

# C++模板

模板是 C++ 支持参数化多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

模板是一种对类型进行参数化的工具；

通常有两种形式：函数模板和类模板；

函数模板针对仅参数类型不同的函数；

类模板针对仅数据成员和成员函数类型不同的类。

使用模板的目的就是能够让程序员编写与类型无关的代码。比如编写了一个交换两个整型 `int` 类型的 `swap` 函数，这个函数就只能实现 `int` 型，对 `double`，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个 `swap` 函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个 `swap` 模板函数，即可以实现 `int` 型，又可以实现 `double` 型的交换。模板可以应用于函数和类。下面分别介绍。

注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在 `main` 函数中声明或定义一个模板。

## 一、函数模板通式

---

### 1、函数模板的格式：

```
template <class 形参名, class 形参名, .....> 返回类型 函数名 (参数列表)
```

```
{  
  
    函数体  
  
}
```

其中 `template` 和 `class` 是关键字，`class` 可以用 `typename` 关键字代替，在这里 `typename` 和 `class` 没区别，`<>` 括号中的参数叫模板形参，模板形参和函数形参很相像，模板形参不能为空。一旦声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，即可以在该函数中使用内置类型的地方都可以使用模板形参名。模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际的模板实参类型就称他实例化了函数模板的一个实例。比如 `swap` 的模板函数形式为

```
template <class T> void swap(T& a, T& b){},
```

当调用这样的模板函数时类型 `T` 就会被被调用时的类型所代替，比如 `swap(a,b)` 其中 `a` 和 `b` 是 `int` 型，这时模板函数 `swap` 中的形参 `T` 就会被 `int` 所代替，模板函数就变为 `swap(int &a, int &b)`。而当 `swap(c,d)` 其中 `c` 和 `d` 是 `double` 类型时，模板函数会被替换为 `swap(double &a, double &b)`，这样就实现了函数的实现与类型无关的代码。

2、注意：对于函数模板而言不存在 `h(int,int)` 这样的调用，不能在函数调用的参数中指定模板形参的类型，对函数模板的调用应使用实参推演来进行，即只能进行 `h(2,3)` 这样的调用，或者 `int a, b; h(a,b)`。

函数模板的示例演示将在下文中涉及！

## 二、类模板通式

---

### 1、类模板的格式为：

```
template<class 形参名, class 形参名, ...> class 类名  
{ ...};
```

类模板和函数模板都是以 `template` 开始后接模板形参列表组成，模板形参不能为空，一旦声明了类模板就可以用类模板的形参名声明类中的成员变量和成员函数，即可以在类中使用内置类型的地方都可以使用模板形参名来声明。比如

```
template<class T> class A{public: T a; T b; T hy(T c, T &d);};
```

在类 A 中声明了两个类型为 T 的成员变量 a 和 b，还声明了一个返回类型为 T 带两个参数类型为 T 的函数 hy。

2、类模板对象的创建：比如一个模板类 A，则使用类模板创建对象的方法为 `A<int> m;`；在类 A 后面跟上一个 `<>` 尖括号并在里面填上相应的类型，这样的话类 A 中凡是用到模板形参的地方都会被 int 所代替。当类模板有两个模板形参时创建对象的方法为 `A<int, double> m;`；类型之间用逗号隔开。

3、对于类模板，模板形参的类型必须在类名后的尖括号中明确指定。比如 `A<2> m;`；用这种方法把模板形参设置为 int 是错误的（编译错误：error C2079: 'a' uses undefined class 'A<int>'），类模板形参不存在实参推演的问题。也就是说不能把整型值 2 推演为 int 型传递给模板形参。要把类模板形参调置为 int 型必须这样指定 `A<int> m。`

### 4、在类模板外部定义成员函数的方法为：

```
template<模板形参列表> 函数返回类型 类名<模板形参名>::函数名(参数列表){函数体},
```

比如有两个模板形参 T1, T2 的类 A 中含有一个 `void h()` 函数，则定义该函

数的语法为：

```
template<class T1,class T2> void A<T1,T2>::h(){}。
```

注意：当在类外面定义类的成员时 `template` 后面的模板形参应与要定义的类的模板形参一致。

5、再次提醒注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在 `main` 函数中声明或定义一个模板。

