Programming C++
Lecture Note 6
ライブラリのアルゴリズムを使う

Jie Huang

記憶クラス

- C++には4つの記憶クラス、auto(自動変数)、static(静的変数)、extern(外部変数)、register(レジスタ変数)がある。
- 自動変数(automatic variable)
  局所変数(local variable)ともいう。関数の内部で宣言される変数は、なにも指定しなければ自動変数となる。そのスコープは関数の内部に限られ、その関数から出る時にその領域は消滅する(開放される)。自動変数が明示的に初期化されていない時は、その値は不定となる。

```
void func(int v1, int v2){
  int a, b;   ...
  {    int c, d;    ... }
  ...
}
```

  この関数funcでは、a、b、c、dは自動変数である。だだし、c、dはスコープ(有効範囲)がブロックの中に限定される。

記憶クラス

- 静的変数(static variable)
  静的変数が関数の外部で宣言された場合は、そのソースプログラム内で宣言された場所からファイルの最後まで有効となる。静的変数はプログラムコンパイル時に記憶領域が確保され、初期化される。プログラムが実行されている間、存在し続ける(従って、初期化は1回だけ行われる)。

```
static int a, b;
void func1(int v1, int v2){
  static int c, d;
  ...
}
void func2(int v3, int v3){
  ...
}
```

  ここでは、a、b、c、dは静的変数である。だだし、c、dはスコープ(有効範囲)が関数func1に限定される。

記憶クラス

- 外部変数（external variable）
  大域変数（global variable）ともいう。外部変数は静的変数と同様にプログラムコンパイル時に記憶領域が確保され、初期化される。プログラムが実行されている間、存在し続ける。外部変数のスコープは静的変数と違って、他のファイルにおいても参照できる。

```
int a, b; // in file 1
```

```
extern int a; // in file 2
void func1(int v1, int v2){  ...}
void func2(int v3, int v3){  extern int b;  ... }
```

ここでは、a、bはすでにファイル1で外部変数として宣言されたもので、ファイル2ではexternを付け加えて参照することができる（externを付けないと2重定義でエラーとなる）。だだし、extern int bは関数func2の内部にだけ有効となる。外部変数は乱用するとプログラムのモジュールの独立性を妨げることになるので、使う時は注意しなければならない。

vectorに要素を挿入する方法

- push_backを使う方法

```
for(vector<string>::const_iterator it = bottom.begin();
    it != bottom.end(); ++it) ret.push_back(*it);
```

この場合、追加するデータの領域確保はpush_back関数が必要に応じて行う。

- insert関数を使った挿入

```
ret.insert(ret.end(), bottom.begin(), bottom.end());
```

領域確保はinsert関数が行う。

- copyと反復子アダプタを使った挿入

```
copy(bottom.begin(), bottom.end(), back_inserter(ret));
```

back_inserter(ret)は追加するデータの領域を確保し、その領域への反復子を返す。

- copyの間違った使い方

```
copy(bottom.begin(), bottom.end(), ret.end()); // エラー
```

ret.end()はretの最後の要素の次を示す反復子で、そこはまだ領域の確保は行われていない。

反復子アダプタ

- 反復子のインターフェースを持ち、より複雑な処理を行うもの
  - 挿入反復子
    コピー先に指定すると、必要に応じてコンテナの領域を自動拡大する。
    back_inserter()、front_inserter()、inserter()など
  - ストリーム反復子
    コピー元やコピー先に入出力ストリームを指定するために使う。

```
vector<string> coll;
copy(istream_iterator<string>(cin),
   istream_iterator<string>(), back_inserter(coll));
```

    この例はcinからstring型のデータを読み込み、vector型の変数collに追加挿入される。
  - 逆反復子
    通常の反復子とは逆の方向に動作するもの。
    内部でインクリメントとデクリメントを入れ替える。
    メンバー関数rbegin()、rend()を使う。

