Programming C++
Lecture Note 4
アルゴリズムをライブラリにする

Jie Huang

参照型変数とパラメータ

- 参照型変数は他の変数の別名として働く

```
float x;
float &a = x;
```

ここで、”&”は変数aはxの別名であることを示す。

- 参照型変数は元の変数と全く同一のもので、メモリ上の同じ場所を占める。従って、別名に値を代入すれば、元の変数も同じようになる。
例えば、

```
a = 3.14;
```

とすると、変数xの値も3.14となる。

- 参照型と似た働きのものとして、ポインタを使うことができる

```
float x;
float *a = &x; //ここの”&”はxのアドレスを示す
*a = 3.14;
```

- 別名は一旦定義した後変更できないのに対して、ポインタは別の変数を指すように変更することができる。

参照型変数とパラメータ

- 参照型変数は必ず同一タイプの他の変数（左辺値）で初期化する必要がある。
- また、初期化しないで宣言することも、左辺値でない式で初期化することもできない。

```
float &c;          // error
float &d = 10;     // error
float &e = x+1;    // error
```

- 左辺値:
  一時的なオブジェクトではない、代入式の左側となれるもの（変数、参照、参照を返す関数の戻り値など）
  例えば以下のような文はx+1は左辺値ではないのでエラーとなる。
  x+1 = 10; // error

参照型変数パラメータ

- C++では、関数のパラメータを参照型にすることができる。

```
#include <iostream.h>
void inc(int &a){ a += 1; }
main(){
   int x = 10;
   inc(x);
   cout << x;
}
```

上のプログラムでは、関数incは参照型のパラメータを持つ。inc(x)と呼び出した時には、aは変数xで初期化されるので、関数呼び出しの結果、元の変数のxの値がインクリメントされる。

- 呼び出しにおける初期化のメカニズムからも推測できるように、参照型パラメータも普通の参照型変数と同じく、定数や式など左辺値でないものを使って呼び出すことはできない。

```
inc(10);   // error
inc(x+1);  // error
```

参照型変数パラメータ

- 参照パラメータに関する補足
  参照パラメータに対する引数は左辺値でなければならない。一時的なアドレスを持った変数は左辺値ではないので、

```
vector<double> emptyvec() {
  vector<double> v;  return v;
}
```

  は以下の関数に引数として渡すことができない。

```
read_hw(cin, emptyvec());   // error
```

  ただし、const指定の参照パラメータに渡す引数は値を代入することがないので、非左辺値でもかまわない(コンパイルエラーにはならない)。
  例えば、

```
  grade(midterm, final, emptyvec());
```

参照型変数とパラメータ

- 参照型戻り値
  以下のプログラムを考える

```
int x[10];
int &f(int i){
  return x[i];
}
```

  関数fの戻り値はint型への参照型として定義されている。関数実行の結果はx[i]への参照型となる。従って、
  f(1) = 1;
  のように、関数コールの結果に値を代入することができるようになる。
- 似た働きをポインタを使って実現すると以下のようになる。

```
int *f(int i){
  return &x[i];
}
*f(1) = 1;
```

参照型変数とパラメータ

- 参照型パラメータのメリットとデメリット
  - メリット
    - 参照型パラメータはコピーを作らないので、関数を高速化することができる。
    - 参照型パラメータは呼び出し側の変数の別名となるので、引数に値を代入することによって、呼び出し側の変数の値を変えることができる。
  - デメリット
    - 定数や式の結果を渡すことができない。
    - 関数の内で、呼び出し側の変数に影響を与えるので、関数の独立性が損なわれる可能性がある（これは変数をconst宣言することで回避することができる）。

関数をライブラリにする

- ライブラリの持つ性質
  例: vector, string, sortなど
  - 特定の問題を解決するためにある
  - 他の大部分のプログラムと独立している
  - ライブラリ名が必要である

関数をライブラリにする

- gradeという関数を作る
  - アルゴリズムの解決する問題は？
  - 他のプログラムと独立にできるか？
  - 名前は関数の機能を反映できるか？

  例:

```
double grade(double midterm, double final,
  double homework){
  return 0.2*midterm+0.4*final+0.4*homework;
}
```