### 三、模板的形参

---

有三种类型的模板形参：类型形参，非类型形参和模板形参。

#### 1、类型形参

1.1 、类型模板形参：类型形参由关键字 `class` 或 `typename` 后接说明符构成，如 `template<class T> void h(T a){}`；其中 `T` 就是一个类型形参，类型形参的名字由用户自己确定。模板形参表示的是一个未知的类型。模板类型形参可作为类型说明符用在模板中的任何地方，与内置类型说明符或类类型说明符的使用方式完全相同，即可以用于指定返回类型，变量声明等。

作者原版：1.2、不能为同一个模板类型形参指定两种不同的类型，比如 `template<class T>void h(T a, T b){}`，语句调用 `h(2, 3.2)`将出错，因为该语句给同一模板形参 `T` 指定了两种类型，第一个实参 `2` 把模板形参 `T` 指定为 `int`，而第二个实参 `3.2` 把模板形参指定为 `double`，两种类型的形参不一致，会出错。

(针对函数模板)

作者原版：1.2 针对函数模板是正确的，但是忽略了类模板。下面

将对类模板的情况进行补充。

本人添加1.2补充版(针对于类模板)、当我们声明类对象为:A<int> a, 比如 template<class T>T g(T a,T b){}, 语句调用 a.g(2, 3.2)在编译时不会出错, 但会有警告, 因为在声明类对象的时候已经将 T 转换为 int 类型, 而第二个实参 3.2 把模板形参指定为 double, 在运行时, 会对 3.2 进行强制类型转换为 3。当我们声明类的对象为: A<double> a,此时就不会有上述的警告, 因为从 int 到 double 是自动类型转换。

演示示例 1 :

TemplateDemo.h



```
1 #ifndef TEMPLATE_DEMO_HXX
2 #define TEMPLATE_DEMO_HXX
3
4 template<class T> class A{
5     public:
6         T g(T a,T b);
7         A();
8 };
9
10 #endif
```



TemplateDemo.cpp



```
1 #include<iostream.h>
```

```

2 #include "TemplateDemo.h"
3
4 template<class T> A<T>::A() {}
5
6 template<class T> T A<T>::g(T a,T b) {
7     return a+b;
8 }
9
10 void main() {
11     A<int> a;
12     cout<<a.g(2,3.2)<<endl;
13 }

```



编译结果:



```

1 -----Configuration: TemplateDemo - Win
32 Debug-----
2 Compiling...
3 TemplateDemo.cpp
4 G:\C++\CDaima\TemplateDemo\TemplateDemo.cpp(12) : wa
rning C4244: 'argument' : conversion from 'const double' to
'int', possible loss of data
5
6 TemplateDemo.obj - 0 error(s), 1 warning(s)

```



运行结果: 5

我们从上面的测试示例中可以看出, 并非作者原作中的那么严密! 此处

仅是本人跟人测试结果！请大家本着实事求是的态度，自行验证！

## 2、非类型形参

2.1 、非类型模板形参：模板的非类型形参也就是内置类型形参，如 `template<class T, int a> class B{};`其中 `int a` 就是非类型的模板形参。

2.2、非类型形参在模板定义的内部是常量值，也就是说非类型形参在模板的内部是常量。

2.3、非类型模板的形参只能是整型，指针和引用，像 `double` , `String`, `String **`这样的类型是不允许的。但是 `double &` , `double *` , 对象的引用或指针是正确的。

2.4、调用非类型模板形参的实参必须是一个常量表达式，即他必须能在编译时计算出结果。

2.5 、注意：任何局部对象，局部变量，局部对象的地址，局部变量的地址都不是一个常量表达式，都不能用作非类型模板形参的实参。全局指针类型，全局变量，全局对象也不是一个常量表达式，不能用作非类型模板形参的实参。

