

# C++ 模板详解

## 分类 编程技术

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函数声明一种一般模式，使用类中的某些数据成员或成员函数的参数数，返回值得任意类型。

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。

模板针对仅数据成员或成员函数类型不同的类。

使用模板的目的就是能够防止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形int类型的swap函数，这个函数就只能实现int型、double、字符串等类型形式实现，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防止程序员的实现与类型无关，比如一个swap模板函数，即可实现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局部，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、函数模板概述

### 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区别，<>括号中的参数叫模板形参，模板形参和函数形参很像，模板形参不能为空，但是一旦声明了模板形参，则使用模板创建对象的方法为A<int> m;在类A后面加上一个<>实参号并在里面写上相参的类型，这样的类A中凡是使用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;类模板之间用逗号隔开。

2. 注意：对于函数模板函数不存在在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导得出类型。

函数模板的示例程序将在下文中涉及！

## 二、类模板概述

### 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开始后按模板形参列表组成，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，即可以在类中使用类内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 f(T c, T d);};
```

在类A中声明了两个类型为T的成员变量a,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f。

2. 类模板的创建: 比如一个模板类A，则使用模板创建对象的方法为A<int> m;在类A后面加上一个<>实参号并在里面写上相参的类型，这样的类A中凡是使用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;类模板之间用逗号隔开。

3. 对于类模板，模板形参的类型必须在类名后面的括号中声明清楚，比如A<2> m;用这种方法来指定模板形参为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导的问题，也就是说不能把整型值当作int型传递传给模板形参，若把类模板形参指定为int型必须这样写A<int> m。

4. 在类模板外部定义成员函数或成员变量的方法为:

```
template<模板形参列表> 函数返回类型 类名<模板形参列表>::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：当在类外定义类成员函数或成员变量时，模板形参列表与类定义中的模板形参列表必须一致。

5. 再次提醒注意：模板的声明或定义只能在局部，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、类型形参: 类型形参由关键字class或typename后接标识符构成，如template<class T> void h(T a,T b);其中T就是一个类型形参，类型形参的字母由用户自己确定，模板形参的值是一个未知的类型，模板形参形参可作为类型说明符用在模板中的任何地方，与内联变量和函数说明符中的类型说明符的作用方式完全相同，即可以用于指定返回类型，变量声明等。

**作者提醒: 1.2** 不能为同一个模板形参指定两种不同的类型，第一个实参在模板形参指定为int型，第二个实参3.2把模板形参指定为double，因为该语句中同一模板形参指定了两种类型，当我们声明的对象为: 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，两种类型的形参不一致，会出现**【针对函数模板】**

作者提醒: 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 (针对类模板)**，当我们声明对象为: A<int> a，比如template<class T> T g(T a, T b);，语句调用g(2, 3.2)在编译时不会报错，但会有警告，因为在声明类对象的时候已将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
#ifndef TEMPLATE_DEMO_HXX
#define TEMPLATE_DEMO_HXX

template<class T> class A{
public:
    T g(T a, T b);
};

// 函数模板
#endif

TemplateDemo.cpp
#include<iostream.h>
#include "TemplateDemo.h"

template<class T> A<T>::A(){}

template<class T> T A<T>::g(T a, T b){
    return a+b;
}

void main(){
    A<int> a;
    cout<<a.g(2,3)<<endl;
}
```

编译结果:

```
-----Configuration: TemplateDemo - Win32 Debug-----
Compiling...
TemplateDemo.cpp
g:\c++\vc\bin\TemplateDemo\TemplateDemo.cpp(12) : warning C4264: 'argument': conversion from 'const double' to 'int', possible loss of data
TemplateDemo.obj - 0 error(s), 1 warning(s)
```

运行结果: 5

我们从上面的示例式例中可以看出，并非作者原作中的那么严密！此处仅是本人测试实例！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: 模板的非类型形参就是内联变量形参，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 \*, 对象的引用和指针是正确的。

2.4、调用非类型模板形参的实参必须是一个常量表达式，即必须能够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，即可以使用非类型模板形参的实参。

2.8、当模板的形参是整型时调用模板形参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，因为这时是常量。

2.9、非类型形参一般不由函数模板中，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推演出出错的错误，对这种模板函数可以用显式模板实参来解决，如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允许互转换

1、从指针到非指针的转换，如: template<int a> class A{}; int b[1]; A<b> m;即数组到指针的转换

2、常量指针到非指针的转换，如: template<const int a> class A{}; int b; A<b> m; 即从int \*到const int \*的转换。

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

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

5、常非指针转换。

### 非类型形参演示示例1:

由用户自己确定非类型形参的大小，有实现性的相关操作。

