず r を初期化してから for 文に入る。for 文内では、まず変数 x に座標 x の役を負わせ、r に x の自乗を足す。次に再び前に戻って、変数 x に座標 y の役を負わせて、r に x の自乗を足す。2 回まわったので、for 文の条件から外れて、for 文の外へ出る。こうして動径方向の自乗和が求まる。

```
     iseed = (iseed*IA+IC)%IM;          /* 乱数発生ルーチ
ン */
     x = (double)iseed/(double)IM;
```

この 2 行で疑似乱数を生成する。乱数発生法はここでは省略する[3]。ここで注目して欲しいのは

[3] この方法は線形合同法と呼ばれる。適当な (IA,IC,IM) の組を用いて、(iseed*IA+IC) を IM で割った時の余り

”x = (double)iseed/(double)IM;” という計算法である。変数 iseed, IM は int 型に対し、変数 x は double 型である。この場合、通常通り ”x = iseed/IM;” と計算すると、iseed と IM が int 型のため、iseed/IM を int 型の答えで一旦出した後に double 型に変換して x に代入される。そのため、x には 0 (まれに 1) しか入らない。そのためここではこれら int 型の変数の前に ”(double)” と付けて double 型とみなして計算し、計算結果も double 型で求める。そうすることで、x には 0 〜 1 の間の任意の実数が代入される。

```
      if(r <= 1.0){                /* r が 1 より小さいか同じなら s に 1 を加
える */
       n++;
    }
```

この部分は 「§3.4」 で挙げた例と同じである。

```
  pi = 4.0*(double)n/(double)NEVENT;      /* n:NEVENT = pi:4.0 */
```

ここで上の関係式を用いて π を導出する。

```
    printf("pi = %f \n",pi);                    /* 結果の表示 */
```

ここでは結果を画面に出力するために、標準関数 printf を用いている。標準関数とは OS であらかじめ用意している関数のことである。標準関数は種類ごとにパッケージにされていて (標準ライブラリと言う)、使用する標準関数に合わせてメイン関数の前に指定する必要がある。

printf の場合、標準ライブラリは stdio.h を指定すれば良い。指定の仕方は、ソースの一番はじめに

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

とおけば良い。

printf の一般的な書式は

printf("データの書式や出力させる文字列等　… \n", 出力される変数, …)

で表される。変数内の値を出力したい時は下の表のような記号をいれる。'\n' は改行を表す。それ以外は、そのまま出力される。

%f	実数
%d	整数
%s	文字列

この例では、画面には

pi = (値)

の様に出力される。

を求める。この余りを IM で割れば 0 から 1 までの任意の数になり、更にこの余りを iseed に代入して、再び同じ計算を繰り返していくことで、0 から 1 までの「ある一定周期内での」ランダムな数列を作ることができる。この周期は (IA,IC,IM) の組合せによって決まる。参考図書 (2) 第 7 章にその具体的な数値が示してある。

§3.7　コンパイルと結果

さて、それでは実際にプログラムをコンパイルして、実行してみよう。

```
c01:~/c_text> gcc -o ludolph ludolph.c
c01:~/c_text> ./ludolph
pi = 3.136000
c01:~/c_text>
```

この例では点の数を 10,000 個としたが、点の数を増やせば実際の値に近づく。点の数を変えるごとにどのように値が変わっていくか、また、その時の決定精度はどのくらいになるか、など興味のあるところだが、それらは読者の自習に任せる。

§4　内積

ここからは、いくつかのプログラムの例を見ながらさらなるCプログラミングの方法を説明していく。

- 配列
- 標準関数
- 関数

この章では内積を計算するプログラムを例に前処理、配列について見ていく。内容は、はじめに 2 つのベクトル vec1，vec2 に初期値を代入し、ベクトル vec3 に計算結果を代入し、最後に結果を出力する。

```
scapro.c ------------------------------------------------------------
#include<stdio.h>

#define L 1
#define M 3
#define N 1

int main(){
  int i,j,k, sum;
  int vec1[L][M]={1,2,3};
  int vec2[M][N]={5,6,7};
  int vec3[L][N];

  for(i = 0; i < L; i++){
    for(j = 0; j < N; j++){
      sum = 0.;
      for(k = 0; k < M; k++){
sum = sum+vec1[i][k]*vec2[k][j];
```

```
      }
      vec3[i][j] = sum;
    }
  }
  printf("vec1 = (%d, %d, %d) \n",vec1[0][0],vec1[0][1],vec1[0][2]);
  printf("vec2 = (%d, %d, %d) \n",vec2[0][0],vec2[1][0],vec2[2][0]);
  printf("vec1・vec2 = %d \n",vec3[0][0]);
}
```

