本文作者：黄邦勇帅

学习本文需要对 C++的构造函数，C++中的各种类成员，C++中类的继承，有一定的了解。
模板是 C++中的重点内容，因此应熟悉 C++模板的使用，本文是 C++的模板专题，因此本文集中介绍了 C++的模板问题，主要介绍了类模板和函数模板的形参与实参(这是学习模板的关键内容)，模板的具体化(特化)方法，模板实参推演，模板函数的匹配，类模板中的模板成员，模板与继承，模板与友元，typename 关键字。本文内容全面，简单易懂，是对于学习模板模棱两可的读者的很好的参考文现。
本文内容完全属于个人见解与参考文现的作者无关，其中难免有误解之处，望指出更正。

主要参考文献：
1、C++.Primer.Plus.第五版.中文版 [美]Stephen Prata 著 孙建春 韦强译 人民邮电出版社 2005 年 5 月
2、C++.Primer.Plus.第四版.中文版 Stanley B.Lippman、Barbara E.Moo 著 李师贤等译 人民邮电出版社 2006 年 3 月
3、C++.Primer.Plus.第三版.中文版 Stanley B.Lippman 等著 潘爱民 张丽译 中国电力出版社 2002 年 5 月
4、C++入门经典 第三版 [美]Ivor Horton 著 李予敏译 清华大学出版社 2006 年 1 月
5、C++参考大全 第四版 [美]Herbert Schidt 著 周志荣 朱德芳 于秀山等译 电子工业出版社 2003 年 9 月
6、21 天学通 第四版 C++ [美]Jesse Liberty 著 康博创作室 译 人民邮电出版社 2002 年 3 月

第 17 章 模板

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、模板函数通式

1、 模板函数的通用形式为：template <class 形参名，class 形参名> 反回类型 函数名(参数列表){函数体}。其中 template 和 class 是关见字，class 可以用 typename 关见字代替，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。一但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)。

二、类模板通式

1. **类模板的通用形式为：**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，一但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，还声明了一个反回类型为 T 带两个类型为 T 的函数 hy。
2. **类模板对象的创建：**比有一个模板类 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，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> 函数反回类型 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、模板的形数

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：类型形参由关见字 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，即可以用于指定反回类型，变量声明等。

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。

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 非模板类型的形参只能是 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。

2.9 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。

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、 类模板类型形参 默认值和函数的默认参数一样 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中的成员时 template 后的形参表应省略默认的形参类型。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(){}。

类模板非类型形参示例

```
//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。
//类模板的定义
template<class T>class A{public:T g(T a, T b); A();};  //定义带有一个类模板类型形参T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

//定义类模板的默认类型形参，默认类型形参不适合于函数模板。
template<class T1,class T2=int> class D{public: void g();}; //定义带默认类型形参的类模板。这里把T2默认设置为int型。
//template<class T1=int, class T2>class E{}; //错误，为T1设了默认类型形参则T1后面的所有形参都必须设置认默值。
```

```
//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 *对象的引用或指针是正确的。
template<class T1,int a> class Ci{public:void g();}; //定义模板的非类型形参，形参为整型
template<class T1,int &a>class Cip{public:void g();};
template<class T1,A<int>* m> class Cc{public:void g();}; //定义模板的模板类型形参，形参为int型的类A的对象的指针。
template<class T1,double *a>class Cd{public:void g();};  //定义模板的非类型形参，形参为double类型的引用。
class E{};    template<class T1,E &m> class Ce{};   //非类型模板形参为对象的引用。

//以下非类型形参的声明是错误的。
//template<class T1,A m>class Cc{}; //错误，对象不能做为非类型形参，非类型模板形参的类型只能是对象的引用或指针。
//template<class T1,double a>class Cc{}; //错误，非类型模板的形参不能是double类型，可以是double的引用。
//template<class T1,A<int> m>class Cc{}; //错误，非类型模板的形参不能是对象，必须是对象的引用或指针。这条规则对于模板型参也不例外。

//在类模板外部定义各种类成员的方法，
//typeid(变量名).name()的作用是提取变量名的类型，如int a，则cout<<typeid(a).name()将输出int
template<class T>   A<T>::A(){cout<<"class A goucao"<<typeid(T).name()<<endl;} //在类模板外部定义类的构造函数的方法
template<class T> T A<T>::g(T a,T b){cout<<"class A g(T a,T b)"<<endl;} //在类模板外部定义类模板的成员
template<class T1,class T2>  void B<T1,T2>::g(){cout<<"class g f()"<<typeid(T1).name()<<typeid(T2).name()<<endl;}
//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
template<class T1,int a>     void Ci<T1,a>::g(){cout<<"class Ci g()"<<typeid(T1).name()<<endl;}
template<class T1,int &a>    void Cip<T1,a>::g(){cout<<"class Cip g()"<<typeid(T1).name()<<endl;}
//在类外部定义类的成员时，template后的模板形参应与要定义的类的模板形参一致
template<class T1,A<int> *m> void Cc<T1,m>::g(){cout<<"class Cc g()"<<typeid(T1).name()<<endl;}
template<class T1,double* a> void Cd<T1,a>::g(){cout<<"class Cd g()"<<typeid(T1).name()<<endl;}

//带有默认类型形参的模板类，在类的外部定义成员的方法。
//在类外部定义类的成员时，template的形参表中默认值应省略
template<class T1,class T2>  void D<T1,T2>::g(){cout<<"class D g()"<<endl;}
//template<class T1,class T2=int> void D<T1,T2>::g(){cout<<"class D k()"<<endl;} //错误，在类模板外部定义带有默认类型的形参时，在template的形参表中默认值应省略。
//定义一些全局变量。
int e=2;   double ed=2.2;      double *pe=&ed;
A<int> mw; A<int> *pec=&mw;    E me;

//main函数开始
int main()
{  // template<class T>void h(){} //错误，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。
 //A<2> m; //错误，对类模板不存在实参推演问题，类模板必须在尖括号中明确指出其类型。
//类模板调用实例
A<int> ma; //输出"class A goucao int"创建int型的类模板A的对象ma。
B<int,int> mb; mb.g(); //输出"class B g() int int"创建类模板B的对象mb，并把类型形参T1和T2设计为int

//非类型形参的调用
//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。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。
//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。
//调用整型int型非类型形参的方法为名为 Ci，声明形式为template<class T1,int a> class Ci
Ci<int,3> mci;mci.g();  //正确，数值3是一个int型常量，输出"class Ci g() int"
const int a2=3; Ci<int,a2> mci1; mci1.g(); //正确，因为a2在这里是const型的常量。输出"class Ci g() int"
//Ci<int,a> mci; //错误，int型变量a是局部变量，不是一个常量表达式。
//Ci<int,e> mci; //错误，全局int型变量e也不是一个常量表达式。
//调用int&型非类型形参的方法类名为 Cip，声明形式为template<class T1,int &a>class Cip
Cip<int,e> mcip;  //正确，对全局变量的引用或地址是常量表达式。
//Cip<int,a> mcip1; //错误，局部变量的引用或地址不是常量表达式。
//调用double*类型的非类形形参类名为 Cd，声明形式为template<class T1,double *a>class Cd
Cd<int,&ed> mcd; //正确，全局变量的引用或地址是常量表达式。
//Cd<int,pe> mcd1; //错误，全局变量指针不是常量表达式。
//double dd=3.3;  Cd<int,&dd> mcd1; //错误，局部变量的地址不是常量表达式，不能用作非类型形参的实参
//Cd<int,&e> mcd;  //错误，非类型形参虽允许一些转换，但这个转换不能实现。
//调用模板类型形参对象A<int> *的方法类名为 Cc，声名形式为template<class T1,A<int>* m> class Cc
Cc<int,&mw> mcc; mcc.g(); //正确，全局对象的地址或者引用是常量表达式
//Cc<int,&ma> mcc;  //错误，局部变量的地址或引用不是常量表达式。
//Cc<int,pec> mcc2;  //错误，全局对象的指针不是常量表达式。
```

```
//调用非类型形参E&对象的引用的方法类名为 Ce。声明形式为template<class T1,E &m> class Ce
E me1; //Ce<int,me1> mce1; //错误，局部对象不是常量表达式
Ce<int,me> mce;  //正确，全局对象的指针或引用是常量表达式。
//非类型形参的转换示例，类名为 Ci
//非类型形参允许从数组到指针，从函数到指针的转换，const修饰符的转换，提升转换，整值转换，常规转换。
const short s=3; Ci<int,s> mci4; }//正确，虽然 short 型和 int 不完全匹配，但这里可以将 short 型转换为 int 型
```