```
TemplateDemo.h
#ifndef TEMPLATE_DEMO_HXX
#define TEMPLATE_DEMO_HXX

template<class T, int MAXSIZE> class Stack{ // MAXSIZE由用户创建对象时自行设置
private:
    T elems[MAXSIZE]; // 包含元素的数组
    int numElems; // 元素的当前个数
public:
    Stack(); // 构造函数
    void push(T const&); // 压入元素
    void pop(); // 弹出元素
    T top() const; // 返回栈顶元素
    bool empty() const; // 返回栈是否为空
    ~Stack(); // 析构函数
    bool full() const; // 返回栈是否已满
    ~Stack(); // 析构函数
};

template<class T, int MAXSIZE>
Stack<T, MAXSIZE>::Stack():numElems(0) // 初始化栈不含元素
{
    // 不能有任何事情
}

template<class T, int MAXSIZE>
void Stack<T, MAXSIZE>::push(T const& elem){
    if(numElems == MAXSIZE){
        throw std::out_of_range("Stack::push(): stack is full");
    }
    elems[numElems] = elem; // 增加元素
    ++numElems; // 增加元素的个数
}

template<class T, int MAXSIZE>
void Stack<T, MAXSIZE>::pop(){
    if(numElems == 0){
        throw std::out_of_range("Stack::pop(): empty stack");
    }
    --numElems; // 减少元素的个数
}

template<class T, int MAXSIZE>
T Stack<T, MAXSIZE>::top() const{
    if(numElems == 0){
        throw std::out_of_range("Stack::top(): empty stack");
    }
    return elems[numElems-1]; // 返回最后一个元素
}

#endif
```

```
TemplateDemo.cpp
#include<iostream.h>
#include "TemplateDemo.h"

int main(){
    try{
        Stack<int, 20> int20Stack; // 可以存储20个int元素的栈
        Stack<int, 40> int40Stack; // 可以存储40个int元素的栈
        Stack<std::string, 40> stringStack; // 可存储40个string元素的栈

        // 使用可存储20个int元素的栈
        std::cout << int20Stack.top() << std::endl; // 1
        int20Stack.pop();

        // 使用可存储40个string的栈
        stringStack.push("Hello");
        std::cout << stringStack.top() << std::endl; // Hello
        stringStack.pop();
        stringStack.pop(); // 返回栈: Stack::pop(): empty stack
        return 0;
    }
    catch (std::exception const& ex){
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE; // 由程序产生并带有ERROR标记
    }
}
```

```
g:\c++\vc\bin\TemplateDemo\Debug\TemplateDemo.exe
?
Hello
Exception: Stack::top(): empty stack
Press any key to continue
```

### 非类型形参演示示例2:

```
TemplateDemo01.h
#ifndef TEMPLATE_DEMO_01
#define TEMPLATE_DEMO_01

template<typename T> class CompareDemo{
public:
    int compare(const T&, const T&);
};

template<typename T>
int CompareDemo<T>::compare(const T& a, const T& b){
    if(a==b){
        return 1;
    }
    else if(a<b){
        return -1;
    }
    else
        return 0;
}

#endif
```

```
TemplateDemo01.cpp
#include<iostream.h>
#include "TemplateDemo01.h"

void main(){
    CompareDemo<int> cd;
    cout<<cd.compare(2,3)<<endl;
}
```

运行结果: -1

```
TemplateDemo01.cpp
#include<iostream.h>
#include "TemplateDemo01.h"

