Programming C++
Lecture Note 5
コンテナの操作と反復子

Jie Huang

クラスの定義と演算子関数

- クラスの前送り宣言(forward class definition)

```
class doctor; // 前送りの不完全なクラス宣言
class patient {
  doctor *d;        // 前送り宣言されたクラスへのポインタ
  ...
public:
  ...
};
// クラス本体の定義
class doctor {
  patient *p;
  ...
public:
  ...
};
```

クラスの定義と演算子関数

- 代入演算子

```
class Fraction {
public:
  int numerator, denominator;
  Fraction(int n = 1, int d = 1); // デフォルト値コンストラクタ
  Fraction operator=(Fraction f); // 代入演算子のプロトタイプ
  operator int(); // 分数をint型に変換する演算子のプロトタイプ
};
// 代入演算子の実体定義
Fraction Fraction::operator=(Fraction f){
  numerator = f.numerator; denominator = f.denominator;
  return *this; //この部分は連続代入ができるようにするため
}
int main(){
  Fraction f1(1,2);  // 1/2で初期化される分数
  Fraction f2, f3;   // デフォルトの値(1)を持つ分数
  f3 = f2 = f1;      // 分数f1の値を分数f2とf3へ代入
}
```

クラスの定義と演算子関数

- 型変換演算子
  - 他の型からの型変換

```
int i = 2;
Fraction f;
// int型からの変換はコンストラクタにより定義されるので、
// 以下のように明示的あるいは暗黙的に行うことができる
f = (Fraction)i; // 明示的に型変化を示す
f = Fraction(i); // 型変換のもう1つの方法
f = i; // コンストラクタを使った暗黙の型変換
```

  - 他の型への型変換は型変換演算子の定義が必要となる

```
// 型変換演算子の定義
Fraction::operator int(){
  return numerator / denominator;
}
i = (int)f;       // 型変換演算子による型変換
i = int(f);       // 型変換演算子による型変換
i = f;            // 明示的に示さなくても良い
```

クラスの定義と演算子関数

- 入出力演算子の定義方法

```
ostream& operator<<(ostream& os, Fraction f){
    if (f.numerator == 0)
        os << “0”;              // 分子が0ならば0
    else if (f.denominator == 1)
        os << f.numerator; // 分母が1ならば、分子の値
    else
        os << f.numerator << “/” << f.denominator;
    return os;
}
```

ここで、ostream型変数osは参照型パラメータで、戻り値もまた参照で返すので、透過型演算子という。
この仕組みによって、

```
cout << x << y;
```

のような連続した記述が可能となる。

クラスの定義と演算子関数

- 多重定義の規則
  - 名前が同じで、パラメータの数と型が違う関数を複数定義することを関数の多重定義という。
  - 多重定義された関数を呼び出す時は、呼ぶ側の引数の数と型によって、どの関数を呼び出すかが決定される。
  - 関数の戻り値の型はチェックの対象とならないので、以下の関数は共存できない。

```
int f( int i );
double f( int i );
```

コンテナの操作と反復子

- 学生を分類する第一の案

```
// 成績の合否を判断する関数
bool fgrade(const Student_info& s){ return grade(s) < 60;}
// 合格と不合格の学生を分ける関数
vector<Student_info>
extract_fails(vector<Student_info>& students){
  vector<Student_info> pass, fail;
  for (std::vector<Student_info>::size_type i = 0;
    i != students.size(); ++i)
    if (fgrade(students[i])) fail.push_back(students[i]);
    else pass.push_back(students[i]);
  students = pass;
  return fail;
}
```

この案の問題点:学生の成績を格納するのにstudents、pass、failの3つのvectorを使うので、余分なメモリが必要となる。

コンテナの操作と反復子

- 学生を分類する第二の案

```
// eraseによって不合格者を削除する
vector<Student_info> extract_fails
  (vector<Student_info>& students) {
  vector<Student_info> fail;
  std::vector<Student_info>::size_type i = 0;
  while (i != students.size()) {
    if (fgrade(students[i])) {
      fail.push_back(students[i]);
      students.erase(students.begin() + i);
    } else ++i;
  }
  return fail;
}
```

この案は、studentsにある不合格者を削除し、failに移すので、メモリの節約にはなるが、vectorの要素削除処理は時間が掛かるので、実行速度が非常に遅くなる。

コンテナ操作と反復子

- コンテナクラスの反復子(iterator)
  - オブジェクトの要素を反復処理する際に使われる
  - 反復子のインターフェースは通常のポインタと同じで、++演算子と*演算子がそのまま使える
  - 反復子はコンテナの内部構造を認識してくれるので、スマートポインタのようなものである
- シーケンシャルアクセス反復子を使った操作
  - インデックスを使った反復処理

```
for (vector<Student_infor>::size_type i = 0;
     i != students.size(); i++)
  cout << students[i].name << endl;
```