五、模板的实例化，区别声明，定义，实例化的概恋

1、声明就是 让编译器知道有这么一个函数或者类 比如 void h(); class A;就声明了一个 无参函数h 和一个类 A。定义就是对函数或者类的实现。比如 void h(){}，class A{int a;}; 就定义了一个 什么也不做的函数h 和有一个变量 a 的类 A。

2、声明和定义 都不会创建实例，只有在使用该函数或 者类的时候才会创建一个实例 比如在 main 函数中调用函数 h，创建类 A 的对象时都会创建函数 h 和类 A 的实例，如果函数 h 是外部函数则如 果在main 函数中 没有调用函数h，就不会 创建函数h 的实例。

3、模板的声明，定义，实例化：声明一个模板的形式为 template<class T> void h(T a); 或 template<class T> class A;这就声明了一个模板函数 h 和一个模板类 A。注意后面有个 分号。模板的定义和函数或类的定义相同。

模板的实 例化发生在调用该模板函数和模板类时,比如 h(2)或 A<int> m;就创建了一个 int 型的 h 函数实 例和类A 的 int 实例版本。

当模板 被实例化之后就会创建该模板的一个实例，在下次调用到相同的模板实例时就不会生成新的实例，而会调用以前创建的那个实例。比如有模板函数 template<class T> void h(T a){}，则有调用 h(2)会生成一个 int 型的模板函数实 例当第二次调用如 h(44)时会使用以 前生成的int 型模板函数实 例而不会创建新的实例。

4、当我们只是声明一个类模板的指针或引用时就没必要知道类模板的定义，也就是说 创建类模板的指针或引用时不会创建类模板的实例,只有指 针被解引用或者该问类中的成员时才需要知道类模板的定义,才会实例化该类模板。比如 class A； A *m; A &n;都不需要知道该类的定义，也不会实 例化该类。但是 m->a 或 n->a 时就会 需要知道该类的定义了，因为这里指针访问类中的成员，须要类的定义，这时也要创建一个类的实例。

六、模板类型形参与实参间所允许的转换或模板实参推演

1、 **模板实参推演：**当函数模板 被调用时，对函数实参类型的检查决定了模板实参的类型和值这个过程叫做模板实参推演。比如 template<class T> void h(T a){}；h(1)；h(2.2)，第一个调用因为实参 1 是 int 型的，所以模板形参 T 被推演为 int 型，因此函数体中的所有 T 被替换为 int。而第二个调用中 double 类型的实数 3.2 决定了 T 的类型为 double。

2、在模板 被实例化后就会生成一个新的实例，但这个新生成的实例不存在类型转换。比如有函数模板 template<class T> void h(T a){}。int a=2; short b=3; 第一次调用 h(a)生成一个 int 型的实 例版本，但是当用 h(b)调用时不会使用 上次生成的 int 实例把 short 转换为 int，而是会 另外生成一个新的short 型的实 例。

3、在模板实参 推演的过程中有时类型并不会完全匹配，这时编译器允许以下几种实参到模板形参的转换，这些转换不会生成新的实例。

3.1、**数组到指针的转换或函数到指针的转换：**比如 template<class T> void h(T * a){}，int b[3]={1,2,3}；h(b);这时数组 b 和类型 T *不是完全 匹配，但允许从数组到指针的转换因此数组 b 被转换成 int *，而类型形参 T 被转换成 int，也就是说函数体中的 T 被替换成 int。

3.2、限制修饰符转换：即把 const 或 volatile 限定符加到指针上。比如 template<class T> void h(const T* a){}，int b=3; h(&b);虽然实参&b 与形参 const T*不完全匹配，但因为允许限制修饰符的转换，结果就把&b 转换成 const int *。而类形型参 T 被转换成 int。如果模板形参是非 const 类型，则无论实参是 const 类型还是非 const 类型调用 都不会产生新的实例。

3.3、**到一个基类的转换(该基类根据一个类模板实例化而来)：**比如 tessmplate<class T1>class A{}; template<class T1> class B:public A<T1>{}; template<class T2> void h(A<T2>& m){}，在 main 函数中有 B<int> n; h(n);函数调用的 子类对象 n 与函数的形参 A<T2>不完全 匹配，但允许到一个基类的转换。在这里转换的顺序为，首先把子类对象 n 转换为基类对象 A<int>，然后再用 A<int>去匹配函数的形参 A<T2>&，所以最后 T2 被转换为 int，也就是说函数体中的 T 将被替换为 int。

4、再次提醒：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)。

模板实参推演示例

```
//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。
//函数模板的定义
template<class T>void h(T a){cout<<"hansu h()"<<typeid(T).name()<<endl;}  //带有一个类型形参T的模板函数的定义方法，typeid(变量名).name()为测试变量类型的语句。
template<class T>void k(T a,T b){T c; cout<<"hansu k()"<<typeid(T).name()<<endl;} //注意语句T c。模板类型形参T可以用来声明变量，作为函数的反回类型，函数形参等凡是类类型能使用的地方。
template<class T1,class T2> void f(T1 a, T2 b){cout<<"hansu f()"<<typeid(T1).name()<<","<<typeid(T2).name()<<endl;}   //定义带有两个类型形参T1，T2的模板函数的方法
template<class T> void g(const T* a){T b;cout<<"hansu g()"<<typeid(b).name()<<endl;}
//template<class T1,class T2=int> void g(){}  //错误，默认模板类型形参不能用于函数模板，只能用于类模板上。
//main函数开始
int main()
{ // template<class T>void h(){} //错误，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。
```

```
//函数模板实参推演示例。
// h(int); //错误，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)。
//h函数形式为：template<class T>void h(T a)
 h(2);//输出"hansu h() int"使用函数模板推演，在这里数值2为int型，所以把类型形参T推演为int型。
 h(2.0); //输出"hansu h() double"，因为2.0为double型，所以将函数模板的类型形参推演为double型
//k函数形式为：template<class T>void k(T a,T b)
 k(2,3); //输出"hansu k() int"
 //k(2,3.0); //错误，模板形参T的类型不明确，因为k()函数第一个参数类型为int，第二个为double型，两个形参类型不一致。
//f函数的形式为：template<class T1,class T2> void f(T1 a, T2 b)
 f(3,4.0); //输出"hansu f() int,double"，这里不存在模板形参推演错误的问题，因为模板函数有两个类型形参T1和T2。在这里将T1推演为int，将T2推演为double。
 int a=3;double b=4;
 f(a,b); //输出同上，这里用变量名实现推板实参的推演。
//模板函数推演允许的转换示例，g函数的形式为template<class T> void g(const T* a)
 int a1[2]={1,2}; g(a1); //输出"hansu g() int"，数组的地址和形参const T*不完全匹配，所以将a1的地址T &转换为const T*，而a1是int型的，所以最后T推演为int。
 g(&b); //输出"hansu g() double",这里和上面的一样，只是把类型T转换为double型。
 h(&b); }//输出"hansu h() double *"这里把模参类型T推演为double *类型。
```