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

运行结果: 1

```
TemplateDemo01.h
#ifndef TEMPLATE_DEMO_01
#define TEMPLATE_DEMO_01

template<typename T> class CompareDemo{
public:
    int compare(T&, T&);
};

template<typename T>
int CompareDemo<T>::compare(T& a, T& b){
    if(a==b){
        return 1;
    }
    else if(a<b){
        return -1;
    }
    else
        return 0;
}

#endif
```

```
TemplateDemo01.cpp
#include<iostream.h>
#include "TemplateDemo01.h"

void main(){
    CompareDemo<int> cd;
    int a=2, b=3;
    cout<<cd.compare(a,b)<<endl;
}
```

### 非类型形参演示示例3:

```
TemplateDemo02.cpp
#include<iostream.h>

const T& max(const T& a, const T& b){
    return a>b ? a:b;
}

void main(){
    cout<<max(int(2.1, 2.2)<<endl; //模板实参被隐式转换为double
    cout<<max(double(2.1, 2.2)<<endl; //显示指定模板实参。
    cout<<max(int(2.1, 2.2)<<endl; //显示指定的模板实参，会将函数参数直接转换为int。
}
```

```
g:\c++\vc\bin\TemplateDemo02\Debug\TemplateDemo02.exe
2.2
2.2
Press any key to continue
```

cout<<max(int(2.1, 2.2)<<endl; //显示指定的模板实参，会将函数参数直接转换为int。

此语句会出现警告:

```
-----Configuration: TemplateDemo02 - Win32 Debug-----
Compiling...
TemplateDemo02.cpp
g:\c++\vc\bin\TemplateDemo02\TemplateDemo02.cpp(11) :
warning C4264: 'argument': conversion from 'const double' to 'const int', possible loss of data
g:\c++\vc\bin\TemplateDemo02\TemplateDemo02.cpp(12) :
warning C4264: 'argument': conversion from 'const double' to 'const int', possible loss of data
TemplateDemo02.obj - 0 error(s), 2 warning(s)
```

## 四、类模板的默认模板类型形参

1. 可以为类模板的类型形参提供默认值，但不能为函数模板的类型形参提供默认值，函数模板和类模板都可以为模板的非类型形参提供默认值。

2. 类模板的类型形参默认值形式为: template<class T1, class T2= int> class A{};为第二个模板类型形参T2提供int型的默认值。

3. 类模板的非类型形参默认值和函数模板参数一样，如果有多个类型形参则从第一个形参设定了默认值之后的所有模板形参都要设定默认值，比如template<class T1= int, class T2= class A{};就是错误的，因为T1给出了默认值，而T2没有设定。

4. 在类模板的外部定义类成员时template后的形参表应省略默认的类型形参，比如template<class T1, class T2= int> class A{public: void h();}; 定义方法为template<class T1, class T2= void A<T1, T2>::h();。

### 定义类模板类型形参:

#### 演示实例1:

```
TemplateDemo.h
#ifndef TEMPLATE_DEMO_HXX
#define TEMPLATE_DEMO_HXX

template<class T> class A{
public:
    T g(T a, T b);
};

// 函数模板
#endif

TemplateDemo.cpp
#include<iostream.h>
#include "TemplateDemo.h"

template<class T> A<T>::A(){}

template<class T> T A<T>::g(T a, T b){
    return a+b;
}

void main(){
    A<int> a;
    cout<<a.g(2,3)<<endl;
}
```

运行结果: 5

### 类模板的默认模板类型形参示例1:

```
TemplateDemo03.h
#ifndef TEMPLATE_DEMO_03
#define TEMPLATE_DEMO_03

// 定义带默认模板类型形参的类模板
template<class T1, class T2= int> class CellDemo{
public:
    int cell(T1, T2);
};

// 在类模板的外部定义类成员时template后的形参表应省略默认的类型形参。
template<class T1, class T2>
int CellDemo<T1, T2>::cell(T1 a, T2 b){
    return a+b;
}

#endif
```

```
TemplateDemo03.cpp
#include<iostream.h>
#include "TemplateDemo03.h"

void main(){
    CellDemo<int, int> cd;
    cout<<cd.cell(2,3)<<endl;
}
```

