在C++发明阶段，C++之父Stroustrup和贝尔实验室的C++小组对原先的宏方法进行了修订，对其进行了简化并将它从预处理范围移入了编译器。这种新的代码替换装置被称为模板，而且它变现了完全不同的代码重用方法：模板对源代码重用，而不是通过继承和组合重用对象代码。当用户使用模板时，参数由编译器来替换，这非常像原来的宏方法，却更清晰，更容易使用。

模板使类和函数可在编译时定义所需处理和返回的数据类型，一个模板并非一个实实在在的类或函数，仅仅是一个类和函数的描述。由于模板可以实现逻辑相同、数据类型不同的程序代码复制，所以使用模板机制可以减轻编程和维护的工作量和难度。模板一般分为模板函数和类模板。以所处理的数据类型的说明作为参数的类就叫类模板，或者模板类，而以所处理的数据类型的说明作为参数的函数，则称为函数模板。

本文包含函数目标和类模板，有些可能会交错设计两个模块的细节。

****1.函数模板****

函数模板定义了参数化的非成员函数，这使得程序员能够用不同类型的参数调用相同的函数，由编译器决定调用哪一种类型，并且从模板中生成相应的代码。

定义：

Template﹤类型参数表﹥返回类型 函数名 （形参表）{函数体}

简单实例，调用函数打印字符串或数字等。

普通函数形式：

1. #include <string>
2. #include <iostream>
3. **void** printstring(**const** std::string& str)
4. {
5. std::cout << str << std::endl;
6. }
7. **int** main()
8. {
9. std::string str("Hello World");
10. printstring(str);
11. **return** 0;
12. }//输出：Hello World

模板函数形式：

1. #include <string>
2. #include <iostream>
3. **using** **namespace** std;
4. **template**<**typename** T> **void** print(**const** T& var)
5. {
6. cout << var << endl;
7. }
8. **int** main()
9. {
10. string str("Hello World");
11. **const** **int** num=1234;
12. print(str);
13. print(num);
14. **return** 0;
15. }
16. //输出：Hello World
17. //       1234

可以看出使用模板后的函数不仅可以输出字符串形式还可以输出数字形式的内容。

上面两个例子介绍了函数模板的简单使用方法，但只有一个参数，如果需要多个参数，相应的函数模板应采用以下形式定义：

Template﹤类型1 变量1，类型2 变量2 ，…﹥返回类型 函数名 （形参表）{函数体}

现在，为了看到模板时如何称为函数的，我们假定min（）函数接受各种类型的参数，并找出其中的最小者，如果不采用模板技术，则只能接受一个特定类型的参数，如果希望也能接受其他类型的参数，就需要对每一种类型的参数都定义一个同功能的函数，其实为函数的重载，这里不在讨论，但这将是一件非常让人麻烦的事情。如：

普通定义：

1. #include <iostream.h>
2. // 定义多态函数，找出三个整数中最小的数
3. **int** min0(**int** ii, **int** jj, **int** kk)
4. {
5. **int** temp;
6. **if**((ii<jj)&&(ii<kk)){temp=ii;}
7. **else** **if**((jj<ii)&&(jj<kk)){temp=jj;    }
8. **else**{    temp=kk;    }
9. **return** temp;
10. }
11. // 定义多态函数，找出三个小数中最小的数
12. **float** min1(**float** ii, **float** jj, **float** kk)
13. {
14. **float** temp;
15. **if**((ii<jj)&&(ii<kk)){temp=ii;}
16. **else** **if**((jj<ii)&&(jj<kk)){temp=jj;    }
17. **else**{    temp=kk;    }
18. **return** temp;
19. }
21. // 定义多态函数，找出三个子符中最小的字符
22. **char** min2(**char** ii, **char** jj, **char** kk)
23. {
24. **char** temp;
25. **if**((ii<jj)&&(ii<kk))    {temp=ii;    }
26. **else** **if**((jj<ii)&&(jj<kk)){temp=jj;}
27. **else**{temp=kk;}
28. **return** temp;
29. }
31. **void** main()
32. {
33. **int** temp1=min0(100,20,30);
34. cout<<temp1<<endl;
35. **float** temp2=min1(10.60,10.64,53.21);
36. cout<<temp2<<endl;
37. **char** temp3=min2('c','a','C');
38. cout<<temp3<<endl;
39. }
40. //以换行形式输出20  10.6  C

使用模板：

1. #include <iostream.h>
2. // 定义函数模板，找出三个值中最小的值，与数据类型无关
3. **template** <**class** T>
4. T min(T ii, T jj, T kk)
5. {
6. T temp;
7. **if**((ii<jj)&&(ii<kk)){temp=ii;}
8. **else** **if**((jj<ii)&&(jj<kk)){temp=jj;}
9. **else**{    temp=kk; }
10. **return** temp;
11. }
12. // 下面是主函数
13. **void** main()
14. {
15. cout<<min(100,20,30)<<endl;
16. cout<<min(10.60,10.64,53.21)<<endl;
17. cout<<min('c','a','C')<<endl;
18. }

输出结果同上，但可以清楚的看到二者之间的工作量大小之差距。

函数模板功能非常强大，但是有时候可能会陷入困境，加入待比较的函数模板没有提供正确的操作符，则程序不会对此进行编译。为了避免这种错误，可以使用函数模板和同名的非模板函数重载，这就是函数定制。函数模板与同名的非模板函数重载必须遵守以下规定：

