C言語の学習
プリプロセッサー

山本昌志*

2004年6月2日

1　本日の学習内容

本日の内容は、教科書の15章である。主にデータ構造に関わる12〜14章は省く。数値計算のデータ構造は単純なので、ほとんど配列で間に合う。興味のある者はそれらのデータ構造に関係する章を自分で学習せよ。

15章　プリプロセッサー

2　プリプロセッサー (15章)

2.1　プリプロセッサの実行タイミング (p.282)

「gcc」を使って、ソースプログラムをマシン語に直すとき、通常はコンパイルすると言うが、実際にはコンパイル以外の作業も行っている。教科書のp.282に示されているように、コンパイルの前にプリプロセッサーがソースファイルを書き直している。そして、コンパイルの後、リンカーがいろいろなファイルを寄せ合わせて実行可能なファイルを作成する。

プリプロセッサーができることは、教科書のp.283の表に示されている。いろいろな機能があるが、本講義で使うのは

- ファイルの挿入を行う`#include`
- マクロ定義を行う`#define`

くらいである。

2.2　プリプロセッサの使い方

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。

*独立行政法人　秋田工業高等専門学校　電気工学科

#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。

- #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。
- 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。
- 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。

2.2.1 ファイルの挿入

#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>で書かれた場合には、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。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。

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。

リスト 1: マクロを使ったプログラム

```
1 #include <stdio.h>
2 #include "myfile.h"
3   return(0);
4 }
```

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 (myfile.h) 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。いかがヘッダーファイルです。

リスト 2: ヘッダーの例

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

2.2.2 マクロ定義

#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。

文字列置換

これは、以下のように記述します。

#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。

引数付き置換

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 マクロ名 (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 マクロ名 (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。

2.3 プリプロセッサを使ったプログラム例

2.3.1 微分方程式

#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、

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

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。次に、$-\Delta x$ を考えます。式 (2) 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、

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

となります。$y = f(x)$ なので、

$$f(x_0+\Delta x) = 2f(x_0) + \frac{\mathrm{d}^2 y}{\mathrm{d}x^2}\Delta x^2 - f(x_0-\Delta x) \tag{6}$$

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。式 (1) を代入すると

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

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、$y(x_0+\Delta x)$ 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、式 (7) 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?x 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。

リスト 3 に示すプログラムでは、

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = -y \tag{8}$$

を計算している。初期条件は、

$$y(0) = 0 \tag{9}$$

$$\frac{\mathrm{d}y}{\mathrm{d}x} = 1 \qquad x = 0 \text{ のとき} \tag{10}$$

です。この初期条件から、y(?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。

$$y(\Delta x) = y(0) + y'(0)\Delta x + \frac{y''}{2}\Delta x^2 \tag{11}$$

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、$y(2\Delta x)$, $y(3\Delta x)$, $y(4\Delta x), \cdots$ を計算するだけです。この方程式の解は、まさに、$\sin x$ です。リストの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。

2.3.2 プログラム例

1 行 解くべき微分方程式、$\mathrm{d}^2y/\mathrm{d}x^2 = -y$ を引数付き置換で定義している。

2 行 微分方程式の初期条件、$y(0) = 0$ を引数付き置換で定義している。

3 行 微分方程式の初期条件、$y'(0) = 1$ を引数付き置換で定義している。

4 行 計算ステップ数を文字型置換で設定している。プログラム中の N が全て 1000 に置き換わる。

5 行 計算の上限を文字型置換で設定している。プログラム中の MAX_N が全て、10.0 に置き換わる。

20-21 行 (x, y) の値をファイルに書き込んでいる。

28 行 微分方程式を計算している。

29 行 計算結果 (x, y) の値をファイルに書き込んでいる。

リスト 3: 微分方程式を計算するプログラム

```
1 #define DY2DX2(x,y) (-y)                    /* 微分方程式 */
2 #define Y0 0                                /* 初期条件 y=0       at x=0 */
```

```
3  #define DYDX0 1                                  /* 初期条件 dy/dx=1  at x=0 */
4  #define N 1000                                   /* 計算ステップ数 */
5  #define MAX_X 10.0                               /* 計算の上限 ここまで計算 */
6  #include <stdio.h>
7
8  int main() {
9    double dx, x, y[N+1];
10   int i; FILE *result;
11
12   result = fopen("cal_result","w");
13
14   /* ------ 計算条件の設定 -------- */
15
16   dx = MAX_X/N;
17   y[0] = Y0;
18   y[1] = Y0+DYDX0*dx+1/2*DY2DX2(0,y[0])*dx*dx;
19
20   fprintf(result,"%e\t%e\n",0.0,y[0]);
21   fprintf(result,"%e\t%e\n",dx,y[1]);
22
23
24   /* ------ 微分方程式の計算 --------- */
25
26   for(i=2; i <= N; i++){
27     x = i*dx;
28     y[i]=2*y[i-1]+DY2DX2(x-dx,y[i-1])*dx*dx - y[i-2];
29     fprintf(result,"%e\t%e\n",x,y[i]);
30   }
31
32   fclose(result);
33   return 0;
34
35 }
```