実行結果 ----------

```
c01:~/c_text> gcc -o scapro scapro.c
c01:~/c_text> ./scapro
vec1 = (1, 2, 3)
vec2 = (5, 6, 7)
vec1・vec2 = 38
c01:~/c_text>
```

---

§4.1 前処理

```
#include<stdio.h>

#define L 1
#define M 3
#define N 1
```

この部分のようにメイン関数に先だって処理される部分を前処理系という。前処理系で処理された実行文は以降の全関数内で有効である。

#include 文は <> 内 (もしくは "" 内) に示されたファイルの中身をその場に置き換える。<> で指定されたファイルは標準ライブラリを示す。また、"" で指定されたファイルはソースファイルと同じディレクトリにあるユーザ定義のファイルを示す。例は 「§6」 に示している。

#define 文はマクロの定義を行う。マクロとは関数内で用いる定数、文字列などを別の名前で置き換えることである。書式は

#define マクロ名 置き換え要素

このようにすることで、頻繁に使う要素を使い易くしたり、プログラム全体の見通しを良くすることができる。ここでは、行列の行、列数を定義した。

§4.2 配列

```
  int vec1[L][M]={1,2,3};
  int vec2[M][N]={5,6,7};
  int vec3[L][N];
```

配列はあるまとまった数値の列を表現するのに用いる。この例のように行列だけではなく、複数個あるデータ列を表すのにも良く使われる。

定義は普通の変数の場合と同じように行う。配列の次元数は配列名の後の [] の数で決まる。[] 内の数値は配列の大きさを表す。この例では 1×3、及び 3×1、 1×1 の 2 次元の配列を定義している。使う時には 0 から (配列の大きさ)-1 で指定する。この例では、

```
vec1[0][0] 〜 vec1[0][2]
vec2[0][0] 〜 vec2[2][0]
vec3[0][0] 〜 vec2[0][0]
```

といった具合である。

初期値を代入する時は上の例のように "={ 数値 }" とする。また、2×2 以上の配列になると、後ろの列から順に代入されていく。

```
MAT[2][2] = {1,2,3,4};

--> MATa[0][0] = 1 , MATa[0][1] = 2
    MATa[1][0] = 3 , MATa[1][1] = 4
```

§4.3 プログラムの中身について

プログラムを見ていて「何でこんな難しい書き方をしているのだろう?」と思った読者も多いかと思う。もちろん内積の計算なんだから簡単に、

```
a = vec1[0][0]*vec2[0][0]+vec1[0][1]*vec2[1][0]+vec1[0][2]*vec2[2][0];
```

とでもすれば十分であると思うかも知れない。しかし、例えば 3×3 の行列の掛け算をしたくなった場合、もちろん一からソースを書くのも良いが、このプログラムを使えば、#define で定義した L,M,N の数値を書き換えるだけで十分なのである。これだけならまだその効果が分からないかも知れないが、後に出てくる関数と組み合わせると、このプログラムだけで、内積の計算や 3×3 の行列の掛け算など、少なくとも 2 次元の行列の掛け算は全て可能となる。この後で出てくる関数、関数への配列の受渡しについて学習した後に、このプログラムを関数にしてみて欲しい。

このように、一般的に使えるようなプログラムは関数としていつでも使えるようにしておくのが便利である。

§5 標準関数を使った例

この章では先に説明した標準関数をいくつか紹介する。

§5.1 標準入力からの入力

「標準入力」はキーボードのことである。余談ではあるが、「標準出力」とはディスプレイのことである。このようにわざわざ難しく言い替えるのは、より一般的な入出力 (主に他のファイル) に対して使うためである。

プログラムを実行した時に、値をプログラム内で代入するのではなく、標準入力から直接与えるための関数 scanf を用いる。scanf は標準ライブラリ stdio.h で定義されている。先の例 ludolph.c