寻找一个参数完全匹配的函数，如有，则调用它

如果失败，寻找一个函数模板，使其实例化，产生一个匹配的模板函数，若有，则调用它

如果失败，再试低一级的对函数重载的方法，例如通过类型转换可产生的参数匹配等，若找到匹配的函数，调用它

如果失败，则证明这是一个错误的调用

现在用上例的模板函数比较两个字符串，但会出现问题：

1. #include <iostream.h>
2. // 定义函数模板，找出三个值中最小的值，与数据类型无关
3. **template** <**class** T>
4. T min(T ii, T jj, T kk)
5. {
6. T temp;
7. **if**((ii<jj)&&(ii<kk)){     temp=ii; }
8. **else** **if**((jj<ii)&&(jj<kk)){ temp=jj; }
9. **else**{    temp=kk;}
10. **return** temp;
11. }
12. **void** main()
13. {
14. cout<<min("anderson","Washington","Smith")<<endl;
15. }

输出anderson 与实际结果不否，原因在于编译器会生成对字符串指针做比较的函数，但比较字符串和比较字符串指针是不一样的，为了解决此问题，我们可以定制函数模板，如：

1. #include <iostream>
2. #include <string>
3. **using** **namespace** std;
4. // 定义函数模板，找出三个值中最小的值，与数据类型无关
5. **template** <**class** T>
6. T min(T ii, T jj, T kk)
7. {
8. T temp;
9. **if**((ii<jj)&&(ii<kk)){        temp=ii;    }
10. **else** **if**((jj<ii)&&(jj<kk)){        temp=jj;    }
11. **else**    {        temp=kk;    }
12. **return** temp;
13. }
14. //非模板函数重载
15. **const** **char**\* min(**const** **char**\* ch1, **const** **char**\* ch2,**const** **char**\* ch3)
16. {
17. **const** **char**\* temp;
18. **int** result1 = strcmp(ch1,ch2);
19. **int** result2 = strcmp(ch1,ch3);
20. **int** result3 = strcmp(ch2,ch1);
21. **int** result4 = strcmp(ch2,ch3);
22. **if**((result1<0)&&(result2<0))    {        temp = ch1;    }
23. **else** **if**((result3<0)&&(result4<0))    {        temp=ch2;    }
24. **else**    {        temp=ch3;    }
25. **return** temp;
26. }
27. **void** main()
28. {
29. cout<<min(100,20,30)<<endl;
30. cout<<min(10.60,10.64,53.21)<<endl;
31. cout<<min('c','a','C')<<endl;
32. cout<<min("anderson","Washington","Smith")<<endl;
33. }

在VS 2010中，最后一行会输出Smith， 与结果先符。

注意：若上例在VC++ 6.0 中运行，其结果最后一行仍会输出anderson，读者可自己上机查看情况并分析原因。

下面给出一些实例：

1. #ifndef HEADER\_MY
2. #define HEADER\_MY
3. #include <string>
4. #include <sstream>
5. **template**<**class** T>
6. T fromString(**const** std::string &s)
7. {
8. std::istringstream is(s);
9. T t;
10. is>>t;
11. **return** t;
12. }
13. **template**<**class** T>
14. std::string toString(**const** T &s)
15. {
16. std::ostringstream t;
17. t<<s;
18. **return** t.str();
19. }
21. #endif

24. #include "HEADER.h"
25. #include <iostream>
26. #include <complex>
27. **using** **namespace** std;
28. **int** main()
29. {
30. **int** i = 1234;
31. cout << "i == \"" << toString(i) << "\"\n";
32. **float** x = 567.89;
33. cout << "x == \"" << toString(x) << "\"\n";
34. complex<**float**> c(1.0, 2.0);
35. cout << "c == \"" << toString(c) << "\"\n";
36. cout << endl;
37. i = fromString<**int**>(string("1234"));
38. cout << "i == " << i << endl;
39. x = fromString<**float**>(string("567.89"));
40. cout << "x == " << x << endl;
41. c = fromString< complex<**float**> >(string("(1.0,2.0)"));
42. cout << "c == " << c << endl;
43. **return** 0;
44. }

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的这个过程叫做模板实参推演。如template <class T> void h(T a){}; h(1); h(0.2);第一个调用因为实参是int型的，所以模板形参T被推演为int型，第二个T的类型则为double。

在使用函数模板时，请注意以下几点：

在模板被实例化后，就会生成一个新的实例，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;第一个调用H(a)生成一个int型的实例版本，但是当调用h(b)的时候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。

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。

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。

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。

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。

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)。

模板实参推演实例，说明内容较长，采用注释形式，但代码较乱：

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T>**void** h(T a){cout<<" 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。
4. **template**<**class** T>**void** k(T a,T b){T c;cout<<" 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。
5. **template**<**class** T1,**class** T2> **void** f(T1 a, T2 b){cout<<" 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<<" g()"<<typeid(b).name()<<endl;}
6. //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。
7. //main函数开始
8. **int** main()
9. { // 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。
10. //函数模板实参推演示例。
11. // 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)。
12. //h函数形式为：template<class T>void h(T a)
13. h(2);//输出" 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。
14. h(2.0);//输出" 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
15. //k函数形式为：template<class T>void k(T a,T b)
16. k(2,3);//输出" k() int"
17. //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。
18. //f函数的形式为：template<class T1,class T2> void f(T1 a, T2 b)
19. f(3,4.0);//输出" 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。
20. **int** a=3;**double** b=4;
21. f(a,b); //输出同上，这里用变量名实现推板实参的推演。
22. //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)
23. **int** a1[2]={1,2};g(a1); //输出" 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。
24. g(&b); //输出" g() double",这里和上面的一样，只是把类型T转换为double型。
25. h(&b); //输出" h() double \*"这里把模参类型T推演为double \*类型。
26. **return** 0;
27. }

