計算天文学 II 第3回 C++言語入門 II

牧野淳一郎

2001 年 10 月 22 日

1　今日の予定

今日は、後の話で必要になる最低限の C++ の知識ということで、配列と再帰処理について簡単にまとめる。

本来、 C++ らしい (C や Fortran ではできないような) プログラムスタイルの根幹をなすのは、「クラス」と「テンプレート」であるが、この辺は実はこの講義ではあまり使わないのでまた後で触れることにしたい。

2　配列

Fortran でも C/C++ でも、数値計算をする上で基本になるデータ構造は多くの場合に配列である。

```
// calculate_sum
#include <iostream>
using namespace std;

int main()
{
    int  n ;
    int  data[100];
    int  i,sum;
    cout << "How many numbers you want to input?";
    cin >> n;
    for (i=0;i<n;i++) cin >>data[i];
    sum = 0;
    for (i=0;i<n;i++){
        sum += data[i];
        cout << "data[" << i <<"]=" << data[i] << " sum=" << sum << endl;
    }
}
```

これは配列を使う簡単な例である。

```
int  data[100] ;
```

と変数宣言で書くと、100 個の整数型変数ができ、それぞれが `data[0]` から `data[99]` までといった風に、番号で指定して操作できる。

これはもちろん Fortran で

```
      integer data(100)
```

と書くのとほとんど変わらないが、どうでもいいような違いがいくつかある。

- 括弧が丸括弧ではなくて角括弧である。
- 添字が 1 からではなく 0 から始まる。

括弧はまあなんでもいいが、添字が 0 からになるのは C/C++ の他の多くの言語とは違った特異な点である。なお、 Fortran の場合は、例えば

```
      integer data(0:100)
```

といった具合に 「最初と最後」を指定できるので、実は 1 から始めないといけないというわけではない。しかし、 C/C++ の場合には必ず 0 から始まり、それを変更する方法はない。

2.1 多次元配列

宣言は Fortran では

```
      integer data(100,100)
```

とか書いたが、C/C++ では

```
int data[100][100];
```

と書く。使い方も同様に Fortran では `data(i,j)` のところを `data[i][j]` と書く。次元が増えても同様である。 Fortran では配列の次元は 7 次元までとかいう制限があるが、 C/C++ では特に制限はない。

なお、他にも細かく違うところはある。実用上で大事なのは、メモリ上にデータがしまわれる順番である。 1 次元配列ではもちろん配列要素が順にメモリ上におかれるが、2 次元配列では 2 つの可能性がある。つまり、最初の要素 `data(1,1)` あるいは `data[0][0]` の次に来るのがなにか ?である。 Fortran では `data(2,1)` が来るが、C/C++では `data[0][1]` が来る。この違いは、いろいろな場面で意識する必要がある。

なお、この C の言語仕様で定義された普通の配列に関する限り、 C/C++ の多次元配列には Fortran のそれに比べた大きな欠点がある。それは、関数の引数として配列を渡す時に、大きさを変数として渡すことができないということである。つまり、 Fortran では

```
      subroutine sub1(a, n, nn)
```

```
      integer n, nn
      real*8 a(nn,nn)
      .....
```

といった形で、次元ごとの大きさを引数として渡すことができる。このために、汎用の行列計算や連立一次方程式の解法などのサブルーチンを準備するのが容易である。ところが、C/C++ では引数としては渡せないので話が面倒になる。

もっとも、これはあくまでも汎用のライブラリを準備し、しかもそれをソースプログラムとしてではなくあらかじめコンパイルしたいわゆるバイナリライブラリとして準備する時にだけ問題になることではある。従って、自分でプログラムを書く場合にはあまり問題にはならない。

コンパイルしたライブラリを作るにはいろいろな方法がある。C++ では、ユーザーが新しいデータ型を定義することができるので、自分で定義してしまえば Fortran の配列よりももっと便利に使えるものができる。また、既に人が作った便利なものがいろいろあるので、高度なことをするにはそういったものの利用を考えるべきである。

