加油！当自己的实力不能满足自己的目标时，就静下心去学习！

@[toc]

# 模板编程

## 模板的概念

模板的特点：

* 模板不可以直接使用，它只是一个框架
* 模板的通用并不是万能的

## 函数模板

* C++另一种编程思想称为 泛型编程 ，主要利用的技术就是模板
* C++提供两种模板机制:**函数模板**和**类模板**

### 函数模板语法

* **函数模板作用**： 建立一个通用函数，其函数返回值类型和形参类型可以不具体制定，用一个**虚拟的类型**来代表。

**语法：**

template<typename T>  
函数声明或定义

**解释：**

template --- 声明创建模板

typename --- 表面其后面的符号是一种数据类型，可以用class代替

T --- 通用的数据类型，名称可以替换，通常为大写字母

**示例：**

//交换整型函数  
void swapInt(int& a, int& b) {  
 int temp = a;  
 a = b;  
 b = temp;  
}  
  
//交换浮点型函数  
void swapDouble(double& a, double& b) {  
 double temp = a;  
 a = b;  
 b = temp;  
}  
  
//利用模板提供通用的交换函数  
template<typename T>  
void mySwap(T& a, T& b)  
{  
 T temp = a;  
 a = b;  
 b = temp;  
}  
  
void test01()  
{  
 int a = 10;  
 int b = 20;  
   
 //swapInt(a, b);  
  
 //利用模板实现交换  
 //1、自动类型推导  
 mySwap(a, b);  
  
 //2、显示指定类型  
 mySwap<int>(a, b);  
  
 cout << "a = " << a << endl;  
 cout << "b = " << b << endl;  
  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

总结：

* 函数模板利用关键字 template
* 使用函数模板有两种方式：自动类型推导、显示指定类型
* 模板的目的是为了提高复用性，将类型参数化

### 函数模板注意事项

注意事项：

* 自动类型推导，必须推导出一致的数据类型T,才可以使用
* 模板必须要确定出T的数据类型，才可以使用

**示例：**

//利用模板提供通用的交换函数  
template<class T>  
void mySwap(T& a, T& b)  
{  
 T temp = a;  
 a = b;  
 b = temp;  
}  
  
  
// 1、自动类型推导，必须推导出一致的数据类型T,才可以使用  
void test01()  
{  
 int a = 10;  
 int b = 20;  
 char c = 'c';  
  
 mySwap(a, b); // 正确，可以推导出一致的T  
 //mySwap(a, c); // 错误，推导不出一致的T类型  
}  
  
  
// 2、模板必须要确定出T的数据类型，才可以使用  
template<class T>  
void func()  
{  
 cout << "func 调用" << endl;  
}  
  
void test02()  
{  
 //func(); //错误，模板不能独立使用，必须确定出T的类型  
 func<int>(); //利用显示指定类型的方式，给T一个类型，才可以使用该模板  
}  
  
int main() {  
  
 test01();  
 test02();  
  
 system("pause");  
  
 return 0;  
}

**总结**：

* 使用模板时必须确定出通用数据类型T，并且能够推导出一致的类型

基于函数模板可以实现多种情况的选择排序  


### 普通函数与函数模板的区别

**普通函数与函数模板区别：**

* 普通函数调用时可以发生自动类型转换（隐式类型转换）
* 函数模板调用时，如果利用自动类型推导，不会发生隐式类型转换
* 如果利用显示指定类型的方式，可以发生隐式类型转换

**示例：**

//普通函数  
int myAdd01(int a, int b)  
{  
 return a + b;  
}  
  
//函数模板  
template<class T>  
T myAdd02(T a, T b)   
{  
 return a + b;  
}  
  
//使用函数模板时，如果用自动类型推导，不会发生自动类型转换,即隐式类型转换  
void test01()  
{  
 int a = 10;  
 int b = 20;  
 char c = 'c';  
   
 cout << myAdd01(a, c) << endl; //正确，将char类型的'c'隐式转换为int类型 'c' 对应 ASCII码 99  
  
 //myAdd02(a, c); // 报错，使用自动类型推导时，不会发生隐式类型转换  
  
 myAdd02<int>(a, c); //正确，如果用显示指定类型，可以发生隐式类型转换  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