函数模板的显示实例化

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。

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。

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。

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。

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。

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。

7、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。

8、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。

下面来看看实例：

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T>**void** g1(T a, T b){cout<<"hansu g1()"<<**typeid**(T).name()<<endl;}
4. **template**<**class** T1,**class** T2,**class** T3>T1 g2(T2 a,T3 b)
5. {T1 c=a;cout<<"hansug2()"<<**typeid**(T1).name()<<**typeid**(T2).name()<<**typeid**(T3).name()<<endl; **return** c;}
6. **template**<**class** T1,**class** T2> **void** g3 ( T1 a ) {cout<<"hansu g3()"<<**typeid**(T1).name()<<**typeid**(T2).name()<<endl;}
7. **template**<**class** T1,**int** a> **void** g4(T1 b, **double** c){cout<<"hansu g4()"<<**typeid**(T1).name()<<**typeid**(a).name()<<endl;}
8. **template**<**class** T1,**class** T2> **class** A{**public**:**void** g();};
9. //模板显示实例化示例。
10. //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。
11. **template** **void** g1<**double**>(**double** a,**double** b); //把函数模板显示实例化为int型。
12. **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。
13. //template class A<int,int>{};  //错误，显示实例化类模板后面不能有大括号{}。
14. //template class A<int,int> m;  //错误，显示实例化类模板后面不能有对象名。
15. //main函数开始
16. **int** main()
17. {//显示模板实参示例。显示模板实参适合于函数模板
18. //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)
19. 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。
20. //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。
21. //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)
22. //g2(2,3);错误，无法推演类型形参T1。
23. //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。
24. //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。
25. g2<**int**,**int**,**int**>(2,3);//正确，将T1，T2，T3 显示指定为int型。输出"hansu g2() intintint"
26. //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)
27. //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
28. //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。
29. //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。
30. //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。
31. g3<**int**,**int**>(2);//正确，显示指定T1和T2的类型都为int型。
32. //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)
33. //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x.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。
34. //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。
35. //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。
36. 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?.2传给函数的第二个形参double c并把c设为?.2。
37. **const** **int** d=1; g4<**int**,d>(2,3.2);//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。
38. **return** 0;
39. }

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（函数定制）：****

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<classT1,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。

1. #include <iostream>
2. **using** **namespace** std;
3. //函数模板特化和类模板特化示例
4. //定义函数g1，g2和类A
5. **template**<**class** T1,**class** T2> **void** g1(T1 a,T2 b){cout<<"g1"<<endl;}
6. **template**<**class** T1,**class** T2,**class** T3>T1 g2(T2 a,T3 b){  **int** c=1;cout<<"g2"<<endl;**return** c;}
7. **template**<**class** T1,**class** T2,**class** T3>**class** A{**public**:**void** h();}
8. //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。
9. //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。
10. **template**<> **void** g1<**int**,**int**>(**int** a,**int** b){cout<<"g1一"<<endl;}
11. //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。
12. **template**<> **void** g1<**int**>(**int** a,**double** b){cout<<"g1二"<<endl;}
13. **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。
14. //template<> void g1<int>(double a,int b){cout<<"g•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。
15. **template**<> **int** g2<**int**>(**int** a,**int** b){**int** c=1;cout<<"g2一"<<endl;**return** c;}
16. **template**<>**double** g2(**int** a,**int** b){**int** c=1;cout<<"g2二"<<endl;**return** c;}
17. //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。
18. //template<class T2> int g2(int a, T2 b){int c=1;cout<<"g2三"<<endl;return c;}
19. //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。
20. //template<> int g2<double>(int a,int b){int c=1;cout<<"two"<<endl;return c;}
21. //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
22. //template<> T1 g2<int>(int a,int b){int c=1;cout<<"two"<<endl;return c;}
23. //类模板的特化和部分特化
24. **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。
25. //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。
26. **template**<**class** T1,**class** T3>**class** A<T1,**double**,T3>{**public**:**void** h();}//特化T2，而T1和T?不特化，注意尖括号中的类型形参是不特化的形参。
27. //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。
28. **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
29. T1是未声明的标识符。\*/
30. //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template<>
31. **template**<**class** T1,**class** T3>**void** A<T1,**double**,T3)::h(){cout<<"class A bute"<<endl;}
32. **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。
33. //main函数开始
34. **int** main()
35. {   //特化的函数模板的调用方式。
36. 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;}
37. 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;}
38. 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;}
39. g1<**double**>(3,2.3);//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。
40. //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。
41. 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;}
42. 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;}
43. 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。
44. //    template<class T1,class T2,class T3>T1 g2(T2 a,T3 b) {int c=1;cout<<"g2"<<endl;return c;}
45. // 类模板特化和部分特化的调用。
46. A<**int**,**int**,**int**> m1; m1.h();//正确，调用类模板的特化版本。
47. A<**int**,**double**,**int**> m; m.h(); //正确，调用类模板的部分特化版本。
48. //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。
49. A<**double**,**double**,**int**> m3; m3.h();//调用类A的部分特化版本。
50. 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。
51. **return** 0;
52. }