2.2 ソート

配列を使う例だが、計算天文学とはいえ、数値計算ばかりでも芸がないので、ここではちょっと違うこともやってみよう。

```
// sort.C

#include <iostream>

double data[100];

int main()
{
    int n;
    int   i,j;
    cin >> n;
    for (i=0;i<n;i++)cin >> data[i];
    cout < "Input data\n";
    for (i=0;i<n;i++) cout << data[i]<<endl;
    for (i=0;i<n-1;i++){
        for (j=i+1;j<n;j++){
            if (data[i] > data[j]){
                double  work;
                work = data[i];
                data[i] = data[j];
                data[j]= work;
            }
        }
```

```
    }
    cout << "Sorted data\n";
    for (i=0;i<n;i++) cout << data[i] << endl;
    return 0;
}
```

このプログラムは、入力した数を小さい順に並べ変えて出力する。並べ変える原理は、以下のようなものである。

1. まず、一番小さい数を先頭に持ってくる。そのために、配列の2番目の要素から最後の要素までを順に見て、それが先頭の値よりも小さければ2つを入れ換える。
2. これで先頭にもっとも小さい数が来た。次に、2番目の位置に残りの数の中で最小のものを持ってくる。これには、配列の3番目の要素から最後の要素までを順に見て、それが2番目の値よりも小さければ2つを入れ換える。
3. 以下、同様に、N-1番目まで順にやっていく。

このプログラムを実行するには、まずデータファイルを別にエディタで作っておく。これは先頭にデータ数を書き、後は一行に一つ数字を書いておけばいい。このデータファイルの名前を `sample.dat`、コンパイルしてできた実行ファイルの名前を `simple_sort` とすれば、

```
simple_sort < sample.dat
```

とシェルウインドウで入力すれば結果が得られるはずである。

注意

コンパイルして作るプログラムの名前には `sort` は使わないこと。これは、この名前の別のプログラムがあらかじめ準備されていて、そちらが実行されてしまうからである。

2.3 練習

1. ソートのプログラムを手続きを使う形に書き直してみよ。メインプログラムは手続きを呼ぶだけの形になるようにすること。
2. ソートのプログラムについて、データの数を非常に大きくした時、実行時間がどうなるかを測定せよ。配列宣言の数字を大きくするのを忘れないこと。また理論上どうなるはずか考えてみよ。

なお、あるプログラムの実行時間を計るには `time` コマンドが使える。

```
time simple_sort < sample.dat
```

という風に入力すると、実行結果のあとにもう一行

```
0.040u 0.080s 0:00.56 21.4% 0+135k 2+0io 21pf+0w
```

というような出力が出てくる。この最初の数字が秒単位の実行時間である。なお、実際にかかった時間はもっと長いことがあり得る。これは、ファイルの読み書きの時間は正確には入らないこと、また何人かで1台の計算機を使っているとその分余計に時間がかかることによる。

3　再帰

配列に関しては C++ は Fortran に比べて必ずしも良くなっているとはいい難い、というか正直いって退化しているが、プログラムの構造という観点からはいくつか Fortran にはない有用な特色を持つ。その一つがここで述べる再帰である。なお、最近の多くの Fortran コンパイラでは、言語仕様では再帰をサポートしていなくても実際には仕様が拡張されていて再帰が使える。

再帰とは、手続きが「自分自身を呼び出す」ことである。これは、プログラミングの理論的な扱いという観点からも、また、実用上からも非常に重要な考え方である。簡単な例から考えてみよう。

3.1　もっとも簡単な例

```
// factrial
#include <iostream>
double factrial(int n)
{
    if (n <1){
        return 1.0;
    }else{
        return  n*factrial(n-1);
    }
}

int main()
{
    int n;
    cout << "Enter n:";
    cin >> n;
    cout <<"N = " << n << "   N! = " << factrial(n) <<endl;
    return 0;
}
```

上の例は、階乗を計算するプログラムである。階乗は、下のように定義される：

$$N! = \begin{cases} 1 & (N = 1) \\ N \cdot (N-1)! & \text{otherwize} \end{cases} \tag{1}$$