运行结果: 2

在模板的外部定义类成员时template后的形参表应省略默认的类型形参，如果没有省略，不会出错编译错误而是提出警告:

```
-----Configuration: TemplateDemo03 - Win32 Debug-----
Compiling...
TemplateDemo03.cpp
g:\c++\vc\bin\TemplateDemo03\templateDemo03.h(12) :
error C2248: 'CellDemo<T1, T2>::cell': unable to resolve function overload
g:\c++\vc\bin\templateDemo03\templateDemo03.cpp(6) :
error C2803: 'cd': undeclared identifier
g:\c++\vc\bin\templateDemo03\templateDemo03.cpp(6) :
error C2228: left of '.cell' must have class/struct/union type
Error executing cl.exe.

TemplateDemo03.obj - 3 error(s), 0 warning(s)
```

从上面的例子我们可以看出，和上例是一样的错误，从实例中我们可以总结如下: 类模板如果有多个类型形参，如果使用类型形参默认值则尽量在参列表的末尾，而且默认的类型形参必须相同，如果从第一个形参设定了默认值之后的所有模板形参都要设定和第一个形参同类型的默认值。(声明: 本人也是用模板++，以上只是我经过实例演示对原作者提出的一些提醒，可能我的举例不到位之处，还望大家手下留情，共同交流进步，给像一样的菜鸟提供一个学习的平台！)

按下来验证“不能为函数模板的类型形参提供默认值”:

### 类模板的默认模板类型形参示例:

```
TemplateDemo04.cpp
#include<iostream.h>

template<class T1, class T2, class T3>
T1 sum(T1 a, T2 b, T3 c){
    return a+b+c;
}

void main(){
    cout<<sum(double, double)(1.1, 2.1, 3.1)<<endl;
}
```

编译错误:

```
-----Configuration: TemplateDemo04 - Win32 Debug-----
Compiling...
TemplateDemo04.cpp
g:\c++\vc\bin\templateDemo04\templateDemo04.cpp(4) :
error C2802: type 'int' unexpected
Error executing cl.exe.

TemplateDemo04.obj - 1 error(s), 0 warning(s)
```

### 修改之后的 TemplateDemo04.cpp

```
TemplateDemo04.cpp
#include<iostream.h>

template<class T1, class T2, class T3>
T1 sum(T1 a, T2 b, T3 c){
    return a+b+c;
}

void main(){
    cout<<sum<double, double>(1.1, 2.1, 3.1)<<endl;
}
```

运行结果: 261.1

原作者演示代码如下:

### 类模板非类型形参示例

类模板的声明或定义只能在局部，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。

类模板的定义

```
template<class T1, class A= public T g(T a, T b); A(); // 定义带有一个类模板类型形参T1的类A
template<class T1, class T2= class B public: void g(); // 定义带有两个类模板类型形参T1, T2的类B
// 定义类模板的默认模板类型形参，默认为int型常量，输出为int型常量。
```

```
template<class T1, class T2= int> class D(public: void g()); // 定义带有一个类模板类型形参的类模板，这里把T2默认设置为int型
// template<class T2= int, class T2= class E{}; // 错误，为T1设定了默认类型形参T2后的所有形参都必须设定默认值。
```

以下为非类型形参的定义

非类型形参只能是整型、指针和引用，像double, String, String \*\*这样的类型是不允许的，但是double &, double \* 对非类型形参是允许的。

模板<class T1, int a> class C(public: void g()); // 定义模板的非类型形参，形参为整型

模板<class T1, int a> class C(public: void g()); // 定义模板的非类型形参，形参为int型的类A的对象

模板<class T1, double a> class C(public: void g()); // 定义模板的非类型形参，形参为double类型的引用。

class T1; // 定义模板的非类型形参，形参为double类型的引用。

template<class T1, A= class C{}; // 错误，对象不能做非类型形参，非类型模板形参的类型只能是对象的引用或指针。

template<class T1, A= int> class C{}; // 错误，非类型模板的形参不能是int，必须是对象的引用或指针。这条规则不适用于模板形参。

非类型模板形参的定义

非类型模板形参的定义

非类型模板形参的定义

非类型模板形参的定义