第1章　この講義の目的

「プログラミング概論 I」 では、C++ の基本的構成要素について、講義してきました。その中で触れた各知識を使えば、一応のプログラムを作成することが出来ます。しかし、それは一応のプログラミングに過ぎません.

何が不足しているのでしょう。もちろん、C++ のより進んだ技術の知識が必要でしょう。特に、C++ の大きな特徴であるクラス (class) を学んでいません。つまり、「オブジェクト指向プログラミング (Object Oriented Programming)」を学んでいないません。また、例えば、関数の多重定義とクラスの継承といった C++ の重要な機能や、、ファイルの入出力、例外処理など実際のプログラミングの際に必要な知識を扱っていません。しかし、何よりも不足しているのは、具体的な対象をプログラムとして捕らえる手法と経験です。

そこで、この「プログラミング概論 II」では、より進んだ C++ の技術とオブジェクト指向プログラミングの技法を、例を中心に講義していきます。単なるプログラミング言語というより、問題解決手法としてのプログラミングという視点を重視します。そのため、狭い意味でのプログラミングをはずれるような知識を必要とします。

「プログラミング概論 II」の講義では、当然のことながら「プログラミング概論 I 」の内容を理解していることを前提として講義を進めていきます。「プログラミング概論 I 」の内容で、理解が不十分な箇所は復習をしておいてください。

第2章　簡単なクラス

2.1　構造体からクラスへ

構造体を使うことで、異なる型のデータを関連付け、まとまったデータのかたまりとして扱うことができました。しかし、構造体を利用するだけでは次のような不都合がおこる可能性があります。

- データへのアクセスを制限することができません。従って、プログラムの中で誤って構造体に保存されたデータを壊してしまう可能性があります。そのため、プログラムの再利用や、大規模プログラムの構成が困難になることがあります。
- データとそれを扱う関数の関係付けができません。そのため、構造体の中を知らないと構造体を使うことができず、ライブラリ作成などに支障が出ることがあります。
- 初期化と構造体の破棄時の操作を毎回記述しなければなりません。

従来のプログラミングの手法は、手続きを主体としています。つまり、

> 関数などの副手続きにデータ構造を渡して加工する

という形式で全体が構成されていきます。これに対して、オブジェクト指向プログラミング (Object Oriented Programming) では、

> データ構造を中心に考え、データ構造に固有な操作を考える

という形式で設計していきます。

オブジェクト指向プログラミングでは、データ構造とその操作は一体として定義されます。例えば、行列には、行列に固有な和や積を定義する必要がありますが、そのような定義を自然な形でプログラミングで導入しようとするのがオブジェクト指向プログラミングの考え方です。

そこで、構造体を拡張して、データとその操作を組にしたクラス (class) を導入します。クラスは、型の一種ですから、具体的なデータは型宣言のように

クラス名 変数名

と宣言されます。具体的なデータ (変数名) を、オブジェクト (object) と呼びます。

ここで、C++ でのクラスの特徴をまとめましょう。

- データと関数 (メソッド、method) を組にして定義する。
- データや関数へのアクセスを制限できる。
- 他のクラスを継承して新しいクラスを定義できる。

- オブジェクトの初期化時に、コンストラクタ (constructer) という特別な関数が自動的に呼ばれ、その中で初期化を行うことができる。
- オブジェクトの破棄時に、デストラクタ (destructer) という特別な関数が自動的に呼ばれ、その中で破棄手続きを実行できる。
- 関数や演算子を上書きすることができる。

2.2　クラスの定義

もっとも基本的なクラスの定義とオブジェクトの生成方法をまとめましょう。まず、次のような宣言をヘッダファイル (例えば `class.h`) に記述します。

```
class クラス名{
private://プライベートデータ及びメソッド
        //他のクラスからはアクセスできない
 変数及び関数の定義

public://パブリックデータ及びメソッド
       //他のクラスに公開されている
 変数及び関数の定義

};
```

ここでは、メソッドの具体的内容は定義する必要はありません。つまり、プロトタイプ宣言だけを行うこともできます。

次にクラスのメソッドを定義するプログラム本体を記述します (例えば `class.cc`)。もちろん、クラス宣言の中でプログラム本体が記述されている場合には、必要ありません。

```
型 クラス名::メソッド名 (変数並び)
{
  メソッドの本体
}
```

クラスは型と同じように宣言し、使うことができます。

```
クラス名 オブジェクト名;
```

クラスに属するメソッドを使うには、構造体のメンバー変数へのアクセスのように

```
オブジェクト名. メソッド
```

とします。メンバーデータへのアクセスは構造体の場合と同じ記法です。

2.3 スタック

前回 (「プログラミング概論 I 」の最後) はスタック (stack) を構造体で定義しました。ここでは、それをクラスとして定義しましょう。今回も、一つ一つのデータを納める Node とスタック全体を表す Stack の二段構えで構成します。

例 2.1 まず、ノードのクラスを宣言をします。ノードは、保持するデータと次のノードへのポインタをメンバ変数として持ちます。これらの変数を private として宣言し、他のクラスや関数からのアクセスを禁止します。

これでは、スタックからさへ、ノードのデータをアクセスすることはできません。これらの要素へのスタックからのアクセスは許さなければならないので、スタックのクラスを friend として宣言しておきます。つまり、friend を宣言されたクラスからプライベートデータやプライベートメソッドへのアクセスが可能となります。

Program 2.3.1 node.h

```
//*** node.h **********
//
// 簡単なクラス (第 12 回講義資料)
// クラスを使ったノードのヘッダファイル
// 作成日:2000/6/30
//
//*************************
class Node {            //一つのデータを保持するノードのクラス
  friend class Stack;  //スタックのクラスからプライベートデータへのアクセスを許す。
private:
//プライベートデータ
  char a;
  Node* next;
public:
//メンバ関数
  Node(char c) {  a=c; next=NULL; };//コンストラクタ (初期化)
  ~Node(void){};                    //デストラクタ (終了時のメモリ解放)
};
```

ノードのクラスでは、二つのメンバ関数だけが定義されています。クラス名と同じ名前を持つメソッド Node は、コンストラクタと呼ばれ、オブジェクトが生成される際に必ず呼ばれる関数です。保持するデータを与えると、ノードが生成されます。次のノードは NULL とします。この関数には、型が無いことに注意してください。

ここで、メソッドからのメンバ変数へのアクセス方法に注意してください。メソッド Node の中で使われている変数 a 及び next は、クラスのメンバ変数です。このように、メソッドの中では、メンバ変数に自由にアクセスすることができます。

もう一つの関数は、クラス名の前に~を付けたメソッド~Node です。これは、デストラクタと呼ばれ、オブジェクトを破棄する際に必ず呼ばれる関数です。ノードクラスの場合には、何もしていません。デストラクタも、型の無い関数です。また、引数を持つこともできません。

ここでの例のように、クラスの定義の中でメンバ関数の実装を書くことが出来ます。ただし、それは短い関数の場合に限られます。また、それらは、inline 関数として扱われます。inline 関数は、コンパイラが可能な限り呼び出される箇所に展開して、コンパイルしようとします。

今回のプログラムでは、node クラスに対する実装部分がクラス宣言の中に全て記述されています。そのため、node.cc が無いことに注意してください。

Program 2.3.2 stack.h

```
//*** stack.h **********
//
// 簡単なクラス (第 12 回講義資料)
// クラスを使ったスタックのヘッダファイル
// 作成日:2000/6/30
//
//*************************
#include "node.h"

class Stack{            //スタッククラス
private:
  Node* root;           //スタックの先頭のノード
public:
  Stack(void);
  ~Stack(void);
  int push(char);       //プッシュ
  char pop(void);       //ポップ
};
```

次に、スタックのクラス定義を行います。メンバ変数は、構造体での定義と同様に、スタックの一番上のノードです。また、必要なメソッドは、コンストラクタとデストラクタの他に、push と pop というスタックを操作するメソッドです。

メソッドの実装をクラス定義の外側で行う場合には、属するクラスを指定する必要があります。

型 クラス名::メソッド (引数並び)

コンストラクタ Stack では、スタックの底のノードを作成しています。

```
root= new Node('\0');
```

このように、オブジェクトのポインタを生成する場合には、

new クラス名 (引数)

という形で行います。

スタックを破棄する場合、作成したノードを全て破棄する必要があります。スタックを単純に破棄すると、スタック先頭のノードへのポインタが単純にプログラムから見ることができなくなるだけで、スタックに保存したデータはそのまま残ってしまいます。そこで、スタックを破棄する際には、一番上のノードから順に pop します。デストラクタ~Stack では、ノードが無くなるまで pop を繰り返しています。

スタックの操作 pop と push の内容は、構造体の場合と同様です。ただし、引数にスタックを渡す必要が無くなりました。これらの操作は、スタッククラスの操作としてクラスと結びつけられているからです。

Program 2.3.3 stack.cc

```
//*** stack.cc **********
//
// 簡単なクラス (第 12 回講義資料)
// クラスを使ったスタックの実装部
// 作成日:2000/6/30
//
//************************
#include <stdlib.h>
#include <iostream.h>
#include "stack.h"
//************************************************
Stack::Stack(void)// スタックの初期化
{
  root= new Node('\0');//スタックの底の記号
}

Stack::~Stack(void){// 終了時の処理
  if(root!=NULL)
    while(pop()!='\0');
}


int Stack::push(char data)
{
  Node* newroot = new Node(data);  //新しいノードの生成
  newroot->next=root;           //新しいノードをスタックの先頭に
  root=newroot;
#ifdef DEBUG
  cerr <<data<<" is push to "<<newroot<<"\n";
#endif
  return 1;
}

char Stack::pop(void)
{
  char c=root->a;               //スタック先頭の文字
  Node* newroot=root->next;
  delete root;                  //スタックの先頭のノードの消去
#ifdef DEBUG
  cerr <<c<<" is pop from "<<root<<"\n";
#endif
  root=newroot;                 //次のノードをスタックの先頭へ
  return c;
}
```

Program 2.3.4 main_stack.cc

```
//*** stack-main.cc **********
//
// 簡単なクラス (第１２回講義資料)
// クラスを使ったスタックのメイン
// 作成日:2000/6/30
//
//************************
#include <stdlib.h>
#include <iostream.h>
#include "stack.h"
//************************************************
int main(int argc,char** argv)
{
  if(argc!=2){// 使い方の説明
    cerr<<"使い方 : "<<argv[0]<<" 文字列\n";
    return 0;
  }

  char* s=argv[1];
  Stack stack;
  while(*s!='\0'){stack.push(*s);s++;}   //文字列をスタックへ保存
  cout <<"\n";

  char c;
  while((c=stack.pop())!='\0')cout  <<c;  //文字列をスタックから読み出す
  cout <<"\n";

  exit(0);
}
```

スタックを使って文字列を反転するプログラムのメインプログラムを示します。Stack を型のように扱ってオブジェクト stack を生成します。オブジェクトが生成される際に、コンストラクタが呼ばれるために、初期化関数を改めて呼ぶ必要はありません。

スタックの操作を行う pop と push というメソッドが構造体のメンバのように呼ばれている点に注意してください。このように、オブジェクトとその操作がプログラムの上で明示的に結びつけられます。

main 関数の最後で、スタックの破棄の手続きが呼ばれていません。しかし、main 関数から抜け出る際に、スタックは破棄されます。その時、デストラクタが自動的に呼び出され、生成したスタックに保存されたデータは全て破棄されます。このように、クラスを使うことで、データ構造の初期化と破棄の手続きを自動的に行うことが可能となります。

最後に、このスタック操作のプログラムをコンパイルするための Makefile を生成する Imakefile を示します。このプログラムは、node.h と stack.h という二つのヘッダファイルと、stack.cc と stack-main.cc というプログラムから構成されています。

Program 2.3.5 Imakefile

```
/* Imakefile for

作成者：只木進一
作成日：1999/6/27


xmkmf -f で新しい Makefile を作成し
make でコンパイル実行
*/
SRCS=stack-main.cc stack.cc  /* ソースファイル一覧 */

CXXEXTRA_DEFINES=   /* コンパイラに渡す define */
CXXEXTRA_INCLUDES=  /* コンパイラに渡す include path */

/* 作成するターゲットの一覧*/
AllTarget(reverse)
/* ソースファイルからオブジェクトファイルへのコンパイル方法の指定 */
NormalCplusplusObjectRule()
/* 実行形式ターゲットのコンパイル */
NormalCplusplusProgramTarget(reverse,stack.o  stack-main.o,\
NullParameter,NullParameter,NullParameter)
/* ファイルの依存関係の確認 */
DependTarget()
```

2.4　まとめ

この章で出てきたクラスを使う利点をまとめましょう。

- データとそれを扱う関数を組にして定義できる。つまり、データ構造とそのインターフェイスを定義できる。
- データへのアクセスを制限できる。
- データ構造の初期化を、宣言時に行うことができる。
- データ構造の破棄手続きを定義することができる。

次にクラスの定義と利用の要点をまとめましょう。

- ヘッダファイルでクラス、そのメンバー変数、メンバーメソッドを宣言する
- クラスのメソッドを定義する
- コンストラクタ (クラス名と同じ名前の関数) で初期化方法を定義する。
- デストラクタ (クラス名の前に~の付いた関数) で破棄方法を定義する。
- 利用の際には、構造体のように使う。

第3章　簡単なクラス2

3.1　復習

複数の異なる型のデータ、または、動的に変化するデータを一つのものとして扱う方法として、構造体とクラスがあります。構造体に比べてクラスは

- データとそれを扱う関数を関連付けることができる。この関数をメソッドと呼ぶ。
- 初期化手順 (コンストラクタ) 及び廃棄手順 (デストラクタ) を定義することができる。
- データへのアクセス制限をすることができる。

という特徴をもっていました。

クラスが実際に値を持ったものをオブジェクトと呼びます。オブジェクトの操作、あるいはオブジェクトの動作を中心にしてプログラム全体を設計する手法をオブジェクト指向プログラミングと呼びます。

本章では、リストを使って簡単なクラスのプログラミングを復習します。また、実際の応用上で重要な、コピーコンストラクタと代入演算子の多重定義を導入します。

3.2　リストの設計

リスト (list) は、要素が一列に並んだものです。スタック (stack) とは異なり、アクセスの制限はありません。ここでは、文字からなるリストを作成します。これまでと同様に、リスト全体と個々のデータを保持するノード (node) の二つの階層に分けて記述します。

x ノードのクラス `Node` を Program 3.2.1 に示します。文字型のデータを保持するメンバ `a` と次のノードへのポインタ `next` がプライベートメンバ、つまり外部からのアクセスを許さない形で定義されています。ただし、リストのクラス `List` からはアクセスが許されています。メンバメソッドは、コンストラクタとデストラクタのみで、ヘッダファイル `node.h` に実装まで記述されています。従って実装を記述するファイル `node.cc` は存在しません。