七、显示实例化，显示模板实参，显示具体化，模板特化，模板函数重载

7.1 函数模板的显示实例化

1、隐式实例化：比如有模板函数 template<class T> void h(T a){}。h(2)这时 h 函数的调用就是 隐式实例化，既参数 T 的类型是隐式确定的。

2、函数模板 显示实例化声明：其语法是：template 函数反回类型 函数名<实例化的类型> (函数形参表); 注意这是声明语句，要以分号结束。例如：template void h<int> (int a);这样就创建了一个 h 函数的 int 实例。再如有模板函数 template<class T> T h(T a){}，注意这里 h 函数的反回类型为 T，显示实例化的方法为 template int h<int>(int a); 把 h 模板函数实 例化为int 型。

3、类模板的显示实例化：和函数模板的 显示实例化一样都是以template 开始。比如 template class A<int,int>;将类 A 显示实例化为两个 int 型的类模板。这里要注意显示实例化后面不能有对象名，且以分号结束。

4、显示实例化可以让程序员控制模板实例化发生的时间。

5、对于给定的函数模板实例，显示实例化声明在一个文件中只能出现一次。

6、在显示实例化声明所在的文件中，函数模板的定义 必须给出；如果定义不可见，就会发生错误。

7、注意：不能在局部范围类显示实例化模板，实例化模板应放在全局范围内，即不能在 main 函数等局部范围中实 例化模板。因为模板的声明或定义不能在局部范围或函数内进行。

7.2 显示模板实参

1、显示模板实参：适用于函数模板，即在调用函数时 显示指定要调用的时参的类型。

2、格式：显示模板实参的格式为在调用模板函数的时候在函数名后用<>尖括号括住要显示表示的类型，比如有模板函数 template<class T> void h(T a, T b){}。则 h<double>(2, 3.2)就把模板形参 T 显示实例化为 double 类型。

3、显示模板实参用于同一个模板形参的类型不一致的情况。比如 template<class T> void h(T a, T b){}，则 h(2, 3.2)的调用会出错，因为两个实参类型不一致，第一个为 int 型，第二个为 double 型。而用 h<double>(2, 3.2)就是正确的，虽然两个模板形参的类型不一致 但这里把模板形参显示实例化为double 类型，这样的话就允许进行标准的隐式类型转换，即这里把第一个 int 参数转换为 double 类型的参数。

4、显示模板实参用法二：用于函数模板的反回类型中。例如有模板函数 template<class T1, class T2, class T3> T1 h(T2 a, T3 b){}，则语句 int a=h(2,3)或 h(2,4)就会出 现模板形参T1 无法推导的情况。而语句 int h(2,3)也会出错。用显示模板实参就参轻松解决这个问题，比如 h<int, int, int>(2,3)即把模板形参 T1 实例化为 int 型，T2 和 T3 也实例化为 int 型。

5、显示模板实参用法三：应用于模板函数的参数中 没有出现模板形参的情况，比如 template<class T>void h(){}如果在 main 函数中 直接调用h 函数如 h()就会出 现无法推演类型形参T 的类型的错 误，这时用显示模板实参就不会出现这种错误，调用方法为 h<int>()，把 h 函数的模板形参实 例化为int 型，从而避免这种错误。

6、显示模板实参用法四：用于函数模板的非类型形参。比如 template<class T,int a> void h(T b){}，而调用 h(3)将出错，因为这个调用无法为非类型形参推演出正确的参数。这时正确调用这个函数模板的方法为 h<int, 3>(4)，首先把函数模板的类型形参 T 推演为 int 型，然后把函数模板的非类型形参 int a 用数值3 来推演，把变量 a 设置为 3，然后再把 4 传递给函数的形参 b，把 b 设置为 4。注意，因为 int a 是非类型形参，所以调用非类型形参的实参应是编译时常量表达式，不然就会出错。

6、在使用显示模板实参时，我们只能省略掉尾部的实参。比如 template<class T1, class T2, class T3> T1 h(T2 a, T3 b){}在显示实例化时 h<int>(3, 3.4)省略了最后两个模板实参 T2 和 T3，T2 和 T3 由调用时的实参 3 和 3.4 隐式确定为 int 型和 double 型，而 T1 被显示确定为 int 型。h<int, , double><2,3.4>是错误的，只能省略尾部的实参。