において、疑似乱数の種 iseed を直接与えられるようにしてみよう。先のプログラムで、変更点のみを以下に示す。

ludolph.c ----------

```
  …
  int iseed;                                  /* 乱数のたね <-- 変更する。*/
  …
  double x,r, pi;                             /* x(y) 座標，動径の大きさ，円周率 */

  /* 「種」の値を決める <-- 以下を挿入する。*/
  printf("乱数の「種」を代入してください (整数型，1 〜 %d-1) : ",IM);
  scanf("%d",&iseed);
  while((iseed >= IM) || (iseed <= 0)){
    printf("値が不適当です。代入しなおしてください (整数型，1 〜 %d-1) : ",IM);
    scanf("%d",&iseed);
  }
```

関数 scanf は次のように書く。

```
scanf("データ書式 データ書式，…",&変数名，&変数名，…);
```

データ書式は関数 printf で説明したものと同じである。変数名の前についている ”&” は今のところおまじないだと思って良い。

今の例では int 型の変数 iseed に代入するようにした。ただし、iseed の範囲は 1 〜 IM-1 なので (理由は乱数の発生方法を見れば一目瞭然である)、それ以外の値を代入した時には再度問い合わせるようにする。その部分は while 文を使って、値が適切なものになるまで繰り返すようにした。

実行結果 ----------

```
c01:~/c_text> gcc -o ludolph ludolph.c
c01:~/c_text> ./ludolph
乱数の「種」を代入してください (整数型，1 〜 259200-1) : -50
値が不適当です。代入しなおしてください (整数型，1 〜 259200-1) : 54321
pi = 3.151200
c01:~/c_text>
```

---

§5.2 算術関数

標準ライブラリ math.h にはいくつかの算術関数が定義されている。算術関数には以下のようなものがある。

sqrt(x)	$\sqrt{x}$
log(x)	$\ln x = \log_e x$
log10(x)	$\log x = \log_{10} x$
exp(x)	e^x
sin(x)	$\sin x$
cos(x)	$\cos x$
tan(x)	$\tan x$
asin(x)	$\arcsin x = \sin^{-1} x$
acos(x)	$\arccos x = \cos^{-1} x$
atan(x)	$\arctan x = \tan^{-1} x$
sinh(x)	$\sinh x$
cosh(x)	$\cosh x$
tanh(x)	$\tanh x$
fabs(x)	$\|x\|$
paw(x,y)	x^y

ここで注意することは、変数 x,(y) は double 型の変数を用いること、関数の返す値も double 型であることである。また、標準ライブラリ math.h 使う時はコンパイル時にオプション “-lm” をつける必要がある。以下に使用例を示す。[4]

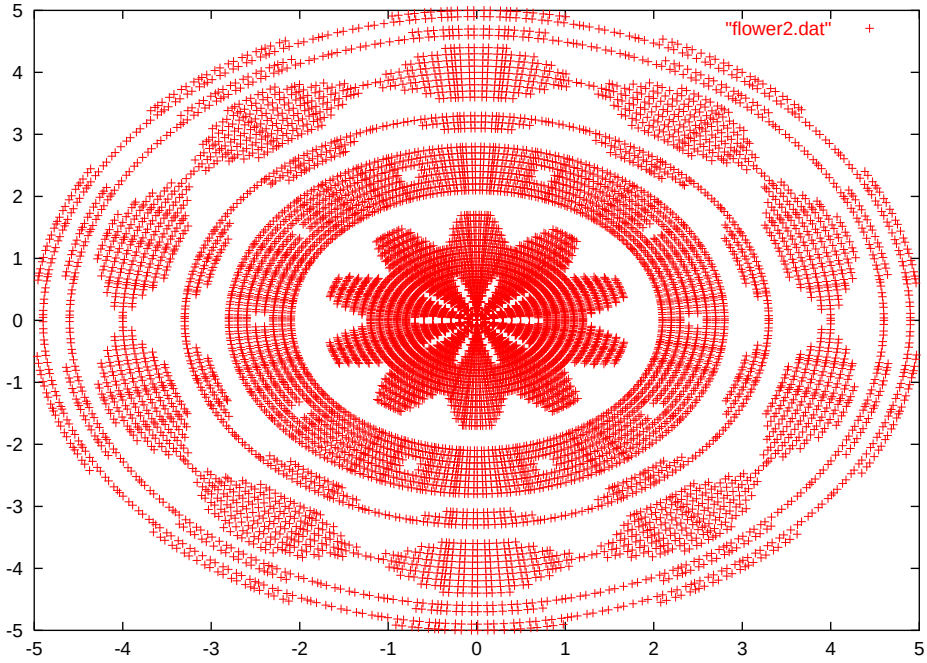
flower.o ----------

```
#include<math.h>                          /* 算術関数を定義した標準ライブラリ */

int main(void){
  double PI = 3.1415926;
  double r,t, x,y, a,b;
  int i;

  for(r = 0.;r < 5.; r+=0.05){           /* 動径成分を for 文で回す */
    for(i = 0; i < 360; i+=1){           /* 角度成分を for 文で回す */
      t = (double)i*180./PI;             /* 度 --> radian 変換 */
      a = cos(10.*t);                    /* cos 関数 */
      b = tan(r*sin(2.*r));              /* sin,tan 関数 */
      if(a < b){                         /* a < b の時にだけ値を出力する */
x = r*cos(t);                  /* (r,t) --> (x,y) に変換 */
y = r*sin(t);
printf("%f %f \n",x,y);
      }
    }
  }
}
```