****1. 类模板****

在此之前我们来看看模板的形参。因为函数模板的参数相对比较简单，故将此内容放置于类模板中。模板形参有三种类型：类型形参、非类型形参和模板形参。先分别解释如下：

类型形参。即由关键字class 或 typename后接的说明符构成，如template <class T>void function(T a);其中T就是类型形参。类型形参的名字由用户自定义，只要是合法的标识符即可。

非类型形参。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就是非类型形参。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内部是常量。使用非类型形参应注意以下几点：

非类型形参只能是整型、指针和应用。如：double,string,string \*\*等都是不允许的，但是double & ,double \*是正确的。

调用非类型模板形参的实参必须是一个常量表达式，即在编译时就能确定其结果。任何局部对象、局部变量、局部变量地址、局部对象地址等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，不能用作非类型形参的实参。但全局变量的地址、全局对象的地址或应用const类型的变量时常量表达式，可用作非类型模板形参的实参。Sizeof表达式的结果也是一个常量表达式，同样也可以用作非类型模板形参的实参。如：

Template <class T,int a>class A{};如果有int b，这时A<int,b> m;就会出错，因为b不是常量，如果有const int b;这时A<int ,b>就是正确的。

非类型形参一般不用于函数模板中。比如有函数模板template <class T,int a>void h(T,b){};若使用h(2)调用就会出错，无法为非类型形参a推演出参数的错误。对这种函数模板可以采用显示模板实参来解决，如h<int ,3>(2)，这样就把非类型形参a设置为整数3。显示模板参数将在后面介绍。

非类型模板形参和实参间允许转换。具体如下；

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。

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。

提升转换。如template <int a>class A{};const shor b;A<b>m;即从short到int的提升转换。

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

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。如template <class T1,class T2=int>class A{};为第二个模板类型形参提供int型的默认值。

类模板的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设定了默认值之后所以的模板形参都应设定默认值。如template <class T1=int,class T2>class D{};就是错误的，因为没有给T2设定默认值。但在外部定义类中的成员时，应省去默认的形参类型。如template <class T1,class T2=int>class A{public:void H();};定义方法是template <class T1,class T2>void A<T1,T2>::H(){};

现将以上小节总结于以下一例，并通过vs2010调试，请读者仔细相关知识点的应用。

1. #include <iostream>
2. **using** **namespace** std;
3. //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。
4. //类模板的定义
5. **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
6. **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
7. //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。
8. **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。
9. //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。
10. //以下为非类型形参的定义
11. //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。
12. **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
13. **template**<**class** T1,**int** &a>**class** Cip{**public**:**void** g();};
14. **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。
15. **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。
16. **class** E{}; **template**<**class** T1,E &m> **class** Ce{}; //非类型模板形参为对象的引用。
17. //以下非类型形参的声明是错误的。
18. //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。
19. //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。
20. //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。
21. //在类模板外部定义各种类成员的方法，
22. //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
23. **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
24. **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
25. **template**<**class** T1,**class** T2>  **void** B<T1,T2>::g(){cout<<"class g f()"<<**typeid**(T1).name()<<**typeid**(T2).name()<<endl;}
26. //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
27. **template**<**class** T1,**int** a>     **void** Ci<T1,a>::g(){cout<<"class Ci g()"<<**typeid**(T1).name()<<endl;}
28. **template**<**class** T1,**int** &a>    **void** Cip<T1,a>::g(){cout<<"class Cip g()"<<**typeid**(T1).name()<<endl;}
29. //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
30. **template**<**class** T1,A<**int**> \*m> **void** Cc<T1,m>::g(){cout<<"class Cc g()"<<**typeid**(T1).name()<<endl;}
31. **template**<**class** T1,**double**\* a> **void** Cd<T1,a>::g(){cout<<"class Cd g()"<<**typeid**(T1).name()<<endl;}
32. //带有默认类型形参的模板类，在类的外部定义成员的方法。
33. //在类外部定义类的成员时，template的形参表中默认值应省略
34. **template**<**class** T1,**class** T2>  **void** D<T1,T2>::g(){cout<<"class D g()"<<endl;}
35. //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。
36. //定义一些全局变量。
37. **int** e=2;  **double** ed=2.2; **double**\*pe=&ed;
38. A<**int**> mw; A<**int**> \*pec=&mw; E me;
39. //main函数开始
40. **int** main()
41. { // 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。
42. //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。
43. //类模板调用实例
44. A<**int**> ma; //输出"class A goucao int"创建int型的类模板A的对象ma。
45. 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
46. //非类型形参的调用
47. //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
48. //    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
49. //        用作非类型模板形参的实参。
50. //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。
51. //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,GHIJKLMJKLNOPQMII//正确，数值R是一个int型常量，输出"class Ci g() int"
52. **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"
53. //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。
54. //Ci<int,e> mci; //错误，全局int型变量e也不是一个常量表达式。
55. //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
56. Cip<**int**,e> mcip;  //正确，对全局变量的引用或地址是常量表达式。
57. //Cip<int,a> mcip1; //错误，局部变量的引用或地址不是常量表达式。
58. //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
59. Cd<**int**,&ed> mcd; //正确，全局变量的引用或地址是常量表达式。
60. //Cd<int,pe> mcd1; //错误，全局变量指针不是常量表达式。
61. //double dd=aNGMIITbULcdefbbHIJKbgMIhh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
62. //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。
63. //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
64. 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
65. //Cc<int,&ma> mcc;  //错误，局部变量的地址或引用不是常量表达式。
66. //Cc<int,pec> mcc2;  //错误，全局对象的指针不是常量表达式。
67. //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
68. E me1; //Ce<int,me1> mce1; //错误，局部对象不是常量表达式
69. Ce<**int**,me> mce;  //正确，全局对象的指针或引用是常量表达式。
70. //非类型形参的转换示例，类名为Ci
71. //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。
72. **const** **short** s=3 ;Ci<**int**,s> mci ;//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
73. **return** 0;
74. }