今回は、リスト本体には、先頭の要素を取り出す関数 `get` と終端に新しい要素を追加する関数 `append`、あるリストの後ろに別のリストを追加する関数 `combine`、リストのコピー、及びリストの代入を作ることにします。

Program 3.2.1 node.h

```
//*** node.h **********
//
// 簡単なクラス 2(第 13 回講義資料)
// クラスを使ったノードのヘッダファイル
// 作成日:2000/9/9
//
//************************
class Node {            //一つのデータを保持するノードのクラス
  friend class List;  //スタックのクラスからプライベートデータへのアクセスを許す。
private:
//プライベートデータ
  char a;
  Node* next;
public:
//メンバ関数
  Node(char c) {  a=c; next=NULL; };//コンストラクタ (初期化)
  ~Node(void){};                     //デストラクタ (終了時のメモリ解放)
};
```

Program 3.2.2 list.h

```
//** list.h **************
//
// 簡単なクラス 2(第 13 回講義資料)
// リストクラス
// 作成日 (2000/9/9)
#include "node.h"

//リストクラス
class List{
private:
  //プライベートメンバー
  Node* head;//リストの先頭ノード
  Node* tail;//リストの最後尾ノード

public:
  //パブリックメンバー
  List(void);   //コンストラクタ
  ~List(void);  //デストラクタ
  List(const List&); //コピーコンストラクタ
  List operator = (const List&);//代入演算子
  int append(char);//データを最後尾に加える
  char get(void);//先頭のデータを取り出す
  int combine(List);//最後尾にリストを加える
};
```

Program 3.3.1 list.cc

```
//** list.cc **************
//
// 簡単なクラス 2(第 13 回講義資料)
// リストクラス
// 作成日 (2000/9/9)
#include <stdlib.h>
#include <iostream.h>
#include "list.h"

//*** リストクラス ****************************
//コンストラクタ
//オブジェクトを生成して、初期化する
List::List(void){head = NULL; tail=head;}

//デストラクタ
//使ったメモリを開放して、オブジェクトを消去する
List::~List(void){while(get()!='\0');}

//コピーコンストラクタ
List::List(const List& l)
{
  head = NULL; tail=head;
  Node* n=l.head;
  while(n!=NULL){append(n->a);n=n->next;}
}

List List::operator = (const List& l){
  while(get()!='\0');
  Node* n=l.head;
  while(n!=NULL){append(n->a);n=n->next;}
  return *this;
}
```

3.3 メソッドの実装

リストクラスは、先頭の要素 head と終端の要素 tail を要素として持ちます。コンストラクタはこれらの要素を NULL とします。要素は動的に生成されるので、終了時には動的に生成された要素を全て消去しなくてはなりません。そこで、デストラクタではメンバ関数 get を使ってリストの要素を全て取り出します。

次にメンバ関数 get を見てみましょう。この関数は、リストの先頭の要素を取り出し、そこに格納されている文字を返します。リストが空である場合には、ヌル文字\0 を返すことにします。リスト先頭の要素が取り出されるので、先頭の要素のポインタを付け替えなくてはならないことに注意します。

メンバ関数 append はリストの終端に新しい要素を追加します。リストが空の場合には、新しい要素が先頭及び終端の要素となります。リストが空でなければ、終端の次に新しい要素を追加します。

Program 3.3.2 list.cc

```
//新しい文字をリストに追加
int List::append(char a)
{
  if(a=='\0')return 1;
  Node* new_node=new Node(a); //新しいノードの生成
  if(new_node==NULL)return 0; //新しいノードが生成出来ない
  if(head==NULL){             //最初のノード
    head=new_node;
    tail=head;
  } else {                    //それ以外
    tail->next=new_node;      //新しいノードを最後尾とする
    tail=new_node;
  }
  return 1;
}

//先頭文字を取り出す
char List::get(void)
{
  if(head==NULL)return '\0';
  char a=head->a;
  Node* next=head->next;
  delete head;
  head=next;
  return a;
}

int List::combine(List l)//最後尾にリストを加える
{
  char c;
  while((c=l.get())!='\0')append(c);
  return 1;
}
```

3.4　リストの連結とコピーコンストラクタ

リストの結合という操作を考えましょう。「結合」という操作には、様々な可能性があります。そこで、どのような操作を行うかを最初に定義しておく必要があります。ここでは、リスト A の後ろにリスト B の要素を追加し、新しいリスト A とする操作を関数 combine として定義することとします。もちろん、この際に、リスト B に変化があってはいけません。

リスト A の後ろにリスト B の要素を追加し、新しいリスト A とする操作を行うので、リスト A のメンバ関数として定義します。

```
int List::combine(List l)//最後尾にリストを加える
{
  char c;
  while((c=l.get())!='\0')append(c);
}
```

使っている関数は get と append です。問題は、get の使用です。get はリストから要素を取り除くので、リスト l は変化してしまいます。

しかし、関数 combine の変数が、参照型ではないので、呼び出し側のリストのコピーであるから、心配は無いように思えます。しかし、それは、コピーがどのような操作なのかを確認してみなくては、分かりません。

今回のリストクラスでは、後述するようにコピーコンストラクタ (copy constructor) が定義されています。しかし、もしもコピーコンストラクタが定義されていない場合には、どのような動作をするかを考えましょう。コピーコンストラクタが定義されていない場合には、コンパイラはデフォルトのコピーコンストラクタを生成します。このデフォルトのコピーコンストラクタは、オブジェクトのコピーに際して、メンバー変数を素直にコピーします。今の場合には、リストの先頭と終端の要素へのポインタの値です。

上記の関数 combine に、呼び出し側の本当のリストのコピーとして、本当の先頭の要素へのポインタを持ったものが渡ってしまっては、呼び出し側のリストが破壊されてしまいます。コピーとして渡して欲しかったものは、呼び出し側と同じ要素をリストとして持つ、全く別のリストです。

そこで、必要になるのが、オブジェクトをコピーする際の動作を記述するコピーコンストラクタ (copy constructor) です。コピーコンストラクタは、引数に自分と同じクラスのオブジェクトへの参照型を持つ、コンストラクタです。

```
List::List(const List& l)
{
  head = NULL; tail=head;
  Node* n=l.head;
  while(n!=NULL){append(n->a);n=n->next;}
}
```

コピー元のオブジェクトを変更しないように、キーワード const を付けます。コピーコンストラクタでは、元のリスト l に保持された文字を一つづつ新しいリストに追加していきます。

コピーコンストラクタは、関数 combine の例のように、関数の引数にオブジェクトが現れた場合に使われるほか、既に存在しているオブジェクトと全く同じものを生成する場合に使うこともできます。下の例では、リスト b はリスト a のコピーとして生成されます。

```
List a;
list b(a);
```

3.5 代入演算子

コピーと似たような機能として代入があります。代入演算子の動作が明示されていない場合には、コンパイラはデフォルトの代入演算子を使います。その場合、コピーの場合と同様に、オブジェクトが持つメンバ変数の値が素直にコピーされます。もしも、代入が、同じ内容を持つ別のオブジェクトの生成であるならば、代入演算子の動きをそのように定義する必要があります。

C++ では、代入演算子=を始め、加減演算子なども、通常の関数として扱われ、引数と組になって区別されます。つまり、多重定義することが可能です。詳細は、別の章で行いますが、ここでは代入演算子だけについて見ていきましょう。

代入演算子は、二項演算子、つまり二つのモノ (被演算子、operand) の間に置かれる演算子 (operator) です。このような場合、C++ では、演算子の左側のオブジェクトのメンバ関数として記述し、演算子の右側のオブジェクトを引数とするように記述します。

```
List List::operator = (const List& l){
  while(get()!='\0');
  Node* n=l.head;
  while(n!=NULL){append(n->a);n=n->next;}
  return *this;
}
```

一旦、=の左辺の内容が全て消去され、右辺のリストの内容が左辺へ追加されます。最後の this は自分自身へのポインタを表します。従って、自分自身が戻り値になります。

Program 3.5.1 main.cc

```
//** main_list.cc **************
//
// 簡単なクラス 2(第 13 回講義資料)
// リストクラス (メインルーチン)
// 作成日 (2000/9/9)


#include <stdlib.h>
#include <iostream.h>
#include "list.h"

int main(int argc,char** argv)
{
  char* string1="This is a pen.";
  char* string2="This is a pencil.";

  //string1 をリストに格納
  char* s=string1;
  List l1;
  while(*s!='\0'){l1.append(*s);s++;}

  //string2 をリストに格納
  s=string2;
  List l2;
  while(*s!='\0'){l2.append(*s);s++;}

  //l3 を定義
  List l3=l2;

  //l1 に l2 を接続する。
  l1.combine(l2);

  char a;
  //l1 の出力
  while((a=l1.get())!='\0')cout << a;
  cout << "\n";
  cout <<"***********\n";
  //l2 の出力
  while((a=l2.get())!='\0')cout << a;
  cout << "\n";
  cout <<"***********\n";
  //l3 の出力
  while((a=l3.get())!='\0')cout << a;
  cout << "\n";

  return 0;
}
```

第4章　行列の演算

4.1　目的

この章では、実数を要素に持つ行列 (matrix) を表すクラスを作成し、具体的な行列演算を行います。このプログラム作成を通じて、演算子の多重定義とクラスの継承の手法を講義します。

まず、もっとも簡単な場合として、2×2 の実正方行列のクラスを定義します。メンバー関数として、代入のほか、行列の加減法と乗法を定義します。このクラスを使った簡単な演算を行いましょう。

続いて、逆行列を定義するとともに、行列とベクトルの積を定義すると、連立方程式を解くことが出来るようになります。

演習 4.1　2×2 の実正方行列について、加減法及び乗法、行列式及び逆行列の計算方法を復習しなさい。

4.2　基本設計

オブジェクト指向プログラミングでは、クラス設計がプログラム設計の基本になります。プログラムに現れるデータをどのようにクラスに分け、どのような形で操作するかを設計します。今回の場合は、行列という数学的に明確なクラスを使うので、素直に行列のクラスを定義していきます。

まず、行列の保持すべきデータを考えましょう。行列の次数 2、配列の大きさ 2×2、そして配列の要素です。配列の要素は `double` 型としておきます。これらはプライベートデータとして、外部からの直接的なアクセスを禁じておきます。

次にメンバ関数を考えましょう。コンストラクタは、引数なしのものの他に、コピーコンストラクタ、更に要素を指定してのコンストラクタが必要でしょう。これらは、コンストラクタへの多重定義となります。更に、データの読みだしの関数を定義して、データへの不用意なアクセスを制限します。

行列の演算としては、例では、加減と乗法を定義します。乗法は、行列同士の乗法の他に、実数との乗法も定義します。

4.3　自動的に生成されるメソッド

クラスが宣言されると、いくつかのメソッドは自動的に生成されます。プログラマが明示的にメソッドを定義しない場合は、自動生成されるメソッドが使われます。自動生成されるメソッドの動

作を以下に列挙します。

- コンストラクタ:クラスの持つメンバ変数の初期化が行われます。どのような値が初期値として設定されるかは、システムに依存します。
- デストラクタ:クラスの持つメンバ変数の領域が開放されます。new で確保された領域は、単にポインタの領域が開放されるだけで、実態は開放されません。クラスのなかで、動的にメモリ割り付けが行われる場合には特に注意が必要です。
- コピーコンストラクタ:新しいオブジェクトを生成し、引数になっているオブジェクトのコピーを行います。クラスの持つメンバ変数の値だけがコピーされるため、配列、構造体、ポインタ、クラスをメンバ変数に持つ場合には注意が必要です。
- 代入演算子:コピーコンストラクタと同様に、メンバ変数の値だけが代入されます。そのため、コピーコンストラクタと同様の注意が必要です。

4.4　関数と演算子の多重定義

C++ では、関数の区別は、関数名と引数並びで行われます。従って、同じ関数名であっても、引数並びが異なるものは、異なる関数として識別されます。C++ では、四則演算や代入などの演算子も関数として扱われます。二項演算子は引数を二つ持つ関数、単項演算子は引数を一つ持つ関数として扱われます。通常の関数同様に、演算子の識別は、被演算子の型と組にして行われます。つまり、システムが最初から持つ演算子に対して、被演算子の型が異なるものの動作を多重定義することができるのです。

4.5 行列クラスの定義

Program 4.5.1 Matrix.h

```
// *** Matrix.h *******
// 実正方行列クラス
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/10 改定 (complex から double へ)
// ********************

class Matrix{
 private:
  const int m;         //次数
  double *element;    //要素
  int n;               //配列サイズ

public:
//コンストラクタ
  Matrix(void):m(2){
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=0.;
  };

  Matrix(double d[]):m(2){ //要素を与えてのコンストラクタ
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=d[i];
  };

  Matrix(const Matrix& m1):m(2){//コピーコンストラクタ
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=m1.element[i];
  };
```

では、具体的に行列クラスの定義を、ヘッダファイル Matrix.h から見ていきましょう。

まず、クラスのメンバの整数定数 m に注意してください。クラスの宣言の中で定数 const と宣言されていますが、その値を与えることはできません。そこで、コンストラクタで定数の値を定義します。

```
  Matrix(void):m(2){
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=0.;
  };
```

の中の m(2) の部分で、定数 m に値を定義しています。

また、コンストラクタが 3 つあることに注意してください。最初のものは、引数なしで、配列の内容を初期化します。次のものは、成分を与えた初期化です。実数配列で成分を与えます。最後のものは、コピーコンストラクタで、成分のコピーを行って初期化します。

Program 4.5.2 Matrix.h 続き

```
  ~Matrix(void){   //デストラクタ
    delete [] element;
  };

  Matrix operator = (const Matrix& m1){//代入演算子
    if(n!=m1.n){
      cerr <<"size mismatch\n"; return *this;}
    for(register int i=0;i<n;i++)element[i]=m1.element[i];
    return *this;
  };

  double operator () (int i) const {return element[i];};//要素の読みだし
  double operator () (int i,int j) const {return element[i*m+j];};
  int size(void) const {return n;};                        //サイズの読みだし
  int order(void) const {return m;};                  //次数の読みだし
};

//行列の演算
Matrix operator + (const Matrix& m1, const Matrix& m2);
Matrix operator - (const Matrix& m1, const Matrix& m2);
Matrix operator * (const Matrix& m1, const Matrix& m2);
//行列と実数の積
Matrix operator * (const double& c, const Matrix& m);
Matrix operator * (const Matrix& m, const double& c);
```

