Programming C++
Lecture Note 8
テンプレートとジェネリック関数

Jie Huang

コンストラクタとクラスの初期化

- なぜコンストラクタが必要か
  コンストラクタはクラスの初期化の手段を与える。コンストラクタによる初期化は、以下の2つのメリットがある。
  - 統一した初期化の手段を与える。
  - 初期化をクラスの設計段階で行うので、クラスを使う段階での初期化のミスを防ぐことができる。
- デフォルト値を持つコンストラクタ

```
class railroad_car {
public: int year_built;
  railroad_car(int year = 0){ year_built = year; }
};
```

以下の2つのコンストラクタと等価

```
  railroad_car(){ year_built = 0; }
  railroad_car(int year){ year_built = year; }
```

コンストラクタとクラスの初期化

- コンストラクタによる変数初期化

```
class sphere {
public: double radius;
  sphere(double r = 0){ radius = r; }
};
class liquid_gas_car: public railroad_car, public sphere {
public: int number;
  liquid_gas_car( int n = 3, double r = 3.5 ){
    number = n; radius = r;
  }
};
#include <iostream.h>
main(){
  liquid_gas_car l(2,5.0) ;
  cout << "year built: " << l.year_built << endl;
  cout << "radius: " << l.radius << endl;
  cout << "number: " << l.number << endl;
}
```

コンストラクタとクラスの初期化

- 初期化リストによる変数の初期化
  コンストラクタ呼び出しに先立って初期化を行う

```
#include <iostream.h>
class sphere {
public:
  double radius;
  sphere(double r = 0): radius(r) {}
};
```

- コンストラクタの中で初期化を行う

```
#include <iostream.h>
class sphere {
public:
  double radius;
  sphere(double r = 0){ radius = r;}
};
```

コンストラクタとクラスの初期化

- 初期化リストによるクラスオブジェクトの初期化

```
class X {
public:
  double x;
  X(double x0 = 0.0): x(x0) {}
};
class sphere {
public:
  double radius;
  X x_obj;
  sphere(double r=0, double x0=1.0):radius(r),x_obj(x0) {}
};
```

コンストラクタとクラスの初期化

- 初期化リストによるconst変数の初期化

```
class railroad_car {
public:
  const int year_built;
  railroad_car(int year = 0): year_built(year){}
};
```

- const変数はコンストラクタの中では初期化(代入)できない

```
class railroad_car {
public:
  const int year_built;
  railroad_car(int year = 0){year_built = year;} // error
};
```

コンストラクタとクラスの初期化

- 初期化リストで基底クラスのコンストラクタにパラメータを渡す

```
class liquid_gas_car: public railroad_car, public sphere {
public:
  int number;
  liquid_gas_car(int=3,double=3.5,int=0);
};
liquid_gas_car::
liquid_gas_car(int n, double r, int y):
 number(n), sphere(r), railroad_car(y){}
main(){
   liquid_gas_car l(2, 5.0, 2000);
   cout << “year built: ” << l.year_built << endl;
   cout << “radius: ” << l.radius << endl;
   cout << “number: ” << l.number << endl;
}
```

コンストラクタとクラスの初期化

- 初期化リストによる初期化のまとめ
  - コンストラクタでは初期化リストかコンストラクタの中で変数の初期化を行うことができる。
  - 初期化リストでクラスオブジェクトの初期化も行うことができる。
  - 定数型変数の初期化は代入ができないので、初期化リストでしか初期化できない。
  - 初期化リストで基底クラスのコンストラクタにパラメータを渡すことができる。
- 派生クラスにおける生成と初期化の順序
  - まず基底クラスの生成
  - 次に派生クラスのクラスオブジェクト
  - そして最後に派生クラス自身の定義部分である。

暗黙の引数と自己参照

- 以下の2つの関数の実装を考える
  - 一般関数によるdisplay_year_builtの実装

```
void display_year_built( railroad_car *r ){
  cout << r->year_built;
}
railroad_car a;
a.year_built = 2000;
display_year_built(&a); //明示的引数が必要
```

  - クラスメンバ関数によるdisplay_year_builtの実装

```
railroad_car:: void display_year_built(){
  cout << year_built;
}
railroad_car b;
b.year_built = 2001;
b.display_year_built(); //明示的引数が必要ない
```

暗黙の引数と自己参照

- thisポインタでの自己参照
  クラスメンバー関数では、自分自身を指す暗黙の引数thisというポインタが存在する。つまり、クラスメンバ関数のdisplay_year_builtは以下のようにも書ける。

```
railroad_car: void display_year_built(){
   cout << this->year_built;
}
```