反復子アダプタ

- 各種挿入のための反復子アダプタの違い
  基本機能は同じで、コンテナを必要に応じて拡大し、その領域への反復子を返す。コピー先などとして安全に使うことができる。
  <iterator>ヘッダで定義されている
  - back_inserter(c)
    コンテナcの最後に要素を付け加えるための反復子を返す。list、vector、stringなどpush_back関数を持つコンテナが使える。
  - front_inserter(c)
    コンテナcの先頭に要素を挿入するための反復子を返す。
    listなどpush_front関数を持つコンテナが使える。
    string、vectorなどは使えない。
  - inserter(c, it)
    コンテナcの要素itの直前に挿入するための反復子を返す。全てのコンテナで使用できる汎用挿入反復子である。ただし、連想コンテナでは要素の正しい位置は値によって決まるので、ヒント(検索開始位置)としてのみ使われる

標準ライブラリを使ったsplit関数

```
#Include <algorithm>
#include <cctype>
#include <string>
#include <vector>
#include “split.h“
using namespace std;
bool space(char c) {return isspace(c);}
bool not_space(char c) {return !isspace(c);}
vector<string> split(const string& str){
  typedef string::const_iterator iter; vector<string> ret;
  iter i = str.begin();
  while (i != str.end()) {
    i = find_if(i, str.end(), not_space);
    iter j = find_if(i, str.end(), space);
    if(i!=str.end()) ret.push_back(string(i,j));
    i = j;
  }
  return ret;
}
```

標準ライブラリを使った回文の判断

- 回文とは前から読んでも場合と後ろから読んでも同じとなる文を言う。例えば、
  - 竹薮焼けた(たけやぶやけた)
  - 英語のワード単位ではCivic, level rotatorなど
  - Was it a rat I saw?
  - Madam, I'm Adam.
- 回文の判断はコンテナの最後から前へ向かっていく逆反復子rbeginを使う

```
bool is_palindrome(const string& s){
  return equal(s.begin(), s.end(), s.rbegin());
}
```

URLを見つける

find_urls:
文の中のurlを見つけ出す関数

以下の下位関数を含む:

url_beg:
まず://を探し(search関数を使う)、
そして前のアルファベット文字を探して先頭を返す。
not_url_charでurlかどうかを判断する

url_end:
not_url_charでurlの終わりを探す。

not_url_char:
urlでは使えない文字かどうかを判断する。

URLを見つける

- URLは以下のような形をしているとする
  protocal-name://resouce-name
- 全体のアルゴリズムを考える

```
vector<string> find_urls(const string& s) {
  vector<string> ret;
  typedef string::const_iterator iter;
  iter b = s.begin(), e = s.end();
  while (b != e) { //終了条件
    b = url_beg(b, e); //urlの先頭を見つける
      if (b != e) {
        iter after = url_end(b, e); //urlの終わりを見つける
        ret.push_back(string(b, after)); //urlをretに追加
        b = after;
    }
  }
  return ret;
}
```

URLを見つける

- URLの先頭を見つける関数

```
string::const_iterator url_beg(string::const_iterator b,
string::const_iterator e){
  static const string sep = “://”;
  typedef string::const_iterator iter;  iter i = b;
  while ((i=search(i,e,sep.begin(),sep.end())) != e){
    if (i != b && i + sep.size() != e) {
      iter beg = i;
      while (beg != b && isalpha(beg[-1])) --beg;
      if (beg != i && !not_url_char(i[sep.size()]))
        return beg;
    }
    i += sep.size();
  }
  return e;
}
```