実行結果 ----------

[4] Mac ユーザなら一目瞭然かも知れない

```
c01:~/c_text> gcc -o flower flower.c -lm
c01:~/c_text> ./flower > out.dat
c01:~/c_text> gnuplot


gnuplot> plot"out.dat"
```



§6　千鳥足

確率・統計の教科書でおなじみの「千鳥足」のシミュレートを行う。ここでは主に関数の説明を行う。

ルールは 0 〜 1 までの疑似乱数を振って

0 以上 0.25 未満	x 方向に -1
0.25 以上 0.5 未満	y 方向に -1
0.5 以上 0.75 未満	x 方向に +1
0.75 以上 1 以下	y 方向に +1

の様に進むことにする[5]。そして、自分の家 ($(x, y) = (10, 10)$) に到着した時点で終了とし、それまでの経路、および歩数をファイルに出力する。ただし、ここでは簡単のため、シェルのリダイレクト ”>” を用いて出力ファイル `out.dat` に出力する。

プログラムはそんなに複雑なものではないが、ここでは

[5] このルールは (1 次元の問題を除けば) 一番簡単なルールである。例えば (r, θ) を乱数で振るようにすれば、ブラウン運動のシミュレートなども可能である (参考図書 (5))。

- random_walk.c – メイン関数。
- ran.c – 疑似乱数を発生させる関数。
- random_walk.h – 定義ファイル。

の 3 つに分けた。「何でそんなことを…」と思うかも知れないが、機能ごとにファイルを分けることはプログラムの見通しを良くする上で大切である。以下にソースと結果を示す。

random_walk.c ----------

```
#include"random_walk.h"

int main(){
  int x = 0, y = 0, i = 0;
  double Prw;

  do{
    printf("(x, y) = (%d, %d) \n",x,y);
    Prw = ran();
    if(Prw < 0.25) x--;
    else if((Prw >= 0.25) && (Prw < 0.5)) y--;
    else if((Prw >= 0.5) && (Prw < 0.75)) x++;
    else y++;

    i++;
  }while((x != XHOME) || (y != YHOME));
  printf("(x, y) = (%d, %d) \n",x,y);
  printf("無事に家に着きました。%d 歩 \n", i);
}
```

ran.c ----------

```
#define IA 7141
#define IC 54773
#define IM 259200

double ran(){
  static long iseed=12354;

  iseed = (iseed*IA+IC)%IM;
  return (double)iseed/(double)IM;
}
```

random_walk.h ----------

```
#define XHOME 10
#define YHOME 10

double ran();
```

実行結果 ----------

```
c01:~/c_text> ./random_walk > out.dat
c01:~/c_text> cat out.dat

0, 0
0, -1
0, 0

…

9, 10
10, 10
無事に家に着きました。9922 歩
c01:~/c_text> gnuplot


gnuplot> plot"out.dat" u 1:2
```

