目录

第9章 Template

相当于定义了一个 函数/类 的集合

9.1 函数模板

9.1.1 函数模板的声明

对不同的类型，执行相同的操作时，使用函数模板，比如swap、sort

```
template <class T> // 或者是<typename T>
void swap(T& x, T& y){
    T temp = x;
    x = y;
    y = temp;
}
```

1. **template**：声明模板
2. **class T**：声明了参数化类型的名字，在模板函数中，用**T**作为一个类型名
    - 作为函数的参数
    - 作为函数的返回值
    - 作为函数的中间变量

9.1.2 函数模板的实例化 template instantiation

1. 将一个真实的类型，带入template中
2. 编译器会生成一个新的 函数/类 的定义
    - 在生成的过程中，会检查 语法错误/不安全的调用
3. 模板特化**specialization**
    - 对特定的类型，执行新的模板函数

```
int i = 3; int j = 4;
swap(i, j);  // 实例化 int swap

float k = 4.5; float m = 3.7;
swap(k, m);  // 实例化 float swap
std::string s(“Hello”);
std::string t(“World”);
swap(s, t);  // 实例化 std::string swap
```

9.1.3 函数模板的使用要求

1. 只有精确匹配，才能调用，不能执行类型转换

```
swap(int, int);       // ok
swap(double, double); // ok
swap(int, double);    // error!
```

2. 函数模板、正常函数的命名可以是相同的，编译器会决定调用哪一个函数
   - 首先，检查与正常函数的参数类型是否完全匹配
   - 然后，检查是否与模板的参数相同
   - 最后，进行类型转换

```
void f(float i, float k) {
    cout << "f(float, float);\n";
};
template <class T>
void f(T t, T u) {
    cout << "f(T, T);\n";
};
f(1.0, 2.0);//编译器默认浮点数为double，此处编译器先判断为f(double, double)，符合模板的定义，输出f(T,T);
f(1, 2);    //此处编译器先判断为f(int, int)，符合模板的定义，输出f(T,T)
f(1, 2.0);  //此处编译器先判断为f(int, double)，不符合模板的定义，进行类型转换，1=>1.0f,2.0=>2.0f，然后输出f(float, float);
```

3. 编译器会推导**T**的实际类型，也可以自己手动定义
   - 如果函数模板没有参数，则需要手动填写参数类型
   - 参数可以有无限多个

```
template <class T>
void foo(void) { /* … */ }
foo<int>();   // type T is int
foo<float>(); // type T is float
```

9.2 类模板

由类型参数化**parameterized**的类

9.2.1 声明

```
template <class T>
class Vector{
public:
    Vector(int);
    ~Vector();
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    T& operator[](int);
private:
    T* m_elements;
    int m_size;
}
```

9.2.2 使用

必须手动填写类型

```
Vector<int> v1(100);
Vector<Complex> v2(256);

v1[20] = 10;
v2[20] = v1[20];  // ok if int->Complex defined
```

9.2.3 定义成员函数

```
template <class T>
Vector<T>::Vector(int size): m_size(size){
    m_elements = new T[m_size];
}
template <class T>
T& Vector<T>::operator[](int index){
    if(index < m_size && index > 0){
        return m_elements[index];
    }else{
        …
    }
}
```

9.2.4 类型参数的传递

```
template <class T>
void sort(vector<T>& arr){
    const size_t last = arr.size() - 1;
    for(int i=0; i<last; i++)
        for(int j = last; i<j; j--)
            if(arr[j] < arr[j-1])
                swap(arr[j], arr[j-1]);
}
vector<int> vi(4);
vi[0] = 4; vi[1] = 3; vi[2] = 7; vi[3] = 1;
sort(vi);  // sort(vector<int>&)

```

```
vector<string> vs;
vs.push_back("Fred");
vs.push_back("Wilma");
vs.push_back("Barney");
vs.push_back("Dino");
vs.push_back("Prince");
sort(vs);   // sort(vector<string>&);
//注意:sort需要有<的定义
```

9.3 templates

1. templates可以有多个参数

```
template <class Key, class Value>
class HashTable{
    const Value& lookup (const Key&) const;
    void install (const Key&, const Value&);
    …
}
```

2. 参数的嵌套

```
Vector< vector<double*> >   //note space > >
```

3. 参数可以很复杂

```
Vector< int (*) (Vector<double>&, int)>
```

4. 参数可以是具体的类型--非类型参数 **non-type parameters**
    - 每给一个bounds的值，编译器就会生成对应的代码，可能会导致编译出的代码过长

```
//声明
template <class T, int bounds = 100>
class FixedVector{
public:
    FixedVector();
    T& operator[](int);
private:
    T elements[bounds]; // fixed size array!
}
//定义
template <class T, int bounds>
T& FixedVector<T, bounds>::operator[] (int i){
    return elements[i]; //no error checking
}
//调用
FixedVector<int, 50> v1;
FixedVector<int, 10*5> v2;
FixedVector<int> v2;  // 使用缺省值
```

9.4 模板与继承

1. 模板类 可以继承于 非模板类

```
template <class A>
class Derived : public Base{…}
```

2. 模板类 可以继承于 模板类
    - 基类的模板参数，从 派生类 中获得

```
template <class A>
class Derived : public List<A>{…}
```

3. 非模板类 可以继承于 模板类
    - 基类的模板参数，需要用户显式指定

```
class SupervisorGroup:public List<Employee*>{…}
```

例：模拟virtual函数的调用(不借用虚函数表)

```
template <class T>
struct Base {
    void implementation() {
        static_cast<T*>(this)->implementation(); // ...
    }
    static void static_func() {
        T::static_sub_func(); // ...
    }
};
struct Derived : public Base<Derived> {
    void implementation();
    static void static_sub_func();
};
```

9.5 注意

1. 要把**模板的定义和实现**都放到**头文件**中
2. 编译器会自动去除模板的重复定义