总结：建议使用显示指定类型的方式，调用函数模板，因为可以自己确定通用类型T

### 普通函数与函数模板的调用规则

调用规则如下：

1. 如果函数模板和普通函数都可以实现，优先调用普通函数
2. 可以通过空模板参数列表来强制调用函数模板
3. 函数模板也可以发生重载
4. 如果函数模板可以产生更好的匹配,优先调用函数模板

**示例：**

//普通函数与函数模板调用规则  
void myPrint(int a, int b)  
{  
 cout << "调用的普通函数" << endl;  
}  
  
template<typename T>  
void myPrint(T a, T b)   
{   
 cout << "调用的模板" << endl;  
}  
  
template<typename T>  
void myPrint(T a, T b, T c)   
{   
 cout << "调用重载的模板" << endl;   
}  
  
void test01()  
{  
 //1、如果函数模板和普通函数都可以实现，优先调用普通函数  
 // 注意 如果告诉编译器 普通函数是有的，但只是声明没有实现，或者不在当前文件内实现，就会报错找不到  
 int a = 10;  
 int b = 20;  
 myPrint(a, b); //调用普通函数  
  
 //2、可以通过空模板参数列表来强制调用函数模板  
 myPrint<>(a, b); //调用函数模板  
  
 //3、函数模板也可以发生重载  
 int c = 30;  
 myPrint(a, b, c); //调用重载的函数模板  
  
 //4、 如果函数模板可以产生更好的匹配,优先调用函数模板  
 char c1 = 'a';  
 char c2 = 'b';  
 myPrint(c1, c2); //调用函数模板  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