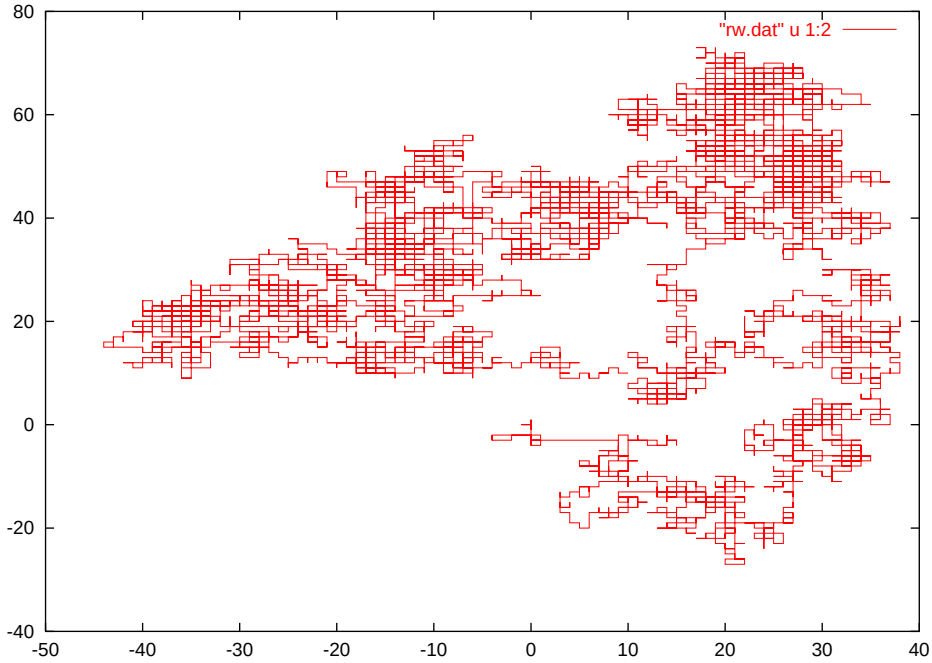
§6.1 繰り返し文 do while

```
do{
  Prw = ran();
  …
  printf("(x, y) = (%d, %d) \n",x,y);
}while((x != XHOME) || (y != YHOME));
```

ここでは、

1. 現在の座標を出力する。
2. 乱数を振って進む方向を決める。
3. 歩数を加算する。
4. 進んだ結果、家に着かなければ 1 にもどる。

を家に着くまで繰り返す。do while 文の一般的な書式は




```
do{
  実行文;
}while(繰り返し条件);
```

となっている。特徴は while 文と逆で実行文を行った後にその判定をするところである。

§6.2 関数

```
Prw = ran();
```

ここでは 変数 Prw に関数 ran() を呼び出し、その出力を代入している。関数は通常

1. 前処理系での関数プロトタイプ宣言
2. 関数の定義
3. 関数の呼び出し

から成る。

1, 前処理系での関数プロトタイプ宣言

```
double ran();
```

関数プロトタイプ宣言では用いる関数について関数の型や、名前、必要な引数を定義する。引数とは呼び出された関数から「引き渡される値」である (呼び出した関数が「引き渡す値」とも言えるが)。

型 関数名 (型 引数, 型 引数, …)

関数の型は変数の定義で説明した基本型に加え void 型 (値を返さない) がある。引数の型も同様である。この例では倍精度実数型の関数 ran を定義している。引数はない。また、前処理系で直接関数を定義する場合は関数プロトタイプ宣言を行う必要はない。

2, 関数の定義

```
double ran(){
  static long iseed=12354;

  iseed = (iseed*IA+IC)%IM;
  return (double)iseed/(double)IM;
}
```

一般的な関数の書式は

```
型 関数名 (型 仮引数, 型 仮引数, …){
   文;
   return 返却値;
}
```

で書かれる。型を省略すると int 型 になる。

仮引数の名前は元の関数の引数と同じである必要はない。関数内で定義された変数の値は、一度関数から抜けると消える。ただし、この例では型の前に"static" と置くことで、関数から抜けた後もその値を保持し続ける (long は扱える桁の範囲の広がった int 型と覚えておけば良い)。

引数は基本的に値のみを関数に渡す。すなわち、関数側は値をいれる格納庫 (仮引数) を用意していて、本体からは値のみが格納される。関数での出力は return 文の返却値を通してのみしか返されない。そのため、仮引数の変更は元の関数の引数には反映されない。仮引数の変更を元の関数の引数に反映させるにはポインタを用いる必要があるが、ポインタについては次章で述べる。