本当は N が整数でないと定義されないし結果も整数だが、整数だとすぐにオーバーフローしてあまり大きな数の階乗は求められないのでプログラムでは `double` を使っている。

C や Pascal のような「近代的」プログラミング言語では、この定義に従って素直にプログラムを書くことができる。上の例では、まさに、`n` が 1 なら 1 を返し、それ以外では `n*factrial(n-1)` を返すというのが、`factrial(n)` の中身になっている。

例えば `factrial(5)` を計算させるとすると、実際の計算は、以下のような手順で進む。

```
factrial(5)
```

```
→ 5*factrial(4)
→ 5*(4*factrial(3))
→ 5*(4*(3*factrial(2)))
→ 5*(4*(3*(2*factrial(1))))
→ 5*(4*(3*(2*1)))
→ 5*(4*(3*2))
→ 5*(4*6)
→ 5*24
→ 120
```

計算機の構造を考えると、こういうことができるというのは一見不思議ではある。つまり、関数 `factrial` の引数 n は同じものなのに、なぜそれが呼ばれるたびに違う値をとることが可能になるのであろうか？

再帰処理を許さない言語の場合、関数の引数や関数中で宣言された変数には「固定したアドレス」が与えられる。このために、関数が自分自身を呼ぶと、引数の値や変数の値が書き変わってしまいなんだかわからない動作をすることになる。

これに対し、C/C++ などの再帰処理を許す言語では、関数の引数や関数中で宣言された変数は固定されたアドレスを持っているわけではなく、関数が呼ばれる度に新しいメモリ領域が割り当てられる。これは別に難しいことではなくて、ある連続したアドレス領域（通常スタック、、、書類を積み上げた山、、、と呼ばれる）をあらかじめ確保しておいて、サブルーチンが呼ばれればそこに新しく場所をとり、サブルーチンから抜ける時に使っていた場所を返すだけである。

ただこれだけのことで、「関数が自分自身を呼ぶ」というなんだか怪しげなことが実現できるわけである。

3.2 練習

1. 階乗を計算する関数を、再帰を使わない形に書き直してみよ。
2. フィボナッチ数列 $F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1$ を再帰、再帰でないものの両方を使って書け。
3. 上のフィボナッチ数列について、n を大きくしていった時に計算の手間がどのように増えるかを、再帰の場合とそうでない場合のそれぞれについて考察せよ。

4 お絵書き

プログラムを書いて実行しても、でるものが数字ばかりではつまらない。最初の講義で Fortran から PGPLOT ライブラリを使って絵を書いてみたが、もうちょっと基礎的なことをやってみることにする。とはいっても、せっかく再帰をやったので、再帰を使ってわりあい簡単なプログラムで複雑な図形を書いてみよう。

```
// tree.C
#include <iostream>
using namespace std;
```

```
#include "cpgplot.h"

#define PI  3.14159265358979

void tree(int n, double x, double y, double angle, double length)
{
    double x1, y1;
    if (n >0){
        x1 = x + length*cos(angle);
        y1 = y - length*sin(angle);
        cpgmove(x,y);
        cpgdraw(x1,y1);
        tree(n-1,x1,y1,angle + 0.25, length*0.7);
        tree(n-1,x1,y1,angle - 0.25, length*0.7);
    }

}

int main()
{
    int n;
    if(cpgopen("?") !=1 ) exit(-1);
    cpgask(0);
    cpgpage();
    cpgenv(0, 1, 0, 1,0,-2);
    cerr << "Level of the tree? :";
    cin >> n;
    tree(n, 0.5, 0.0, -PI/2, 0.25);
    cpgend();
   return 0;
}
```

これは、樹形図を書くプログラムである。原理は、指定した長さと角度で一本線を引き、その先から角度をつけて2本線を引き、それぞれの端からまた角度をつけて、、、というのを繰り返すだけである。