関数をライブラリにする

- テンプレートを使った関数定義

```
double median(vector<double> vec) {
  typedef std::vector<double>::size_type vec_sz;
  vec_sz size = vec.size();
  if (size == 0)
    throw domain_error("median of an empty vector");
  sort(vec.begin(), vec.end());
  vec_sz mid = size/2;
  return size % 2 == 0 ?
    (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}
```

- テンプレートは変数の型をパラメーターとするので、テンプレートのパラメータを変えるだけで、いろんなタイプのデータを定義することができる。

```
vector<double> vec; //double型要素のvectorオブジェクト
```

関数をライブラリにする

- 多重定義を利用して関数にラップをかける

```
double grade(double midterm, double final,
             const vector<double>& hw) {
  if (hw.size() == 0)
    throw domain_error("student has done no homework");
  return grade(midterm, final, median(hw));
}
double grade(const Student_info& s) {
  return grade(s.midterm, s.final, s.homework);
}
```

- grade関数の階層構造で大きな処理を分解する
  - `double grade(const Student_info& s);`
  - `double grade(double midterm, double final,`
    `const vector<double>& hw);`
  - `double grade(double midterm, double final, double homework);`

関数をライブラリにする

- エラー処理の問題点
  - 戻り値でエラーを表すのは限界がある。例えばよく戻り値-1でエラーの発生を表すが、マイナスの値を返す関数では使えない。
  - エラー処理の部分と通常の処理の部分が混在するプログラムは分かりにくい。
  - 一般的にエラーの検出は下位の関数で、エラーに対応する処理は上位の関数で行う。エラー検出側の関数でエラー処理の全てを行うのが一般的に不可能である。
  - 同じエラーに対してもそれぞれの高レベルの関数では異なる処理を行わなければならない場合はよくある。
  - モジュールの観点からでも、エラー検出主体とエラー処理主体が高い独立性を維持すべきである。自分を呼び出した不特定の実行主体がエラー処理に関わる可能性があるからである。

関数をライブラリにする

- try - throw - catch によるエラー処理

```
vector<double> homework;
read_hw(cin, homework);
try { //throwはこのブロックの中で発生する
  double final_grade = grade(midterm, final, homework);
  streamsize prec = cout.precision();
  cout << “Your final grade is ” << setprecision(3)
       << final_grade << setprecision(prec) << endl;
} catch (domain_error) {
  cout << endl << “You must enter your grades. ”
          “Please try again.” << endl; return 1;
}
return 0;
```

- try:実行部分
  throw:エラーの発生を知らせる
  catch:エラーの発生をキャッチして、エラー処理を行う部分

関数をライブラリにする

- try-throw-catchエラー処理の流れ
  - throwが発生すると、まずthrowのオブジェクトをセーブし、現在のtryブロックの終わりの部分を探す。
  - 現在の関数内にtryブロックが見付からない場合は、制御は上位関数へ移る。さらに上位関数でもtryブロックが見付からない場合はさらに上位関数へ制御が移る。
  - tryブロックが見付かると、今度はtryブロックのすぐ後ろのcatch記述を探す。throwで送出されるオブジェクトと型が一致する場合は、制御はcatchの部分に移る。一致しない場合は、次のcatch記述を探す。
  - どのcatchとも一致しない場合は、プログラムは次のレベルのブロックで適切なcatchを探し続ける。最後に一致しない場合はプログラムは終了する。

関数をライブラリにする

- try-throw-catchエラー処理の流れ

```
#include <iostream>
using namespace std;
void f2() { throw “no data”; }
void f1() {
  try {f2();} catch (int n) {cout << “f2 error!” << endl;}
}
int main(){
  try{f1();}
  catch(char *s){cout << “f1 error:(” << s << “)!” << endl;}
  return 0;
}
```

この例では、”f1 error(no data)!”が出力される。

- 目的の型のcatchが実行されると、他のcatchブロックは無視されるので、同じ型のcatchブルックを二重には定義できない。
- 全ての例外を受け取るcatch

```
catch(...){ここでデフォルトの例外処理を行う}
```