代入演算子=は、内容をコピーします。前回出てきたように、this は、オブジェクト自身を表すポインタです。

配列の要素を読み出す方法として、ここでは、括弧を多重定義しています。このようにすることで、行と列を与えて直接読み出すことが出来ます (main.cc 参照)。その他に、サイズと次数を読み出す関数が定義されています。

これらの関数の const キーワードは、行列が定数として宣言された際に、内容を読み出す際に必要です。

4.6 行列の演算の定義

Program 4.6.1 Matrix.cc

```
// *** Matrix.c *******
// 実正方行列クラス
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/10 改定 (complex から double へ)
// ********************
#include <stdlib.h>
#include <iostream.h>
#include "Matrix.h"

Matrix operator + (const Matrix& m1, const Matrix& m2){//行列和
  double* c=new double[m1.size()];
  for(register int i=0;i<m1.size();i++)c[i]=m1(i)+m2(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator - (const Matrix& m1, const Matrix& m2){//行列の差
  double* c=new double[m1.size()];
  for(register int i=0;i<m1.size();i++)c[i]=m1(i)-m2(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator * (const Matrix& m1, const Matrix& m2){//行列の積
  int n=m1.order();
  double* c=new double[m1.size()];
  for(register int i=0;i<n;i++)
    for(register int j=0;j<n;j++){
      int r=i*n+j;
      c[r]=0.;
      for(register int k=0;k<n;k++)c[r]+=m1(i,k)*m2(k,j);
    }
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator * (const double& a, const Matrix& m){//行列と実数の積
  double* c=new double[m.size()];
  for(register int i=0;i<m.size();i++)c[i]=a*m(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator * (const Matrix& m,const double& c){
  return c*m;
}
```

行列同士の演算は、クラスのメンバ関数ではなく、通常の関数として定義されています。メンバ関数として定義する場合は、代入演算子の場合のように、演算子の左辺が自分自身です。しかし、通常の関数として定義する場合は、例えば+ の場合は、二つの引数が必要になります。

どちらを選択するかは、読みやすさで決まります。一般には、代入や比較などはメンバ関数として、四則演算は普通の関数として定義します。

ここでは、行列同士の加減、及び行列同士の積と行列と実数の積を定義しています。

Program 4.6.2 main.cc

```
// *** main.cc *******
// 実正方行列クラスのメイン部
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/10 改定 (complex から double へ)
// ********************
#include <stdlib.h>
#include <iostream.h>
#include "Matrix.h"

int main(int argc,char** argv)
{
  double ad[]={1.,1.,1.,1.};
  double ud[]={1.,0.,0.,1.};
  Matrix a(ad);
  Matrix u(ud);

  Matrix c=a+2.*a*u;

  for(register int i=0;i<2;i++){
    for(register int j=0;j<2;j++)
      cout <<c(i,j)<<" ";
    cout <<"\n";
  }
}
```

演習 4.2　行列式を求める関数 det を定義するとともに、行列の割算をする演算子/を多重定義しなさい。

4.7　多重定義の注意

ここで見てきたように、C++ では、同じ関数名を多重に定義出来ます。それらは、関数名と引数並びで区別されます。しかし、多重定義を無闇に行うことは避けなければなりません。つまり、多重定義は、常識的に行う必要があります。例えば行列の加法は、数学的定義と一致するように定義します。

第5章　デバッグ

5.1　デバッグとは

これまでのプログラムは、小さなサイズでした。そのため、コンパイルできなかったり、あるいは実行できない場合は、プログラムをじっと眺めて間違いのある個所を探すことで対応して来たと思います。しかし、プログラムがある程度大きくなれば、そのような方法はとても使えません。

プログラムの誤りのことをバグ (bug、虫) と呼びます。プログラムからこのバグを取り除くことをデバッグ (debug) といいます。デバッグは、プログラムがコンパイルできるまでのもの、実行できるようになるまでのもの、更に実行可能となった後のものに分けることができます。

もちろん、デバッグの前に、プログラムの全体構成、アルゴリズム、クラス設計などが十分に行われていなければいけません。

5.2　プログラム入力に際して

Microsoft Visual C++の環境では、括弧の対応の様子とプログラム構造が正しいかをインデントの情報でみることもできます。例えば `for` ループや条件分岐が正しく行われているか否かをある程度確認することができます。

5.3　コンパイルできるまで

プログラムに文法的誤りがあれば、コンパイル (翻訳) ができません。コンパイラは、誤りのあった個所と理由をある程度指示してくれます。コンパイラのエラーに従ってプログラムを直します。

初心者が陥り易い文法的誤りとしては次のようなものがあります。

- 関数のスペルミス：関数がシステムが持っているものである場合、下記のリンク時までわからない可能性があります。
- 括弧の不一致：これは、編集時にある程度確認できます。
- 型の不一致
- 関数呼び出しの不整合：関数をプロトタイプ宣言することで、関数の呼び出しと関数の定義の不一致を防ぐことができます

プログラムをコンパイルして実行可能にするには、コンパイルとリンクの二つの作業が必要です。各プログラムファイルがコンパイルされると、それらが結合されるとともに、必要なライブラ

リが結合される過程がリンクです。このとき、関数の呼び出しの不一致や、関数の不足などが検出されます。

例として、前回の例題に簡略化演算子+=と-=を例にしてデバッグの様子を見ていきましょう。原因が特定しやすいように、間違いがわざと混じっています。

まず、ヘッダファイルです。これは、前回使用した Matrix.h に、簡略化演算子+=と-=の定義を追加したものです。

Program 5.3.1 Matrix.h

```
// *** Matrix.h *******
// 実正方行列クラス
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/30 改定 (complex から double へ)
// ********************

class Matrix{
 private:
  const int m;        //次数
  double *element;   //要素
  int n;              //配列サイズ

public:
//コンストラクタ
  Matrix(void):m(2){
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=0.;
  };

  Matrix(double d[]):m(2){ //要素を与えてのコンストラクタ
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=d[i];
  };

  Matrix(const Matrix& m1):m(2){//コピーコンストラクタ
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=m1.element[i];
  };

  ~Matrix(void){   //デストラクタ
    delete [] element;
  };
```

Program 5.3.2 Matrix.h 続き

```
  Matrix operator = (const Matrix& m1){//代入演算子
    if(n!=m1.n){
      cerr <<"size mismatch\n"; return *this;}
    for(register int i=0;i<n;i++)element[i]=m1.element[i];
    return *this;
  };
  double operator () (int i) const {return element[i];};//要素の読みだし
  double operator () (int i,int j) const {return element[i*m+j];};
  int size(void) const {return n;};                        //サイズの読みだし
  int order(void) const {return m;};                  //次数の読みだし
  Matrix& operator += (const Matrix& m1);

  Matrix& operator -= (const Matrix& m1);

};

//行列の演算
Matrix operator + (const Matrix& m1, const Matrix& m2);
Matrix operator - (const Matrix& m1, const Matrix& m2);
Matrix operator * (const Matrix& m1, const Matrix& m2);
//行列と実数の積
Matrix operator * (const double& c, const Matrix& m);
Matrix operator * (const Matrix& m, const double& c);
```

次に、プログラム本体 (Program 5.3.3) です。このプログラムの前回の Matrix.cc に簡略化演算子の部分を追加したものです。この簡略化演算子の定義には文法的誤りがあります。

このプログラムをコンパイルすると次のようなエラーが出ます。

```
Matrix.cpp
C:Matrix.cpp(57) : error C2511: '+=' : オーバーロードされたメンバ関数が '<Unknown>' にありません。
C:Matrix.cpp(65) : error C2511: '-=' : オーバーロードされたメンバ関数が '<Unknown>' にありません。
cl.exe の実行


compile_error.exe -  2、警告 0
```

エラーメッセージからファイルのパスを消しています。最初のメッセージは、ファイル Matrix.cpp の 47 行目の宣言

```
Matrix & Matrix::operator +=(Matrix &)
```

が、プロトタイプ宣言と不整合であることを示しています。ヘッダファイルを見ると

```
Matrix & Matrix::operator +=(const Matrix &)
```

とすべきことが分かります。次のメッセージは Matrix.cc の 54 行目です。同じく、プロトタイプ宣言との不一致が原因です。

Program 5.3.3 Matrix.cc

```
// *** Matrix.c *******
// 実正方行列クラス
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/10 改定 (complex から double へ)
// ********************
#include <stdlib.h>
#include <iostream.h>
#include "Matrix.h"

Matrix operator + (const Matrix& m1, const Matrix& m2){//行列和
  double* c=new double[m1.size()];
  for(register int i=0;i<m1.size();i++)c[i]=m1(i)+m2(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator - (const Matrix& m1, const Matrix& m2){//行列の差
  double* c=new double[m1.size()];
  for(register int i=0;i<m1.size();i++)c[i]=m1(i)-m2(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}
```

関数プロトタイプとの不一致の部分を訂正して再コンパイルすると次のエラーメッセージが現れます。

```
Matrix.cpp
Matrix.cpp(61) : error C2440: 'return' : 'class Matrix *const ' から
'class Matrix &' に変換することはできません。(新しい動作 ; ヘルプを参照)
        'const' に対してではない参照は 非 lvalue へバインドできません。
Matrix.cpp(69) : error C2440: 'return' : 'class Matrix *const ' から
'class Matrix &' に変換することはできません。(新しい動作 ; ヘルプを参照)
        'const' に対してではない参照は 非 lvalue へバインドできません。
cl.exe の実行エラー

compile_error.exe - エラー 2、警告 0
```

このエラーメッセージはかなり分かりにくいですが、return の仕方が悪いというのがなんとか分かります。関数の型は Matrix &つまり、Matrix クラスへの参照型です。一方、return で戻るのは this つまり Matrix クラスへのポインタです。つまり、型宣言と戻り値が一致していません。

return で戻すのを*this と直すと正しくコンパイルすることができます。これらのエラーメッセージに応じて直したコードの対応部分を示します (Program 5.3.5)。

5.4 実行時のエラー

コンパイルができても実行できるとは限りません。前節でコンパイルできるようになった行列クラスを使おうとしてメインプログラム main.cc を使うとエラーが出ます。エラーメッセージは

```
DAMAGE:after normal block (#55) at 0x00441AD0
```

などと出ます。どうも、メモリ関係のエラーだということが分かります。

```
Segmentation fault (core dumped)
```

このように、実行時のエラー等で実行できなことがあります。主な原因として考えられるのは、以下のようなものです。

- ゼロでの割り算などによるオーバーフロー
- 配列のインデクスのオーバーフロー
- 関数呼び出しの不整合
- 領域割付の失敗
- ファイルのオープン失敗

Microsoft Visual C++の環境では、一行ずつプログラムを追うことでデバッグを行うことができます。Microsoft Visual C++のデバッグ環境では、実行時のエラー箇所をプログラムの行番号で直接表すような機能は提供されていません。そこで、エラーが起こりそうな箇所を予想してデバッグします。

今回の場合は、新しい演算子を定義しました。その場所を調べましょう。新しい演算子+=の場所にカーソルを置き、マウスを右クリックします。「ブレークポイントの挿入／削除」を選びます。すると、ソースファイルの左に●印が挿入されます。

次に「ビルド」メニューから「デバッグの開始」中の「実行」を選択します。デバッグモードでプログラムが実行され、問題の演算子+=の箇所で停止します。

続いて、F11 を順に押すと、for ループの中での、変数 i の値と m1(i) の値が順に表示されます。本来は i は 0 から 3 まで変化するはずですが、4 まで変化し、その際の m1(i) の値としておかしな値が入っているのが分かります。これが、メモリエラーを引き起こしていました。

5.5 実行できた後

プログラムが実行できたからと言ってプログラム開発が終るわけではありません。正しく動作しているか否かの確認が必要です。

- 要求された機能を持っているか。
- 利用者が正しく使えるようになっているか。
- 手で解ける例題が正しく動作するか。
- 結果が非常識でないか。

Program 5.3.4 Matrix.cc 続き

```
Matrix operator * (const Matrix& m1, const Matrix& m2){//行列の積
  int n=m1.order();
  double* c=new double[m1.size()];
  for(register int i=0;i<n;i++)
    for(register int j=0;j<n;j++){
      int r=i*n+j;
      c[r]=0.;
      for(register int k=0;k<n;k++)c[r]+=m1(i,k)*m2(k,j);
    }
  Matrix m3(c);
  delete [] c;
  return m3;
}
Matrix operator * (const double& a, const Matrix& m){//行列と実数の積
  double* c=new double[m.size()];
  for(register int i=0;i<m.size();i++)c[i]=a*m(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator * (const Matrix& m,const double& c){
  return c*m;
}

Matrix& Matrix::operator += (Matrix& m1){
for(register int i=0;i<=n;i++){
element[i]+=m1(i);
}
return this;
}


Matrix& Matrix::operator -= (Matrix& m1){
for(register int i=0;i<=n;i++){
element[i]-=m1(i);
}
return this;
}
```

Program 5.3.5 Matrix.cc の訂正部分

```
Matrix& Matrix::operator += (const Matrix& m1){
  for(register int i=0;i<=n;i++){
    element[i]+=m1(i);
  }
  return *this;
}
Matrix& Matrix::operator -= (const Matrix& m1)
{
  for(register int i=0;i<=n;i++){
    element[i]-=m1(i);
  }
  return *this;
}
```

Program 5.4.1 main.cc

```
// *** main.cc *******
// 正方行列クラスのメイン部
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/10 改定 (complex から double へ)
// ********************
#include <stdlib.h>
#include <iostream.h>
#include "Matrix.h"

int main(int argc,char** argv)
{
  double ad[]={1.,1.,1.,1.};
  double ud[]={1.,0.,0.,1.};
  Matrix a(ad);
  Matrix u(ud);

  Matrix c=a+2.*a*u;

  for(register int i=0;i<2;i++)c+=u;

  for(register int i=0;i<2;i++){
    for(register int j=0;j<2;j++)
      cout <<c(i,j)<<" ";
    cout <<"\n";
  }
}
```

Program 5.4.2 正しい簡略化演算子の定義

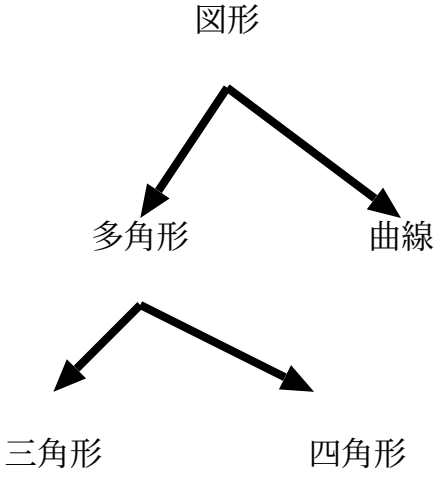
```
Matrix& Matrix::operator += (const Matrix& m1){
  for(register int i=0;i<n;i++){
    element[i]+=m1(i);
#ifdef DEBUG
    cerr <<i<<"\n";
#endif
  }
  return *this;
}
```