2.6、全局变量的地址或引用，全局对象的地址或引用 `const` 类型变量是常量表达式，可以用作非类型模板形参的实参。

2.7 、`sizeof` 表达式的结果是一个常量表达式，也能用作非类型模板形参的实参。

2.8 、当模板的形参是整型时调用该模板时的实参必须是整型的，且在编译期间是常量，比如 `template <class T, int a> class A{};`如果有 `int b` , 这时

A<int, b> m;将出错，因为 b 不是常量，如果 const int b，这时 A<int, b> m;就是正确的，因为这时 b 是常量。

2.9 、非类型形参一般不应用于函数模板中，比如有函数模板  
template<class T, int a> void h(T b){}，若使用 h(2)调用会出现无法为非类型形参  
a 推演出参数的错误，对这种模板函数可以用显示模板实参来解决，如用 h<int,  
3>(2)这样就把非类型形参 a 设置为整数 3。显示模板实参在后面介绍。

### 2.10、非类型模板形参的形参和实参间所允许的转换

1、允许从数组到指针，从函数到指针的转换。如：template <int  
\*a> class A{}; int b[1]; A<b> m;即数组到指针的转换

2、const 修饰符的转换。如：template<const int \*a> class A{}; int  
b; A<&b> m; 即从 int \*到 const int \*的转换。

3、提升转换。如：template<int a> class A{}; const short b=2; A<b>  
m; 即从 short 到 int 的提升转换

4、整值转换。如：template<unsigned int a> class A{}; A<3> m;  
即从 int 到 unsigned int 的转换。

5、常规转换。

非类型形参演示示例 1：

由用户自己亲自指定栈的大小，并实现栈的相关操作。

TemplateDemo.h



```
1 #ifndef TEMPLATE_DEMO_HXX
2 #define TEMPLATE_DEMO_HXX
3
```



```
4 template<class T,int MAXSIZE> class Stack{//MAXSIZE  
由用户创建对象时自行设置
```

```
5     private:  
6         T elems[MAXSIZE];    // 包含元素的数组  
7         int numElems;    // 元素的当前总个数  
8     public:  
9         Stack();    //构造函数  
10        void push(T const&);    //压入元素  
11        void pop();    //弹出元素  
12        T top() const;    //返回栈顶元素  
13        bool empty() const{    // 返回栈是否为空  
14            return numElems == 0;  
15        }  
16        bool full() const{    // 返回栈是否已满  
17            return numElems == MAXSIZE;  
18        }  
19 };
```

20

```
21 template <class T,int MAXSIZE>  
22 Stack<T,MAXSIZE>::Stack():numElems(0){    // 初始时栈不含元素  
23     // 不做任何事情  
24 }  
25
```

```
26 template <class T,int MAXSIZE>  
27 void Stack<T, MAXSIZE>::push(T const& elem){
```

```

28     if(numElems == MAXSIZE){
29         throw std::out_of_range("Stack<>::push(): stack is full");
30     }
31     elems[numElems] = elem;    // 附加元素
32     ++numElems;                // 增加元素的个数
33 }
34
35 template<class T,int MAXSIZE>
36 void Stack<T,MAXSIZE>::pop(){
37     if (numElems <= 0) {
38         throw std::out_of_range("Stack<>::pop(): empty stack");
39     }
40     --numElems;                // 减少元素的个数
41 }
42
43 template <class T,int MAXSIZE>
44 T Stack<T,MAXSIZE>::top()const{
45     if (numElems <= 0) {
46         throw std::out_of_range("Stack<>::top(): empty stack");
47     }
48     return elems[numElems-1]; // 返回最后一个元素
49 }
50
51 #endif

```



## TemplateDemo.cpp



```
1 #include<iostream.h>
2 #include <iostream>
3 #include <string>
4 #include <cstdlib>
5 #include "TemplateDemo.h"
6
7 int main(){
8     try {
9         Stack<int,20>  int20Stack;  // 可以存储 20 个 int
元素的栈
10        Stack<int,40>  int40Stack;  // 可以存储 40 个 int
元素的栈
11        Stack<std::string,40> stringStack; // 可存储 4
0 个 string 元素的栈
12
13        // 使用可存储 20 个 int 元素的栈
14        int20Stack.push(7);
15        std::cout << int20Stack.top() << std::endl;
//7
16        int20Stack.pop();
17
18        // 使用可存储 40 个 string 的栈
19        stringStack.push("hello");
20        std::cout << stringStack.top() << std::endl;
//hello
21        stringStack.pop();
```

```

22         stringStack.pop();    //Exception: Stack<>::~p
op<>: empty stack

23         return 0;

24     }

25     catch (std::exception const& ex) {

26         std::cerr << "Exception: " << ex.what() << st
d::endl;

27         return EXIT_FAILURE; // 退出程序且有 ERROR 标记

28     }

29 }

