C 言語のサンプルプログラム (5)

計算機応用　5E 2003.05.29

1．コンソール入力

コンソール(ディスプレイやキーボート)の入出力は、いろいろあります。ここでは、数値計算法を学ぶために必要なことのみ述べます。これ以上の詳細は、C 言語の解説書を読んでください。

コンソール入出力のことを、標準入出力と言う場合があります。また、後で(P.11)述べますが、ファイルの入出力先を指定するために、この標準入出力には、名前が付いています。C 言語では、これらの名前を用いて、入出力先の指定を行います。これらの関係を、以下に示します。

キーボード	→	標準入力	→	stdin
ディスプレイ	→	標準出力	→	stdout

1.1 書式付入力(scanf)

入力は、少し癖は有りますが、文字や数字が同じように入力できる scanf 関数を用います。この関数を使う上での注意は、

- データを入れる引数は、ポインターである。
- %s の書式(文字列)で文字列を読み込む場合、空白は読み込みません。空白の前まで、読み込みは終わりです。
- %c の書式(1 文字)であれば、空白を読み込みます。
- 復改文字(\n)がバッファーに残ります。これを、読み捨てないと、次の入力でこれが読み込まれます。

等です。

この書式付入力 scanf 関数の定義は、

```
int scanf(書式指定, 引数並び)
```

です。戻り値は、正常時は入力した項目数ですが、異常時は EOF を返します。

入力の文字や数値の書誌指定は、

```
% [代入抑止文字] [最大フィールド幅] [変換修飾子] 変換指定子
```

です。[ ]内は省略可能です。それぞれについては、表 1～4 のようになります。

1 個の scanf 関数で複数のデータを入力することが可能です。そのときのデーターの区切りは、空白、Tab、改行で行われます。特別な文字で、データを区切りたいときには、書式に書きます。printf と同じです。

いろいろありますが、数値計算で使うのは、主に、

%lf	倍精度実数型(double)入力
%d	10 進数整数入力
%s	文字列入力
%c	指定文字数入力。1 文字のとき、よくつかう。

です。

表 1 代入抑止文字

*	代入を抑止する。データが読み捨てられる。

表 2 最大フィールド幅

整数	指定数の幅(column)で入力データを区切る

表 3 変換修飾子

h	short 修飾 d,i,n 変換 : short int 型 o,u,x 変換 : unsigned short int 型
l	long 修飾 d,i,n 変換 : long int 型 o,u,x,X 変換 : unsigned long int 型 e,E,f,g,G 変換 : double 型 その他 : 未定義
L	long double 修飾 e,E,f,g,G 変換 : double 型 その他 : 未定義

表 4 変換指定子

d	有符号 10 進(Decimal)整数変換
i	有符号 8,10,16 進整数（integer）変換 • 数値の入力は以下の通り。いずれの場合も同じ数値。 8 進数 07777 8 進数を表す接頭語 0 で始まる 10 進数 4095 通常 16 進数 0xfff 16 進数を表す接頭語 0x（or 0X）で始まる • 負の場合は、先頭に-をつける。
o	有符号 8 進（Octal）整数変換 • 8 進数を表す接頭語は、つけてもつけなくても良い。
u	無符号(Unsigned)10 進整数変換
x, X	有符号 16 進(heXadecimal)整数変換 • 16 進数を表す接頭語は、つけてもつけたくても良い。
e,E,f g,G	有符号 10 進浮動小数点変換(Floating-point)数変換
s	非空白文字列(String)変換 • 空白以外の文字からなる文字列を読み、指定された配列に入れる。 • 最後に文字列区切り記号の\0 を付加する。
c	文字(Character)変換 • 最大フィールド幅(デフォルト 1)で指定された個数の文字列を読み込み、配列に入れる。空白文字も読み込む。 • 文字列区切り記号の\0 を付加されない。
p	ポインター(Pointer)変換 • ポインターとして入力する
n	入力バイト数変換
%	%変換 • 入力された%を読み捨てる。代入と変換は行われない。

1.2 書式付入力サンプルプログラム

サンプルプログラム(char.c)

1 文字入力のプログラムです。読み込んだ文字を出力します。temp には、復改文字(\n)が代入されます。この復改文字は、通常のプログラムでは不要なので、読み捨てます。もし読み捨てないと、バッファーに残り、次の読み込みで不都合が生じます。

```
#include <stdio.h>

main(){
   char a, temp;

   scanf("%c%c", &a, &temp);

   printf("%c\n",a);

}
```

サンプルプログラム(string.c)

最大 256 文字までの文字列を読み込みます。そして、読み込んだ文字を出力します。

```
#include <stdio.h>

main(){
   char a[256], temp;

   scanf("%s%c", a, &temp);

   printf("%s\n",a);
}
```

サンプルプログラム(string2.c)