与函数模板相同，类模板的声明语句也必须至于类声明的前面。有两个以上模板参数时，应使用逗号分开。使用含类模板的类定义对象时也必须在类名的后面带上“﹤实际类型﹥”的参数列表。类模板最常用于各种类包容关系的设计模型中。定义：

Template ﹤类型参数表﹥ class 类名 {类声明体}

在使用类模板时，应注意以下几点：

在所有出现类模板的地方不能直接用类名表示，都应加上﹤…﹥

在类模板定义体中，可以省略﹤…﹥

一个类模板的各个实例之间没有特殊的联系（形成一个独立的类）如：Queue<int> qi 和Queue<string> qs，分别表示整数队列和字符队列

实例化时机：在需要时实例化，比如定义指针或引用是不需要实例化，定义具体的变量或常量是会实例化，而访问对象的成员时会实例化。如 Queue<int> \*q //不实例化Queue<> ,Queue<int> iq //实例化Queue<>，iq->add(2) //实例化Queue<>

****类模板中的友元****：

非模板函数、类成为所有实例类的友元。如：

1. Class Foo { **void** bar();};
2. Template <**class** Type>
3. Class QueueItem
4. {
5. Friend **class** Foobar;  //类Foobar不需要先定义或声明，并没有<>
6. Frined **void** foo();    //函数foo（）
7. Frined **void** Foo::bar();//类Foo必须先定义
8. }

模板函数、模板类成为同类型实例类的友元。如：

1. Template <**class** Type> **class** Foo {…};
2. Template <**class** Type> **void** foo(QueueItem<Type>);
3. Template <**class** Type> **class** Queue{ **void** bar();};
4. Template <**class** Type> **class** QueueItem
5. {
6. Friend **class** Foo<Type>;  //模板类Foo需要先定义或声明，并带有<>
7. Friend **void** foo(QueueItem<Type>); //模板函数foo()需要先定义或声明
8. Friend **void** Queue<Type>::bar();   //模板类Queue必须先定义
9. }

模板函数、模板类成为不同类型实例类的友元。如：

1. Template <**class** T> **class** QueueItem
2. {
3. Template <**class** Type> **friend** **class** Foo;
4. Template <**class** Type> **friend** **void** foo(QueueItem<Type>);
5. Template <**class** Type> **friend** **void** Queue::bar();
6. }

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。显示实例化可以让程序员控制模板实例化发生的时间。

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。

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。

5、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。

****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。

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T1,**class** T2> **class** A{**public**:**int** a,b; **static** **int** e;};
4. **template**<**class** T3,**class** T4> **class** B;
5. **template**<**class** T5,**class** T6> **void** g(T5 a,T6 b);
6. **class** C{**public**:**void** gc(){cout<<"class C gc()"<<endl;}};
7. **void** g1(){cout<<"putong g1()"<<endl;} }
8. **template**<**class** T1,**class** T2>**template**<**class** T3,**class** T4}**class** A<T1,T2>::B{**public**:**void** gb(){cout<<"moban class B gb()"<<endl;}}
9. //在类模板外面定义类模板的模板成员类的方法
10. **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
11. **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。
12. **int** main()
13. {
14. A<**int**,**int**> ma;
15. 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。
16. 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。
17. cout<<"ma.e="<<ma.e<<endl;
18. A<**int**,**double**> ma1;
19. 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。
20. A<**int**,**int**>::B<**int**,**int**> mb;  //声明模板类中模板成员类的方法。
21. mb.gb();//调用嵌套类B的成员函数
22. //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
23. **return** 0;
24. }

简单模板实例，参数列表为基本类型：

1. #include<iostream.h>
2. **template**<**class** T>
3. **class** Array
4. {    T \*ar;
5. **public**:
6. Array(**int** c){ar=**new** T[c];}
7. **void** init(**int** n,T x){ar[n]=x;    }
8. T& operator[](**int** n){**return** ar[n];}
9. };
10. **void** main()
11. {
12. Array<**int**> array(5);
13. cout<<"Please input every element's value:"<<endl;
14. **for**(**int** i=0;i<5;i++)
15. {
16. cout<<"No."<<i+1<<':';
17. cin>>array[i];
18. }
19. }

类模板参数是类：

1. #include<iostream.h>
2. **class** A
3. {
4. **int** j;
5. **public**:
6. A(){}
7. A(**int** x):j(x){}
8. A(A \*x){j=x->j;}
9. **void** operator!(){cout<<"J="<<j<<endl;}
10. };
11. **template**<**class** T>
12. **class** B
13. {
14. **int** i;
15. T \*x;
16. **public**:
17. B(**int** xa,T \*p):i(xa){x=**new** T(p);}
18. **void** operator!(){cout<<"I="<<i<<endl;!\*x;}
19. };
21. **void** main()
22. {
23. A a(1);        //最后的显示结果为：
24. B<A> b(2,&a);  //I=2
25. !b;           //J=1
26. }