§7　ポインタ、及び構造体

関数：

もう一度関数についておさらいをすると、C 言語はループや判別などの制御構造と、様々な機能を一つにまとめた関数というものから成り立っている。関数には入力と出力の機能があり、値の受け渡しができるようになっている。ここで簡単な関数を書いてみる。受け渡された値を二倍して返すというものである。

```
kannsuu.c-----------------------------------------------------------
#include <stdio.h>

int nibai(int c){     //与えられた数を二倍する。戻り値は二倍した値
  c=c*2;
  return(c);
}
```

```
int main(void){
  int a,b;
  a=7;
  b=nibai(a);
  printf("a=%d, b=%d \n", a, b);
}
```
実行結果 ----------
```
~/> ./a
a=7, b=14
```

ここで気をつけたいのは、関数に渡した値は関数の中でどんなに値を変更しても戻ってきたときにはその変更は反映されていないということである。先の例でいうと、関数 nibai() の中で、main() 関数から渡された値 7(これを引数という) が二倍にされているが、main() 関数の b=nibai(a); の次の行でも a の値は 7 のままである。ではどうやって受け取るか？二倍した結果は return(c); で戻され、b に代入される。(これを戻り値という) ただし、戻り値は一つしかあり得ないので複数の結果を得たいときには次のポインターという概念を用いる。

ポインター：

pointer.c----------
```
#include <stdio.h>

int irekae(int *a,int *b){  //値を入れ替える関数。
  int c;
  c  = *a;
  *a = *b;
  *b = c;
  return 0;                 //戻り値には特に意味はない
}

int main(void){
  int *p, *m;
  int i=5, j=8;
  p=&i;  m=&j;
  printf("始め    %d, %d \n", i, j);
  irekae(p, m);
  printf("終わり  %d, %d \n", i, j);
}
```
実行結果 ----------
```
~/> ./a
始め    5, 8
終わり  8, 5
```

ポインターとはアドレス (メモリ上の番地) を扱うための変数で、ポインターの型宣言は”int *p;”の

ようにする。普通の変数は別なところにあり (上の例では i=5)、そのアドレス (&i で取得できる, 例えば 1000) をポインターに代入 (p=&i) する (これで p の値は 1000 になる, 更に*p とすればアドレス 1000 の所にある値 5 を得る)。そして徐にポインターを関数に渡せば、関数での中でいじった結果を受け取ることができるようになる。つまり、変数の値を渡すのではなく、変数がどこにあるのかを渡すのである。

配列 (関数への受渡し)：

　配列の考え方は先に述べたとおりである。ただし関数に配列を渡すには、配列を全部渡すのではなく、配列の名前と配列の大きさを渡す。

```
hairetu.c-----------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>

int saidai(int b[], int n){
  int i, j=0;
  for(i=0; i<n; ++i){
    if(j<b[i]){
      j=b[i];
    }
  }
  return j;
}

int main(void){
  int i=5, j;
  int a[i];
  a[0]=3; a[1]=5; a[2]=1; a[3]=7; a[4]=2;
  j = saidai(a,i);
  printf("最大値 %d \n", j);
}
実行結果 ------------------------------------------------------------
~/> ./a
最大値 7
--------------------------------------------------------------------
```

配列とポインター：

　じつは配列の名前というのは配列の先頭アドレスを示すので、配列の中身をポインターを使って扱うことができる。どのようにするのかというと、ポインターというのは単なるアドレスの入れ物ではなくアドレスの演算 (足したり引いたり) を行うことができるので、メモリの中を自由に行き来きして値を読み書きするのである。先ほどの関数”saidai()”をポインターを用いて書き換えてみよう。

```
hai_poin.c----------------------------------------------------------
#include <stdio.h>
```

```
#include <stdlib.h>

int saidai(int *b, int n){
  int i, j=0;
  for(i=0; i<n; ++i){
    if(j<*(b+i)){  //*(b+i) とは、
      j=*(b+i);    //配列の先頭アドレス b から i だけ進んだところの値
    }
  }
  return j;
}

int main(void){
  int i=5, j;
  int a[i];
  a[0]=3; a[1]=5; a[2]=1; a[3]=7; a[4]=2;
  j = saidai(a,i);
  printf("最大値 %d \n", j);
}
```