  - 反復子を使った効率的に処理することができる

```
for (vector<Student_info>::const_iterator iter =
  students.begin(); iter != students.end(); ++iter)
  cout << (*iter).name << endl;
```

コンテナ操作と反復子

- 反復子の操作
  - 次の要素を指すように反復子をインクリメントする
    iter++;
  - 反復子がコンテナの最初の要素を指すかどうかをチェックする
    if (iter == students.begin())
  - 反復子がコンテナの最後の次の要素を指すかどうかをチェックする
    if (iter == students.end)
  - 反復子の指す内容を返す。ポインタの使い方と同じ
    (*iter)
    例:
    (*iter).name
    また、iter->nameの使い方も同じ
    ただし、*iter.nameは*(iter.name)と構文的には同じなので、
    エラーとなる
  - 以下の+操作はランダムアクセスのできるコンテナにしか許されない
    iter+i

コンテナ操作と反復子

- 反復子を使った学生を分類する第三の案

```
vector<Student_info>
extract_fails(vector<Student_info>& students){
  vector<Student_info> fail;
  std::vector<Student_info>::iterator iter =
    students.begin();
  while (iter != students.end()) {
    if (fgrade(*iter)) {
      fail.push_back(*iter);
      iter = students.erase(iter);
    } else ++iter;
  }
  return fail;
}
```

このプログラムは反復子を使って処理を改善しているが、studentsにある不合格者を削除する仕組みは変わらないので、第二の案の遅いところは改善されていない。

コンテナ操作と反復子

- list型コンテナを使った第四の案
  - vectorはランダムアクセスを効率的に行うコンテナであるが、挿入と削除の効率はよくない。これに対し、listはランダムアクセスはできないが、削除と挿入を効率よく行うことができる。

```
list<Student_info>
extract_fails(list<Student_info>& students){
  list<Student_info> fail;
  std::list<Student_info>::iterator iter =
    students.begin();
  while (iter != students.end()) {
    if (fgrade(*iter)) {
      fail.push_back(*iter);
      iter = students.erase(iter);
    } else ++iter;
  }
  return fail;
}
// listを使うことで、実行速度を改善する。
// 変更はvectorをlistにするだけで、他の変更は全く必要ない
```

コンテナ操作と反復子

- listコンテナとvectorコンテナにおける反復子の違い
  - Listの反復子はシーケンシャルアクセスのみで、vectorの反復子はランダムアクセスが可能である。
  - vectorの要素をeraseした場合、削除される要素とその後の要素を指す反復子が無効となる。
  - vectorはpush_backを使って要素を追加した場合は全反復子が無効となる（要素の追加操作は場合によっては全要素を別の領域に移すことがあるから）。
  - 一方、listのeraseとpush_back操作は他の反復子を無効にはしない。eraseの場合で無効となるので、実際に削除される要素だけである。
  - listは要素の削除、挿入がvectorよりも遥かに効率よいものである。次の表は違うサイズのファイルを処理するのに必要な時間を示す

ファイルサイズ	list	vector
735	0.1	0.1
7,350	0.8	6.7
73,500	8.8	597.1

コンテナ操作と反復子

- listにおけるソート
  - 今まで使った標準ライブラリのソート関数sortはランダムアクセス反復子を要求するので、vectorのような以下のソートの仕方はlistでは使えない。

```
vector<Student_info> students;
sort(students.begin(), students.end(), compare);
```

  - その代わり、listはソートを行うメンバー関数が用意されている

```
list<Student_info> student;
students.sort(compare);
```

ここで、compareは同じく比較方法を指定する関数

stringコンテナの操作

- stringオブジェクトを操作する
  - stringコンテナも反復子を持ち、ランダムアクセスができる
  - stringの連結
    `s1 += s2;`
  - 範囲[i, i+j)のサブstringを取り出す
    `s.substr(i, j)`
  - stringオブジェクトsの長さはs.size()で得られる
  - stringのi番目の文字はs[i]
- 関連関数
  - 文字がスペースである判定はisspace(s[i])関数を使う
  - 行単位で文字列を読み込む関数は
    `getline(cin, s);`

stringコンテナの操作

- スペースで区切られる文字列をstring型vectorに分割する関数

```
vector<string> split(const string& s){
  vector<string> ret; //分割されたstringを格納するvector
  typedef string::size_type string_size;
  string_size i = 0;
  while (i != s.size()) {
    //先頭のスペースを飛ばす
    while (i != s.size() && isspace(s[i])) ++i;
    string_size j = i;
    //次のスペースを探す
    while (j != s.size() && !isspace(s[j])) ++j;
    //文字の部分列を取り出し、vectorに入れる
    if (i != j){ret.push_back(s.substr(i, j - i)); i = j;}
  }
  return ret;
}
```