関数をライブラリにする

- throwの再発生

```
#include <iostream>
using namespace std;
void f2() { throw “no data”; }
void f1() {
  try {f2();} catch (char *s) {
    cout << “f2 error(” << s <<“)!” << endl;
    throw; //同じ型のthrowの再発生
  }
}
int main(){
  try{f1();}
  catch(char *s){cout << “f1 error:(” << s << “)!” <<
endl;}
  return 0;
}
```

この例では、”f2 error(no data)!f1 error(no data)!”が出力される。

関数をライブラリにする

- ストリームからデータを読み込む際の終了処理

```
istream& read_hw(istream& in, vector<double>& hw){
  if (in) {
    hw.clear();
    double x;
    //下のwhileブロックはinからdouble型の値を変数xに読み込み、
    //vector変数のhwに追加することエラーが起こるまで繰り返す。
    //入力データが変数のタイプが異なる場合もエラーとなるので、
    //それを使ってループの終了条件とすることができる。
    while (in >> x)
      hw.push_back(x);
    in.clear(); //エラーのクリア
  }
  return in;
}
```

関数をライブラリにする

- sort関数に対する比較方法の指定
  - ソートするデータ型

```
struct Student_info {
    string name;
    double midterm, final;
    vector<double> homework;
};
vector<Student_info> students;
```

  - 比較演算子が定義されていないデータ型をソートする場合は、比較関数を指定してあげる必要がある。

```
bool compare(const Student_info& x,const Student_info& y){
    return x.name < y.name;
}
sort(students.begin(), students.end(), compare);
// ここで、compareは関数のポインタを渡す
```

プログラムをファイルに分割する

- 関数のプロトタイプと実体を分ける
  - Student_info.cc には関数の実体を記述する。
  - Student_info.h には関数のプロトタイプを記述する。
  - 関数内で使う構造体などの定義はヘッダファイルにて行う。

プログラムをファイルに分割する

- 重複定義を避けるためにifndef-define-endifでガードをかける

```
#ifndef GUARD_Student_info
#define GUARD_Student_info
// `Student_info.h' header file
#include <iostream>
#include <string>
#include <vector>
struct Student_info {
     std::string name;
     double midterm, final;
     std::vector<double> homework;
};
bool compare(const Student_info&, const Student_info&);
std::istream& read(std::istream&, Student_info&);
std::istream& read_hw(std::istream&,std::vector<double>&);
#endif
```

プログラムをファイルに分割する

- Makefileの記述
  - コンパイラの指定
    CC = g++
  - ターゲットと依存ファイル
    ターゲットが存在しないか、依存ファイルが更新された(新しい日付)場合にターゲットファイルを生成する。
  - さらに依存ファイルがまた別のファイルに依存する場合はそのファイルの更新もチェックされる。
  - ターゲットファイルの生成方法は依存関係の次の行に書く。生成方法がデフォルトで決まっている場合は空行を必ず入れる
    Student_info.o: Student_info.cc Student_info.h
    ←---- ここは空行、空ける必要がある
  - デフォルト方法が決まっていない場合の空行は生成しないことを意味する
    all: main
    ←---- ここは空行、空ける必要がある
    ターゲットallは特に生成されないが、依存ファイルmainの依存ファイルがさらにチェックされ、必要な場合には生成を行う。

プログラムをファイルに分割する

- Makefileの例

```
CXX = g++
CC = g++
all:            main

Student_info.o:      Student_info.cc Student_info.h

grade.o:     grade.cc grade.h median.h Student_info.h

median.o:    median.cc median.h

main.o:      main.cc grade.h median.h Student_info.h

main:        main.o grade.o median.o Student_info.o

test:           all
        ./main < gradedata  #この行の先頭はtab
clobber:
        rm -f *.o *.exe core main #この行の先頭はtab
```

プログラムをファイルに分割する

- Makefileによる自動コンパイルは
  make: 最初のターゲットallを生成する。
  make all: makeと同じ働き。
  make grade.o: 依存ファイルをチェックし、grade.oを生成する
  make test: make allと実行ファイルの起動
  make clobber: ファイルの削除