第6章　派生クラス(Derived Class)

6.1　派生クラスとは

クラスは、プログラミングを行う上でのデータ構造の概念に対応します。我々の日常的な概念操作の場合、その概念には階層的 (hierarchical) な構造が存在します。つまり、一般的な概念から様々な個別的な概念が派生 (derived) します。例えば図形は図 6.1 のように階層的な概念に整理されます。

図 6.1: 平面図形の階層的概念



概念の個別化の場合、基になる概念の要素が継承され、更に基になる概念が特殊化されたり、新たな性質や機能が追加されます。このような概念の個別化をプログラムの中で実現するのがクラスの派生の方法です。

クラスの派生の手法を使うことによって、基底クラス (基になるクラス) のプログラムを再利用することができるようになり、プログラミング効率を上げることができます。本章では、行列クラスについて、クラスの派生の例を見ることにします。

6.2　回転を表す行列を定義する

前回まで、実数を要素とする 2×2 の正方行列を作ってきました。正方行列の中には、要素の関係が特殊であり、そのために演算規則が簡略化されているものがあります。2 次元平面内の回転を

表す行列もその一つです。

2 次元面内のベクトルや行列を角度 θ 回転する変換は、2×2 の正方行列 R で表されます。

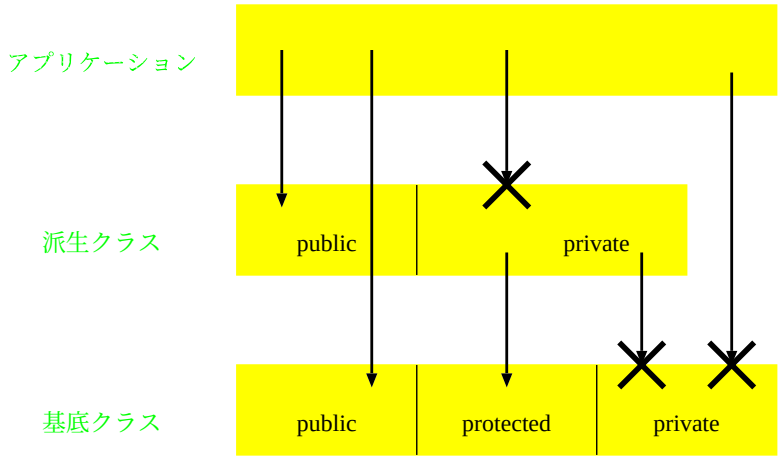
$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{6.1}$$

この行列の独立な変数は θ 一つです。従って行列を初期化する際に、角度だけを与えて初期化できるのが望ましいでしょう。

また、回転を表す行列の行列式は常に 1 であり、逆行列は角度 $-\theta$ の回転を表す行列です。従って、一般的な行列式や逆行列の式を使う必要もありません。

そこで、一般的な 2×2 の実正方行列の派生クラスとして回転行列のクラスを作成しましょう。

図 6.2: メンバへのアクセス制限



6.3　派生クラスの定義

新しいクラス `newClass` を基底クラス `baseClass` からの派生クラスとして定義するには、クラス宣言において次のような宣言をします。

```
class newClass : public baseClass{
  メンバーの宣言
};
```

基底クラスの前のキーワード `public` は、派生クラスを通じて基底クラスが見えることを宣言しています。つまり、基底クラスの `public` メンバーは、派生クラスの `public` メンバーのように振舞い、派生クラスを通じてアクセスすることができます。

もしも、基底クラスの全てのメンバーへの派生クラスを通じてのアクセスを禁じたければ、キーワード `private` を付けます。

いずれの場合も、派生クラスから基底クラスの private メンバーへのアクセスは禁じられています。では、基底クラスのメンバーを派生クラスにだけ公開するにはどのようにしたら良いでしょうか。private の代わりに protected とすることで、可能になります。

前回までの実正方行列クラスのヘッダを、派生クラスを作ることが可能にした場合を示します。行列式や逆行列などの演算が宣言されています。

Program 6.3.1 Matrix.h

```
// *** Matrix.h *******
// 実正方行列クラス
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/11/11 改定 (complex から double へ)
// ********************

class Matrix{
 protected:
  const int m;         //次数
  double *element;    //要素
  int n;               //配列サイズ

public:
//コンストラクタ
  Matrix(void):m(2){
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=0.;
  };

  Matrix(double d[]):m(2){ //要素を与えてのコンストラクタ
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=d[i];
  };

  Matrix(const Matrix& m1):m(2){//コピーコンストラクタ
    n=m*m;
    element=new double[n];
    for(register int i=0;i<n;i++)element[i]=m1.element[i];
  };

  ~Matrix(void){   //デストラクタ
    delete [] element;
  };
```

Program 6.3.2 Matrix.h 続き

```
  Matrix operator = (const Matrix& m1){//代入演算子
    if(n!=m1.n){
      cerr <<"size mismatch\n"; return *this;}
    for(register int i=0;i<n;i++)element[i]=m1.element[i];
    return *this;
  };

  double operator () (int i) const {return element[i];};//要素の読みだし
  double operator () (int i,int j) const {return element[i*m+j];};
  int size(void) const {return n;};                        //サイズの読みだし
  int order(void) const {return m;};                    //次数の読みだし
};

//行列の演算
Matrix operator + (const Matrix& m1, const Matrix& m2);
Matrix operator - (const Matrix& m1, const Matrix& m2);
Matrix operator * (const Matrix& m1, const Matrix& m2);
//行列と実数の積
Matrix operator * (const double& c, const Matrix& m);
Matrix operator * (const Matrix& m, const double& c);
//符号を変える
Matrix operator - (const Matrix& m);
//行列式
double det(const Matrix& m);
//逆行列
Matrix inverse(const Matrix& m);
//印刷
ostream& operator <<(ostream&, const Matrix&);
```

Program 6.3.3 Matrix.cc

```
// *** Matrix.c *******
// 実正方行列クラス
//
// 「プログラミング概論 II」第 1 回講義資料
//  1999/10/7  作成
//  2000/10/11 改定
//  2003/10/10 改定 (complex から double へ)
// ********************
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include "Matrix.h"

Matrix operator + (const Matrix& m1, const Matrix& m2){//行列和
  double* c=new double[m1.size()];
  for(register int i=0;i<m1.size();i++)c[i]=m1(i)+m2(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator - (const Matrix& m1, const Matrix& m2){//行列の差
  double* c=new double[m1.size()];
  for(register int i=0;i<m1.size();i++)c[i]=m1(i)-m2(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}
```

Program 6.3.4 Matrix.cc 続き

```
Matrix operator * (const Matrix& m1, const Matrix& m2){//行列の積
  int n=m1.order();
  double* c=new double[m1.size()];
  for(register int i=0;i<n;i++)
    for(register int j=0;j<n;j++){
      int r=i*n+j;
      c[r]=0.;
      for(register int k=0;k<n;k++)c[r]+=m1(i,k)*m2(k,j);
    }
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator * (const double& a, const Matrix& m){//行列と実数の積
  double* c=new double[m.size()];
  for(register int i=0;i<m.size();i++)c[i]=a*m(i);
  Matrix m3(c);
  delete [] c;
  return m3;
}

Matrix operator * (const Matrix& m,const double& c){
  return c*m;
}

Matrix operator - (const Matrix& m){//符号を変える
return -1.*m;
}

double det(const Matrix& m){//行列式
  return m(0)*m(3)-m(1)*m(2);
}

Matrix inverse(const Matrix& m1) {//逆行列
  double* c=new double[m1.size()];
  double d=det(m1);
  if(fabs(d)>0.){
    c[0]=m1(3)/d; c[1]=-m1(1)/d; c[2]=-m1(2)/d; c[3]=m1(0)/d;
    Matrix m2(c);
    delete [] c;
    return m2;
  } else {
    Matrix m2;
    delete [] c;
    return m2;
  }
}

ostream& operator << (ostream& os, const Matrix& m){//出力
  return os << m(0)<< " " << m(1) <<"\n"<<m(2)<<" "<<m(3);
}
```

次に、回転を表す行列クラス RotateMatrix を定義します。ここでは、角度が新たなメンバー変数として定義され、角度を与えてのコンストラクタが定義されています。

Program 6.3.5 RotateMatrix.h

```
// *** RotateMatrix.h *******
// 座標の回転を表す正方行列クラス
//
// 「プログラミング概論 II」講義資料
//  1999/10/27  作成
//  2003/11/11  改定
// ********************
#include <math.h>
#include "Matrix.h"

class RotateMatrix:public Matrix{//クラス Matrix を継承して宣言
private:
  double theta; //回転角
  friend RotateMatrix inverse(const RotateMatrix& m1);
public:
  RotateMatrix(double t){//角度を与えてのコンストラクタ
    theta=t;
    element[0]=cos(theta);element[1]=-sin(theta);
    element[2]=sin(theta);element[3]=cos(theta);
  };
};

double det(const RotateMatrix& m){//行列式
  return 1.;}

RotateMatrix inverse(const RotateMatrix& m1){//逆行列
  return RotateMatrix(-m1.theta);}
```

派生クラスの初期化の順番を見てみましょう。派生クラスのコンストラクタは、まず、基底クラスのコンストラクタを実行します。ここでの例の場合、引数なしのコンストラクタ Matrix(void) が実行されます。つまり、要素が全て 0 の実行列が生成されます。その後、派生クラスのコンストラクタで、要素が与えられています。行列の要素 element は、基底クラスで protected と宣言されているので、派生クラスから直接アクセス可能です。

引数ありのコンストラクタを呼んで基底クラスを初期化する必要がある場合には、コンストラクタの記述として

派生クラス::派生クラス (引数並び):基底クラス (引数並び)

とします。

デストラクタの場合は、先に派生クラスのデストラクタが呼ばれ、その後に基底クラスのデストラクタが呼ばれます。

6.4 派生クラスとメンバ関数

派生クラスでの一般のメンバ関数の呼び出しはどのように行われるかを次に見ていきます。

派生クラスは、基底クラスのメンバ関数を使うことができます。派生クラスと基底クラスで同じ名前、同じ引数並びのメンバ関数が定義された場合、派生クラスのオブジェクトから呼ばれるのは派生クラスのメンバ関数です。派生クラスから基底クラスのメンバ関数を明示的に使うには

基底クラス::メンバ関数

という形で呼びます。

プログラム 6.3 を見てみましょう。行列式を計算するメンバ関数 det が定義されています。基底クラスでも行列式が定義されていますが、回転を表す行列の場合、常に行列式は 1 のため、新たに定義しています。

クラスの外で定義された関数に関しては、通常の多重定義の規則が使われます。プログラム 6.3 では、private メンバ theta を使って逆方向へ回転させる必要があります。そこで、private メンバへアクセスしてくる関数であることを friend キーワードで宣言しています。

最後に今回の新しい機能を使った main.cc を示します。

Program 6.4.1 main.cc

```
#include <stdlib.h>
#include <iostream.h>
#include "RotateMatrix.h"

int main(int argc,char** argv)
{
  double ad[]={1.,0.,0.,1.};
  Matrix a(ad);
  double pi=4.*atan(1.);
  double theta=pi/2.;
  RotateMatrix u(theta);
  Matrix iu=inverse(u);

  Matrix c=iu*u*a;
  cout <<c<<"\n";

  iu=1.*inverse(u);
  c=iu*u*a;
  cout <<c<<"\n";
  cout <<"Determinant\n";
  cout <<det(u)<<" "<<det(c)<<"\n\n";

  iu=1.*inverse(u);
  c=-iu*u*a;
  cout <<c<<"\n";
}
```

演習 6.1 行列クラスを使って二元連立方程式を解きなさい。

演習 6.2　一般の次数の正方行列クラスを定義し、必要な演算を多重定義しなさい。

第7章　ファイルへの入出力

7.1　はじめに

これまで、入出力には標準入力 (キーボード) と標準出力 (端末画面) のみを使ってきました。これら二つの入出力だけでも、プログラムが一つの入力と一つの出力を使うだけのものであれば、UNIX のリダイレクト機能を使ってファイルを入出力に使うことができます。しかし、複数のファイルを入出力に使うためには、直接的にプログラムの中からファイルの開閉を行う必要があります。

本章では、C++ での基本的なファイルの利用方法を例に基づいて見ていきましょう。また、C++ の入出力より C のそれが便利である場合もあります。そこで、C での基本的なファイル操作の方法も確認します。

プログラミング言語でファイルの入出力を行う場合、通常は直接にファイルを読み書きすることはありません。C/C++ では、ストリーム (stream) と呼ばれる文字列を保持するデータ構造を使っています。ストリームはファイルと接続され、プログラムからストリームを通じて入出力を行います。

入出力がファイルと直接に行われないために、バッファリング (buffering) という現象が起きます。出力はストリームに一旦格納され、適当な大きさを超えて始めて物理的媒体であるファイルは書き出されます。バッファリングの途中でプログラムがアボート (異常終了) した場合には、何も書き出されません。このような方法を取っているのは、ハードディスクなどの物理媒体への読み書きのコストが高いためです。

標準入出力を含めて、ストリームへの読み書きは、通常は文字列で行われます。つまり、文字列以外のデータは文字列へ変換する必要があります。整数や浮動小数点データを文字列に変換する方式を書式 (format) と呼びます。このことについて、日常的に使う書式について見ていきます。

ファイルという物理媒体が関係することと、ファイルの形式と読みだしプログラムの不一致が起こる可能性があるため、ファイルに関連して実行時のエラーが起こる可能性が高くなります。このことについては、次章以降で説明します。

7.2　標準入出力の復習

標準出力、つまり端末画面への出力には、これまで `cout` というクラスを使ってきました。

```
cout << 変数;
```

演算子<<は、標準で定義されている型を適切な書式に従って標準出力ストリームへ印刷する機能を有しています。ユーザーが新しい型 (クラス) を定義する場合には、演算子<< を多重定義することで、出力方法を定義することができます。

標準出力クラス cout は、通常のデータ出力のためのストリームです。従ってバッファリングが行われます。それに対して、エラー出力のためのストリーム cerr はエラー表示という目的のため、バッファリングが行われません。

標準出力と標準エラー出力は、通常は端末画面に表示されるために、区別がつきません。しかし、システムとしては厳密に区別されています。標準出力へのデータ出力を有するプログラム prog を考えます。実行時に

```
prog > output
```