  ここで、

```
b.display_year_built();
```

  と実行した時に、thisは暗黙的にクラスオブジェクトbを指すようになる。
  thisは自分自身を指すので、自己参照(self-reference)とも言う。

自己参照の使い方

- 自分自身の値を返す場合

```
Fraction Fraction::operator=(Fraction f){
  numerator = f.numerator; denominator = f.denominator;
  return *this;
}
Fraction f1(1,2), f2, f3;
f3 = f2 = f1;
```

ここでは、自分自身の値を返すことによって、代入演算子の連続代入が可能となる。

- クラスメンバー関数において自分自身を関数への引数として渡す場合

```
func(this);
```

- クラスメンバー関数において自分自身を他のオブジェクトへ代入あるいはコピーを行う場合

```
a = *this;
```

テンプレートとジェネリック関数

- Cでは不特定の変数型に対応した関数できないので、代わりにマクロを使って定義する
```
#define min(a,b) (((a)<(b))?(a):(b))
```
- C++ではテンプレート関数を使って不特定型の関数を定義することができる
```
template<class T> T min(const T a, const T b){
  return a < b ? a : b;
}
```
なお、<calss T>の代わりに<typename T>を使うこともある。
- テンプレート関数を呼び出す場合、C++が適切な型の関数を自動的に生成する。例えば、
```
int i, j, k; double u, v, w;
k = min(i, j);
w = min(u, v);
```
の場合は自動的に以下の関数が生成される:
```
int min(const int a, const int b);
double min(const double a, const double b);
```
テンプレート関数は通常の関数や追加の関数テンプレートによってオーバーロードできる。

テンプレートとジェネリック関数

- ジェネリックアルゴリズムとは
  - 特定のコンテナに付随しているのではなく、多くのコンテナに共通して作用できるアルゴリズムのことをいう。
  - ジェネリック関数は関数テンプレートによって実現され、関数の引数や戻り値の型が使う時に決められるものである。
- 例えば、未知の型のメジアンを求める

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

ここで、typenameはテンプレートを含んだtypedefで使う必要がある。

テンプレートとジェネリック関数

- 順次読込み専用アクセス

```
template <class In, class X>
In find(In begin, In end, const X& x){
  while (begin != end && *begin != x) ++begin;
  return begin;
}
```

ここで、反復子beginに++演算子、beginの要素を指すのに*演算子を使っているのみである。このようなコンテナの要素を順次読込むための反復子は入力反復子と呼ぶ。

- 順次書き込み専用アクセス

```
template <class In, class Out>
Out copy(In begin, In end, Out dest){
  while (begin != end)  *dest++ = *begin++;
  return dest;
}
```

ここで、dest反復子は*dest = valueという操作を許すので、出力に使うことができ、出力反復子と呼ぶ。

テンプレートとジェネリック関数

- 順次読み書きアクセス

```
template <class For, class X>
void replace(For beg, For end, const X& x, const X& y){
  while (beg != end) {if (*beg == x) *beg = y; ++beg; }
}
```

ここで、begは入力にも、出力にも使われ、また、++の演算子しか許せないので、前方向演算子と呼ぶ。

- 可逆アクセス

```
template <class Bi>
void reverse(Bi begin, Bi end) {
  while (begin != end) {
    --end;
    if (begin != end) swap(*begin++, *end);
  }
}
```

ここで、endという反復子には、順方向に加えて--という逆方向の演算子も許されるので、双方向反復子と呼ぶ。

テンプレートとジェネリック関数

- ランダムアクセスを許す反復子

```
template <class Ran, class X>
bool binary_search(Ran begin, Ran end, const X& x){
  while (begin < end) {
    Ran mid = begin + (end - begin) / 2;
    if (x < *mid) end = mid;
    else if (*mid < x) begin = mid + 1;
    else return true;
  }
  return false;
}
```

ここで、反復子begin、midとendについては+, -などの演算子が自由に使えるので、ランダムアクセス反復子という。

- 反復子の範囲指定を[begin, end)で行う理由
  - 範囲に要素が全くない場合の例外処理が簡単
  - 範囲内にあるかどうかの判断が一回の比較で済む
  - 見付からなかった時に範囲外を示す反復子を返すことができる

テンプレートとジェネリック関数

- 入力反復子と出力反復子を使って入出力

```
vector<int> v;
//反復子による入力
copy(istream_iterator<int>(cin),
     istream_iterator<int>(), back_inserter(v));
//反復子による出力
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

- 柔軟性のための反復子

```
template <class Out> void split(const string& str, Out os) {
  typedef string::const_iterator iter; 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()) *os++ = string(i, j);
    i = j;
  }
}
```

- 以下のように使うことができる

```
list<string> word_list;
split(s, back_inserter(word_list));
split(s, ostream_iterator<string>(cout, “¥n”));
```