08OOP 之泛型编程

多态的概念：

前面我们讲面向对象编程的时候，它的第三个特征为动态绑定。事实上，动态绑定属于运行期多态。前面我们讲过的函数重载属于编译期多态。

这里必须注意，传统的说法，OOP 的三大特征封装、继承、多态中的多态仅包含运行期多态。**编译期多态不是面向对象编程特征的一部分**。

下面我们来学习一种新的编译期多态-模板。它是泛型编程的重要组成部分。

函数模板

考虑一个最简单的求和问题，写成函数，可以有以下的版本：

```
int add(int a, int b){
    return a + b;
}

double add(double a, double b){

    return a + b;
}

string add(const string &a, const string &b){
    return a + b;
}
```

一个简单的求和程序，有没有办法写成一个统一的版本呢？方法

就是采用模板。

```
template <typename T>
T add(const T &a, const T &b){
    return a + b;
}
```

这就是我们写的第一个模板程序。我们观察这个程序，第一行的template <typename T>称为模板形参表，用来表示模板函数实际使用的类型或者值。

当我们使用函数模板的时候，编译器会推断哪个模板实参绑定到模板形参，一旦编译器确定了实际的模板形参，就称它为实例化。

对于 add(string(“abc”), string(“def”))的调用，编译器自动辨别出string 类型，使用它替代模板函数中的 T，并编译 string 版本的函数。

注意，调用模板函数时，参数必须完全匹配，例如，当调用 add(1, 3.4)时，会发生编译错误，原因在于编译器推断第一个类型为 int，第二个类型为 double，没有一个函数符合这个要求。

解决办法是可以写这样一个函数：

```
template <typename T1, typename T2>
T1 add(const T1 &a, const T2 &b){
    return a + b;
}
```

非类型模板形参

模板的形参未必都是类型，也可以是值。

例如：

```
template <typename T, size_t N>
```

```
void array_init(T (&parm)[N])
{
    for(size_t ix = 0; ix != N; ++ix){
        parm[ix] = 0;
    }
}
```

这是一个用来初始化数组的模板函数，它的第二个模板形参不是类型，而是一个数值。函数本身接收一个形参，该形参是数组的引用。当调用 array_init 的时候，编译器自动推导 T，并且计算出 N 的数值。

模板内部用 typename 指定类型

看下面的代码：

例如：

```
template <class Parm, class U>
Parm fcn(Parm *array, U value){
    Parm::size_type * p;
}
```

这段代码存在歧义，因为 Parm::size_type * p 既可以解释成定义一个变量，也可以解释成两个变量相乘。

解决方案是前面加一个 typename，来显式说明这里 Parm::size_type 是一个类型而不是变量。

编写模板程序的两条原则：

1.模板的形参是 const 引用

2.函数体中的比较只使用<比较

这样做的原因在于：使用 const 引用，模板参数就可以接受不可以复制的类型，而且采用引用参数，避免了对象的复制，可以加快程序

的效率。

只使用<的原因，可以让模板的用户更加清楚必须定义那些操作符。可以减少对类型的要求。

例如一个函数内同时具备>和<操作，如果我们直接用这两个运算符，那么用户必须为他们的类型同时重载两个操作符，但是只采用一个，就减少了用户的负担。

注意，**通过<实际上可以推断出所有的比较符**

例如 v1 < v2，那么 v1 > v2 可以看做 v2 < v1

v1 <= v2 可以看做 !(v1 > v2)

v1 >= v2 可以看做 !(v1 < v2)

练习：编写一个泛型的 swap 函数

类模板

我们在运算符重载一节，曾经编写了一个简单的智能指针类，下面我们把它改造成泛型的智能指针。

这里需要注意的是，我们要把类的定义实现全部放到同一个文件中，可以为 h 文件，但最好使用 hpp 文件。

```
#ifndef SMARTPTR_H_
#define SMARTPTR_H_

#include <string>
#include <iostream>

template <typename T>
class SmartPtr {
public:
	SmartPtr();
	explicit SmartPtr(T *ptr);
	~SmartPtr();

	void reset_ptr(T *ptr);
	const T *get_ptr() const;

	T *operator->();
	const T *operator->() const;

	T &operator*();
	const T &operator*() const;

private:
	T *ptr_;

private:
	//prevent copy
	SmartPtr(const SmartPtr &);
	SmartPtr &operator=(const SmartPtr &);
};




template <typename T>
inline SmartPtr<T>::SmartPtr() :
		ptr_(NULL) {

}

template <typename T>
inline SmartPtr<T>::SmartPtr(T *ptr) :
		ptr_(ptr) {
```

```
}

template <typename T>
inline SmartPtr<T>::~SmartPtr() {
	delete ptr_;
}

template <typename T>
inline void SmartPtr<T>::reset_ptr(T *ptr) {
	if (ptr != ptr_) {
		delete ptr_;
		ptr_ = ptr;
	}
}

template <typename T>
inline const T *SmartPtr<T>::get_ptr() const {
	return ptr_;
}

template <typename T>
inline T *SmartPtr<T>::operator->() {
	return ptr_;
}

template <typename T>
inline const T *SmartPtr<T>::operator->() const {
	return ptr_;
}

template <typename T>
inline T &SmartPtr<T>::operator*() {
	return *ptr_;
}

template <typename T>
inline const T &SmartPtr<T>::operator*() const {
	return *ptr_;
}

#endif /* SMARTPTR_H_ */
```

这里需要注意的是：

1.SmartPtr 不再是一个完整的类，正确使用方式是：SmartPtr<T>，所以我们以前接触的 STL，都是采用这种方式编写。

2.函数要声明为 inline。

3.每个函数前面都要加上模板参数。

课堂练习：

使用模板技术编写一个泛型队列。