注:beg[-1]は*(beg-1)と等価である。コンテナがインデックスを持つ時に限って反復子もインデックスが使える。

URLを見つける

- URLの終りを見つける

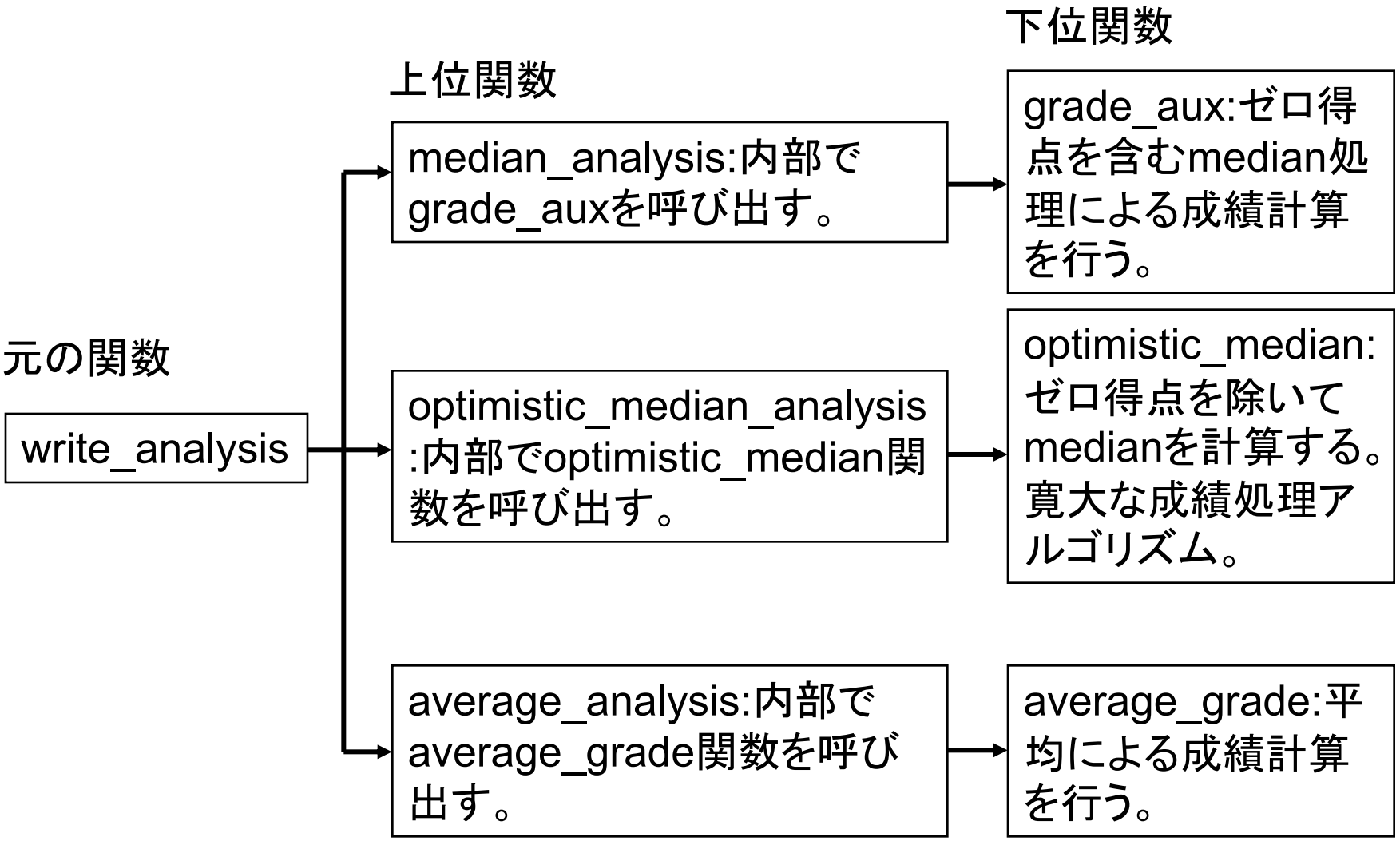
```
string::const_iterator url_end(string::const_iterator b,
string::const_iterator e) {
  return find_if(b, e, not_url_char);
}

bool not_url_char(char c) {
  static const string url_ch = "~;/?:@=&$-_.+!*'(),";
  return !(isalnum(c) ||
    find(url_ch.begin(), url_ch.end(), c) != url_ch.end());
}
```