最大 256 文字までの文字列を、2 文字列読み込みます。そして、読み込んだ文字を出力します。

```
#include <stdio.h>

main(){
   char a[256], b[256];
   char temp;

   scanf("%s%s%c", &a, &b, &temp);

   printf("%s\n",a);
   printf("%s\n",b);
}
```

サンプルプログラム(double2.c)

double 型で、2 個の数値を読み込みます。そして、読み込んだ数値を出力します。

```
#include <stdio.h>

main(){
   double a, b;
   char temp;

   scanf("%lf%lf%c",&a,&b,&temp);

   printf("%f\n",a);
   printf("%f\n",b);
}
```

サンプルプログラム(integer.c)

8, 10, 16 進数の整数を読み込みます。そして、読み込んだ整数を 10 進数で出力します。

```
#include <stdio.h>

main(){
   int a;
   char temp;

   scanf("%i%c", &a, &temp);

   printf("%d\n",a);
}
```

実行結果

8 進数の 7777 と, 10 進数の 4095, 16 進数の fff が同じであることが分かります。

8 進数の場合	10 進数の場合	16 進数の場合
07777	4095	0xfff
4095	4095	4095

2．コンソール出力

2.1 書式付出力(printf)

ディスプレイへの出力は、今までおなじみの printf を使います。この書式付出力 printf 関数の定義は、

```
int printf(書式指定、引数並び)
```

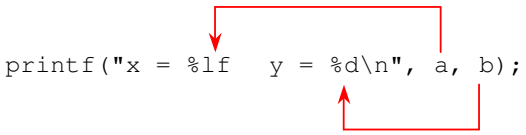
です。戻り値は、正常時は出力した文字数ですが、異常時は負の数を返します。

出力の文字や数値の書誌指定は、

```
% [フラグ] [最小フィールド幅] [精度] [変換修飾子] 変換指定子
```

です。[ ]内は省略可能ですそれぞれについては、表 5～9 のようになります。メモリーの内容を変換指定子で定められた書式に、変換し、文字出力します。

書式指定中の書式指定と変数との対応は、以下の通りです。いままで、さんざん練習してきましたので、わかっていますよね。書式指定と変数は、1 対 1 で順番通りに対応しています。



いろいろありますが、数値計算で使うのは、主に、

```
%f    浮動小数点出力  桁数が不足する場合があるの%e の方が望ましい。
%e    指数出力
%d    10 進数整数出力
%s    文字列出力
%c    文字出力。1 文字のとき、よくつかう。
```

です。

表 5 フラグ

-	左詰めフラグ：変換結果をフィールド内に左詰めする。 ・このフラグがない場合は、右詰めになる。
+	符号フラグ：正の数の場合でも、符号'+'をつける。
空白	符号フラグ：正の数の場合、符号'+'ではなく、空白を出力する ・負の数との位置をそろえるときに便利です。
0	0 フラグ　　ゼロ詰めをする。 ・数値出力のとき、ゼロパディング(先行 0 埋め)を行う。
#	表記フラグ：表記法がわかる形式に変換する。 o 変換　　：8 進数データに、接頭語 0 をつける。 x,X 変換　　：16 進数データに、接頭語 0x または 0X をつける。 e,E,F 変換　：出力に必ず小数点をつける。 g,E 変換　　：出力に必ず小数点をつけ、さらに続く 0 を省略しない。

表 6 最小フィールド幅

数値	出力の最小幅を指定する。 ・変換データの幅が、これよりも小さい場合は、フラグに従う。 ・変換データの幅が、これよりも大きい場合は、データサイズで出力される。
*	対応する int 実引数で最小フィールド幅を指定する。 ・負の場合は、-フラグ、絶対値はフィールド幅になる。

表 7 精度

.数値	数値により、小数点以下の桁数または最大表示文字数を指定する。 d,i,o,u,x,X 変換　：少なくとも、指定の数の数値を書く。不足分は先行ゼロ埋めで補う。 e,E,f 変換　　：小数点以下のの桁数。 g,E 変換　　：最大有効桁数。 s 変換　　：最大表示文字数 その他　　：未定義
*	対応する int 実引数で.数値の数値を指定する。

表 8 変換修飾子

h	short 修飾 d,i,o,u,x,X 変換　：short int、または unsigned short int 型 p 変換　　：short int 型のポインター その他　　：未定義
l	long 修飾 d,i,o,u,x,X 変換　：long int、または unsigned long int 型 その他　　：未定義
L	long double 修飾 e,E,f,g,G 変換　：double 型 その他　　：未定義