とすると、出力はファイル output へ出力されます。しかし、エラーが起こった場合は、エラーは端末画面に出力されてきます。

```
(prog > output) >& errors
```

とするとエラーは別のファイル errors へ出力されます。

標準入力、つまりキーボードからの入力は、クラス cin を使って行われます。

```
cin >> 変数名;
```

7.3　ファイルへの出力

ここでは、乱数を生成して、ファイルへ書き出すプログラムを例として示します。

Program 7.3.1 out.cc

```
// *** out.cc *******
// ファイルへの出力
//
// 「プログラミング概論 II」第 4 回講義資料
//  1999/11/11  作成
//  2003/11/24  VC++対応
// ********************
#include <fstream.h>
#include <stdlib.h>
#ifdef WIN32
//drand48 及び srand48 を持たない場合に対応
// 2003/11/24 追加
double drand48(void){return rand()/double(RAND_MAX);}
void srand48(unsigned int s){srand(s);}
#endif

int main(int argc,char** argv)
{
  const int num_data=100;
  long int seed=98300;


  //ファイルを開く
  ofstream fout;
  fout.open("out");
```

Program 7.3.2 out.cc 続き

```
  fout <<"#乱数\n";
  fout <<"#データ数\n"<<num_data<<"\n";
  fout <<"#\n";

  //乱数初期化
  srand48(seed);

  for(register int i=0;i<num_data;i++){
    double r=drand48();
    fout.setf(ios::scientific);  //e 変換
//    fout.setf(ios::fixed);     //f 変換
    fout.width(15);              //総桁数
    fout.precision(6);           //小数点以下の桁数
    fout << r <<"\n";
  }

  fout.close();//ファイルを閉じる
}
```

ファイルへの入出力を行うには、標準入出力の場合に iostream.h をインクルードした代わりに fstream.h をインクルードします。

ファイルへの出力は、クラス ofstream を使います。ファイル出力用オブジェクトを生成した後に、そのストリームとファイルを結合します。

```
ofstream fw;
fw.open("ファイル名");
```

ファイルへの書き出しは、標準出力と同様に演算子<<を使います。

```
fw<<変数名;
```

ファイルを使い終ったら、ファイルを閉じます。

```
fw.close();
```

出力用ストリームのコンストラクタにファイル名を指定することも可能です。

```
ofstream fw("ファイル名");
```

ストリームクラスのデストラクタは自動的にストリームを閉じます。

メンバ関数 open は、本来は引数を 3 つ持っています。

```
ofstream::ofstream(const char *name, int mode=ios::out,
                                     int prot=filebuf::openprot);
```

二番目の引数はファイルを開ける際のモードで、表 7.1 を指定する。何も指定しないと `ios::out` となります。

フラグ	内容
ios::app	既に存在しているファイルに追加する
ios::ate	ファイルを開き、その終端に移動する
ios::in	入力用にファイルを開く
ios::out	出力用にファイルを開く
ios::binary	バイナリファイルとして開く
ios::trunc	ファイルを開く際に内容を一旦消去する
ios::nocreate	指定したファイルが存在しない場合失敗する
ios::noreplace	ファイルに上書きできない

表 7.1: ファイルを開ける際のモード

3 番目の引数は、UNIX ではファイルのパーミッションを指定し、何も指定しないと `0644` となっています。

7.4　変換ルーチン

C++ では、ストリームクラスの定義で、データの変換方法が指定されています。ユーザーがその変換方法を変更するためのメンバ関数が用意されています。

```
fw.setf(flags);      //フラグの設定
fw.unsetf(flags);    //フラグの設定解除
```

設定できるフラグを表 7.2 に示します。

ここでは、特に

```
ios::scientific
ios::fixed
```

の二つについて説明しましょう。浮動小数点型データは、符号 (±)、仮数部 ($f.fff$)、指数部 ($\pm e$) の三つの部分から構成されています。

$$\pm f.fff \times 10^{\pm e}$$

e の値が小さければ、

$$\pm f.fff$$

の形式で出力ができますが、e が大きくなればそのようは出力はできません。そこで、C/C++ などでは

$$\pm f.fff\mathrm{e} \pm e$$

という表記が使われます。例えば、10^5 は

フラグ	内容
ios::skipws	入力時に先頭の空白を読み飛ばす
ios::left	左詰めで出力する
ios::right	右詰めで出力する
ios::internal	符号を左詰め、データを右詰めで出力し、中間を埋め込み文字で埋める
ios::dec	10 進変換で出力する
ios::ict	8 進変換で出力する
ios::hex	16 進変換で出力する
ios::showbase	基数をつけてデータを出力する。例えば 16 進ならば 0x が付く
ios::showpint	必ず小数点を表示する
ios::uppercdase	16 進数を表示する際に大文字を使う
ios::shopos	正の数を表示する際に+を表示する
ios::scientific	浮動小数点を表示する際に e 変換に相当する表示を行う
ios::fixed	浮動小数点を表示する際に、固定小数形式を使う

表 7.2: I/O 変換フラグ

```
0.1000e+06
```

となります。このような表記方法を指定するのが ios::scientific です。

数値計算の場合、有効桁は非常に重要です。数値計算に伴う数値エラーなどが避けられない場合が多いからです。そこで、出力の総桁数や有効桁数を制御する必要があります。出力ストリームのメンバ関数 width は総桁数を、precision は小数以下の有効桁数を指定することができます。

指数が小さい場合、指数表現でないほうが便利な場合があります。その場合は、ios::fixed を使います。

これらの、変換方法のフラッグ類は、一度設定されると別の変更があるまでそのままのものと、出力が行われるたびにリセットされるものがあります。それは、コンパイラに依存しています。基本的に、リセットされるものと考えておいたほうが良いでしょう。

7.5 ファイルからの入力

ファイルからの入力は、入力ストリームクラス ifstream を使います。

Program 7.5.1 in.cc

```
// *** in.cc *******
// ファイルへの出力
//
// 「プログラミング概論 II」第 4 回講義資料
//  1999/11/11  作成
// ********************
#include <fstream.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char** argv)
{
  char* fname="out";


  //ファイルを開く
  ifstream fin;
  fin.open("out");
  if(fin.bad()){
    cerr <<"Can not open "<<fname<<"\n";
    exit(8);
  }

  char buf[256];

  fin >>buf;
  fin >>buf;

  int n;
  fin >>n;
  cout <<"Number of data is "<<n<<"\n";
  fin >>buf;
  double* data=new double[n];

  for(register int i=0;i<n;i++){
    fin>>data[i];
  }

  fin.close();

//平均と標準偏差を計算する


}
```

第 7.3 節で出力したファイルを読み込みましょう。次のように、ファイル名を指定してファイルを開きます。

```
  char* fname="out";


  //ファイルを開く
```

```
ifstream fin;
fin.open(fname);
if(fin.bad()){
  cerr <<"Can not open "<<fname<<"\n";
  exit(8);
}
```

出力の場合は、ファイルが存在しなければ新しいファイルを生成してそこに書き込めば良かったのですが、入力の場合、データファイルが無ければ読み込むことができません。open メンバ関数を実行した際に、ファイルが無ければフラッグの値がセットされます。bad メンバ関数でそのフラグを調べて、ファイルのオープンに失敗した場合に、作業が止まるようにプログラムしています。

また、入力ファイルには、コメントはデータの総数が記入されているので、そのデータファイルに合わせて、データを読み込んでいく必要があります。

演習 7.1　プログラム 7.5.1 の平均と標準偏差を計算する部分を完成させなさい。

第8章　ファイルへの入出力(その2)

8.1　はじめに

前章では、基本的なファイルのオープン・クローズ、及び入出力の方法を説明しました。本章では、ファイルの入出力に関係した幾つかの重要な技術をまとめます。例えば、ファイルからの入力に際しては、データの総数が不明の場合を考えます。ファイルの終端に至ったことをストリームの状態を調べることで判断し、ファイルの終端まで読んだ所で、入力を終了する方法が必要です。

問題によってはC++ よりもC の機能を使ったほうが簡単にプログラムできる場合があります。そこで、C の入出力の基本について簡単にまとめます。

8.2　ストリームの状態

入出力ストリームには、いくつかの状態があります。その状態を見ることでファイルの接続に成功したかやファイルのの終端に至ったかなどを見ることができます。g++ の場合、

`/usr/include/g++/streambuf.h`

で記述されているクラス `ios` に状態が定義されています。Microsoft Visual C++の場合

`C:\Program Files\Microsoft Visual Studio\VC98\Include\IOS.H`

に定義されています。

表 8.1: ストリームの状態

ビット名	内容
goodbit	良好
eofbit	終端に到達した
failbit	次の入力は失敗する
badbit	ストリームが壊れている

ストリームの状態は4 つのビットで定義されています (表8.1)。ストリームの状態を調べる関数もストリームクラスのメンバ関数として用意されています。

```
bool good() const;     //次の操作は成功する
bool eof() const;      //ストリーム終端に達した
bool fail() const;     //次の操作は失敗する
```

```
bool bad() const;     //ストリームが壊れた

iostate rdstate() const;//ストリーム状態のビットを返す
void clear(iostate f=goodbit); //ストリーム状態のビットを設定する
void setstate(iosstate f) {clear(rdstate()|f);//ストリーム状態のビットを追加する

operator void* () const { return fail() ? (void*)0 : (void*)(-1); }
int operator !() const { return fail(); }
```

Program 8.2.1 read2ddata.cc

```
// *** read2ddata.cc *******
// ファイルからの入力
//
// 「プログラミング概論 II」第 5 回講義資料
//  1999/11/17  作成
//  2002/11/26  改定
// ********************
#include <fstream.h>
#include <stdlib.h>
int main(int argc,char** argv)
{
  char* fname="out2d";    //ファイル名指定

  //ファイルを開く
  ifstream fin;
  fin.open(fname);
  if(fin.bad()){          //ファイルを開けられない場合の処理
    cerr <<"Can not open "<<fname<<"\n";
    exit(8);
  }

  int n=0;  //データ総数
  int m=0;  //扇型の内部の点の総数
  double x,y;
  while(!fin.eof()){      //ファイル終端まで読む
    fin >>x;
    fin >>y;
    if(fin.good()){       //本当にデータが読めた場合
      cerr<<x<<" "<<y<<"\n";
      if((x*x+y*y)<=1.)m++;// 扇型の内部ならば m を加算
      n++;
    }
  }
  fin.close();            //ファイルを閉じる

  cout <<"pi = "<<4.*double(m)/double(n)<<"\n";
}
```

これらを使うと、ファイルの終端までデータを読むことが可能となります。

```
  while(!cin.eof()){
    cin >> data;
  }
```

最後の二つの演算子は少し分かりにくいと思います。これらは、ストリームそのものを条件判断に用いる際に使います。例えば

```
while(cin>>x){
  ....
}
```

において、命令 cin>>x は結果として cin への参照を返します。この時使われるのが operator void* () で、ストリームからの読み込みが成功したか否かを判定することができます。

プログラム 8.2.1 では、0 以上 1 未満の乱数が一行に二つづつ空白に区切られて入っているファイル out2d を読み込む例を示します。

このプログラムでは、データファイルに保存されている 2 次元のランダムデータを使って、それが 90 度の扇型に入っているか否かを判定します。データの総数に対してこの扇型に入った点の割合は、乱数が一様であってかつデータ数が十分に多い極限で $\pi/4$ になるはずです。このことを使って、π の計算を行っています。

このように乱数を使って、積分などを計算する方法を一般に Monte Carlo 法と呼びます。Monte Carlo 法は、雑音のある系のシミュレーションや組合せ最適化問題の解法など、広く使われる技法の一つです。

入力の部分で、一旦 while(!fin.eof()) でファイル終端で無いことを確認した後、更に if(fin.good()) とストリームの状態を調べているのは、データファイルの最後のデータを含む行が改行されている場合と改行されていない場合に対応するためです。

8.3 文字の入力

これまで入力は、標準入力の場合

```
cin >> 変数名;
```

という形で行ってきました。この場合、スペースや改行で区切られたものを一つの単位として変数へ代入が行われます。文字や文字列の場合、一文字だけ読みたい、あるいはスペースを含む一行を一度に読みたい場合があります。

一文字だけ読み込む場合には、ストリームクラスのメンバ関数 get(char) を使います。次のプログラムは標準入力から標準出力へ一文字づつコピーします。

```
int main()
{
  char c;
  while(cin.get(c))cout.put(c);
}
```

put(char) は出力ストリームクラスのメンバ関数で、一文字ストリームへ出力します。
　一行すべてを読み込むには、getline を使います。

```
int main()
{
  const int size=1024;
  char buf[size];
  while(cin.getline(buf,size))cout <<buf<<"\n";
}
```

関数 getline では、改行記号の直前までが読み込まれます。改行記号を特に指定する場合には、3 番目の変数として指定します。

```
getline(char *,int,char);
```

8.4　C の出力

　ここでは、C の出力のうち、特に有用なものをまとめます。
　標準出力へは次のマクロを使って出力を行います。

printf(書式, 変数並び)

変数はいくつでも並べることができます。ただし、文字列である書式と整合している必要があります。例えば、整数型変数と浮動小数点型変数を出力する場合、次のように使います。

```
int main()
{
  int n=1;
  double f=0.1;
  printf("%d %f\n",n,f);
}
```

　主な書式は次のようになっています。

%s	文字列
%d	整数
%f	浮動小数点型 (指数型ではない)
%e	浮動小数点を指数型で表示
%g	浮動小数点型を%f か%e の短い表現法で表す

幅や小数以下の桁数を指定する場合は

```
%13.6e
```

のように指定します。
　ファイルへの出力を行う場合は、fopen を使ってファイルとストリームを結びつけた後、printf の代わりに fprintf を使います。

```
FILE* fw;
fw = fopen("file","w");
fprintf(fw, 書式, 変数並び);
fclose(fw);
```

fprintf は、最初の引数がストリームである点以外は、printf を同じです。
特に有用な機能は、文字列変数へ出力する機能です。

```
char line[1024];
sprintf(line, 書式, 変数並び);
```

つまり、sprintf を使うと、ストリームへではなく、第一引数で指定された文字列変数に書き出すことが可能です。

8.5 C の入力

Program 8.5.1 read2ddata-2.cc

```
// *** readd2ddata-2.cc *******
// ファイルへの出力
//
// 「プログラミング概論 II」第 5 回講義資料
//  1999/11/17  作成
//  2000/11/8   改定
// ********************
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char** argv)
{
  char* fname="out2d";    //ファイル名指定

  //ファイルを開く
  ifstream fin;
  fin.open(fname);
  if(fin.bad()){          //ファイルを開けられない場合の処理
    cerr <<"Can not open "<<fname<<"\n";     exit(8);
  }
```