実行結果 ----------

```
~/> ./a
最大値 7
```

構造体：

座標や複素数など複数の変数が組になったものを一貫して扱うには構造体を用いるとよい。配列を使ってある点の x 座標,y 座標を (double 型の)a[0],a[1] に入れて、別な点の x 座標,y 座標を b[0],b[1] に入れるとしてもいいが、その地点の名前 (char) や他の情報も一纏めにしておきたい場合もある。ここでは、ある点の名前とその x 座標,y 座標を一纏めにしたような新たな型をまず定義して、そのような型を持つ変数 (構造体変数) をつくる。大まかにはこの構造体変数を用いて行うが、実際に値が入っているのは構造体の中の変数 (メンバ) である。構造体のメンバへのアクセスは (構造体変数).(メンバの名前) によって行う。

kouzou.c----------

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

typedef struct{    //新たに構造体の型を定義する
                   //この構造体には以下のような変数 (メンバ) が入ってます
  double x_axis;   //x 座標の値を入れる変数
  double y_axis;   //y 座標の値を入れる変数
  char   name;     //地点の名前を入れる変数
}ZAHYOU;           //新たに作った構造体の型の名前
```

```
ZAHYOU kaiten(ZAHYOU a, int psi){  //座標回転の関数
  ZAHYOU b;
  double theta;
  theta    = psi*(2*M_PI/360);  //M_PI=3.14159265358979323846
  b.x_axis = a.x_axis*cos(theta)-a.y_axis*sin(theta);
  b.y_axis = a.x_axis*sin(theta)+a.y_axis*cos(theta);
  b.name   = a.name +1;  //アルファベットを一つずつ進む (A,B,C,…)
  return b;
}

int main(){
  ZAHYOU sitten1, sitten2, sitten3;   //三つの ZAHYOU 型構造体変数を作る
  int phi=45;                 //回転させる角度は 45 °
  sitten1.x_axis=1;           //構造体のメンバに値を代入
  sitten1.y_axis=0;
  sitten1.name  ='A';         //はじめの地点は"A"
  sitten2 = kaiten(sitten1, phi);
  sitten3 = kaiten(sitten2, phi);
  printf("%c 地点 x=%g y=%g \n", sitten1.name, sitten1.x_axis, sitten1.y_axis);
  printf("%c 地点 x=%g y=%g \n", sitten2.name, sitten2.x_axis, sitten2.y_axis);
  printf("%c 地点 x=%g y=%g \n", sitten3.name, sitten3.x_axis, sitten3.y_axis);
}
実行結果 ------------------------------------------------------------
~/> ./a
A 地点 x=1 y=0
B 地点 x=0.707107 y=0.707107
C 地点 x=5.77609e-17 y=1
-------------------------------------------------------------------
```

ファイルの読み込み：

ファイルを読み込むためにはまずファイルをオープンする必要がある。そのための関数が fopen で、ファイル構造体へのポインタを返してくるので、以降ファイルをいじる場合にはこのポインタを使う。たとえば一行単位の読み込みの場合には、fgets 関数の引数としてデータを格納する領域とその大きさ、そしてファイルポインタを渡す。ファイルを閉じる場合には fclose 関数を用いる。

```
file.c-----------------------------------------------------------------
#include <stdio.h>

int main(void){
  FILE *fp;
  int nbyte=128;
  char buff[128];
  char fname[15]="sample.txt";      //オープンするファイルの名前
  if((fp=fopen(fname,"r"))==NULL){//ファイルのオープン "r"は読み込み用
```

```
                                        //"w"なら書き込み用
    printf("Cannot open \"%s\"\n",fname);
    return;
  }
  while(1){   //ここで無限ループに入る
    if(fgets(buff, nbyte, fp) == NULL) break;
      //fgets はファイルポインタ fp のある位置から
      //改行文字を読み込むか nbyte だけ読み込み、
      //buff に格納する
      //fp がファイルの終端を読み込むと fgets は NULL を返すので
      //break でループから抜け出す
    printf("%s", buff);
  }
  fclose(fp);
}
```

実行結果 ----------