再帰を使うことで、角度、長さを変えて自分自身を呼び出すという形で簡潔にアルゴリズムが表現できている。

4.1 グラフィックライブラリ

ここでは、第一回の講義でもちょっとつかった PGPLOT の基本的な機能を使って画面に線を書いている。とりあえず、ここで使っている関数についてそれらが何をしているかを見ておこう。cpg で始まっている関数がすべて PGPLOT が提供しているものである。

```
    if(cpgopen("?") !=1 ) exit(-1);
    cpgask(0);
    cpgpage();
```

この 3 行はグラフィックスを使うための準備である。大抵のプログラムではここはこのままでいいはずである。

```
    cpgenv(0, 1, 0, 1,0,-2);
```

これは、画面（ウインドゥ）内での座標系を決めている。最初の 4 つの引数は x 座標の最小値、x 座標の最大値、y 座標の最小値、y 座標の最大値であり、最後の 2 つは画面をいっぱいに使うか、それとも縦横のスケールを合わせるかなどの細かい指定である。

```
        cpgmove(x,y);
        cpgdraw(x1,y1);
```

`cpgdraw` は「現在位置」から指定した座標まで線を引く。その後ではこの指定された座標が「現在位置」になる。`cpgmove` のほうは、なにもしないでただ指定された座標が「現在位置」になる。従って、これらを上の順で呼ぶことで指定した座標から指定した座標までの線分を引くことが出来る。

なお、`cpgdraw` を繰り返して呼ぶと折れ線が引ける。グラフを書いたりするにはこれが便利である。

```
    cpend();
```

これによりグラフィックの利用を終える。PS ファイルなどに結果を書く時には、これを呼ばないと正しいファイルが作られない。画面に出す分にはこれがなくても動くので注意すること。

同工異曲だが、こんなものも書ける。

```
// fract.C
#include <iostream>
using namespace std;

#include "cpgplot.h"

#define PI  3.14159265358979

void fractal(int n,
     double x0,
```

```
     double y0,
     double x1,
     double y1)
{
    double dx, dy, xa, ya;
    if (n > 0){
dx = (x1 - x0)/2;
dy = (y1 - y0)/2;
xa =  x0 + dx  - dy ;
ya =  y0  + dx +dy ;
fractal(n-1,x0, y0, xa, ya);
fractal(n-1,xa, ya, x1, y1);
    }else{
cpgmove(x0,y0);
cpgdraw(x1,y1);
    }
}


int main()
{
    if(cpgopen("?") !=1 ) exit(-1);
    cpgask(0);
    cpgpage();
    cpgenv(0, 1, 0, 1,1,-2);
    int n;
    cerr << "Level of the tree? :";
    cin >> n;
    fractal(n, 0.6, 0.3, 0.6, 0.7);
    cpgend();
    return 0;
}
```

4.2 練習

1. この木では枝が2本ずつ出ているが、もっとたくさん出すにはどうすればいいか ?
2. この木は枝分かれが完全に規則的で不自然である。自然にするにはどうすればいいだろうか ? (ヒント :drand48() という関数を呼ぶと、0 から x までの範囲の乱数 — デタラメな数 — を返す。これを使って枝分かれの長さ、角度を変えて見よう)
3. 他のフラクタル曲線、たとえばコッホ曲線やペアノ曲線を書くプログラムを作ってみる。

5　レポート

前回と今回の「練習」の中から、2つ以上を選んで解き、

1. プログラム
2. 実行結果
3. 考察

をまとめて提出すること。なお、提出は、メイルで

`makino@s01.astron.s.u-tokyo.ac.jp`

あてに、Subject を kadai-1 として提出すること。グラフィックスの出力結果は、PS ファイルにすることができるので TeX でレポートを書くなら文章中にとりこむことが出来る。TeX の場合、あるいは MS Word などで書いた場合でもそうだが、TeX ソースではなく PS または PDF ファイルにしたものを提出すること。

なお、〆切ですが、一応「課題を出してから2回後の講義の時間まで」を目安にする。遅れても単位をつけないということはないが、あまり溜めると出すのが大変になると思うのでそのへんは各自判断すること。

来週からは偏微分方程式に戻る。