Typename的使用：

1. #include <iostream>
2. **template**<**class** T> **class** X
3. {
4. **typename** T::id i; //如无typename看看情况如何
5. **public**: **void** f()
6. {
7. i.g();
8. }
9. };
11. **class** Y
12. {
13. **public**: **class** id
14. {
15. **public**: **void** g()
16. {
17. std::cout<<"Hello World!"<<std::endl;
18. }
19. };
20. };
22. **int** main()
23. {
24. //Y y;
25. X<Y> xy;
26. xy.f();
27. **return** 0;
28. }

Typename关键字告诉编译器把一个特殊的名字解释成一个类型，在下列情况下必须对一个name使用typename关键字：

一个唯一的name（可以作为类型理解），嵌套在另一个类型中。

依赖于一个模板参数，就是说，模板参数在某种程度上包含这个name。当模板参数使编译器在指认一个类型时便会产生误解。

在定义模板时，typename和class作用基本相同，至于二者的其他关系没有什么区别，仅是历史原因，typename仅是一个新生代。

复杂的模板类实例：

1. #include<iostream.h>
2. #include<string.h>
3. **class** Student
4. {
5. **int** number;
6. **static** Student \*ip;
7. Student \*p;
8. **public**:
9. Student(){p=NULL;}
10. Student(**int** n);
11. **static** Student\* get\_first(){**return** ip;}
12. **int** get\_number(){**return** **this**->number;}
13. Student\* get\_next(){**return** **this**->p;}
14. };
16. Student::Student(**int** n):number(n)  //依据学号的大小顺序将学生对象插入链表
17. {
18. p=NULL;
19. **if**(ip==NULL)ip=**this**;  //如果是第一个则使头指针指向该对象
20. **else**{Student \*temp=ip;
21. **if**(n<ip->number){ip=**this**;p=temp;}//如学号小于第一个学生对象的学号则使头指针指向该对象
22. **else** {
23. **while**(temp)
24. {
26. **if**(n<temp->p->number)
27. {
28. p=temp->p;  //链中间对象的插入
29. temp->p=**this**;
30. **break**;
31. }**else**
32. {
33. **if**(temp->p->p==NULL)  //最后一个链的插入
34. {
35. temp->p->p=**this**;**break**;
36. }                    }
37. temp=temp->p;
38. }
39. }
40. }
41. }
42. Student\* Student::ip=NULL;
43. **template**<**class** T>
44. **class** Class
45. {
46. **int** num;
47. T \*p;
48. **public**:
49. Class(){}
50. Class(**int** n):num(n){p=NULL;}
51. T\* insert(**int** n){p=**new** T(n);**return** p;}
52. **void** list\_all\_member(T\* x)
53. {   T \*temp=x;
54. **while**(temp) { cout<<temp->get\_number()<<",";temp=temp->get\_next();}
55. }
56. };
57. **void** main()
58. {
59. Class<Student> x97x(9707);
60. x97x.insert(23);
61. x97x.insert(12);
62. x97x.insert(38);
63. x97x.insert(22);
64. x97x.insert(32);
65. x97x.list\_all\_member(Student::get\_first());
66. }

现在来讨论模板安全：模板根据参数的类型进行实例化。因为通常事先不知道其具体类型，所以也无法确切知道将在哪儿产生异常。程序员需要知道程序在什么地方发生了异常。下面看一个简单的模板类：

1. Template <**typename** T>
2. Class Wrapper
3. {
4. Public:
5. Wrapper(){}
6. T get(){**return** value\_;}
7. T set(T **const** &value){value\_=value;}
8. Private:
9. T value\_;
10. Wrapper(Wrapper **const** &);
11. Wrapper &operator=(Wrapper **const** &);
12. };

实例化过程很简单，如Wrapper <int> i；因为Wrapper <int>只接受int或其引用，所以不会触及异常，Wrapper <int>不抛异常，也没有直接或者间接调用任何可能抛异常的函数，因此Wrapper <int>是异常安全的。

现在再来看Wrapper<X>x,这里X是一个类。在这个定义里，编译器实例化了：

1. Template <> Class Wrapper<X>
2. {
3. Public:
4. Wrapper(){}
5. X get(){**return** value\_;}
6. X set(X **const** &value){value\_=value;}
7. Private:
8. T value\_;
9. Wrapper(Wrapper **const** &);
10. Wrapper &operator=(Wrapper **const** &);
11. };

现在就有问题出现了：

Wrapper<X> 包含了一个X的子对象。这个子对象需要构造，意味着调用X的构造函数，这个构造函数可能抛出异常。

Wrapper<X>::get()产生并返回了一个X的临时对象。为了这个临时对象，get()调用了X的拷贝构造函数，这个函数可能抛出异常。

Wrapper<X>::set()执行了表达式value\_=value，他实际上调用了X的赋值运算。这个运算可能抛出异常。

可以看到，同样的模板和同样的语句，但其含义不同。由于这样的不确定性，我们需要采用保守策略：假设Wrapper会根据类来实例化，而这些类在其成员上没有进行异常规格申明，则他们可能抛出异常。

再假设Wrapper的异常规格申明承诺其成员不产生异常。至少必须在其成员上加上异常规格申明throw()，所以需要修补掉这些可能导致异常的地方：

在Wrapper：：Wrapper()中构造value\_的过程。

在Wrapper::get()中返回value\_的过程。

在Wrapper::set()中队value\_的赋值过程。

另外，在违背throw()的异常规格申明是，还要处理std::unexpected.