7、显示模板实参最好用在存在二义性或模板实参推演不能进行的情况下。

```
//函数显示模板实参示例适用于函数模板
template<class T>void g1(T a, T b){cout<<"hansu g1()"<<typeid(T).name()<<endl;}
template<classT1,classT2,classT3>T1 g2(T2a,T3b)
```

```
{T1 c=a;cout<<"hansug2()"<<typeid(T1).name()<<typeid(T2).name()<<typeid(T3).name()<<endl;return c;}
template<class T1,class T2> void g3(T1 a){cout<<"hansu g3()"<<typeid(T1).name()<<typeid(T2).name()<<endl;}
template<class T1,int a> void g4(T1 b,double c){cout<<"hansu g4()"<<typeid(T1).name()<<typeid(a).name()<<endl;}
template<class T1,class T2> class A{public:void g();};
//模板显示实例化示例。
//因为模板的声明或定义不能在局部范围或函数内进行。所以模板实例化都应在全局范围内进行。
 template void g1<double>(double a,double b); //把函数模板显示实例化为int型。
 template class A<double,double>; //显示实例化类模板，注意后面没有对象名，也没有{}大括号。
 //template class A<int,int>{}; //错误，显示实例化类模板后面不能有大括号{}。
 //template class A<int,int> m; //错误，显示实例化类模板后面不能有对象名。
//main函数开始
int main()
{//显示模板实参示例。显示模板实参适合于函数模板
//1、显示模板实参用于同一个模板形参的类型不一致的情况。函数g1形式为template<class T>void g1(T a, T b)
g1<double>(2,3.2); //输出"hansu g1() int"两个实参类型不一致，第一个为int第二个为double。但这里用显示模板实参把类型形参T指定为double，所以第一个int型的实参数值2被转换为double类型。
//g1(2,3.2); //错误，这里没有用显式模板实参。所以两个实参类型不一致。
//2、用于函数模板的反回类型中。函数g2形式为template<class T1,class T2,class T3> T1 g2(T2 a,T3 b)
//g2(2,3); //错误，无法推演类型形参T1。
//int g2(2,3); //错误，不能以这种方法试图推导类型形参T1为int型。
//int a=g2(2,3); //错误，以这种方式试图推演出T1的类型为int也是错误的。
g2<int,int,int>(2,3); //正确，将T1，T2，T3显示指定为int型。输出"hansu g2() intintint"
//3、应用于模板函数的参数中没有出现模板形参的情况其中包括省略的用法，函数g3的形式为template<class T1,class T2> void g3(T1 a)
//g3(2); //错误，无法为函数模板的类型形参T2推演出正确的类型
//g3(2,3); //错误，岂图以这种方式为T2指定int型是错误的，因为函数只有一个参数。
//g3<,int>(2);//错误，这里起图用数值2来推演出T1为int型,而省略掉第一个的显示模板实参，这种方法是错误的。在用显示模板实参时，只能省略掉尾部的实参。
//g3<int>(2); //错误，虽然用了显示模板实参方法，省略掉了尾部的实参，但该方法只是把T1指定为int型，仍然无法为T2推演正确的类型。
g3<int,int>(2);//正确，显示指定T1和T2的类型都为int型。
//4、用于函数模板的非类型形参。g4函数的形式为template<class T1,int a> void g4(T1 b,double c)
//g4(3,3.2);//错误，虽然指定了两个参数，但是这里仍然无法为函数模板的非类型形参int a推演出正确的实参。因为第二个函数参数3.2是传递给函数的参数double c的，而不是函数模板的非类型形参int a。
//g4(3,2); //错误，起图以整型值把实参传递给函数模板的非类型形参是不行的，这里数值2会传递给函数形参double c并把int型转换为double型。所以非类型形参int a仍然无实参。
//int d=1; g4<int,d>(3,3.2); //错误，调用方法正确，但对于非类型形参要求实参是一个常量表达式，而局部变量c是非常量表达式，不能做为非类型形参的实参，所以错误。
g4<int,1>(2,3.2); //正确，用显示模板实参，把函数模板的类型形参T1设为int型，把数值1传给非类型形参int a，并把a设为1，把数值2传给函数的第一个形参T1 b并把b设为2，数值3.2传给函数的第二个形参double c并把c设为3.2。
const int d=1; g4<int,d>(3,3.2); }//正确，这里变量d是const常量，能作为非类型形参的实参，这里参数的传递方法同上面的语句。
```

7.3 显示具体化(模板特化，模板说明) 和函数模板的重载

1、 具体化或特化或模板说明指的是一个意思，就是把模板 特殊化，比如有模板 template<class T>void h(T a){}，这个模板适用于所有类型，但是有些特殊类型不需要与这个模板相同的操作或者定义，比如 int 型的 h 实现的功能和这个模板的功能不一样，这样的话我们就要重定义一个 h 模板函数的 int 版本，即特化版本。

特化函数模板：

2、 显示特化格式为：template<> 反回类型 函数名<要特化的类型>(参数列表) {函数体}，显示特化以 template<>开头，表明要 显示特化一个模板，在函数名后<>用尖括号括住要特化的类型版本。比如 template <class T> void h(T a){}，其 int 类型的特化版本为 template<> void h<int>(int a){}，当出现 int 类型的调用时就会调用 这个特化版本 而不会调用通用的模板，比如 h(2)，就会调用 int 类型的特化版本。

3、 如果可以从实参中推演出模板的形参，则可以 省略掉显示模板实参的部分。比如：template<> void h(int a){}。注意函数 h 后面没有<>符号，即显示模板实参部分。

4、 对于反回类型为模板形参时，调用该函数的 特化版本必须要用显示模板实参调用，如果不这样的话就会出现其中一个形参 无法推演的情况。如 template<class T1,class T2,class T3> T1 h(T2 a,T3 b){}，有几种特化情况：

情况一：template<> int h<int,int>(int a, in b){}该情况下把 T1，T2，T3 的类型 推演为 int 型。在主函数中的调用方式应为 h<int>(2,3)。

情况二：template<> int h(int a, int b){}，这里把 T2,T2 推演为 int 型，而 T1 为 int 型，但在调用时必须用显示模板实参调用，且在<>尖括号内必须指定为 int 型，不然就会调用到通用函数模板，如 h<int>(2,3)就会调用函数模板的 特化版本，而 h(2,3)调用会出错。h<double>(2,3)调用则会调用 到通用的函数模板版本。

这几种情况的特化版本是错误的，如 template<> T1 h(int a,int b){}，这种情况下 T1 会成为不能 识别的名字，因而出现错误，template<> int h<double>(int a,int b){}在这种情况下反回类型为 int 型，把 T1 确定为 int 而尖括号内又把 T1 确定为 double 型，这样就出现了冲突。

5、具有相同名字和相同数量反回类型的非模板函数(即普通函数)，也是函数模板 特化的一种情况 这种情况将在后面参

数匹配问题时讲解。

函数模板重载

1、 **函数模板可以重载，注意类模板不存在重载问题，**也就是说出现这两条语句时 template<class T>class A{};template<class T1,class T2>class A{};将出错。

2、 **模板函数重载的形式为：**template<class T> void h(T a, int b){}。Template<class T>void h(T a, double b){}等。

3、 **重载模板函数要注意二义性问题**比如 template<class T> void h(T a, int b){}和 template<class T>void h(T a, T b){}这两个版本就存在二义性问题，当出现语句h(2,3)时就不知道调用哪个才正确，在程序中应避免这种情况出现。

4、 重载函数模板的第二个二义性问题是 template<class T>void h(T a, T b){} 与 template<class T1, class T2>void h(T1 a,T2 b){}，当出现h(2,4)这样的调用时就会出现二义性。解决这个问题的方法是使用显示模板实参，比如要调用第一个 h 函数，可以使用语法 h<int>(2,3)，调用第二个 h 函数的方法为 h<int, int>(2,3)。

5、 函数模板的特化也可以理解为函数模板重载的一种形式。只是特化以 template<>开始。

6、 重载的特殊情况：比如 template<class T1,class T2> void h(T1 a, T2 b){}，还有个版本如 template<class T1>void h(T1 a, int b){}这里两个函数具有两同的名字和相同的形参数量，但形参的类型不同，可以认为第二个版本是第一个版本的重载版本。

7、 函数模板的重载和特化很容易混晓，因为特化很像是一个函数的重载版本，只是开头以 template<>开始而已。

特化类模板：

6、 **特化整个类模板：**比如有 template<class T1,class T2> class A{};其特化形式为 template<> class A<int, int>{};特化形式以 template<>开始，这和模板函数的形式相同，在类名 A 后跟上要特化的类型。

7、 **在类特化的外部定义成员的方法：**比如 template<class T> class A{public: void h();};类 A 特化为 template<> class A<int>{public: void h();};在类外定义特化的类的成员函数h 的方法为：void A<int>::h(){}。在外部定义类特化的成员时应省略掉 template<>。

