* 没有退路可言、不要给自己找接口、以及**顾虑太多**
* 缺乏目的、目标、方法、效率再勤奋也只是一种时间的牺牲

@[toc]



# 函数对象

## 函数对象

### 函数对象概念

**概念：**

* 重载**函数调用操作符** () 的类，其对象常称为**函数对象**
* **函数对象**使用重载的()时，行为类似函数调用，也叫**仿函数**
* 在源码中 pred 代表谓词

**本质：**

函数对象(仿函数)是一个**类**，不是一个函数

### 函数对象使用

**特点：**

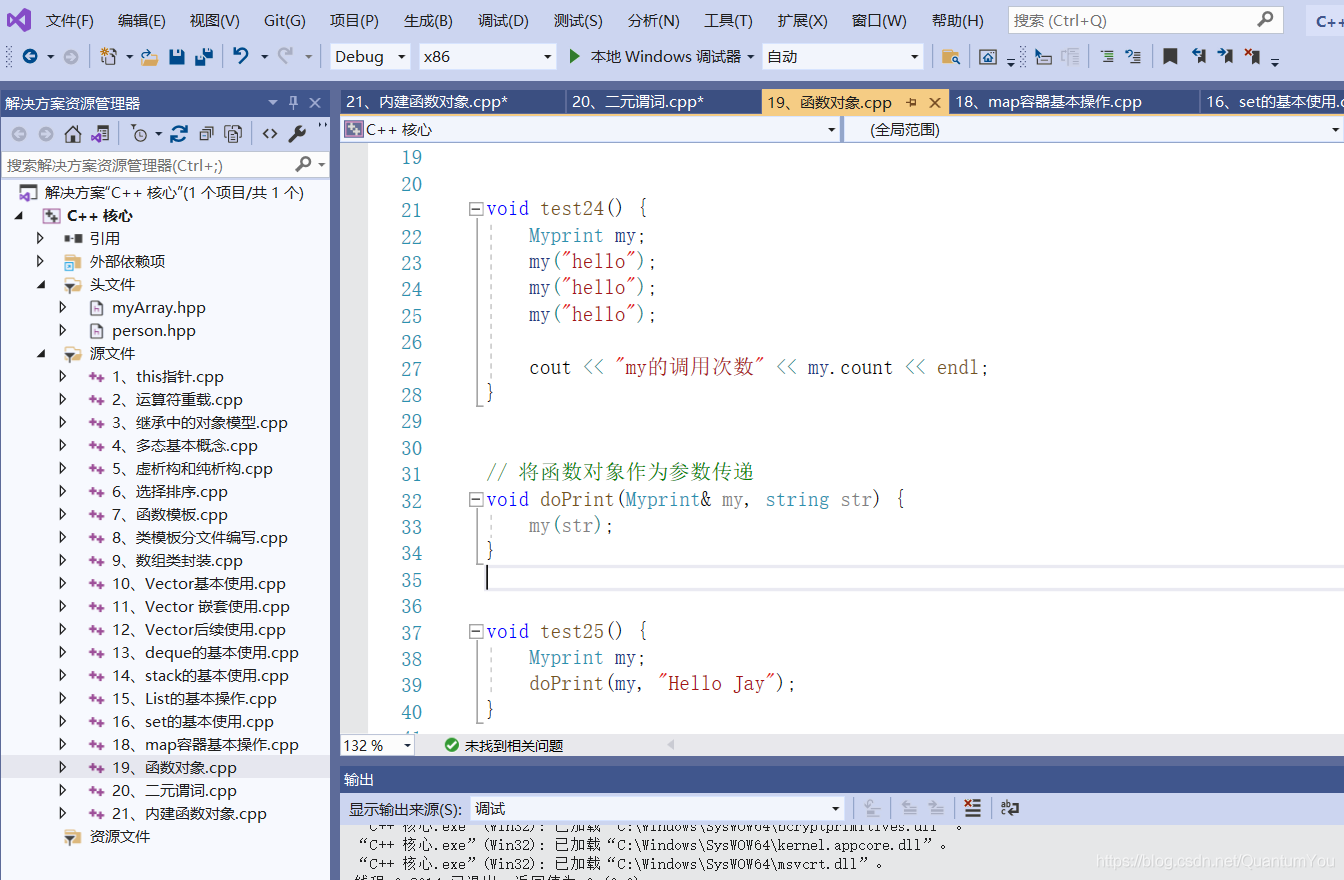
* 函数对象在使用时，可以像普通函数那样调用, 可以有参数，可以有返回值
* 函数对象超出普通函数的概念，函数对象可以有自己的状态
* 函数对象可以作为参数传递