2.3.3 実行と表示

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 1 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
   ```

[**練習 1**] リスト 3 のプログラムを実行せよ。

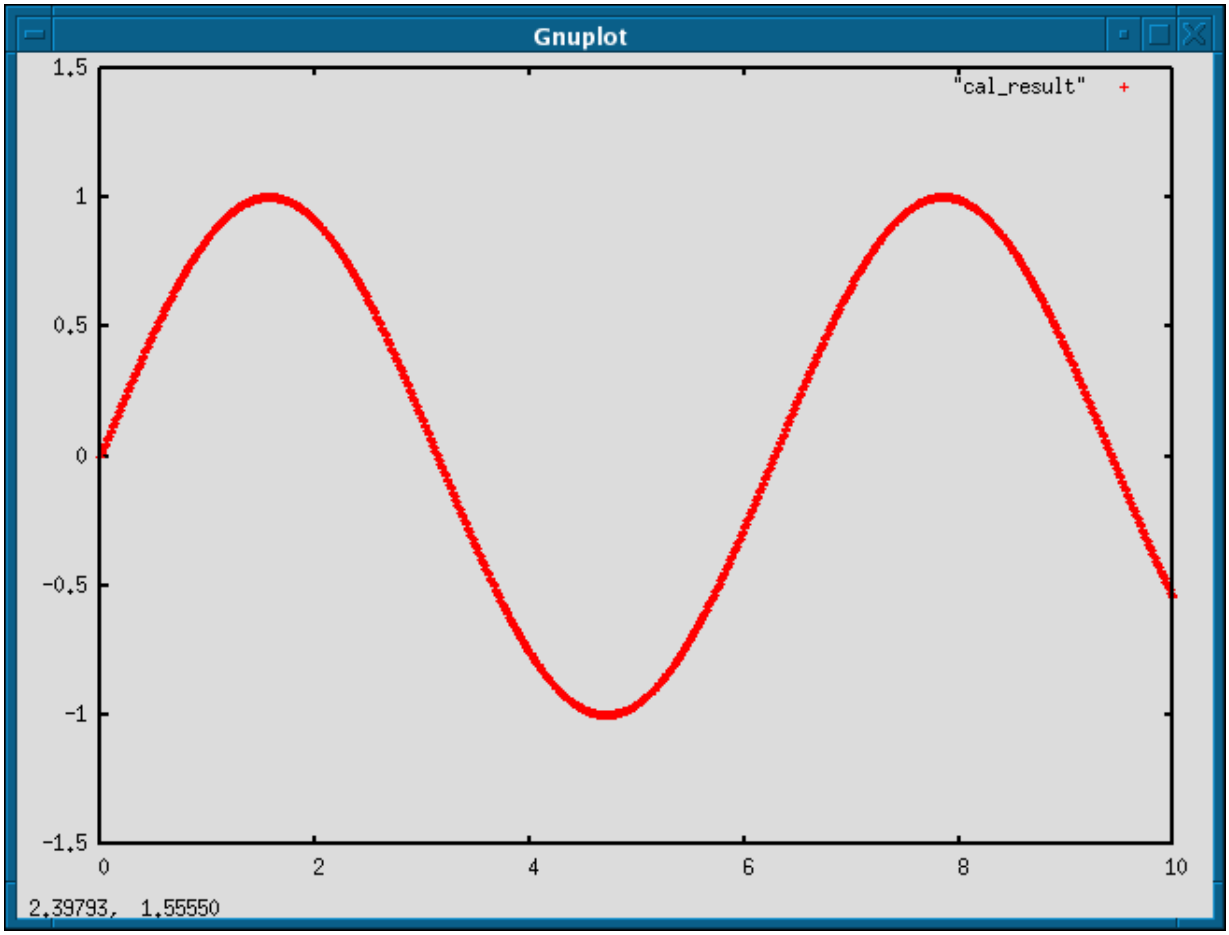

図 1: gnuplot による計算結果のグラフ

[**練習 2**] リスト 3 と式の関係を調べよ。

[**練習 3**] リスト 3 を書き換えて、他の微分方程式を解いて見よ。

[**練習 4**] 計算ステップ数と誤差の関係を調べよ。