表 9 変換指定子

o	8 進（Octal）整数変換
d,i	10 進（Decimal）整数(Integer)変換
x,X	16 進(heXadecimal)整数変換 x ：16 進数を小文字(a～f)出力する。 X ：16 進数を大文字(A～F)出力する。
u	無符号(Unsigned)10 進整数変換変換
c	文字(Character)変換
s	文字列(String)変換
f	浮動小数点(Floating point)変換 • 引数は、float 型でも double 型でも良い。 • 標準の精度は、6 桁
e,E	指数(Exponet)変換 • 標準の精度は、6 桁
g,G	実数変換 • 通常は f 変換 • 指数部が−5 以下か、有効精度異常の場合には e または E 変換 • この変換の意図は、出力をなるべく短く表現することにある。
p	ポインター(Pointer)変換 • 実際の出力は、処理系に依存する。
n	変数変換 • %n がくるまでに出力さた文字の数を対応する int *型引数に書き込む。 • 文字は、なにも出力されない。
%	%変換 • %を書くときに使う。"%%"で%を出力する。

2.2 書式付出力サンプルプログラム

書式付出力の printf 関数は、さんざん使ってきたので、必要なことはほとんど説明済みです。詳細な出力指定をしたいときは、表 5〜9 を見て、自分で考えてください。

ここでは、少し変わったサンプルプログラムを示します。三角関数のグラフを書きます。

サンプルプログラム(sin.c)

最小フィールド幅*を使って、三角関数のグラフを書きます。よく FORTRAN の教科書にあるやつです。数学関数を使っているので、コンパイルにはオプション(-lm)が必要です。

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

main(){
   int i, iy, n, column;
   double x;
   double pi=4*atan(1);

   n = 25;

   for(i=0; i <= n; i++){
      x = 2*pi/n*i;
      iy = 30*sin(x)+40;

      printf("%*c\n",iy,'*');
   }
}
```

サンプルプログラム(outnum.c)

整数と実数をいろいろな形式で出力します。

```
#include <stdio.h>
main()
{
  char tmp;
  int i;
  double x;

  printf("write integer number  = ");
  scanf("%d%c%", &i, &tmp);

  printf("write real number     = ");
  scanf("%lf%c%", &x, &tmp);

  printf("\n");

  printf("decimal      : %d\n",i);
  printf("octal        : %o\n",i);
  printf("hexadecimal  : %x\n",i);

  printf("\n");

  printf("float        : %f\n",x);
  printf("exponent     : %e\n",x);
  printf("g type       : %g\n",x);
}
```

実行結果

いろいろな数字を入力して、出力を試して見ましょう。

```
write integer number = 123456
write real number    = 123.456e-7

decimal       :  123456
octal         :  361100
hexadecimal   :  1e240

float         :  0.000012
exponent      :  1.234560e-05
g type        :  1.23456e-05
```

3．ファイル処理関数

ここでは、数値計算に必要なハードディスクへのデータの書き込みと、そこからの読み込みについて述べます。それ以外のデバイスもほとんど同じです。各自、必要になったら学習すればよいでしょう。

数値計算では、大量の計算結果をファイルに書き込んだり、ファイルから計算すべきデータを読み込んだりします。また、他のプログラムとのデータの受け渡しに、ファイルが使われます。例えば、計算結果をグラフにする場合、一度ファイルにデータを書き込んで、グラフを書くソフトウェアーに渡します。そうすると、グラフを書くソフトウェアーを書く手間が省けます。

ファイルへのアクセスの方法は、順番通りアクセスするシーケンシャルアクセスと、いろいろなところからアクセスするランダムアクセスがあります。数値計算で、ランダムアクセスを使うことは希なので、ここでは取り扱わないこととします。

3.1 ファイルのオープンとクローズ

ほとんどのプログラム言語では、ファイルの処理は、図 1 のようになっています。これは、約束事なので覚えてください。当然、複数個のファイルをオープンすることもできます。

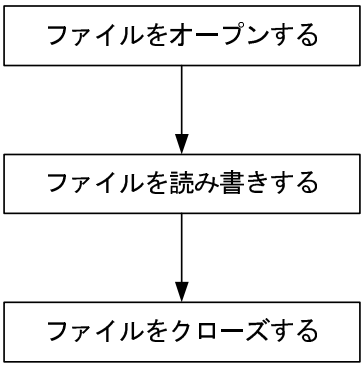


図 1 ファイル処理の流れ

これら 3 個の処理のうち、簡単なオープンとクローズについて、ここで説明します。ファイルの読み書きは、これらより少し複雑なので、次節以降で説明します。

サンプルプログラム(1)でファイルの読み書きについて、簡単に説明しましたが、ここではもう少し詳しく述べます。

C 言語では、かなり細かいファイルの処理が出来ます。そのために、ファイルの情報は、ファイルポインターと言われる構造体に記述されます。これは、stdio.h というヘッダーファイルに定義されています。すべてこのファイルポインターを使って、ファイル関係の処理は実施されます。名前の通り、これはポインターです。この値は、fopen という関数の戻り値です。

それでは、例を図 2 示して、ファイルのオープンとクローズを示します。このプログラムは、

- ファイルポインター（図 2 のプログラムの変数 fp)を定義しています。ポインターだから、*fp と変数の前にアスタリスクがついています。ファイルポインターの型は、FILE とすべて大文字で書きます。
- 次に、fopen 関数でファイルをオープンしています。この関数の戻り値が、ポインターです。
- 最後に、fclose 関数でファイルをクローズしています。

となっています。

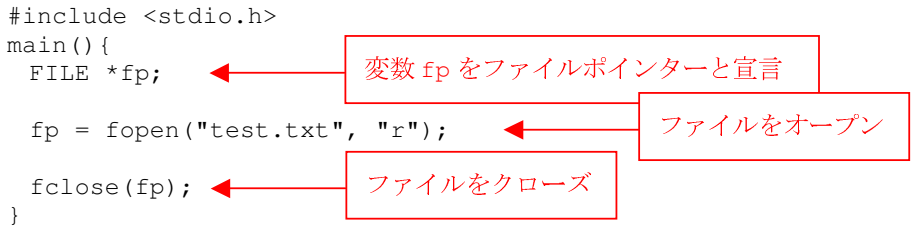


図 2 ファイルのオープンとクローズ

ファイルをオープンする関数 fopen の書式は、次の通りです。

