C 言語のサンプルプログラム(6)

計算機応用　5E 2004.06.23

1．プリプロセッサ(15 章)　---------------------　2
1.1 プリプロセッサーとは　2
1.2 プリプロセッサーコマンドの書き方　3
1.3 ファイル挿入(#include)　4
1.4 定義(#define)　5
1.5 常微分方程式の数値計算　6

１．プリプロセッサー

１.１ 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 1 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。

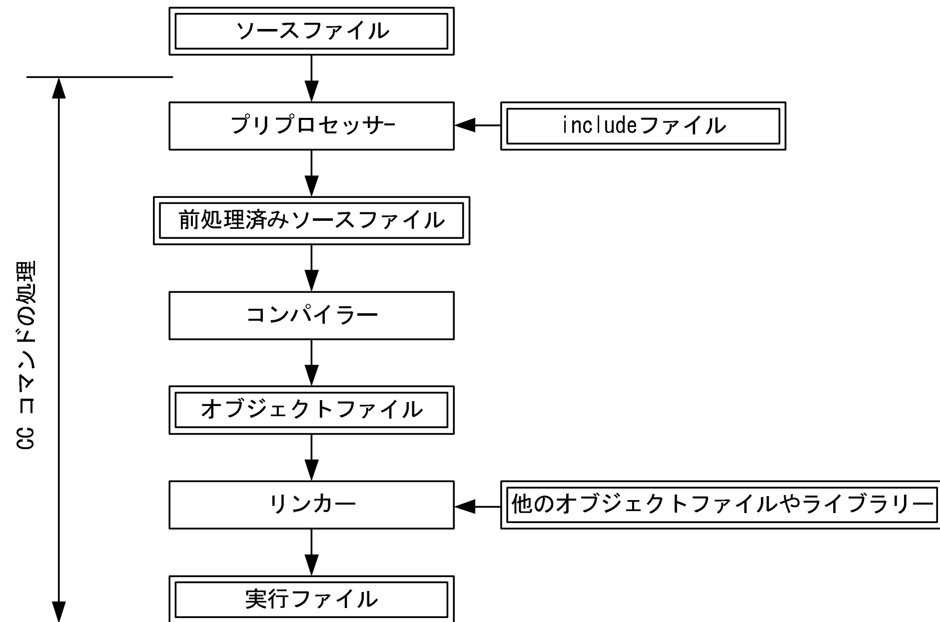

図 1 実行ファイルができあがるまでのプロセス

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１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１ 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指示する
#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

1.2 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。

1.3 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"
return(0);
}
```

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。

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

1.4 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)

int main(){

   y=FN(a);

   return(0);
}
```

プリプロセッサー処理 →

```
int main(){

   y=(a*a-2*a+5);

   return(0);
}
```

1.5 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、

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

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、

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

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、

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

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、

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

となります。この式の 4 次以上の項を無視して、ちょっとだけ変形すると、

$$f(x_0+\Delta x)=2f(x_0)+f''(x_0)\Delta x^2-f(x_0-\Delta x) \qquad (5)$$

となります。$y=f(x)$なので、

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

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 $y(x_0\text{-}\Delta x)$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、

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

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。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>

int 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);

   return(0);
}
```

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 2 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
```

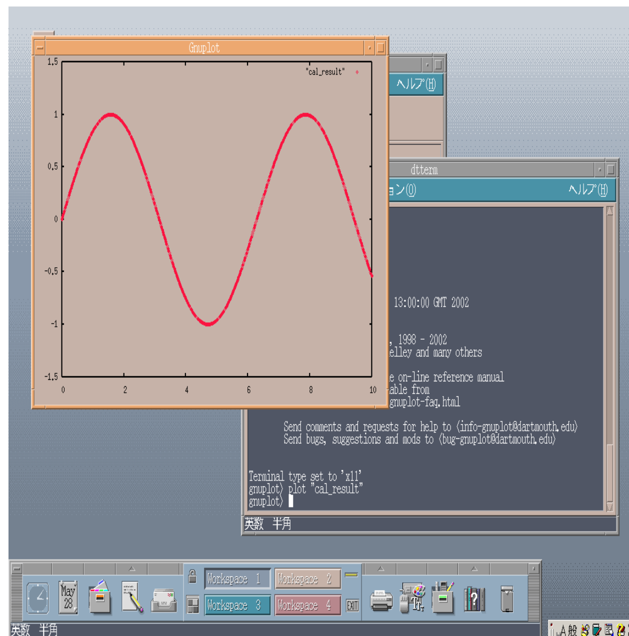

図 2 gnuplot による計算結果のグラフ化