学生の成績処理の階層的な仕組み

- 目的
  全部の宿題を完成した学生とそうでない学生についてmedian_analysis、optimistic_madian_analysisとaverage_analysisの方法で成績を分析する。
- 全部の宿題をしたかどうかを判断する

```
using std::find;
bool did_all_hw(const Student_info& s){
   return((find(s.homework.begin(),s.homework.end(), 0))
     == s.homework.end()); //0点の宿題がない
}
// 以下はmain関数での処理
vector<Student_info> did, didnt;
Student_info student;
while (read(cin, student)) {
   if (did_all_hw(student)) did.push_back(student);
   else didnt.push_back(student);
}
```

学生の成績処理の階層的な仕組み



学生の成績処理の階層的な仕組み

- 関数をパラメータとする処理関数
  今までのcompare関数と同じように、関数パラメータを使って、どの関数を使うかを関数コール時に決めることができる。

```
void write_analysis(ostream& out, const string& name,
        double analysis(const vector<Student_info>&),
        const vector<Student_info>& did,
        const vector<Student_info>& didnt)
{
  out << name << “: median(did) = “
      << analysis(did)
      << ”, median(didnt) = ” << analysis(didnt)
      << endl;
}
```

  ここで、analysisは成績処理の関数をパラメータとするので、コール時にmedian_analysisやaverage_analysisなどの関数を渡すことができる。

学生の成績処理の階層的な仕組み

- 関数をパラメータとする関数の使い方

```
write_analysis(cout, "median", median_analysis,
    did, didnt);
write_analysis(cout, "average", average_analysis,
    did, didnt);
write_analysis(cout, "median of homework turned in",
    optimistic_median_analysis, did, didnt);
```

学生の成績処理の階層的な仕組み

- 違う種類の成績処理関数を用意する

```
double median_analysis(const vector<Student_info>& students){
  vector<double> grades;
  transform(students.begin(), students.end(),
      back_inserter(grades), grade_aux);
  return median(grades); // 全学生の成績のmedianを返す
}
double average_analysis(const vector<Student_info>& students){
  vector<double> grades;
  transform(students.begin(), students.end(),
      back_inserter(grades), average_grade);
  return median(grades); // 全学生の成績のmedianを返す
}
double optimistic_median_analysis
(const vector<Student_info>& students){
  vector<double> grades;
  transform(students.begin(), students.end(),
      back_inserter(grades), optimistic_median);
  return median(grades); // 全学生の成績のmedianを返す
}
```

学生の成績処理の階層的な仕組み

いくつかの関数の説明

remove_copy(b, e, d, t):
[b, e)の範囲にある、tに等しくない要素だけをdへコピーする。

remove_copy_if(b, e, d, p):
[b, e)にある、関数pによってfalseを返す要素だけをdへコピーする。

transform(b, e, d, f):
[b, e)の範囲の要素に関数fを適用し、その戻り値をdで示す場所に格納する。

accumulate(b, e, t):
tの値に範囲[b, e)の全ての要素の値を足し、その結果を戻す。
戻り値はtと同じ型になる。

学生の成績処理の階層的な仕組み

- さらに下位処理関数を用意する

```
double grade_aux(const Student_info& s){
  try { return grade(s);
  } catch (domain_error) {
    return grade(s.midterm, s.final, 0);
  }
}
double average_grade(const Student_info& s){
  return grade(s.midterm, s.final,average(s.homework));
}
double optimistic_median(const Student_info& s){
  vector<double> nonzero;
  remove_copy(s.homework.begin(), s.homework.end(),
        back_inserter(nonzero), 0);
  if (nonzero.empty()) return grade(s.midterm, s.final, 0);
  else return grade(s.midterm, s.final, median(nonzero));
}
```