8、 **类的特化版本应与类模板版本有相同的成员定义，**如果不相同的话那么当类特化的对象访问到类模板的成员时就会出错。因为当调用类的特化版本创建实例时创建的是特化版本的实例，不会创建类模板的实例，特化版本如果和类的模板版本的成员不一样就有可能出现这种错误。比如：模板类 A 中有成员函数 h()和 f()，而特化的类 A 中没有定义成员函数 f()，这时如果有一个特化的类的对象访问到模板类中的函数 f()时就会出错，因为在特化类的实例中找不到这个成员。

9、**类模板的部分特化：**比如有类模板 template<class T1, class T2> class A{};则部分特化的格式为 template<class T1> class A<T1, int>{};将模板形参 T2 特化为 int 型，T1 保持不变。部分特化以 template 开始，在<>中的模板形参是不用特化的模板形参，在类名 A 后面跟上要特化的类型。如果要特化第一个模板形参 T1，则格式为 template<class T2> class A<int, T2>{};部分特化的另一用法是 template<class T1> class A<T1,T1>{};将模板形参 T2 也特化为模板形参 T1 的类型。

10、**在类部分特化的外面定义类成员的方法：**比如有部分特化类template<class T1> class A<T1,int>{public: void h();};则在类外定义的形式为 template<class T1> void A<T1,int>::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样，这里就与部分特化的类 A 的一样 template<class T1>。

其他说明：

11、**可以对模板的特化版本只进行声明而不定义。**比如 template<> void h<int>(int a);注意，声明时后面有个分号。

12、**在调用模板实例之前必须要先对特化的模板进行声明或定义。**一个程序不允许同一模板实参集的同一模板既有显示特化又有实例化。比如有模板 template<class T> void h(T a){}在 h(2)之前没有声明该模板的 int 型特化版本，而是在调用该模板后定义该模板的 int 型特化版本，这时程序不会调用该模板的特化版本，而是调用该模板产生一个新的实例。这里就有一个问题，到底是调用由 h(2)产生的实例版本呢还是调用程序中的特化版本。

13、注意：因为模板的声明或定义不能在局部范围或函数内进行。所以特化类模板或函数模板都应在全局范围内进行。

14、**在特化版本中模板的类型形参是不可见的。**比如 template<> void h<int,int>(int a,int b){T1 a;}就会出现错误，在这里模板的类型形参 T1 在函数模板的特化版本中是不可见的，所以在这里 T1 是未知的标识符，是错误的。

//函数模板特化和类模板特化示例

//定义函数g1，g2和类A

```
template<class T1,class T2> void g1(T1 a,T2 b){cout<<"g1洞"<<endl;}
template<class T1,class T2,class T3> T1 g2(T2 a,T3 b){int c=1;cout<<"g2洞"<<endl;return c;}
template<class T1,class T2,class T3> class A{public:void h();};
```

//函数模板的特化定义。函数模板的特化可以理解为函数模板重载的另一种形式。

```
//下式为g1的类型形参显示指定其类型，把T1，T2在模板实参的尖括号中设为int型。
template<> void g1<int,int>(int a,int b){cout<<"g1一"<<endl;}
//下式显示设定g1的类型形参T1，并设为int型，T2由函数参数double推演为double型。
template<> void g1<int>(int a,double b){cout<<"g1二"<<endl;}
template<> void g1(double a,double b){cout<<"g1三"<<endl;}  //g1的类型形参都由g1的形参推演出来。
//template<> void g1<int>(double a,int b){cout<<"g3一"<<endl;}  //错误，在显示模板实参的尖括号中显示把类型形参T1的类型设为int型，而又在函数的形参中把类型形参T1的类型推演为double型，这样就发生了冲突，出现错误。
template<> int g2<int>(int a,int b){int c=1;cout<<"g2一"<<endl;return c;}
template<> double g2(int a,int b){int c=1;cout<<"g2二"<<endl;return c;}
//注意，下式正确，该式并不是对函数模板g2的部分特化，而是g2的重载。
//template<class T2> int g2(int a, T2 b){int c=1;cout<<"g2三"<<endl;return c;}
```

```
//下式错误，函数反回类型和<double>尖括号中的double类型不同，发生冲突。
//template<> int g2<double>(int a,int b){int c=1;cout<<"two"<<endl;return c;}
//下式错误，函数模板的类型形参在特化版本中是不可见的，也就是说这里的会把类型形参T1理解为未声明的标识符
//template<> T1 g2<int>(int a,int b){int c=1;cout<<"two"<<endl;return c;}
```

//类模板的特化和部分特化

```
template<> class A<int,int,int>{public:void h();}; //特化整个类模板的格式，注意类名后的尖括号中必须指定所有的类模板的类型形参。
//template<> class A<int>{}; //错误，在特化的类名后的尖括号中指定的类模板类型形参的数量不够。要想只特化其中一个类模板的类型形参，就要使用类模板的部分特化。
template<class T1,class T3>class A<T1,double,T3>{public:void h();}; //特化T2，而T1和T3不特化，注意尖括号中的类型形参是不特化的形参。
```

//在类模板的特化或部分特化版本的外部定义成员函数的方法。

```
void A<int,int,int>::h(){cout<<"class A tehua"<<endl; /*  T1 c; 错误，在特化版本中模板的类型形参是不可见的，也就是说在这里T1是未声明的标识符。*/}
//template<> void A<int,int,int>::h(){} //错误，在类模板的特化版本外面定义类模板的成员时应省略掉emplate<>
template<class T1,class T3> void A<T1,double,T3>::h(){cout<<"class A bute"<<endl;}
template<class T1,class T2,class T3> void A<T1,T2,T3>::h(){cout<<"class A putong"<<endl;} //定义普通类模板中的成员函数。
```

//main函数开始

```
int main()
{//特化的函数模板的调用方式。
 g1(2,2); //输出"g1一"，调用函数模板g1的第一个特化版本template<> void g1<int,int>(int a,int b){cout<<"g1一"<<endl;}
 g1(2,3.2); //输出"g1二"，调用函数模板g1的第二个特化版本template<> void g1<int>(int a,double b){cout<<"g1二"<<endl;}
 g1(3.3,4.4); //输出"g1三"，调用函数模板g1的第三个特化版本template<> void g1(double a,double b){cout<<"g1三"<<endl;}
 g1<double>(4,3.5); //输出"g1三"，这里用显示模板实参把第一个实参指定为double型，这样g1的两个实参都是double型，所以将调用g1的第三个特化版本。
// g2(3,3); //错误，在调用反回类型为类型形参的时候必须用显示模板实参的形式为反回类型的形参显示指定类型。在这里就会出现无法为T1确定类型的情况。
 g2<int>(2,3);  //正确，把g2的类型形参T1设显示指定为int，调用g2的第一个特化版本。template<> int g2<int>(int a,int b){int c=1;cout<<"g2一"<<endl;return c;}
 g2<double>(2,3);//正确，把g2的类型形参T1设显示指定为double，调用g2的第二个特化版本。template<> double g2(int a,int b){int c=1;cout<<"g2二"<<endl;return c;}
 g2<char>(2,3);  //正确，把g2的类型形参T1设显示指定为char，对于char版本的g2函数没有特化版本，因此调用g2的通用版本。
template<class T1,class T2,class T3> T1 g2(T2 a,T3 b){int c=1;cout<<"g2洞"<<endl;return c;}
```