```



运行结果:

```

"G:\C++\CDaima\TemplateDemo\Debug\TemplateDemo.exe"
7
hello
Exception: Stack<>::~pop<>: empty stack
Press any key to continue

```

非类型形参演示示例 2:

TemplateDemo01.h



```

1 #ifndef TEMPLATE_DEMO_01

2 #define TEMPLATE_DEMO_01

3

4 template<typename T> class CompareDemo{

5     public:

6         int compare(const T&, const T&);

7 };

8

```

```

9 template<typename T>
10 int CompareDemo<T>::compare(const T& a,const T& b){
11     if((a-b)>0)
12         return 1;
13     else if((a-b)<0)
14         return -1;
15     else
16         return 0;
17 }
18
19 #endif

```



TemplateDemo01.cpp



```

1 #include<iostream.h>
2 #include "TemplateDemo01.h"
3
4 void main(){
5     CompareDemo<int> cd;
6     cout<<cd.compare(2,3)<<endl;
7 }

```



运行结果:                -1



```

1 #include<iostream.h>
2 #include "TemplateDemo01.h"
3

```

```
4 void main() {  
5     CompareDemo<double> cd;  
6     cout<<cd.compare(3.2,3.1)<<endl;  
7 }
```



运行结果: 1

TemplateDemo01.h 改动如下:



```
1 #ifndef TEMPLATE_DEMO_01  
2 #define TEMPLATE_DEMO_01  
3  
4 template<typename T> class CompareDemo{  
5     public:  
6         int compare(T&, T&);  
7 };  
8  
9 template<typename T>  
10 int CompareDemo<T>::compare(T& a,T& b) {  
11     if((a-b)>0)  
12         return 1;  
13     else if((a-b)<0)  
14         return -1;  
15     else  
16         return 0;  
17 }  
18
```

```
19 #endif
```



TempalteDemo01.cpp



```
1 #include<iostream.h>
2 #include "TemplateDemo01.h"
3
4 void main() {
5     CompareDemo<int> cd;
6     int a=2,b=3;
7     cout<<cd.compare(a,b)<<endl;
8 }
```



非类型形参演示示例 3:

TemplateDemo02.cpp

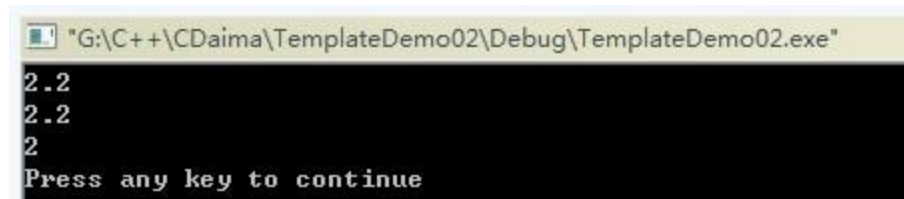


```
1 #include<iostream.h>
2
3 template<typename T>
4 const T& max(const T& a,const T& b) {
5     return a>b ? a:b;
6 }
7
8 void main() {
9     cout<<max(2.1,2.2)<<endl;//模板实参被隐式推演成 double
10 }
```

```
10      cout<<max<double>(2.1,2.2)<<endl;//显示指定模板参  
数。  
  
11      cout<<max<int>(2.1,2.2)<<endl;//显示指定的模板参  
数，会将函数函数直接转换为 int。  
  
12 }
```



运行结果:



```
"G:\C++\CDaima\TemplateDemo02\Debug\TemplateDemo02.exe"  
2.2  
2.2  
2  
Press any key to continue
```