```
FILE *open(char *filename, char *openmode)
```

戻り値は FILE 型のポインター、ファイル名を表す第一引数は char 型のポインター、オープンモードを表す第 2 引数は char 型のポインターと言う意味です。もし、オープンに失敗すると、NULL という戻り値になります。

オープンモードについては、表 10 にまとめておきます。あと、バイナリーモードというものがありますが、余り関係ないので説明は省略します。

こう言うと分かりにくいのですが、実際は、図 2 に示したように書きます。これを、真似ればよいでしょう。

ファイルをクローズする関数 fclose の書式は、簡単で、次の通りです。

```
int fclose(FILE *filepointer)
```

戻り値は、int 型で、クローズに成功すると 0、失敗すると EOF が返されます。引数は、ファイルポインターです。これも、図 2 を真似すればよいでしょう。

表 10 オープンモード

モード	処理	ファイルが無いとき	ファイルが有るとき
"r"	読み込み (read)	NULL を返す	正常処理
"w"	書き込み (write)	新規作成	前の内容は捨てる
"a"	追加書き込み (append)	新規作成	前の内容の後に追加する
"r+"	読み書き (更新)	NULL を返す	正常処理
"w+"	読み書き (更新)	新規作成	前の内容は捨てる
"a+"	追加のための書き込み	新規作成	前の内容の後に追加する

3.2 特別なファイル(標準入力、標準出力、標準エラー出力)

C 言語でファイルを取り扱う場合、以下のようにプログラムを作成しなくてはなりません。

① ファイルポインター用の変数を FILE 型で宣言する。
② ファイルをオープンする。
③ ファイルの読み書き
④ ファイルのクローズ

しかし、特別な 3 個のファイル(標準入力、標準出力、標準エラー出力)は、この①と②、④が不要です。この 3 個のファイルは、実行時、自動的に①③④の処理が行われます。

コンソール入力で述べたように、標準入力はキーボード、標準出力はディスプレイです。C 言語では、広くは UNIX では、キーボードやディスプレイもファイルとして扱われ、読み書きします。それどころか、すべてのデバイスがファイルとして扱われます。そうすると、シンプルな取り扱いができます。

これら、特別な 3 個のファイルについて、表 11 にまとめておきます。

表 11 標準入出力ファイル

ファイル	ファイルポインター	デバイス(通常)
標準入力	stdin	キーボード
標準出力	stdout	ディスプレイ
標準エラー出力	stderr	ディスプレイ

3.3 ファイル入出力関数

いよいよ、ファイルのデータを読み書きする、ファイル入出力関数です。これは、難しそうですが、簡単です。いままで、標準入力と標準出力に使ってきた関数、printf と scanf とほとんど同じです。コンソール入出力と言ってきたのは、標準入出力です。キーボードやディスプレイもファイルの取り扱いなので、同じ手法がハードディスクにも使えます。

まず、入力からですが、一般のファイルと標準入力の場合を並べて書くと

```
一般のファイル    int fscanf(ファイルポインター, 書式指定, 引数並び)
標準入力          int scanf(書式指定, 引数並び)
```

となります。ファイルポインターを指定する以外、すべて標準入力の場合と同じです。簡単でしょう。もちろん、fscanf 関数でファイルポインターとして stdin を指定した場合、scanf と同じ動作をします。

次に、出力ですが、これもまったく同じです。

```
一般のファイル    int fprintf(ファイルポインター, 書式指定, 引数並び)
標準出力          int printf(書式指定、引数並び)
```

これで、コンソール入出力をしつこく詳細に説明した理由がわかったでしょう。これも、fprintf 関数でファイルポインターとして stdout を指定した場合、printf 関数と同じ動作をします。

最後に標準エラー出力について述べて起きます。標準エラー出力とは、エラーが発生した場合のメッセージなどを出力先です。プログラム中で処理にエラーが発生した場合、そのメッセージの出力先に指定します。printf 関数を使うよりも、