```
~/> cat sample.txt
hogehoge
aiueo
~/> ./a
hogehoge
aiueo
```

---

§8 リンクと make

ここでは、これらのいくつかのプログラムをコンパイルして実行ファイルを作成する方法を説明する。

「§6」のプログラムは 2 つのファイルから成っている。そのため、これらのファイルをそれぞれコンパイルしてオブジェクトファイルを作り、リンクするという作業を行わなければいけない。これまで「コンパイル」と言ってきた操作は、

- オブジェクトファイルの作成
- オブジェクトファイルのリンク

の 2 つの操作を指してきたが、厳密には前者の操作を「コンパイル」と言う。標準ライブラリを使う場合ももちろん同様の操作をしなければいけないが、通常はプログラマが手を煩わせる事は無いはずである (math.h の場合は別だったが)。コンパイルとリンクの方法はいくつかあるが、ここでは 2 つの方法を紹介する。

1 つは文字通り 1 つずつコンパイルしてオブジェクトファイルを作り、リンクする方法である。すなわち、

```
c01:~/c_text> gcc -c ran.c -o ran.o
c01:~/c_text> gcc -o random_walk random_walk.c ran.o
```

である。もちろん、この 2 つを毎回行うのはある意味確実な方法だが、作業効率が悪い。ある人はこれを shell script に書いてしまえば良いと考えるかも知れない。それも悪くは無いが、根本的には何も解決していない。ここで言う「作業効率が良い」とは、どれだけシステムに無駄な作業をさせないかという事を意味している。

そこであげるもう 1 つの方法が、make を使う方法である。詳しい解説は参考図書 (4) に譲るとして、ここでは最も初歩的な使い方について説明する。make とはCの様なコンパイル言語のコンパイル、リンクの作業を支援するツールである。その大きな特徴は必要となるファイル (コンポーネント) と出来上がるファイル (ターゲット) との時間関係、階層構造をチェックし、必要最小限のコンパイルなどの操作を行う点である。

make ではあらかじめ Makefile というファイルを用意する。以下に今回のプログラムに用いられる Makefile の例を示す。

```
Makefile ------------------------------------------------------------

random_walk: random_walk.c ran.o
(tab)gcc -o random_walk random_walk.c ran.o

ran.o : ran.c

(tab)gcc -c ran.c -o ran.o

clean:
(tab)rm *.o *~

----------------------------------------------------------------------
```

ここでは

ターゲット : コンポーネント コンポーネント …
(tab) 実行コマンド

を 1 つの単位とする。”(tab)” は「タブキー」をいれることを表す。実行コマンドの前にはタブをいれなければいけない。make を実行する時はコマンドプロンプトに ’make ターゲット’ と叩く。そうすると、ターゲットと各コンポーネントとの時間関係を調べ、ターゲットに対してコンポーネントが更新されていれば実行コマンドを実行する。そうでなければ、何もせずに終了する。また、コンポーネントが下の階層でターゲットとして指定されていれば、そちらから参照して、同様の操作を行い、上の階層に進んで行く。

ここの例ではコマンドプロンプトから ’make random_walk’ と叩けばよい。例えば、全てのオブジェクトファイル、実行ファイルが無い場合は、

```
c01:~/c_text> make random_walk
gcc -c ran.c -o ran.o
gcc -o random_walk random_walk.c ran.o
c01:~/c_text>
```

となる。’random_walk.c’ のみを更新した場合は、

```
c01:~/c_text> make random_walk
gcc -o random_walk random_walk.c ran.o
c01:~/c_text>
```

の様になる。

また、上の Makefile にあるように、ターゲットとして'clean' を定義しておくと、コマンドプロンプトで 'make clean' と打てば、いらなくなったオブジェクトファイルや " " の付いたファイルを消去してくれるので便利である (rm * で全てのファイルを消去してしまう危険性が格段に減る!!)。

参考図書

(1) はじめての " C"(椋田實 技術評論社)
(2) NUMERICAL RECIPES in C[日本語版](William H. Oress, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery(丹慶勝市・奥村晴彦・佐藤俊郎・小林誠 訳) 技術評論社)
(3) 計算物理 (早野龍五・高橋忠幸 共立出版)
(4) make(Andrew Oram, Steve Talbott(菊池彰 訳) オライリージャパン)
(5) 理工系数学のキーポイント● 6 キーポイント 確率・統計 (和達三樹・十河清 岩波書店)