// 类模板特化和部分特化的调用。

```
 A<int,int,int> m1; m1.h();//正确，调用类模板的特化版本。
 A<int,double,int> m; m.h(); //正确，调用类模板的部分特化版本。
 //A<int,int> m2; //错误，类模板有三个类型形参，这里只提供了两个，数量不够，错误。
 A<double,double,int> m3; m3.h(); //调用类A的部分特化版本。
 A<double,int,int> m4; m4.h();}//调用类A的普通版本，在这里没有A<double,int,int>型的特化或者部分特化版本可用。
```

八、匹配问题即函数的重载解析问题

1、当模板函数即有 重载版本又有非模板函数的重载版本时就会出现参数的匹配问题 模板函数 最后会选择最佳匹配的函数调用。过程为先进行参数匹配，参后 从匹配好的函数列表中选择完全匹配的函数，最后再在完全匹配的函数中选择最佳匹配的函数。

2、参数的 匹配过程如下。

a、创建候选函数列表。其中包含与被调用函数的名称相同的函数和模板函数(其中模板函数推演成功才会加入到候选函数列表中)。

b、使用 候选函数列表创建可行函数列表。这些都是参数数目正确的函数，为此有一个隐式转换序列，其中包括实参类型与相应的形参类型完全 匹配的情况

c、确定是 否有最佳的可行函数。如果有则使用之，否则出错。

3、确定最佳可行函数的步骤，首先应确定哪些是完全匹配的，完全匹配从最佳到最差的顺序为

a、完全匹配，但普通函数优于模板函数及模板函数的特化版本

b、提升转换(如：char 和 short 转换为 int，及 float 转换为 double)

c、标准转换(如：int 转换为 char，及 long 转换为 double)

d、用户定义的 转换，如类声明中定义的 转换

注意，完全匹配允许一些无关紧要的转换，这些无关紧要的转换如下。其中包括 const type 到 const type&的转换

形参	实参	**形参**	实参	**形参**	实参	**形参**	实参
type &	type	**type**	type &	**type ***	type[]	**type(参数列表)**	type(*)(参数列表)
const type	type	**volatile type**	type	**const type ***	type *	**volatile type***	type *

比如：有函数 void h(int a), void h(int &a)，其中 int b=3;则调用 h(b)将出错。因为两个函数 都是完全匹配且是最佳匹配的

完全匹配示例：比如有如 下函数定义：

#1 void may(int a); #2 float may(float a, float b=3); #3 void may(char a); #4 char *may(const char* a);

#5 char may(const char & a);　　#6 template<class T>void may(const T &a);　　#7 template<class T>void may(T * a);
则如调用 may('b'); 对这个调用#4 和#7 不可行，因为整数不能被隐式转换为指针类型。
#1 调用优先于#2，因为 char 到 int 是提升转换，而 char 到 float 是标准转换，提升转换优先于标准转换，所以#1 优先于#2。
#3,#5,#6 都优先于#1 和#2，因为#3,#5,#6 都是完全匹配，其中#3 和#5 优先于#6，因为#6 是模板函数。其中#3 和#5 都是完全匹配，且是最佳的，这样就出现了二义性问题的错误。
注意#5 是 const char &类型的形参，在这里应用了完全匹配的无关紧要的转换，type &形参到 type 实参的转换和 const type 形参到 type 实参的转换。

4、但是完全匹配并不是最佳匹配，最佳匹配的原则如下

a、如果多个函数都是完全匹配，则非 const 数据的指针或引用形参优先于 const 的指针和引用的参数匹配。这条规则只适合于指针或引用参数，当参数是非指 针或引用时将出现二义性。比如 void h(int& a)和 void h(const int & a)其中 int b=2;当有调用 h(b)时，h(int & a)优先于 h(const int& a)因为在这里变量b 没有被声明为 const，而第一个函数 又是非 const 指针函数，所以第一个函数优先于第二个函数。但对于 h(2)这样的调用，则第二个函数将 优先于第一个函数的调用，因为这里数值2 本身就是 const 常量，所以第二个函数更优先。而 h(int a)和 h(const int a)当出现 h(b)时就会出现二义性问题。

b、注意这四个函数的调用 1#　h(int)，2#　h(const int)，3#　h(int &)，4#　h(const int&)，
情况 1：如果只有#1 和#2 则调用 h(2)和 h(b)的调用 都会出错。
情况 2：如果只有 1#和 3#则 h(2)将调用 1#，而 h(b)调用会出错。
情况 3：如果只有 1#和 4#则 h(3)和 h(b)都将出错。
情况 4：如果只有 2#和 3#则 h(2)将调用 2#，而 h(b)将出错。
情况 5：如果只有 2#和 4#，则 h(3)和 h(b)都将出错。
对于以上五种情况 h(b)的调用 都是错误的，而对 h(3)的调用中只要 引用参数不带const 则不会出错。具体原因还不清楚，有待考证。可能与 按值传递有关。
情况 6：如果只有 3#和 4#，则 h(3)将调用 4#，因为数 值3 是 const 常量；h(b)将调用 3#，因为就量 b 本身不是 const 常量。

c、普通函数优先于模板函数及特化的模板函数。

d、当完全匹配的函数都是模板函数时，则更具体模板函数优先。更具体指的是编译器推断使用哪种类型时执行的转换更少。比如：template<class T>void h(T* a){}与 template<class T>void h(T a){} ,其中 int b=3;调用 h(&b)将调用第 1 个模板函数，因为第一个模板函数被转换为 h<int>(int *)传送给函数，而第二个模板函数 被转换为h<int *>(int *)传送给函数，第二个 转换把模板形参T 具体化为 int 型的指针，而第一个模板函数的形参 T 被转换为 int 型，因此第一个转换更具体，需要的转换更少。注意，如果把 T* a 换成 T& a 则会出 现类似于b 的情况。

//函数模板的参数的匹配问题示例

```
template<class T1,class T2> void g1(T1 a,T2 b){cout<<"g1一"<<endl;}
template<> void g1(int a,int b){cout<<"g1二"<<endl;}
void g1(int a,int b){cout<<"g1三"<<endl;}
```

//g2函数的7个重载版本

```
void g2(int a){cout<<"g2#1"<<endl;}  //#1
void g2(float a){cout<<"g2#2"<<endl;} //#2
void g2(char &a){cout<<"g2#3"<<endl;} //#3
void g2(const char *a){cout<<"g2#4"<<endl;} //#4
void g2(const char &a){cout<<"g2#5"<<endl;} //#5
template<class T> void g2(const T& a){cout<<"g2#6"<<endl;} //#6
template<class T> void g2(const T* a){cout<<"g2#7"<<endl;} //#7

void g3(int a){cout<<"g3#1"<<endl;}
void g3(float a){cout<<"g3#2"<<endl;}
```