再来看默认构造函数：

1. Wrapper() **throw**()
2. Try:T() {}
3. Catch (…){}

虽然看上去不错，但它不能工作，根据C++标准：对构造函数或析构函数上的function-try-block，当控制权到达了异常处理函数的结束点是，被捕获的异常被再次抛出。对于一般的函数，此时是函数返回，等同于没有返回值的return 语句，对于定了返回类型的函数此时的行为未定义。换句话说，上面的程序相当于：

1. X::X() **throw**()
2. Try: T (){}
3. Catch (…){ **throw**;}

这不是程序本来想要的结果，换成以下代码：

1. X::X() **throw**()
2. Try: T (){}
3. Catch (…){ **return**;}

但是它却违背了标准：如果在构造函数上的function-try-block的异常处理函数体中出现了return语句，则程序是病态的。最终：无法用function-try-block快来实现构造函数的接口安全。

引申原则1：尽可能使用构造函数不抛异常的基类或成员子对象。

引申原则2：为了帮助别人实现原则1，不要从构造函数中抛出任何异常。

其他方面的不再讨论，比如析构与关键字new等。总之，良好的设计必须满足以下两个原则：

通过异常对象的存在来注视异常状态，并适当的做出反应。

确保创造和传播异常对象不会造成更大的破坏。

最终代码的参考将如下：

1. **template** <**typename** T>
2. **class** wrapper
3. {
4. **public**:
5. wrapper()     **throw**() : value\_(NULL)
6. {
7. **try**
8. {
9. value\_ = **new** T;
10. }
11. **catch** (...) {      }
12. }
13. ~wrapper() **throw**()
14. {
15. **try**
16. {
17. **delete** value\_;
18. }
19. **catch** (...){operator **delete**(value\_);}
20. }
21. **bool** get(T &value) **const** **throw**(){**return** assign(value, \*value\_);}
22. **bool** set(T **const** &value) **throw**(){**return** assign(\*value\_, value);}
23. **private**:
24. **bool** assign(T &to, T **const** &from) **throw**()
25. {
26. **bool** error(**false**);
27. **try**{to = from; }
28. **catch** (...) { error = **true**;}
29. **return** error;
30. }
31. T \*value\_;
32. wrapper(wrapper **const** &);
33. wrapper &operator=(wrapper **const** &);
34. };
35. **void** main()
36. {
37. wrapper<**int**> mywrapper();
38. }

可以像使用普通类的方法来使用模板类，这一点毫无疑问，例如：可以继承、创建一个从现有模板继承过来的并已经初始化的模板。现在，我们来看看模板的继承，如果vector已经为你做了很多事，但你还想加入sort()的功能，则可用下面代码来扩充。

1. #ifndef SORTED
2. #define SORTED
3. #include<vector>
5. **template** <**class** T>
6. **class** Sorted:**public** std::vector<T>
7. {
8. **public**:
9. **void** sort();
10. };
12. **template** <**class** T>
13. **void** Sorted<T>::sort()
14. {
15. **for** (**int** i=size();i>0;i--)
16. {
17. **for** (**int** j=1;j<i;j++)
18. {
19. **if** (at(j-1)>at(j))
20. {
21. T t=at(j-1);
22. at(j-1)=at(j);
23. at(j)=t;
24. }
25. }
26. }
27. }
29. #endif

实现文件

1. #include "123.h"
2. #include <iostream>
3. #include <string>
4. **using** **namespace** std;
5. **char**\* words[] = {"is", "running", "big", "dog", "a"};
6. **char**\* words2[] = { "this", "that", "theother" };
7. **int** main()
8. {
9. Sorted<**int**> is;
10. **for**(**int** i = 15; i >0; i--)    is.push\_back(i);
11. **for**(**int** l = 0; l < is.size(); l++)  cout << is[l] << ' ';
12. cout << endl;
13. is.sort();
14. **for**(l = 0; l < is.size(); l++) cout << is[l] << ' ';
15. cout << endl;
16. Sorted<string\*> ss;
17. **for**(i = 0; i < 5; i++) ss.push\_back(**new** string(words[i]));
18. **for**(i = 0; i < ss.size(); i++) cout << \*ss[i] << ' ';
19. cout << endl;
20. ss.sort();
21. **for**(i = 0; i < ss.size(); i++) cout << \*ss[i] << ' ';
22. cout << endl;
23. Sorted<**char**\*> scp;
24. **for**(i = 0; i < 3; i++) scp.push\_back(words2[i]);
25. **for**(i = 0; i < scp.size(); i++) cout << scp[i] << ' ';
26. cout << endl;
27. scp.sort();
28. **for**(i = 0; i < scp.size(); i++) cout << scp[i] << ' ';
29. cout << endl;
30. **return**  0;
31. }

以上简单实现了模板的继承，读者可自行编写相关代码进行测试，并分析模板继承情况下，析构函数和构造函数等的消坏和初始化情况，这里不这讨论。

注意：子类并不会从通用的模板基类继承而来，只能是从基类的某一个实例继承而来。

现将模板的继承方式总结以下几点：

基类是模板类的一个特定实例化的版本。比如：template <class T1> class B:public A<int>{}.