C の入力は scanf で行います。

```
scanf(書式, 変数へのポインタの並び)
```

ここで、変数そのものではなく、ポインタ並びを指定することに注意が必要です。

```
int main()
{
```

Program 8.5.2 read2ddata-2.cc つづき

```
  char buf[256];
  int n=0;  //データ総数
  int m=0;  //扇型の内部の点の総数
  double x,y;
  while(fin.getline(buf,256,'\n')){
    if(sscanf(buf,"%le %le",&x,&y)==2){
      cerr<<buf <<" : "<<x<<" "<<y<<"\n";
      if((x*x+y*y)<=1.)m++;
      n++;
    }
  }

  fin.close();              //ファイルを閉じる
  cout <<"pi = "<<4.*double(m)/double(n)<<"\n";
}
```

```
  int n;
  float f;
  scanf("%d %f\n",&n,&f);
}
```

上の例では、整数型と浮動小数点型への入力を読んでいます。scanf は書式に一致した変数の数を戻り値として返します。上の例では、2 が返ってきます。この機能を利用すると、入力のパターンを判別し、プログラムが予想した正しいデータ形式であることを確認しながらデータを読み込むことができます。

出力と同様にファイルストリームからの入力 fscanf と文字列変数からの入力 sscanf が用意されています。

プログラム 8.5.1 では、乱数が一行に二つづつ入ったファイルを C の入力を使って読み込む例を示します。fscanf の戻り値を使って、ファイルに浮動小数点型データが二つならんでいることを確認しながら読み込んでいます。

演習 8.1　標準入力から整数を読み、その数だけ乱数を発生するプログラム mkrand を作りなさい。また、その出力、つまり幾つ乱数が入っているか不明なファイルを読み込み、0.1 刻みのヒストグラムを出力するプログラム hist を作成しなさい。

演習 8.2　popen と pscanf を使うと、外部コマンドの実行結果を読み込むことができます。データファイルが圧縮されている場合に、解凍しながらファイルを読み込むプログラムを作成しなさい。

第9章　ファイルへの入出力(その3)

9.1　はじめに

ファイルの入出力のまとめとして、簡単なプロッタを作ることにしましょう。2 次元のデータ (二つの `double` 型) がファイルに書かれているときに、それを順に直線で結ぶものです。簡単な Tcl/Tk の GUI を付けましょう。

9.2　仕様

9.2.1　データ形式

`double` 型データが二つづつ組になって、データファイルに記述されているとします。それ以外のデータは一切含まれていないとします。作図結果は、PostScript ファイル `out.ps` に出力されるようにします。

9.2.2　作図形式

作図範囲は、X 軸及び Y 軸について、それぞれ最小値及び最大値を指定します。その範囲内のデータだけが表示されるようにします。

9.2.3　GUI

GUI からファイルを選択することで作図を行えるようにします。また、作図結果をプレビューする機能も持たせます。

Tcl/Tk との連係を計る方法として、Tcl/Tk の基本機能を持ったシェル `wish` に作図機能を追加したものを c++ で作成します。

9.3　ファイル構成

`Imakefile`

`main.tcl` Tcl/Tk のメインスクリプト

`MkListBox.tcl` リストボックスを生成するスクリプト

図 9.1: 簡易プロッタ

`Files.tcl` ファイル一覧を生成するスクリプト

`main.cc` 機能を追加した Tcl/Tk の新しいシェル `simpleplot` のメイン

`plot.h` プロット関数のヘッダファイル

`plot.cc` プロット関数のプログラム

`head.ps` 作図データの Postscript ヘッダファイル

`param.ps` 作図パラメタの Postscript ヘッダファイル

`mkdata.cc` 作図データ作成プログラム

9.4 新しい Tcl/Tk シェルを作る

Program 9.4.1 main.cc

```
// *** main.cc ************************************
//
// 簡易プロッタ用 main
//
// 「プログラミング概論 II」第 6 回講義資料
//  1999/11/24  作成
//  2000/11/8   改定
//************************************************
//
#include <stdlib.h>
#include <tk.h>
#include "plot.h"

int Tcl_AppInit(Tcl_Interp*);

int main(int argc,char *argv[])
{
  Tk_Main(argc,argv,Tcl_AppInit);
  exit (0);
}

int Tcl_AppInit(Tcl_Interp* interp)
{
  if (Tcl_Init(interp) == TCL_ERROR) return TCL_ERROR;
  if (Tk_Init(interp) == TCL_ERROR) return TCL_ERROR;

//新たに加える Tcl/Tk 関数を登録
  Tcl_CreateCommand(interp, "Plot", Plot,
                    (ClientData)Tk_MainWindow(interp),
    (Tcl_CmdDeleteProc *)NULL);
  return TCL_OK;
}
```

このメイン部では、新しい Tcl/Tk シェルを定義し、その Tcl/Tk シェルに新しい関数を追加するしています。新しい関数の追加を行う関数が Tcl_CreateCommand です。ここでは、データを作図する Plot を追加しています。

関数 Plot は、plot.h でプロトタイプ宣言が行われ、plot.cc で定義されています。

plot.cc では、まず、作図のための様々な Postscript のマクロの定義されたファイル head.ps とパラメタを指定する param.ps を作図の出力ファイル out.ps へコピーをしています。

データの総数は不明ですから、while(!fdata.eof()) でストリームの状態を見ながら、終端までデータを読みプロットします。

演習 9.1 コマンドラインからデータファイル名を指定すると、out.ps へ Postscript 作図データを出力するプログラムを作成しなさい。

Program 9.4.2 plot.h

```
// ************** plot.h ******************
// 2000/11/8  作成
// ****************************************
//新たに加える Tcl/Tk 関数の宣言
int Plot(ClientData clientData, Tcl_Interp* interp, int argc, char** argv);
```

Program 9.4.3 plot.cc

```
// ************** plot.cc ******************
// 2000/11/8  作成
// ***************************************
#include <stdlib.h>
#include <fstream.h>
#include <stdio.h>
#include <tk.h>
#include "plot.h"

//プロット関数
int Plot(ClientData clientData, Tcl_Interp* interp,
                 int argc, char** argv)
{
  char c;
  char line[1024];
  ofstream fw("out.ps");//出力ファイル

  //ヘッダファイルの読み込み
  sprintf(line,"%s/head.ps",argv[2]);
  ifstream fr(line);

  while(fr.get(c))fw.put(c);
  fr.close();

  //作図パラメタの設定
  sprintf(line,"%s/param.ps",argv[2]);
  fr.open(line);

  while(fr.get(c))fw.put(c);
  fr.close();

  fw <<"XAXIS YAXIS\n";
  fw <<"gsave Rangepath clip newpath\n";

  //データの作図
  ifstream fdata(argv[1]);

  double x,y;
  fdata>>x; fdata>>y;
  fw <<x<<" "<<y<<" M\n";
  while(!fdata.eof()){
    fdata>>x; fdata>>y;
    fw <<x<<" "<<y<<" L\n";
  }

  fw <<"stroke grestore showpage\n";

  return TCL_OK;
}
```

Program 9.4.4 main.tcl

```
#!simpleplot
source MkListBox.tcl
source Files.tcl

set TOPDIR [pwd]
set t [frame .topfmame]
pack $t
set bt [frame $t.bframe]
pack $bt
button $bt.quit -text QUIT -command exit
button $bt.prev -text PREVIEW -command prev
pack $bt.quit $bt.prev -side left

frame $t.file
pack $t.file
MkFileList $t.file

proc prev { } {
    exec /usr/X11R6/bin/gv out.ps
}
```

Program 9.4.5 Files.tcl

```
#ファイル操作
proc MkFileList {w} {
    mklistbox $w
    bind $w.listbox <Double-Button-1> {selectfile %W %y}
    list_file $w.listbox
}

proc list_file {w} {
    set flist [exec ls -a]
    $w delete 0 end
    foreach line $flist {
        if {[file isdirectory $line]} {
            $w insert end ${line}/
        } else {
            $w insert end ${line}
        }
    }
}

proc selectfile {w y} {
    global selectedfile
    global TOPDIR
    set i [$w nearest $y]
    set fname [$w get $i]
    if {[file isdirectory $fname]} {
        cd $fname
        list_file $w
    } else {
        if {[file isfile $fname]} {
            set selectedfile $fname
    Plot $selectedfile $TOPDIR
        }
    }
}
```

Program 9.4.6 MkListBox.tcl

```
#リストボックス作成

proc mklistbox {w} {
    scrollbar $w.xscroll -command "$w.listbox xview" -orient horizontal
    scrollbar $w.yscroll -command "$w.listbox yview" -orient vertical
    pack $w.xscroll -side bottom -fill x
    pack $w.yscroll -side right -fill y

    listbox $w.listbox -xscroll "$w.xscroll set" -yscroll "$w.yscroll set"
    pack $w.listbox -side left -fill both -expand true -anchor nw
}
```

Program 9.4.7 mkdata.cc

```
#include <stdlib.h>
#include <fstream.h>
#include <math.h>

int main(int argc,char** argv)
{
  ofstream fw("data");

  for (register int i=0;i<1000;i++){
    double x=100.*sin(0.01*double(i)+3.);
    double y=2.*cos(pow(0.01*double(i),3.));
    fw <<x<<" "<<y<<"\n";
  }
}
```

Program 9.4.8 Imakefile

```
/* Imakefile for

作成者：只木進一
作成日：1999/11/11


xmkmf -f で新しい Makefile を作成し
make でコンパイル実行
*/
SRCS=main.cc plot.cc mkdata.cc/* ソースファイル一覧 */
OBJS=main.o plot.o
TCLTKINCLUDE=-I/usr/local/include/tcl8.0jp  -I/usr/local/include/tk8.0jp
TCLTKLIB=-L/usr/local/lib -ltcl80jp -ltk80jp
CXXEXTRA_DEFINES=    /* コンパイラに渡す define */
EXTRA_INCLUDES=$(TCLTKINCLUDE)
CXXEXTRA_INCLUDES=$(EXTRA_INCLUDES)   /* コンパイラに渡す include path */

/* 作成するターゲットの一覧*/
AllTarget(simpleplot mkdata)
/* ソースファイルからオブジェクトファイルへのコンパイル方法の指定 */
NormalCplusplusObjectRule()
/* 実行形式ターゲットのコンパイル */
NormalCplusplusProgramTarget(simpleplot,$(OBJS),NullParameter,$(TCLTKLIB)   -lX11,-lm)
NormalCplusplusProgramTarget(mkdata,mkdata.o,NullParameter,NullParameter,-lm)
/* ファイルの依存関係の確認 */
DependTarget()
```

第10章　C++パーサを作る (その1)

10.1　はじめに

最後のまとめのプログラムとして、C++ プログラムの括弧の対応等をチェックするプログラムを作成しましょう。ここで基本となる技術は、ファイルから一文字づつ読み出すことと、左右の括弧の対応を確認するためのスタック操作です。

まず、括弧の対応を調べる方法を考えましょう。C/C++ では、()、{}及び [] の三種類の括弧が入れ子 (nesting) になって使われます。入れ子の深さはコンパイラによって制限がありますが、言語仕様としては制限がありません。このような入れ子構造を許す記号の対応を調べる方法として文脈自由文法 (Context Free Language) やプッシュダウンオートマトン (Pushdown Automaton) を使う方法があります。詳しくは、計算の理論の教科書などを参照してください。ここでは、必要な部分だけを説明します。

スタックは、先入れ後出しというアクセス制限のある無限メモリです。皿を積むことを考えてください。後から積んだ皿を先に取り出さなければ、下の皿は取り出すことはできません。このようなスタックを使って、括弧の対応を調べます。

先頭からプログラムを一文字づつ読み出しているとしましょう。問題としているのは括弧だけですから、それ以外の記号は読み飛ばします。左括弧を読んだ場合、それは新しい括弧の対応関係の始まりですから、スタックに入れます (push)。右括弧を読んだ場合、スタックの一番上に対応する左括弧があれば正しく括弧が閉じられたことになります。そこで、スタックの一番上の左括弧を取り去ります (pop)。もしも、対応する左括弧がスタックの一番上に無ければ、括弧の対応が誤っていることが分かります。

10.2　仕様

10.2.1　対象とするプログラム

C 及び C++ のプログラムを対象として、上記の三種類の括弧の対応を調べます。ただし、コメント文には括弧は無いとします。また、文字定数や文字列定数にも括弧は現れないとします。つまり、左括弧は必ず対応する右括弧と組になっていると仮定します。

10.2.2　解析内容

プログラム中の括弧の対応を調べます。その際、誤りがあった行番号を表示できるようにします。従って、単にスタックに括弧を保存するだけでは不足であり、同時にその括弧が現れた行番号

文字列	スタックの状態
$\bar{\{}(())\}$	Z_0
$\{\bar{(}())\}$	$\{Z_0$
$\{(\bar{(})())\}$	$(\{Z_0$
$\{(()\bar{(})()\}$	$((\{Z_0$
$\{(())\bar{(}\}$	$(\{Z_0$
$\{(()\bar{(})\}$	$((\{Z_0$
$\{(())\bar{)}\}$	$(\{Z_0$
$\{(())\bar{)}\}$	$\{Z_0$
$\{(())\}$	Z_0

表 10.1: 文字列 {(()())} を読む時の様子。¯ はこれから読む文字を示す。スタックの底には Z_0 が書かれている。

を保存することにします。

エラーとして考えられるのは、左括弧が対応する括弧で閉じられていない場合と、左括弧が閉じられる前にファイルの終端に到達する場合です。

10.2.3　実行方法

コマンドラインから解析するファイル名を指定できるようにします。

```
simpleparse file.cc
```

解析結果は標準エラーに出します。

10.3　スタックの設計

まず、左括弧を保存するスタックを設計しましょう。前期の講義で扱ったスタックを思い出してください。一つのデータを保存するノードクラスをまず作ります。ノードクラスは、保存するデータと次のノードへのポインタを持っています。スタック全体のクラスは一番上のノードを知っているだけです。スタックへのプッシュによって新しいノードが生成されスタックの一番上に積まれます。スタックからポップすることで、スタックの一番上のデータが取り除かれます。以前の例では、ポップするとスタックの一番上に保存されていた文字が返されました。

これから作成するプログラムでは、解析結果として、誤りのあった行番号を表示する必要があるので、括弧と同時にその括弧が現れたプログラムの行番号を保存します。そこで、クラス `Node` には、保存する括弧 `Node::t` だけでなく、その行番号 `Node::line` をメンバ変数として定義します。