//完全匹配的模板

```
template<class T>void g4(T* a){cout<<"g4一"<<typeid(T).name()<<endl;}
template<class T> void g4(T a){cout<<"g4二"<<typeid(T).name()<<endl;}

int main()
{//综全应用函数g2。
char b='a';  g2(b);/*调用g2#3，这里#4和#7匹配，因为变量b不是地址。其中#1优先于#2，因为#1是提升转换，#2是标准转换，提升转换优先于标准转换。而#3,#5,#6都优先于#1和#2，其中#3和#5优先于#6，因为#6是模板，在这个调用中#3优先于#5，因为变量b不是const类型的，所以#3更优先。*/
g2('b'); //调用g2#5，这里的匹配方式为上面的相同，只是最后#3和#5的区别，在这里调用#5的原因是字符'b'是常量，而#5是常量引用，所以调用#5。
//下面分别应用完全匹配与最佳匹配。
g3('b'); //调用g3(int)，因为g3(int)是提升转换即char到int的转换，而g3(float)是标准转换即char到float的转换，提升转换优先于标准转换所以调用g3(int)。
g1(2,2); //调用g1的普通函数，在完全匹配的情况下普通函数优先于模板函数及模板的特化函数。
g1(2,3.2); //调用g1的通用模板函数，因为对于实数.2来说没有完全匹配的特化版本和普通函数。
```

//完全匹配的模板函数的调用

```
int c=3; g4(&c); }//输出"g4一", 因为g4的第一个模板函数更具体。
```

九、类模板中的模板成员(模板函数,模板类)和静态成员

1、 类模板中的模板函数和模板类的声明：与普通模板的声明方式相同，即都是以 template 开始

2、 在类模板外定义类模板中的模板成员的方法：比如 template<class T1> class A{public:template<class T2> class B; template<class T3> void g(T3 a);};则在类模板外定义模板成员的方法为,template<class T1> template<class T2> class A<T1>::B{};定义模板函数的方法为：template<class T1> template<class T3> void A<T1>::g(T3 a){}其中第一个 template 指明外围类的模板形参，第二个 template 指定模板成员的模板形参，而作用域解析运算符指明是来自哪个类的成员。

3、 实例化类模板的模板成员函数：比如上例中要实例化函数 g()则方法为, A<int> m; m.g(2);这里外围类 A 的模板形参由尖括号中指出，而类中的模板函数的参数由 整型值 2 推演出为 int 型。

4、 创建类模板中的模板成员类的对象的方法：比如上例中要创建模板成员类 B 的方法为，A<int>::B<int> m1；A<int>::B<doble>m2; A<double>::B<int> m3;在类模板成员 B 的前面要使用作用域解析运算符以指定来自哪个外围类，并且在尖括号中要指定创建哪个外围类的实例的对象。这里说明在类模板中定义模板类成员时就意味意该外围模板类的一个实 例比如int 实例将包含有多个模板成员类的实例。比如这里类 A 的 int 实例就有两个模板成员类 B 的 int 和 double 两个实 例版本。

5、 要访问类模板中的模板成员类的成员遵守嵌套类的规则，因为类模板中的模板成员类就是一个 嵌套类 即外围类和嵌套类中的成员是相 互独立的, 要访问其中的成员只能通过嵌套类的指针，引用或对象的方式来访问。具体情况见嵌套类部分。

6、 类模板中的 静态成员是类模板的所有实例所共享的。

//类模板中的模板成员示例

```
template<class T1,class T2> class A{public:int a,b; static int e;
template<class T3,class T4> class B;
template<class T5,class T6> void g(T5 a,T6 b);
class C{public:void gc(){cout<<"class C gc()"<<endl;}};
void g1(){cout<<"putong g1()"<<endl;}  };

template<class T1,class T2>template<class T3,class T4> class A<T1,T2>::B{public:void gb(){cout<<"moban class B gb()"<<endl;}};
//在类模板外面定义类模板的模板成员类的方法
template<class T1,class T2>template<class T5,class T6> void A<T1,T2>::g(T5 a,T6 b){cout<<"moban g()"<<endl;}//在类模板外面定义类模板的模板成员函数的方法
template<class T1,class T2> int A<T1,T2>::e=0; //在类模板外面定义静态成员的方法。

int main()
{A<int,int> ma;
 ma.g(2,3);  //创建模板类中模板成员函数的方法，在这里模板类A的模板形参被设为int，而模板成员函数的模板形参则由两个int型的整数推演为int型。
 ma.e=1; A<int,int>::e=2;  //把类模板A的int,int型实例的静态成员设为。
 cout<<"ma.e="<<ma.e<<endl;
 A<int,double> ma1;
 cout<<"ma.e="<<ma1.e<<A<int,int>::e<<A<int,double>::e<<endl; //因为类模板A的int,int型实例和int,double实例是两个实例，所以这里的静态常量e的值不是三个二。
 A<int,int>::B<int,int> mb;  //声明模板类中模板成员类的方法。
 mb.gb();//调用嵌套类B的成员函数
 //mb.g(); //错误，函数g()是外围类的成员，嵌套类不能访问外围类的成员}
```

十、友元和类模板

1、类模板中有 普通友元函数, 友元类，模板友元函数和友元类，普通友元函数和友元类不做介绍。

2、可以建立两种类模板的友元模板，即约束型的友元模板和非约束型的友元模板。

3、非约束型友元模板：即类模板的 友元模板类或者友元模板函数的任一实例都是外围类的任一实例的友元, 也就是外围类和友元模板类或友元模板函数之间是多对多的关系

4、约束型友元模板：即类模板的 友元模板类或友元模板函数的一个特定实例只是外围类的相关的一个实例的友元。即外围类和 友元模板类或友元模板函数之间是一对一的关系。

3、约束型友元模板函数或友元类的建立：比如有 前向声明：template<class T1> void g(T1 a); template<class T2> void g1(); template<class T3>class B;则 template<class T>class A{friend void g<>(T a); friend void g1<T>(); friend class B<T>;};就建立了三个 约束型友元模板，其中 g 和 g1 是函数，而 B 是类。注意其中的语法。这里 g<int>型和类 A<int>型是一对一的友元关系，g<double>和 A<double>是一个一 对一的友元关系。

6、非约束型友元模板函数或友元类的建立：非约束型友元模板和外围类具有不同的模板形参，比如 template<class T>class A{template<class T1> friend void g(T1 a); template<class T2> friend class B;}注意其中的语法，非约束型友元模板都要以 template 开头。要注意 友元模板类, 在类名 B 的后面 没有尖括号。

7、不存在部分约束型的友元模板或者友元类：比如 template<class T> class A{template<class T1>friend void g(T1 a, T b); template<class T3>friend class B<T3,T>;}其中函数 g 具有 template<class T1,class T2>void g(T1 a,T2 b)的形式。其中的函数 g 试图把第二个模板形参部分约束为类 A 的模板形参类型，但是这是无笑的，这种语法的结果是 g 函数的非 约束型类友元函数，而对类 B 的友元声明则是一种语法错误。

//友元模板示例。

```
template<class T1>class B;                    template<class T2> void g(T2 b);
template<class T3> class C;                   template<class T4> void g1(T4 c);
template<class T7,class T8>class D;           template<class T5,class T6> void g2(T5 d,T6 e);

template<class T> class A{int a;
```

//类模板的 约束友元模板函数和友元模板类的声明形式。

```
public:  friend class B<T>;  //注意语法，这里没有以template<>开始。
     //template<> friend class B<T>; //错误，要把类友元类B约束为类A的T类型请使用以上的语法。
friend void g<>(T b); //注意语法，这里没有以template<>开始，这里把友元模板函数的类型形参T2约束为类A的类型T
//friend void g<T>(T b);//笑果同上，如果函数没有形参的情况使用该语法。
//template<> friend void g<T>(T b);//错误，要把函数g与类A的类型形参T相关联请使用上面的语法。
```