**示例:**

#include <string>  
  
//1、函数对象在使用时，可以像普通函数那样调用, 可以有参数，可以有返回值  
class MyAdd  
{  
public :  
 int operator()(int v1,int v2)  
 {  
 return v1 + v2;  
 }  
};  
  
void test01()  
{  
 MyAdd myAdd;  
 cout << myAdd(10, 10) << endl;  
}  
  
//2、函数对象可以有自己的状态  
class MyPrint  
{  
public:  
 MyPrint()  
 {  
 count = 0;  
 }  
 void operator()(string test)  
 {  
 cout << test << endl;  
 count++; //统计使用次数  
 }  
  
 int count; //内部自己的状态  
};  
void test02()  
{  
 MyPrint myPrint;  
 myPrint("hello world");  
 myPrint("hello world");  
 myPrint("hello world");  
 cout << "myPrint调用次数为： " << myPrint.count << endl;  
}  
  
//3、函数对象可以作为参数传递  
void doPrint(MyPrint &mp , string test)  
{  
 mp(test);  
}  
  
void test03()  
{  
 MyPrint myPrint;  
 doPrint(myPrint, "Hello C++");  
}  
  
int main() {  
  
 //test01();  
 //test02();  
 test03();  
  
 system("pause");  
  
 return 0;  
}

总结：

* 仿函数写法非常灵活，可以作为参数进行传递。



## 谓词

### 谓词概念

**概念：**

* 返回bool类型的仿函数称为**谓词**
* 如果operator()接受一个参数，那么叫做一元谓词
* 如果operator()接受两个参数，那么叫做二元谓词

### 一元谓词

**示例：**

#include <vector>  
#include <algorithm>  
  
//1.一元谓词  
struct GreaterFive{  
 bool operator()(int val) {  
 return val > 5;  
 }  
};  
  
void test01() {  
  
 vector<int> v;  
 for (int i = 0; i < 10; i++)  
 {  
 v.push\_back(i);  
 }  
  
 vector<int>::iterator it = find\_if(v.begin(), v.end(), GreaterFive());  
 if (it == v.end()) {  
 cout << "没找到!" << endl;  
 }  
 else {  
 cout << "找到:" << \*it << endl;  
 }  
  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