总结：既然提供了函数模板，最好就不要提供普通函数，否则容易出现二义性

### 模板的局限性

**局限性：**

* 模板的通用性并不是万能的

**例如：**

template<class T>  
 void f(T a, T b)  
 {   
 a = b;  
 }

在上述代码中提供的赋值操作，如果传入的a和b是一个数组，就无法实现了

再例如：

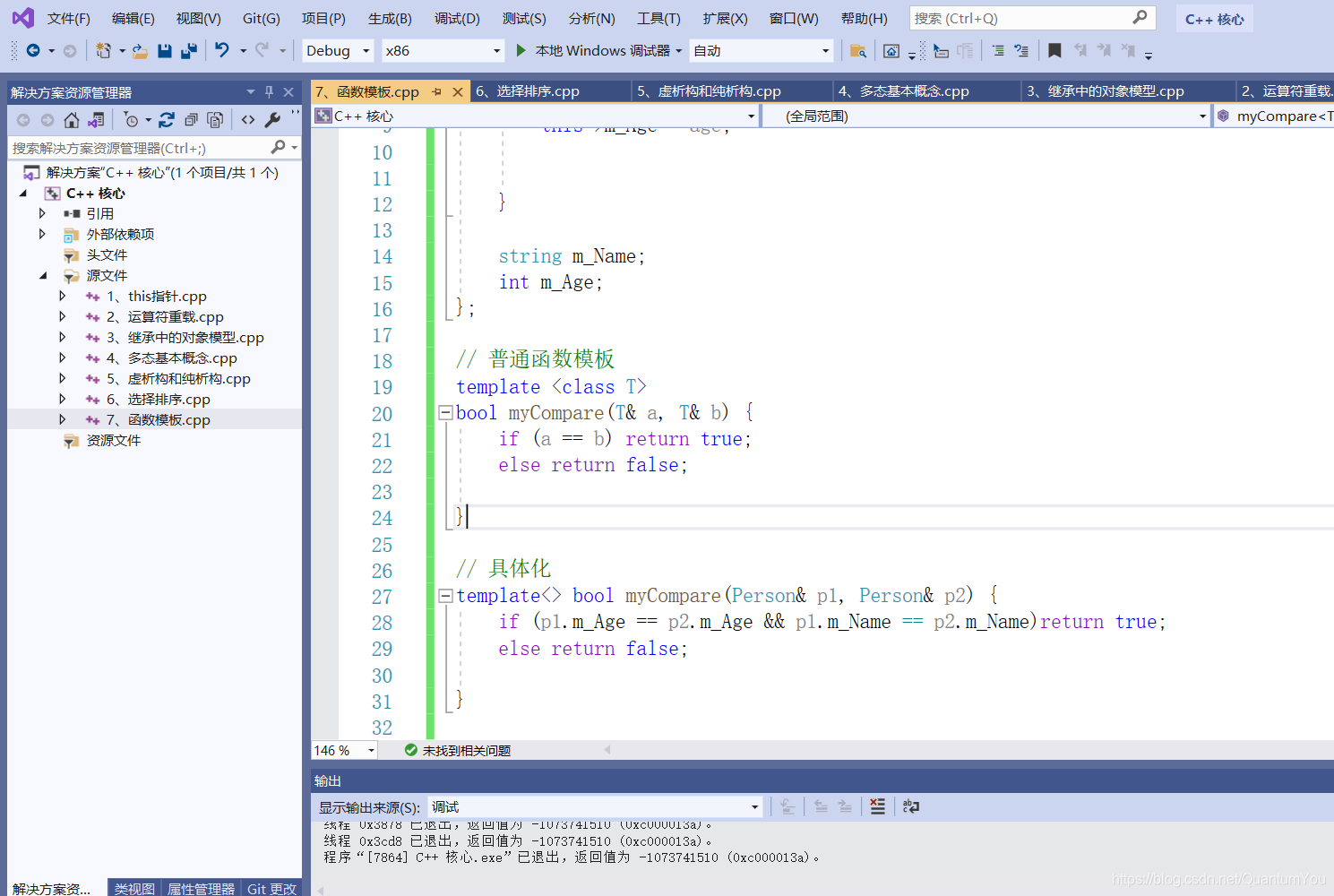
template<class T>  
 void f(T a, T b)  
 {   
 if(a > b) { ... }  
 }

在上述代码中，如果T的数据类型传入的是像Person这样的自定义数据类型，也无法正常运行

因此C++为了解决这种问题，提供模板的重载，可以为这些**特定的类型**提供**具体化的模板**

**示例：**

#include<iostream>  
using namespace std;  
  
#include <string>  
  
class Person  
{  
public:  
 Person(string name, int age)  
 {  
 this->m\_Name = name;  
 this->m\_Age = age;  
 }  
 string m\_Name;  
 int m\_Age;  
};  
  
//普通函数模板  
template<class T>  
bool myCompare(T& a, T& b)  
{  
 if (a == b)  
 {  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}  
  
  
//具体化，显示具体化的原型和定意思以template<>开头，并通过名称来指出类型  
//具体化优先于常规模板  
template<> bool myCompare(Person &p1, Person &p2)  
{  
 if ( p1.m\_Name == p2.m\_Name && p1.m\_Age == p2.m\_Age)  
 {  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}  
  
void test01()  
{  
 int a = 10;  
 int b = 20;  
 //内置数据类型可以直接使用通用的函数模板  
 bool ret = myCompare(a, b);  
 if (ret)  
 {  
 cout << "a == b " << endl;  
 }  
 else  
 {  
 cout << "a != b " << endl;  
 }  
}  
  
void test02()  
{  
 Person p1("Tom", 10);  
 Person p2("Tom", 10);  
 //自定义数据类型，不会调用普通的函数模板  
 //可以创建具体化的Person数据类型的模板，用于特殊处理这个类型  
 bool ret = myCompare(p1, p2);  
 if (ret)  
 {  
 cout << "p1 == p2 " << endl;  
 }  
 else  
 {  
 cout << "p1 != p2 " << endl;  
 }  
}  
  
int main() {  
  
 test01();  
  
 test02();  
  
 system("pause");  
  
 return 0;  
}



总结：

* 利用具体化的模板，可以解决自定义类型的通用化(具体化优先于常规模板)
* 学习模板并不是为了写模板，而是在STL能够运用系统提供的模板