```
fprintf(stderr, "ファイルの読み込みに失敗し増した\n")
```

とした方が、プロっぽくて良いです。エラーメッセージのみ、リダイレクトすることができプログラムの保守性が上がります。

コーヒーブレイク

リダイレクトとは、以下のように出力先を変えることです。ここの command は、実行ファイルと同じです。これは、プログラム中に書くのではなくて、UNIX のターミナルからのコマンドです。

- 標準出力を hogehoge というファイルに出力する場合

```
command > hogehoge
```

- 標準エラー出力を hogehoge というファイルにする場合

```
command 2> hogehoge
```

- 標準入力の変わりに、hogehoge というファイルを使う場合

```
command < hogehoge
```

サンプルプログラム(inout.c)

このプログラムの動作は、①標準入力から整数を読む　②それを標準出力に出力　③それをファイル one に出力 ④ファイル one を読み込む　⑤読み込んだファイル one の中身をファイル two に出力します。

ファイルへのデータの書き込みのデータ区切りに/t (tab タブ)を使っています。データの区切りとして、タブは良く使われます。これは、データをエディターで見るときに、データの並びがそろっているため、視認性がよくなるからです。

```
#include <stdio.h>
main()
{
  int a[4], b[4];

  char tmp;
  FILE *out1, *in, *out2;

  fscanf(stdin,"%d%d%c", &a[0], &a[1], &tmp);  /* read from keybord */
  fscanf(stdin,"%d%d%c", &a[2], &a[3], &tmp);

  printf("\n");
  fprintf(stdout,"%d %d\n", a[0], a[1]);       /* write to display */
  fprintf(stdout,"%d %d\n", a[2], a[3]);

  out1 = fopen("one","w");
  fprintf(out1,"%d\t%d\n", a[0], a[1]);        /* write to file one */
  fprintf(out1,"%d\t%d\n", a[2], a[3]);
  fclose(out1);

  in = fopen("one","r");         /* read from one and write to two */
  out2 = fopen("two","w");
  fscanf(in,"%d\t%d", &b[0], &b[1]);
  fscanf(in,"%d\t%d", &b[2], &b[3]);
  fprintf(out2,"%d\t%d\n", &b[0], &b[1]);
  fprintf(out2,"%d\t%d\n", &b[2], &b[3]);
  fclose(in);
  fclose(out2);
}
```

実行結果

コンソールは以下のようになります。ファイル one と two についても調べましょう。

```
1 23
456 789

1 23
456 789
```

4. プリプロセッサー

4.1 プリプロセッサーとは

C 言語をコンパイルするときの順序は、

①前処理
②コンパイル

です。その後、リンカーに処理がわたり、必要なライブラリーなどをリンクして、実行ファイルができ上がります。CC -o hogehoge fugaguga.c は、コンパイルからリンクまでの処理をしています。プリプロセッサーというのは、前処理のことを言います。

も少し詳しく、実行ファイルができあがるまでのプロセス示すと、図 3 のようになります。この図から分かるように、プリプロセッサーは、いろいろな処理をしたファイルをコンパイラーに渡しています。プリプロセッサーが行う処理は、

- ソースファイルに他のテキストファイルを入れる
- 単語の置き換え
- コンパイル条件の指示

のようなものがあります。要するに、コンパイルを実行する前のテキストファイルの処理です。

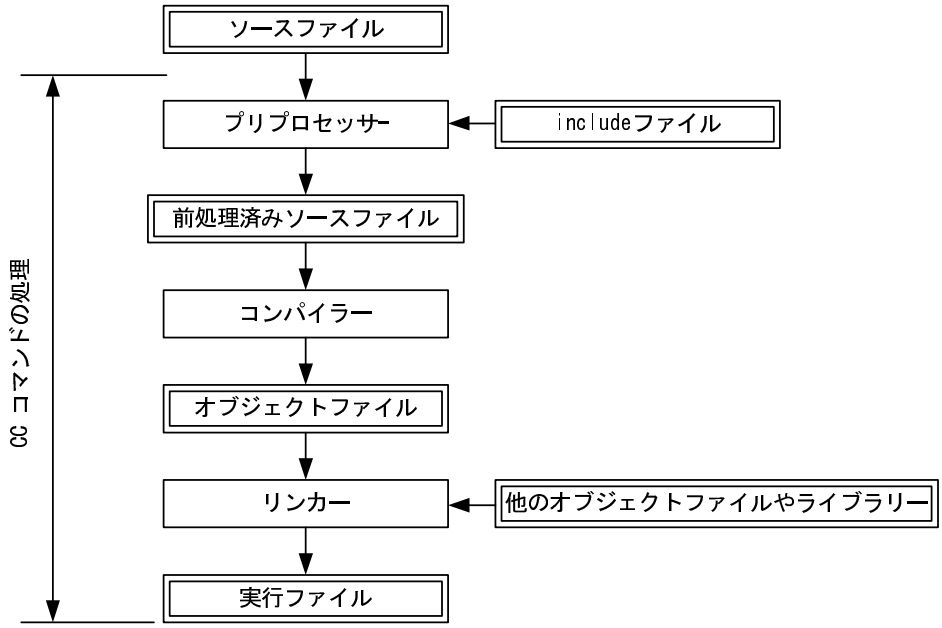


図 3 実行ファイルができあがるまでのプロセス

それでは、実際にプリプロセッサーのコマンドを見てみましょう。コマンドとその概略の機能を表 12 に示します。
いろいろ有りますが、良く使われるのは、後で説明する#include と#define です。他は、必要になったときに、各自勉強してください。

表 12 プリプロセッサーコマンド

プリプロセッサーコマンド	機能
#include	ファイルを挿入する
#define	置き換えとマクロ定義を行う
#undef	#define によるマクロ定義を無効にする
#line	行番号およびファイル名の変更設定する
#pragma	コンパイラへのオプションをしじする
#error	プリプロセッサーの記述エラーの表示
#if #ifdef #ifndef #else #endif	条件付コンパイルの時、これらのコマンドを使用する

4.2 プリプロセッサーコマンドの書き方

プリプロセッサーの書式は、以下の通りです。今まで、さんざん書いてきているので、大体理解はできると思います。