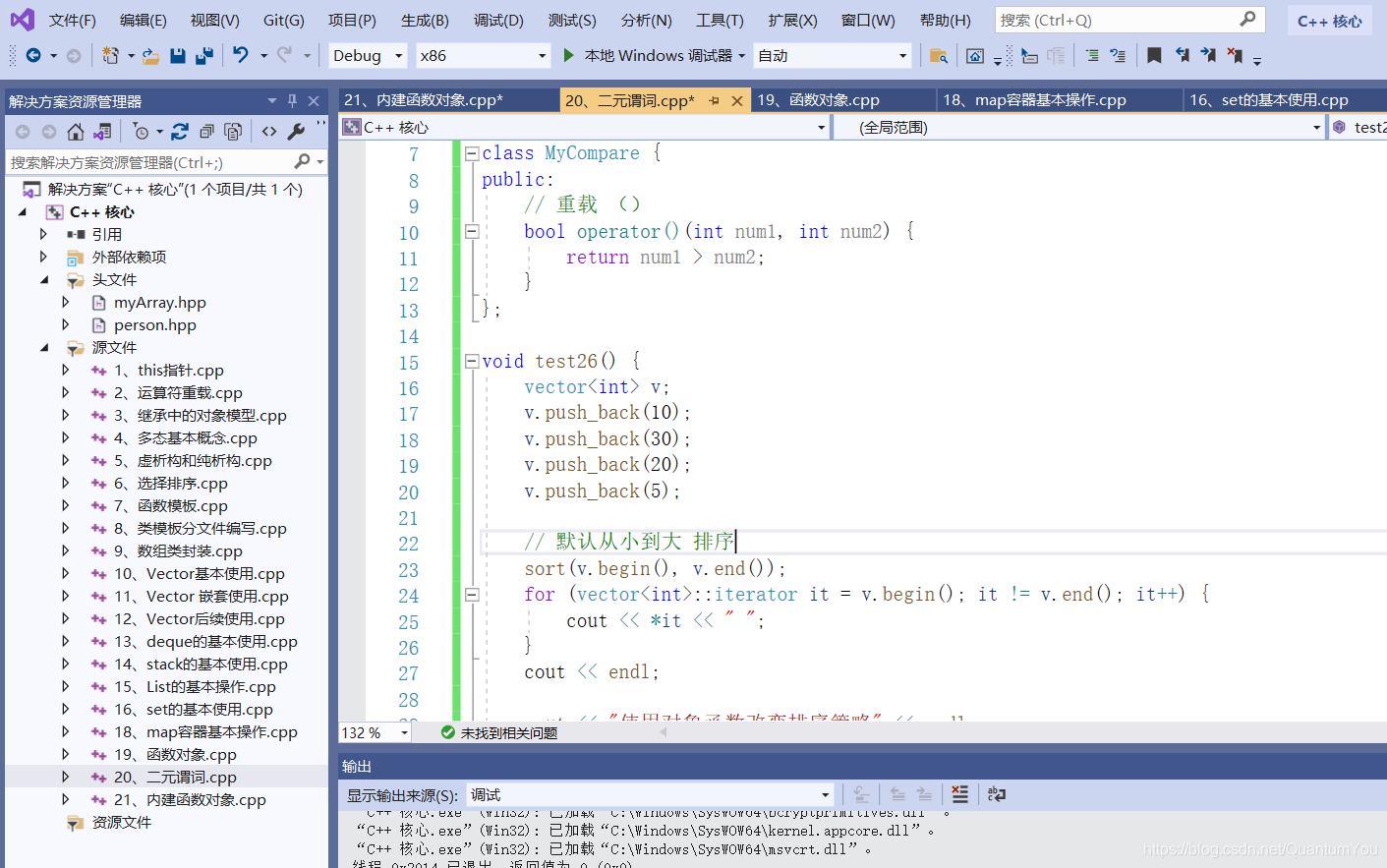
总结：参数只有一个的谓词，称为一元谓词

### 二元谓词

**示例：**

#include <vector>  
#include <algorithm>  
//二元谓词  
class MyCompare  
{  
public:  
 bool operator()(int num1, int num2)  
 {  
 return num1 > num2;  
 }  
};  
  
void test01()  
{  
 vector<int> v;  
 v.push\_back(10);  
 v.push\_back(40);  
 v.push\_back(20);  
 v.push\_back(30);  
 v.push\_back(50);  
  
 //默认从小到大  
 sort(v.begin(), v.end());  
 for (vector<int>::iterator it = v.begin(); it != v.end(); it++)  
 {  
 cout << \*it << " ";  
 }  
 cout << endl;  
 cout << "----------------------------" << endl;  
  