スタックのクラス `Stack` は上記の `Node` クラスの列のうち、一番上の `Node` を保持しています。スタックからポップした場合には、括弧とその括弧が現れたプログラムの行番号を戻す必要があります。そこで、ポップの戻り値も `Node` クラスにします。

Program 10.3.1 stack.h

```
// *** stack.h ************************************
//
// 括弧などの対応を見るためのスタック
//
// 「プログラミング概論 II」第 7 回講義資料
//  1999/11/29  作成
//************************************************
class Node {
public:
  char t;
  int line;
  Node* next;
  Node(char c,int j){t=c;line=j;next=NULL;}
  Node(char c,int j,Node* n){t=c;line=j;next=n;}
};

class Stack {
private:
  Node* top;

public:
  Stack(void);
  ~Stack(void);
  int push(char c,int j);
  Node pop(void);
};
```

Program 10.3.2 stack.cc

```
// *** stack.cc ***********************************
//
// 括弧などの対応を見るためのスタック
//
// 「プログラミング概論 II」第 7 回講義資料
//  1999/11/29  作成
//************************************************
#include <stdlib.h>
#include <iostream.h>
#include "stack.h"

Stack::Stack(void) {top=new Node('\0',0);}// コンストラクタ

Stack::~Stack(void)// デストラクタ
{
  int id=1;
  while(id){
    if(top==NULL)id=0;
    else{
      Node node=pop();
      if(node.t=='\0')id=0;
    }
  }
}
```

Program 10.3.3 stack.cc(その 2)

```
int Stack::push(char c,int j)//新しいレコードの登録
{
#ifdef DEBUG
  cerr <<"push:"<<c<<"\n";
#endif
  Node* newnode=new Node(c,j,top); top=newnode;
  return 1;
}

Node Stack::pop(void){// トップにあるノードを返す
  char it = top->t;
#ifdef DEBUG
  cerr <<"pop :"<<it<<"\n";
#endif
  int line=top->line;
  Node* next=top->next;
  delete top;
  top=next;
  return Node(it,line);
}
```

10.4　解析プログラムの本体

関数 main の引数 argc と argv は、実行時のコマンドライン引数の数とその内容を表します。今回のプログラムは、解析対象のファイル名を引数とするので、argc は 2 で、そのファイル名は argv[1] に格納されています。ちなみに、argv[0] には、プログラム名そのもの、今回の場合には simpleparser が入っています。

解析対象のファイル名を渡されたら、そのファイルをストリームに接続し、開けることができるかを ifstream::bad() で調べます。

exit で与える値は、/usr/include/sysexits.h で定義されているので、その値を使います。

解析のメイン部分は関数 parse です。fr.get で一文字づつ読みだし、文字の種類でスタックの操作をします。ここでは文字の種類ごとの条件判断を switch 文で行っています。

Program 10.4.1 main.cc(その 1)

```
// *** main.cc ************************************
//
// 括弧などの対応を見るためのプログラム
//
// 「プログラミング概論 II」第 7 回講義資料
//  1999/11/30  作成
//************************************************
#include <stdlib.h>
#ifdef WIN32
#include "sysexits.h"
#else
#include <sysexits.h>  //exit コード
#endif
#include <fstream.h>
#include <string.h>
#include "stack.h"

#ifdef WIN32
char *strdup( const char *string ){return _strdup(string );}
#endif

//プロトタイプ宣言
int parse(ifstream&);
int b_close(char,int,char,Stack*);
int parse_error(char,int,char,int);
int parse_error(char,int);
int parse_ok(char,int,char,int);

int main(int argc,char** argv)
{
  if(argc!=2){//必ず引数にファイル名を指定する
    cerr << "正しい使い方:" <<argv[0]<<" ファイル名\n";
    exit(EX_USAGE);
  }

  //入力ファイルの指定
  char* filename=strdup(argv[1]);
  ifstream fr(filename);
  if(fr.bad()){
    cerr <<argv[1]<<" を開くことができません。\n";
    exit(EX_NOINPUT);
  }

  parse(fr);
  exit(EX_OK);
}
```

Program 10.4.2 main.cc(その 2)

```
int parse(ifstream& fr)
{
  Stack stack;
  int n=1;//行数のカウンタ
  char c;
  while(fr.get(c)){//一文字読み込む
    switch(c){
    case '\n'://改行
      n++;
      break;

//括弧開く
    case '(':
    case '{':
    case '[':
      {
        Node node=stack.pop();
        stack.push(node.t,node.line);stack.push(c,n);
      }
      break;

//括弧閉じる
    case ')':
      b_close(c,n,'(',&stack); break;

    case '}':
      b_close(c,n,'{',&stack); break;

    case ']':
      b_close(c,n,'[',&stack); break;

    default:
      break;
    }
  }

  int id=1;
  while(id){//括弧の閉じ忘れの確認
    Node node=stack.pop();
    if(node.line!=0)parse_error(node.t,node.line);
    else id=0;
  }
  return 1;
}
```

関数 b_close は括弧の対応を調べる副関数です。対応が取れていなければエラーメッセージを出す関数 parse_error を呼びます。

エラーメッセージを出す関数 parse_error は、引数の数の異なるものが二種類あります。括弧の対応が誤っている場合と、右括弧が対応していない左括弧がある場合です。

Program 10.4.3 main.cc(その 3)

```
int b_close(char c,int n,char b,Stack* stack)//括弧の対応を調べる
{
  Node node=stack->pop();
  if(node.t!=b){
    parse_error(c,n,node.t,node.line);exit(EX_OK);}
  else {
#ifdef DEBUG
    parse_ok(c,n,node.t,node.line);
#endif
  }
  return 1;
}

int parse_error(char a,int n,char b,int m)//括弧の閉じ方が誤っている場合
{
  cerr <<m<<" 行の記号 "<<b<<" が";
  cerr <<n<<" 行の記号 "<<a<<" で誤って閉じられています\n";
  return 1;
}

int parse_error(char a,int n)//括弧が閉じられていない場合
{
  cerr <<n<<" 行の記号 "<<a<<" が閉じられていません。\n";
  return 1;
}

int parse_ok(char a,int n,char b,int m)//括弧が正しく閉じられている場合
{
  cerr <<m<<" 行の記号 "<<b<<" が";
  cerr <<n<<" 行の記号 "<<a<<" で閉じられています\n";
  return 1;
}
```

10.5 コンパイルのために

今回のプログラムはスタッククラスのプログラム stack.cc とメインプログラム main.cc から構成されています。また、デバッグ用の cpp ディレクティブを含んでいます。そこで、Imakefile を使って、効率良くコンパイルをします。

Program 10.5.1 Imakefile

```
/* Imakefile for

作成者：只木進一
作成日：1999/11/30


xmkmf -f で新しい Makefile を作成し
make でコンパイル実行
*/
SRCS=main.cc stack.cc/* ソースファイル一覧 */
OBJS=main.o stack.o
EXTRA_DEFINES=
CXXEXTRA_DEFINES= $(EXTRA_DEFINES)  /* コンパイラに渡す define */
EXTRA_INCLUDES=
CXXEXTRA_INCLUDES=$(EXTRA_INCLUDES)  /* コンパイラに渡す include path */

/* 作成するターゲットの一覧*/
AllTarget(simpleparse)
/* ソースファイルからオブジェクトファイルへのコンパイル方法の指定 */
NormalCplusplusObjectRule()
/* 実行形式ターゲットのコンパイル */
NormalCplusplusProgramTarget(simpleparse,$(OBJS),NullParameter,NullParameter,-lm)
/* ファイルの依存関係の確認 */
DependTarget()
```

Imakefile を用意して

```
xmkmf -a
```

を実行すると Makefile が生成されます。UNIX のユーティリティーである make は、ファイルの依存関係を調べ、コンパイルを効率良く行うツールです。Makefile ができていれば

```
make
```

でコンパイルを実行することができます。また、

```
make clean
```

で、実行結果を全て消去することもできます。今回のプログラムでは、デバッグ用のコードが書かれています。デバッグを有効にするには

```
make CXX="g++ -DDEBUG"
```

とします。

コンパイルが成功すると実行可能プログラム simpleparser が生成されます。

```
simpleparser stack.cc
```

などとファイル名を指定すると解析を行うことができます。

Program 10.5.2 sysexits.h

```
#define EX_OK           0       /* successful termination */

#define EX__BASE        64      /* base value for error messages */

#define EX_USAGE        64      /* command line usage error */
#define EX_DATAERR      65      /* data format error */
#define EX_NOINPUT      66      /* cannot open input */
#define EX_NOUSER       67      /* addressee unknown */
#define EX_NOHOST       68      /* host name unknown */
#define EX_UNAVAILABLE  69      /* service unavailable */
#define EX_SOFTWARE     70      /* internal software error */
#define EX_OSERR        71      /* system error (e.g., can't fork) */
#define EX_OSFILE       72      /* critical OS file missing */
#define EX_CANTCREAT    73      /* can't create (user) output file */
#define EX_IOERR        74      /* input/output error */
#define EX_TEMPFAIL     75      /* temp failure; user is invited to retry */
#define EX_PROTOCOL     76      /* remote error in protocol */
#define EX_NOPERM       77      /* permission denied */
#define EX_CONFIG       78      /* configuration error */

#define EX__MAX 78      /* maximum listed value */
```

第11章　C++パーサを作る(その2)

11.1　はじめに

前章では、括弧の対応だけを調べるC/C++パーサを作成しました。今回は、引用符の対応も調べることができるように拡張しましょう。また、前回は、括弧が文字列定数やコメント内に現れた場合には対応できませんでした。今回は、解析する文字列が文字列定数やコメント内に現れる場合も想定してプログラムしていきます。

11.2　設計

基本的な仕様は、前回と同様ですが、上述のように、括弧の対応を調べるとともに、文字列定数やコメント内に解析対象が現れた場合にも対応できるようにします。

スタックの仕様は前回と同様です。今回は、三種類の括弧の他に二種類の引用符"と’もスタックに入れます。

コメントは、二つの方法

```
//コメント
```

と

```
/*
コメント
*/
```

に対応できるようにします。

11.3　引用符への対応

引用符への対応を入れる場合、二つのことを考える必要があります。引用符は文字定数用の’と文字列定数用の"があります。

第一に考えるべきことは、C/C++ のプログラムの中で、文字定数としての引用符をどのように記述するかです。もちろん

```
char c=’’’;
```

は正しくありません。文字定数として"や’を定義する場合、バックスラッシュ\を使って表します。

```
char double_quote='\"';
char single_quote='\'';
```

文字列の中でも同様です。

注意しなければならないのは、プログラムを単なるファイルとして見ると、これらの記号は二つの文字、つまり\と"や' に分かれて読み込まれるということです。つまり、引用符の確認をする場合には、注目している文字と一つ前の文字の組合せで引用符そのものか、あるいは文字定数としての引用符かを判断する必要があります。

注意の第二は、引用符の対応の確認と、引用符内での括弧や文字定数としての引用符を確認対象から除外する方法です。対応を確認するので、文字定数及び文字列定数の始まりを表す左側の引用符をスタックにプッシュする必要があります。文字定数及び文字列定数の終りを表す右側の引用符が現れると、スタックから取り出します。この間に現れる括弧は文字定数としてのものなので無視する必要があります。

11.4　コメントへの対応

次にコメントの扱いを考えましょう。まず

```
//コメント
```

型のコメントについて考えましょう。記号列//が現れると、その行の行末までがコメントになります。そこで、記号列//が現れた場合、適当なフラッグを立てて一時的に解析を停止するとともに、行末まで読み飛ばす仕組みを作りましょう。

行末まで読み飛ばす必要があるので、今回はファイルから一文字ずつ読み込むのではなく、一行単位で読み込むことにします。

つぎに、

```
/*
コメント
*/
```

型のコメントについて考えましょう。記号列/*が現れた後、記号列*/が現れるまでがコメントになります。その間は解析を停止しなければなりません。そこで、このコメントの開始及び終了に対応したフラッグも用意しましょう。

11.5 メインプログラム

Program 11.5.1 stack.h

```
// *** main.cc ************************************
//
// 括弧などの対応を見るためのプログラム
//
// 「プログラミング概論 II」第 8 回講義資料
//  1999/12/7  作成
// ************************************************
#include <stdlib.h>
ifdef WIN32
#include "sysexits.h"
#else
#include <sysexits.h>
#endif
#include <fstream.h>
#include <string.h>
#include "stack.h"

#ifdef WIN32
char *strdup(const char *string){return _strdup(string);}
#endif

int parse_main(ifstream&);
int parse(char,char,int,Stack*);
int b_open(char,int, Stack*);
int b_close(char,int,char,Stack*);
int q_close(char,int,char,Stack*);
int parse_error(char,int,char,int);
int parse_error(char,int);
int parse_ok(char,int,char,int);

int main(int argc,char** argv)
{
  if(argc!=2){//必ず引数にファイル名を指定する
    cerr << "正しい使い方:" <<argv[0]<<" ファイル名\n";
    exit(EX_USAGE);
  }

  //入力ファイルの指定
  char* filename=strdup(argv[1]);
  ifstream fr(filename);
  if(fr.bad()){
    cerr <<argv[1]<<" を開くことができません\n";
    exit(EX_NOINPUT);
  }

  parse_main(fr);
  exit(EX_OK);
}
```

解析の主たる部分を今回は二つの副プログラム parse_main と parse に分けます。関数 parse_main はファイルから一行読み出して、コメント中で無ければ一文字ずつ関数 parse に渡し、括弧及び引用符の対応を調べます。

関数 parse_main では、一行ずつ ifstream::getline によって文字列 line にファイルの内容が読み込まれます。一行読み込まれる度に行数のカウンタ n が増やされるとともに、//によるコメント中を表すフラッグ id_1 が 1 に戻されます。

Program 11.5.2 main.cc(その 2)

```
int parse_main(ifstream& fr)//解析プログラムの主部分
{
  Stack stack;
  int n=0;//行数のカウンタ
  const int L=256;
  char line[L];

  int id_2=1;// コメントタイプ 2
  while(fr.getline(line,L)){
    n++;
    char* s=line;
    char cp='\0';//一つ前の文字
    int id_1=1;// コメントタイプ 1
    while(*s!='\0'&&id_1==1){
      char c=*s;
      //コメントタイプ 1
      if(id_2==1 && cp=='/' && c=='/')id_1=0;

      //コメントタイプ 2
      if(id_1==1){
        if(cp=='/' && c=='*')id_2=0;
        if(id_2==0 && cp=='*' && c=='/')id_2=1;
      }

      //コメント以外の場合に解析
      if((id_1*id_2)==1)parse(c,cp,n,&stack);
      cp=c;
      s++;
    }
  }

  //閉じられていない括弧や引用符の確認
  int id=1;
  while(id){
    Node node=stack.pop();
    if(node.line!=0)parse_error(node.t,node.line);
    else id=0;
  }
  return id;
}
```