```
#コマンド　パラメーターリスト
```

このプリプロセッサーの書式には、以下の約束があります。

①#の前後に空白が有っても良い。#とコマンドの間に空白が有っても良いのですが、プログラムがわかりにくくなりますので、通常は#とコマンドの間に空白は置きません。
②プリプロセッサーコマンドは、その行で終わりです。C 言語の文のように';'が来るまで有効ということは有りません。したがって、文の区切りを表す';'もありません。
③ただし、プリプロセッサーコマンドを複数行にわたって、記述したい場合、行の終わりに'\'をつけて、次行と接続することができます。

4.3 ファイル挿入(#include)

#include)ファイル名` は、指定されたテキストファイルとこの行を置き換えます。ファイル名が<filename> でかかれた場合には、システムに定義されたファイルです。unix では、/usr/include にあるファイルで置き換えられます。これは、おなじみですね。

プログラマーが作成したファイルを挿入したい場合は、

```
#include "myfile.h"
```

のように記述します。プログラマーが自分でヘッダーファイルを書くことがあります。例えば、大規模なプログラムを作成する場合、ファイルは１つではなく、いろいろな部分を別々のファイルに書いて、それをあとであわせて、１つのプログラムにすることがあります（分割コンパイル）。その場合には、共有する構造体や大域変数を１つのファイルにして、myfile.h というファイルを作り、それぞれのファイルで#includeすることがよくあります。このようにすると、同じ文をそれぞれのファイルに記述する必要が無くなり、タイピングが楽になるとともに、ミスが減ります。

ここでの学習では、分割コンパイルをするほどのプログラムを書くことはないでしょう。将来、比較的大きなプログラムを書くときに、勉強してください。

本来、次のサンプルのような使い方はしませんが、#includeの動作の理解のために、実行させてみてください。#includeにより、そこの行が指定のファイルに置き換わっているのが理解できます。

実行の結果、使い方によっては、便利そうであることが分かりましたか?。ただ、注意して欲しいことは、この行の置き換えはコンパイルの前に行われることです。コンパイルにより実行ファイルが出来上がった後に、ヘッダーファイルを書き換えても、それは反映されません。言いたいことは、ここにデータを書いた場合、その変更を反映させるためには、再度コンパイルが必要ですということです。

サンプルプログラム(include.c)

Hello World !!を出力するおなじみのプログラムです。プログラムの一部が、myfile.h にかかれています。

以下が、c言語のソースプログラムです。変なプログラムですが、ヘッダーファイルがちゃんと書かれていれば、実行可能です。

```
#include <stdio.h>
#include "myfile.h"
}
```

これをコンパイルして、実行ファイルを作るためには、以下のヘッダーファイルも必要です。当然、ファイル名はソースプログラムの指定通りにする必要があります。

```
main()
{
       printf("Hello World !!\n");
```

4.4 定義(#define)

#includeはその行を指定のファイルで置き換えましたが、#defineはファイル内の文字列を指定の通りに置き換えます。単純な文字列の置き換えと、引数を含む文字列の置き換えがあります。この引数を含む文字列の置き換えをマクロ定義と言います。それぞれについて、説明します。

4.4.1 単純な文字列の置き換え

これは、以下のように記述します。