 //使用函数对象改变算法策略，排序从大到小  
 sort(v.begin(), v.end(), MyCompare());  
 for (vector<int>::iterator it = v.begin(); it != v.end(); it++)  
 {  
 cout << \*it << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

总结：参数只有两个的谓词，称为二元谓词



## 内建函数对象

### 内建函数对象意义

**概念：**

* STL内建了一些函数对象

**分类:**

* 算术仿函数
* 关系仿函数
* 逻辑仿函数

**用法：**

* 这些仿函数所产生的对象，用法和一般函数完全相同
* 使用内建函数对象，需要引入头文件 #include<functional>

### 算术仿函数

**功能描述：**

* 实现四则运算
* 其中negate是一元运算，其他都是二元运算

**仿函数原型：**

* template<class T> T plus<T> //加法仿函数
* template<class T> T minus<T> //减法仿函数
* template<class T> T multiplies<T> //乘法仿函数
* template<class T> T divides<T> //除法仿函数
* template<class T> T modulus<T> //取模仿函数
* template<class T> T negate<T> //取反仿函数

**示例：**

#include <functional>  
//negate  
void test01()  
{  
 negate<int> n;  
 cout << n(50) << endl;  
}  
  
//plus  
void test02()  
{  
 plus<int> p;  
 cout << p(10, 20) << endl;  
}  
  
int main() {  
  
 test01();  
 test02();  
  
 system("pause");  
  
 return 0;  
}

总结：使用内建函数对象时，需要引入头文件 #include <functional>

### 关系仿函数

**功能描述：**

* 实现关系对比

**仿函数原型：**

* template<class T> bool equal\_to<T> //等于
* template<class T> bool not\_equal\_to<T> //不等于
* template<class T> bool greater<T> //大于
* template<class T> bool greater\_equal<T> //大于等于
* template<class T> bool less<T> //小于
* template<class T> bool less\_equal<T> //小于等于

**示例：**

#include <functional>  
#include <vector>  
#include <algorithm>  
  
class MyCompare  
{  
public:  
 bool operator()(int v1,int v2)  
 {  
 return v1 > v2;  
 }  
};  
void test01()  
{  
 vector<int> v;  
  
 v.push\_back(10);  
 v.push\_back(30);  
 v.push\_back(50);  
 v.push\_back(40);  
 v.push\_back(20);  
  
 for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
  
 //自己实现仿函数  
 //sort(v.begin(), v.end(), MyCompare());  
 //STL内建仿函数 大于仿函数  
 sort(v.begin(), v.end(), greater<int>());  
  
 for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

总结：关系仿函数中最常用的就是greater<>大于

### 逻辑仿函数

**功能描述：**

* 实现逻辑运算

**函数原型：**

* template<class T> bool logical\_and<T> //逻辑与
* template<class T> bool logical\_or<T> //逻辑或
* template<class T> bool logical\_not<T> //逻辑非

**示例：**

#include <vector>  
#include <functional>  
#include <algorithm>  
void test01()  
{  
 vector<bool> v;  
 v.push\_back(true);  
 v.push\_back(false);  
 v.push\_back(true);  
 v.push\_back(false);  
  
 for (vector<bool>::iterator it = v.begin();it!= v.end();it++)  
 {  
 cout << \*it << " ";  
 }  
 cout << endl;  
  
 //逻辑非 将v容器搬运到v2中，并执行逻辑非运算  
 vector<bool> v2;  
 v2.resize(v.size());  
 transform(v.begin(), v.end(), v2.begin(), logical\_not<bool>());  
 for (vector<bool>::iterator it = v2.begin(); it != v2.end(); it++)  
 {  
 cout << \*it << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

总结：逻辑仿函数实际应用较少，了解即可

Tips :

Visual Studio注释代码段快捷键 默认注释快捷组合键 注释： Ctrl+K → 选定需要注释的代码段 →Ctrl+C 取消注释：  
Ctrl+K → 选定需要取消注释的代码段 →Ctrl+U