//类模板的非 约束友元模板函数和友元类的声明形式

```
template<class T3>friend class C;   //friend template<class T3>class C;//语法错误，friend的位置不对。
template<class T4>friend void g1(T4 c);
//template<class T4>friend void g1(T c); //正确，注意这里不是把g1的T4模板形参约束为T类型，这里的笑果同以上的g1函数，非约束友元类的声明中使用类的形参和函数的形参没有区别。这种情况只适合于函数模板。
//template<class T3>friend class C<T>;//错误，这是语法错误，这里试图把T3的类型约束为类A的T类型，这种方法是错误的，具体情况请参看类模板的部分特化部分
//template<class T3>friend class C<T3>; //语法错误，不存在这样的类模板语法，该语法即像部分特化又不是部分特化。
```

//类模板的友元模板和友元函数不存在部分约束的情况

```
template<class T5> friend void g2(T5 d,T e); //在这里试图对g2函数的T6形参进行特化为T类型，但这里不会成功
//template<class T7> friend class D<T7,T>; //错误试图把友元类D的T8形参部分特化为T将是不成功的。};
```

//定义各种 友元模板和函数

```
template<class T2> void g(T2 b){A<int> d;A<double> mg;mg.a=1;cout<<mg.a<<endl;}//T b;//错误，友元虽然可以访问类对象的私有成员但是对于类型形参T来说友元是不可见的，在友元中不能使用类的类型形参T来声明变量
template<class T1> class B{public:void gb(){A<int> mb; mb.a=2;cout<<mb.a<<endl;}};
template<class T5,class T6>void g2(T5 d,T6 e){A<int> mg2;A<double>mg22;mg22.a=33;cout<<mg22.a<<endl; mg2.a=3;
cout<<mg2.a<<endl;}
template<class T4>void g1(T4 c){A<int> mg1;A<double> mg2; mg1.a=4;cout<<mg1.a<<endl;}
template<class T7,class T8>class D{public:void gd(){A<int> md; md.a=5;cout<<md.a<<endl;}};
template<class T3>class C {public:void gc(){A<int> mb; mb.a=8;cout<<mb.a<<endl;}};

int main()
{g(4.8);g(3.3);//正确，
 //g(3);//错误，函数g是约束型的友元函数，也就是说这里的调用只能访问类A的int型的对象的私有成员即A<int> d对象的私有成员，但函数g中有A<double> mg;对象的私有成员的访问，所以该调用将是错误的。同样如果该函数中有A<int> d;对象对类A的私有成员的访问时上面的g函数调用也是错误的。
 B<int> mbb;   mbb.gb();
 B<double> mbb1;   // mbb1.gb();//该函数调用错语，错误原因同上面的g(3)调用。
g1(2); g1(3.3); //这里的两种调用都是正确的，因为g1不是约束型的友元函数。
g2(3,3);   g2(3,3.3);//从该调用可以看出，试图对函数g2进行部分约束将是不可行的。
C<int> mc;  mc.gc();    C<double> mc1;  mc1.gc();  //两种调用方式都正确，原因同g1函数的调用。
D<int,int> md; //md.gd(); //错误，类D不是类A的友元，不能访问类A的私有成员。}
```

十一、模板与继承

1、当从模板类派生出子类时必须注意的是：子类并不会从通用的模板基类继承而来，只能从基类的某一实例继承而来。这样的话就有以下几种继承方式：

a、基类是模板类的一个特定实例版本比如 template<class T1>class B:public A<int>{};

b、基类是一个和子类相关的一个实例，比如 template<class T1>class B:public A<T1>{};这时当实例化一个子类 B 的时候基类就相应的被实例化为一个和基类相同的实例版本，比如 B<int> m；模板类 B 被实例化为 int 版本，这时基类 A 也相应的 被实例化为int 版本。

c、如果基类是一个特定的实例的版本，这时子类可以不是一个模板，比如 class B:public A<int>{}。

//模板与继承的示例

```
template<class T> class A{  public: void ga(){cout<<"class A "<<typeid(T).name()<<endl;}};
```

//以下是几种继承方式

```
template<class T1> class B:public A<int>{public: void gb(){cout<<"class B "<<typeid(T1).name()<<endl;}};
template<class T2> class C:public A<T2>{public: void gb(){cout<<"class C "<<typeid(T2).name()<<endl;}};
//template<class T3> class D:public A<T>{}; //错误，模板形参T在这里是不可见的说明符。错误原因在于，继承并不会从通用的基类中继承而来，只能是从某一特定的基类中继承而来。
class E:public A<double>{}; //从特定实例继承时子类可以是一般的类
int main()
{B<double> m;
m.ga();  //输出class A int;
m.gb();  //输出class B double;
C<double> mc;
```

```
mc.ga(); //输出class A double
mc.gb();} //输出 class A double
```

十二、模板中的关见字typename 和 class 的区别

1、首先 typename 是一个较新的关见字，而 class 是比较老的关见字，在类模板的声明中使用 哪个都是一样的

2、必须使用 typename 关见字的情况：当有一个模板，且模板的类型形参和某一个类名同名，而且这个同名的类中又有一个嵌套类的时候就要使用关见字typename 了，比如有类 class T{public: class A{};}; 则 template<class T> class B{T::A m;};这时的语句就会出错，因为模板类 B 无法区别出 T::A 表示的是一个类型 还是一个数据成员 在 C++里面默认情况下是一个数据成员，在这里的语句T::A m 就会出错，会出现 m 前缺少类型说明符的错误。要使该式 正确就必须在 T::A 前加上 typename 关见字以说明 T::A 表示的是一个类型。即 template<class T> class B{typename T::A m;}这样就正确地声明了一个类 T 的嵌套类 A 的对象 m。当然如果模板的类型形参和那个类名不同名时不会出现这种错误，比如把嵌套类 B 的外围类的类名 改为D，则在模板类 B 中的语句 D::B m;将是正确的语句。

//关见字 typename 和 class 的区别的示例

```
class T{public:static int e;class B{public:void gb(){cout<<"class B"<<endl;}};}; //定义一个带有嵌套的类T。
int T::e=3;

template<class T1>class A{public:T::B a;}; //模板形参与带有嵌套的类T不同名，所以这里的语句T::B a是正确的。当然在前面加上typename关见字也不会出错。
template<class T>class D{public: typename T::B b;}; //模板形参与带有嵌套类的类名同名，在语句T::B b;前必须加上typename以指定这里的嵌套类B是一个类型而不是一个表达式。
template<class T>class F{public: void g(){int a=T::e+2;cout<<a<<endl;}}; //这个范例可以看出如果不在T::e前加typename则系统默认表示的是e是一个数据成员，而不是一个类型。

int main()
{A<int> ma;    ma.a.gb(); //输出class B
D<int> md;     md.b.gb(); //输出class B
F<int> mf;     mf.g();  }  //输出数字 5，在这里可以清楚的看到T::e 默认表示的是一个数据成员而不是类型。
```

作者：黄邦勇帅