```
#define 文字列1  文字列2
```

このプリプロセッサーコマンドがあると、この行以降の'文字列1'が'文字列2'に、

```
#undef 文字列1
```

があるまで、置き換わります。もし、#undefが無いと、そのファイルの最後の行までコマンドは有効になります。

通常、文字列1はすべて大文字で記述します。別に小文字でも良いのですが、ほとんどのプログラマーは大文字を使います。その方が、プログラムがわかりやすくなります。

4.4.2 マクロ定義

先に説明した単純な文字列の置き換えと似ています。ただ、関数のように引数が使えます。記述は、以下の通りです。

```
#define マグロ名(引数)  引数を含む文字列
```

このプリプロセッサーコマンドも、

```
#undef マグロ名(引数)
```

があるまで、有効です。もし、#undefが無いと、そのファイルの最後の行までコマンドは有効になります。

これは、引数があるので少し難しくなります。例で示します。以下のような置き換えが行われます。

```
#define FN(x) (x*x-2*x+5)

main(){                                          main(){
   y=FN(a);        ---- プリプロセッサー処理 ---->       y=(a*a-2*a+5);
}                                                }
```

4.5 常微分方程式の数値計算

#define を使ったサンプルとして、2 階の常微分方程式の解を数値計算で求めます。2 階の常微分方程式の一般形は、

$$\frac{d^2y}{dx^2} = g(x, y) \tag{1}$$

です。いきなり、この微分方程式の解を求めるとなると、難しそうですが、いたって簡単です。次ページに、プログラムを載せてあります。このプログラムのなかで、微分方程式を計算しているのは、たった 1 行です。プログラムの後半にある

```
y[i]=2*y[i-1]+DY2DX2(x-dx,y[i-1])*dx*dx - y[i-2];
```

だけです。たったこの 1 行で、皆さんが苦労した微分方程式が計算できます。

簡単なだけではなく、どんな形の微分方程式でも解けます。皆さんが数学の授業で苦労した微分方程式は、解析解(解が初等関数の組み合わせ)があるものばかりです。この方法を使うと、解析解が無いものまで計算可能です。驚いたでしょう。このようなことから、コンピューターによる数値計算では、実に有益な情報が得られることが理解できるでしょう。

それでは、重要なこの 1 行の内容を示します。やっと数値計算の授業らしくなってきました。＼(＾　＾)／ まず、微分方程式を数値計算で解く場合、テイラー展開が重要になります。テイラー展開は、

$$\begin{aligned} f(x_0+\Delta x) &= f(x_0) + f'(x_0)\Delta x + \frac{f''(x_0)}{2!}\Delta x^2 + \frac{f^{(3)}(x_0)}{3!}\Delta x^3 + \frac{f^{(4)}(x_0)}{4!}\Delta x^4 \cdots \\ &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}\Delta x^n \end{aligned} \tag{2}$$

です。これは、$x=x_0$ に関するすべての導関数の値が分かれば、他の場所の関数の値がわかるというものです。不思議ですねーw(°0°)w 。次に、$-\Delta x$ を考えます。(2)式と同じように、

$$f(x_0-\Delta x) = f(x_0) - f'(x_0)\Delta x + \frac{f''(x_0)}{2!}\Delta x^2 - \frac{f^{(3)}(x_0)}{3!}\Delta x^3 + \frac{f^{(4)}(x_0)}{4!}\Delta x^4 \cdots \tag{3}$$

となります。次に、(2)と(3)式の各辺どおしを足し合わせます。すると、

$$f(x_0+\Delta x) + f(x_0-\Delta x) = 2f(x_0) + f''(x_0)\Delta x^2 + O(\Delta x^4) \tag{4}$$

となります。この式の 4 次の微小量以降の項を無視して、ちょっとだけ変形すると、

$$f(x_0+\Delta x) = 2f(x_0) + f''(x_0)\Delta x^2 - f(x_0-\Delta x) \tag{5}$$

となります。$y = f(x)$なので、

$$y(x_0+\Delta x) = 2y(x_0) + \left.\frac{d^2y}{dx^2}\right|_{x=x_0} \Delta x^2 - y(x_0-\Delta x) \tag{6}$$

となります。この式の右辺第 2 項は、与えられた微分方程式から、計算できます。(1)式を代入すると

$$y(x_0+\Delta x)=2y(x_0)+g(x_0,y_0)\Delta x^2-y(x_0-\Delta x) \qquad (7)$$

となります。

この式の右辺は、$y(x_0)$と $y(x_0-\Delta x)$と $g(x_0, y_0)$の値がわかれば、$y(x+\Delta x)$の値が計算できることを示しています。関数 $g(x,y)$は問題により与えれています。残りは、通常初期条件として、与えられます。すると、$y(x_0+\Delta x)$が計算できます。この値をまた、(7)式に代入して、今度は、$y(x_0+2\Delta x)$を計算します。これを順次繰り返せば、いくらでも計算できます。このΔx ごとの計算結果が、まさに 2 階の常微分方程式(1)の解になっています。

サンプルプログラムでは、

$$\frac{d^2y}{dx^2}=-y \qquad (8)$$

を計算してます。初期条件は、

$$y(0)=0;$$
$$\left.\frac{dy}{dx}\right|_{x=0}=1 \qquad (9)$$

です。この初期条件から、$y(\Delta x)$を計算する必要があります。そのために、テーラー展開を使います。以下の通りです。

$$y(\Delta x)=y(0)+y'(0)\Delta x+\frac{y''(0)}{2}\Delta x^2 \qquad (10)$$

これで、$y(0)$と $y(\Delta x)$が初期条件より決められたので、あとは順次、$y(2\Delta x)$, $y(3\Delta x)$, $y(4\Delta x)$・・・を計算するだけです。

この方程式の解は、まさに、$\sin(x)$です。サンプルプログラムで確認してみましょう。

実は、常微分方程式を数値計算により解く、もっと強力な方法があります。教科書の p.130 以降にある 4 次のルンゲ・クッタの方法です。初期条件もこちらのほうが、簡単に表現できます。ただし、2 階の微分方程式を解くときには、ほんのちょっとだけ、工夫が必要です。今の段階で、それを説明するのは、早すぎます。ルンゲ・クッタの方法を学習するときに説明します。

サンプルプログラム(define.c)

実際に微分方程式を計算するプログラムです。第 1 行目が微分方程式です。2 と 3 行目がこの方程式の初期条件です。他に計算条件を指定しています。これらを変えることにより、いろいろな微分方程式を、様々な精度で計算可能です。

プログラムの中身は、先に示した数式の通りです。