一行の内容を保持する文字列 line から一文字ずつ変数 c に入れると同時に、一つ前の文字を cp

に入れます。文字列//が現れれば、行末までコメントと判断し、id_1=0 とします。文字列/*が現れれば、文字列*/が現れるまでコメントと判断し、id_2=0 とします。コメント中でなければ解析ルーチン parse へ、変数 c と cp 及び行番号 n を渡します。

Program 11.5.3 main.cc(その 3)

```
int parse(char c,char cp,int n,Stack* stack)//解析プログラムの副部分
{
  switch(c){
    //括弧開く
  case '(':
  case '{':
  case '[':
    b_open(c,n,stack); break;

    //括弧閉じる
  case ')':
    b_close(c,n,'(',stack); break;

  case '}':
    b_close(c,n,'{',stack); break;

  case ']':
    b_close(c,n,'[',stack); break;

    //引用符
  case '\"':
    if(cp!='\\')q_close(c,n,'\"',stack); break;

  case '\'':
    if(cp!='\\')q_close(c,n,'\'',stack); break;

  default:
    break;
  }

  return 1;
}
```

解析の実行部分 parse の中では、場合分けが行われます。左括弧の場合には関数 b_open が、右括弧の場合には関数 b_close が、引用符の場合には関数 q_close が呼ばれます。

左括弧の場合 (関数 b_open)、スタック上に引用符があれば文字定数であると判断して、スタックにプッシュしません。スタック上に引用符が無ければ、その左括弧をスタックにプッシュします。

右括弧の場合 (関数 b_close)、スタック上に引用符があれば文字定数として無視します。そうでない場合には、対応する左括弧がスタックにあることを確認します。

引用符の場合 (関数 q_close)、直前の文字 cp が\であれば文字定数であるので無視します。そうでない場合には、対応する引用符がスタックにあることを確認します。

Program 11.5.4 main.cc(その 4)

```
int b_open(char c,int n,Stack* stack)
{
  Node node=stack->pop();
  if(node.t=='\"' || node.t=='\'')//文字列定数内の場合
    {stack->push(node.t,node.line);}
  else {stack->push(node.t,node.line);stack->push(c,n);}
  return 1;
}

int b_close(char c,int n,char b,Stack* stack)//閉じ括弧の確認
{
  Node node=stack->pop();
  if(node.t=='\"' || node.t=='\'')//文字列定数内の場合
    {stack->push(node.t,node.line);}
  else
    if(node.t!=b){
      parse_error(c,n,node.t,node.line);exit(EX_OK);}
    else {
#ifdef DEBUG
      parse_ok(c,n,node.t,node.line);
#endif
    }
  return 1;
}

int q_close(char c,int n,char b,Stack* stack)//引用符号の確認
{
  Node node=stack->pop();
  if(node.t==b){
#ifdef DEBUG
    parse_ok(c,n,node.t,node.line);
#endif
  }
  else {stack->push(node.t,node.line);stack->push(c,n);}
  return 1;
}

//メッセージ
int parse_error(char a,int n,char b,int m)
{
  cerr <<m<<" 行の記号 "<<b<<" が";
  cerr <<n<<" 行の記号 "<<a<<" で誤って閉じられています\n";
  return 1;
}
int parse_error(char a,int n)
{
  cerr <<n<<" 行の記号 "<<a<<" が閉じられていません。\n";
  return 1;
}

int parse_ok(char a,int n,char b,int m)
{
  cerr <<m<<" 行の記号 "<<b<<" が";
  cerr <<n<<" 行の記号 "<<a<<" で閉じられています\n";
  return 1;
}
```

第12章　HTMLパーサを作る

12.1　はじめに

前回までは、C/C++において括弧の対応などを調べるプログラムを作成しました。その場合、一文字からなる記号 (括弧) の対応を調べるだけで済みました。今回は、文字列からなる記号の対応を調べるプログラムを作りましょう。例として、WWW(World Wide Web) の記述言語であるHTML(Hyper Text Markup Language) を取り上げます。

12.2　HTML小論

HTML のタグは<タグ名>という形で記述されます。タグには二つの種類があります。第一は、開始と終了があり、その間の属性を記述するものです。幾つか例を示します。

まず、文字の大きさを変化させる<Hn>という形があります。n には自然数が入り、大きな数ほど小さな文字が使われます。

```
<H1>
おおきな文字
</H1>
```

属性の指定された領域の終了が</Hn>というタグ名の前にスラッシュが付いてたタグであることに注意してください。次の例は、アンカーと呼ばれ、他のページへのリンクを記述するものです。

```
<A HREF="http://www.is.saga-u.ac.jp">知能情報システム学科</A>
```

この場合、タグ A の後ろにスペースが空いて、オプションが記述されています。属性の終了が</A>であることに注意してください。

このように、一般に

```
<タグ名 オプション並び>
文字列
</タグ名>
```

という形で属性変化が行われます。LaTeX で begin と end でグループを作るに似ています。

HTML のタグには、上述のように、開始と終了で組で現れないものもあります。例えば、箇条書きの項目 (LaTeX の場合の\item に相当) を表す<LI> や、段落の終りを表す<P>、強制的な改行を表す
などがあります。今回は以下のものについて、単独で現れるタグとして定義しておきましょう。

<P>,
, <LI>, <DT>, <DD>, <META>, <HR>

HTML 文書にもコメントを書くことができます。

```
<!--
コメント
//-->
```

最後に、HTML 文書においては、タグ名の大文字小文字の区別が無いことに注意しておきます。

Program 12.2.1 stack.h

```
// *** stack.h ************************************
//
// 括弧などの対応を見るためのスタック
//
// 「プログラミング概論 II」第 7 回講義資料
//  1999/11/29  作成
//************************************************
class Node {
public:
  char* t;
  int line;
  Node* next;
  Node(char* c,int j);
  Node(char* c,int j,Node* n);
};

class Stack {
private:
  Node* top;

public:
  Stack(void);
  ~Stack(void);
  int push(char* c,int j);
  Node pop(void);
};
```

12.3　設計

今回のプログラムでは、タグを表す括弧<及び>は入れ子にならないとしましょう。しかし、タグそのものは入れ子になります。例えば次のような場合はあり得ます。

<H2><A HREF="http://www.is.saga-u.ac.jp"> ホーム</A></H2>

そこで、開始と終了のあるタグについては、スタックを使って対応関係を調べることにします。したがって、今回のスタックは、文字列を保存するようなスタックになります。

タグは複数行に跨ることもあり得ます。そこで、ファイルから一文字ずつ読み、タグの開始<からタグの終り>までの文字列を保持する仕組みが必要です。タグが切り取れたら、スタックの操作を行います。

スタックにはタグの開始<からタグの終り>の文字列を積んでも問題ありません。しかし、タグの終了との確認には注意が必要です。つまり、領域の開始を表すタグはオプションを含むことと、大文字小文字の区別が無いことです。そこで、タグの対応を調べる際には、オプション部分を外し、更に全て大文字に変換して比較することにしましょう。

12.4 スタック

今回は、タグという文字列を保存する必要があります。前回のスタックを改良します。

Program 12.4.1 stack.cc

```
// *** stack.cc ************************************
//
// 括弧などの対応を見るためのスタック
//
// 「プログラミング概論 II」第 7 回講義資料
//  1999/11/29  作成
//*************************************************
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include "stack.h"

Node::Node(char* c,int j){t=strdup(c);line=j;next=NULL;}
Node::Node(char* c,int j,Node* n){t=strdup(c);line=j;next=n;}

Stack::Stack(void) {top=new Node("0",0);}//コンストラクタ

Stack::~Stack(void)// デストラクタ
{
  int id=1;
  while(id){
    if(top==NULL)id=0;
    else{
      Node node=pop();
      if(strcmp(node.t,"\0")==0)id=0;
    }
  }
}
```

12.5 タグの操作

関数 parse がタグの切り出し及び解析の主プログラムになっている。

最初に、関数 gettag がタグを切り出す。一文字ずつ入力を得て、<から>までをタグとして切り

Program 12.4.2 stack.cc(その 2)

```
int Stack::push(char* c,int j)//新しいレコードの登録
{
#ifdef DEBUG
  cerr <<"push:"<<c<<"\n";
#endif
  Node* newnode=new Node(c,j,top); top=newnode;
  return 1;
}

Node Stack::pop(void){// トップにあるノードを返す
  char* it = top->t;
#ifdef DEBUG
  cerr <<"pop :"<<it<<"\n";
#endif
  int line=top->line;
  Node* next=top->next;
  delete top;
  top=next;
  return Node(it,line);
}
```

出す。<が現れてタグのに入ると id=1 となり、タグの終了>で id=-1 を返す。タグが一つ切り出されると、そのタグは checktag に送られる。

関数 checktag では、関数 gettagname を使って、オプションが切り取られ、全て大文字に変換される。次に関数 clasifytag でタグの種類が調べられる。タグの種類は、コメント、単一で使われるタグ、開始タグ及び終了タグの 4 種類に分けられる。開始タグであればスタックに積まれ、終了タグであればタグの対応が関数 t_close で調べられる。

12.6　メイン

Program 12.5.1 parse.cc

```
// *** parse.cc ************************************
//
// HTML のタグの対応の解析
//
// 「プログラミング概論 II」第 9 回講義資料
//  1999/12/13  作成
// ************************************************
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include "stack.h"
int gettag(char c,char string[],int id);
int checktag(char line[],int,Stack& stack);
int t_close(char* tagname,int, Stack& stack);
int clasifytag(char* tag);

enum {tag_comment,tag_close,tag_open,tag_single};//  タグの種類
//閉じなくて良いタグ
const int ntags=7;
const char* singletags[ntags]={"<P>","<BR>","<LI>","<HR>","<META>",
     "<DT>","<DD>"};
int clasifytag(char* tag)//タグの分類
{
  if(tag[1]=='!')return tag_comment; //コメント
  if(tag[1]=='/')return tag_close;   //タグ閉じる
  for(register int i=0;i<ntags;i++)  //閉じなくて良いタグ
    if(strcmp(tag,singletags[i])==0)return  tag_single;
  return tag_open;                   //タグ開く
}


int parse(char c,char line[],int nline,int id,Stack& stack){
  if((id=gettag(c,line,id))==-1){// タグを得る
    checktag(line,nline,stack);  //タグを確認する
    id=0;
  }
  return id;
}
```

Program 12.5.2 parse.cc(その 2)

```
int gettag(char c,char string[],int id)
//一文字ずつ切り出して string にタグを格納する
{
  char buf[2];
  buf[1]='\0';
  switch (c) {
  case '<':
    string[0]='<';string[1]='\0';
    id=1;
    break;
  case '>':
    buf[0]='>';
    strcat(string,buf);
    id=-1;
    break;
  default:
    if(id==1){buf[0]=c;strcat(string,buf);}
    break;
  }
  return id;
}

int checktag(char line[],int nline,Stack& stack)//タグの確認
{
  char* gettagname(char []);
  char* tagname=gettagname(line);
  switch(clasifytag(tagname)){
  case tag_comment: //コメント
  case tag_single:  //単一で使われるタグ
    break;
  case tag_open:    //タグ始め
    stack.push(line,nline);
    break;
  case tag_close:   //タグ終り
    t_close(line,nline,stack);
    break;
  default:
    break;
  }
}
```

Program 12.5.3 parse.cc(その 3)

```
char* gettagname(char line[])//大文字のタグの名前を切り出す
{
  char* tag=strdup(line);
  char* s=tag;
  int id=1;
  while(*s!='\0' && id==1){
    if(*s==' '){*s='>';*(s+1)='\0'; id=0;}
    if(*s=='>'){*(s+1)='\0'; id=0;}
    else
      if(*s!='/')
if(*s>=97 && *s<=122){
  *s-=32;
}
    s++;
  }
  return tag;
}

int t_close(char* line,int nline,Stack& stack)
//タグが閉じられているか確認
{
  extern int parse_error(char* a,int n,char* b,int m);
#ifdef DEBUG
  extern int parse_ok(char* a,int n,char* b,int m);
#endif
  Node node=stack.pop();
  char* opentag=gettagname(node.t);
  char* closetag=gettagname(line);
  if(strcmp((closetag+2),(opentag+1))!=0)
    parse_error(line,nline,node.t,node.line);
#ifdef DEBUG
  else
    parse_ok(line,nline,node.t,node.line);
#endif
  return 1;
}
```

Program 12.6.1 main.cc

```
// *** main.cc ************************************
//
// HTML のタグの対応をみるプログラム
//
// 「プログラミング概論 II」第 9 回講義資料
//  1999/12/13  作成
// ************************************************
#include <stdlib.h>
#include <sysexits.h>
#include <fstream.h>
#include <string.h>
#include "stack.h"
int parse(char c,char line[],int,int id,Stack&);
int parse_error(char* a,int n,char* b,int m);
int parse_error(char* a,int n);
int parse_ok(char* a,int n,char* b,int m);

int main(int argc,char** argv)
{
  if(argc!=2){//必ず引数にファイル名を指定する
    cerr << "正しい使い方:" <<argv[0]<<" ファイル名\n";
    exit(EX_USAGE);
  }

  //入力ファイルの指定
  char* filename=strdup(argv[1]);
  ifstream fr(filename);
  if(fr.bad()){
    cerr <<argv[1]<<" を開くことができません\n";
    exit(EX_NOINPUT);
  }

  char line[256];
  char buf[256];
  Stack stack;
  int nline=0;
  int id=0;
  while(fr.getline(buf,256)){
    nline++;
    char* s=buf;
    while(*s!='\0'){
      parse(*s,line,nline,id,stack);
      s++;
    }
  }
  //閉じられていないタグの確認
  id=1;
  while(id){
    Node node=stack.pop();
    if(node.line!=0)parse_error(node.t,node.line);
    else id=0;
  }
  exit(EX_OK);
}
```

Program 12.6.2 main.cc(その 2)

```
//メッセージ
int parse_error(char* a,int n,char* b,int m)
{
  cerr <<m<<" 行の記号 "<<b<<" が";
  cerr <<n<<" 行の記号 "<<a<<" で誤って閉じられています\n";
  return 1;
}
int parse_error(char* a,int n)
{
  cerr <<n<<" 行の記号 "<<a<<" が閉じられていません。\n";
  return 1;
}

int parse_ok(char* a,int n,char* b,int m)
{
  cerr <<m<<" 行の記号 "<<b<<" が";
  cerr <<n<<" 行の記号 "<<a<<" で閉じられています\n";
  return 1;
}
```