```
#define DY2DX2(x,y) (-y)    /* differential equation        */
#define DYDX0 1             /* initial condition dy/dx      */
#define Y0 0                /* initial condition y(0)       */

#define N 1000              /* number of calculation steps  */
#define MAX_X 10.0          /* calculation limit x          */

#include <stdio.h>

main()
{
   double dx, x, y[N+1];
   int i;
   FILE *result;

   result = fopen("cal_result","w");

   dx = MAX_X/N;


   /* ------ set intial conditions -------- */

   y[0] = Y0;
   y[1] = Y0+DYDX0*dx+1/2*DY2DX2(0,y[0])*dx*dx;

   fprintf(result,"%e\t%e\n",0,y[0]);
   fprintf(result,"%e\t%e\n",dx,y[1]);


   /* ------ solve the equation --------- */

   for(i=2; i <= N; i++){
      x = i*dx;
      y[i]=2*y[i-1]+DY2DX2(x-dx,y[i-1])*dx*dx - y[i-2];

      fprintf(result,"%e\t%e\n",x,y[i]);
   }

      fclose(result);
}
```

実行結果

作成したプログラムを実行すると、計算結果のファイル(cal_result)が作成されます。これは、各行に(x,y)の値がかかれています。これを、gnuplot というグラフ作成のプログラムを用いて、可視化しましょう。方法は、以下の通りです。

① ターミナルから gnuplot を起動させます。コマンドは、gnuplot です。

```
[yamamoto]$ gnuplot
```

② gnuplot が起動したら、計算結果の描画です。コマンドは、plot "ファイル名"です。コマンドを送ると、図 4 のようなグラフが書かれるはずです。

```
gnuplot> plot "cal_result"
```

③ gnuplot を終了するときにコマンドは、exit です。

```
gnuplot> exit
```

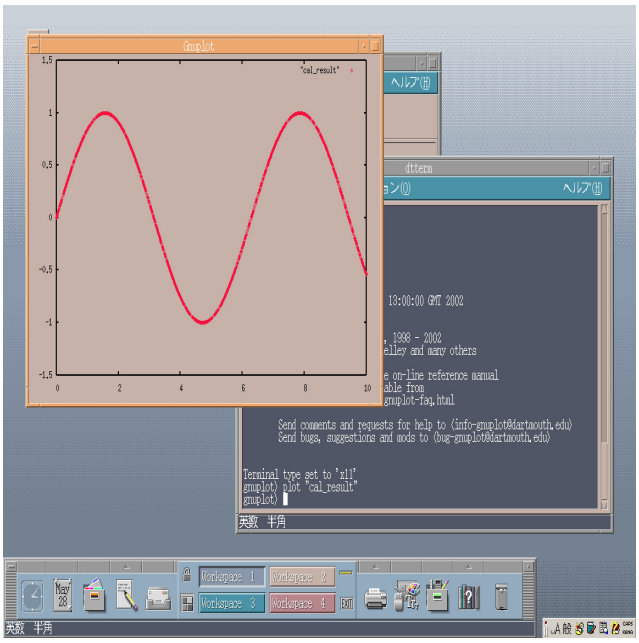


図 4 gnuplot による計算結果のグラフ化