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>{}。这时实例化基类就相应的被实例化一个和基类相同的实例版本，比如：B<int> b;模板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。

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

每次实例化一个模板，模板的代码都会被重新生成（除了inline标记的函数），如果一个模板某些函数不依赖于特定的类型参数而存在，那它们就可以放置在一个通用的基础类中，来阻止无意义的代码重生。

Inline函数因不产生新的代码所以它们是自由的，在整个过程中，功能性的代码只是在我们创建基础类代码时产生了一次，而且，所属权的问题也因为增加了新的析构函数而解决。通常模板只有在需要的时候才实例化，对函数模板来说，这就意味着调用它时才被实例化，但对类模板来说，它更加明细化，只有在使用到模板中的某个函数式，函数才会被实例化，换句话说：只有用到的成员函数被实例化了。例如：

1. #include <iostream.h>
2. **class** X
3. {
4. **public**:
5. **void** x()
6. {
7. cout<<"This is fuction x()"<<endl;
8. }
9. };
10. **class** Y
11. {
12. **public**:
13. **void** y()
14. {
15. cout<<"This is fuction y()"<<endl;
16. }
17. };
19. **template** <**typename** T> **class** Z
20. {
21. T t;
22. **public**:
23. **void** a() { t.x(); }
24. **void** b() { t.y(); }
25. };
27. **int** main()
28. {
29. Z<X> zx;
30. zx.a(); // Doesn't create Z<X>::b()
31. Z<Y> zy;
32. zy.b(); // Doesn't create Z<Y>::a()
33. **return** 0;
34. }

最后用模板技术演示list的的使用。

1. // Template class for storing list elements
2. #include <iostream>
3. #include <string>
4. **using** **namespace** std;
5. **template** <**class** T>                          // Use template keyword
6. **class** ListElement                            //定义类ListElement,用于表示list对象
7. {
8. **public**:
9. T data;
10. ListElement<T> \* next ;
11. ListElement(T& i\_d, ListElement <T>\* i\_n)
12. : data(i\_d),next(i\_n) { }
13. ListElement<T>\* copy()                    // copy includes all next elements
14. {
15. **return** **new** ListElement(data,(next?next->copy():0));
16. }
17. };
18. **template** <**class** T>
19. **class** ListIterator        //定义类ListIterator,用于访问和操作list对象
20. {
21. **public** :
22. //ListIterator(List<T>& l) ;
23. T operator()() ;
24. **int** operator++() ;
25. **int** operator!() ;
26. **public**:
27. ListElement<T>\* rep ;
28. };
29. **template** <**class** T>
30. T ListIterator<T>::operator() ()
31. {
32. **if** (rep) **return** rep->data;
33. **else**
34. {
35. T tmp ;    **return** tmp ;                // Default value
36. }
37. }
38. **template** <**class** T>
39. **int** ListIterator<T>::operator++()
40. {
41. **if** (rep)
42. rep = rep->next ;
43. **return** (rep != 0) ;
44. }
45. **template** <**class** T>
46. **int** ListIterator<T>::operator!()
47. {
48. **return** (rep != 0) ;
49. }
50. **template** <**class** T>
51. **class** List                //定义类List
52. {
53. **public**:
54. **friend** **class** ListIterator<T> ;
55. List();
56. List(**const** List&);
57. ~List();
58. List<T>& operator=(**const** List<T>&);
59. // typical list ops
60. T head();
61. List<T> tail();
62. **void** add(T&);
63. **friend** ostream& operator<<(ostream&, **const** List<T>&);
64. **public**:
65. **void** clear() ;                            // Delete all list elements
66. ListElement<T>\* rep ;
67. };
68. // Default Constructor
69. **template** <**class** T>
70. List<T>::List(){ rep = 0 ; }
71. // Copy Constructor
72. **template** <**class** T>List<T>::List(**const** List<T>& l)
73. {
74. rep = l.rep ? l.rep->copy() : 0 ;
75. }
76. // Overloaded assignment operator
77. **template** <**class** T>
78. List<T>& List<T>::operator=(**const** List<T>& l)
79. {
80. **if** (rep != l.rep)
81. {
82. clear() ;
83. rep = l.rep ? l.rep->copy() : 0 ;
84. }
85. **return** \***this** ;
86. }
87. // Destructor
88. **template** <**class** T>
89. List<T>::~List(){  clear() ;}
90. // Delete representation
91. **template** <**class** T>
92. **void** List<T>::clear()
93. {   **while** (rep)
94. {   ListElement<T>\* tmp = rep ;
95. rep = rep->next ;
96. **delete** tmp ;
97. }
98. rep = 0 ;
99. }
100. // Add element to front of list
101. **template** <**class** T>
102. **void** List<T>::add(T& i)
103. {
104. rep = **new** ListElement<T>(i,rep) ;
105. }
106. // Return head of list or default value of type T
107. **template** <**class** T>
108. T List<T>::head()
109. {
110. **if** (rep) **return** rep->data ;
111. **else**
112. {
113. T tmp ;
114. **return** tmp ;
115. }
116. }
117. // Return tail of list or empty list
118. **template** <**class** T>
119. List<T> List<T>::tail()
120. {    List<T> tmp ;
121. **if** (rep)
122. **if** (rep->next) tmp.rep = rep->next->copy() ;
123. **return** tmp ;
124. }
125. // Output operator
126. **template** <**class** T>
127. ostream& operator<<(ostream& os, **const** List<T>& l)
128. {
129. **if** (l.rep)
130. {
131. ListElement<T>\* p = l.rep ;
132. os << "( " ;
133. **while** (p){ os << p->data << " " ;   p = p->next ; }
134. os << ")\n" ;
135. }
136. **else**
137. os << "Empty list\n" ;
138. **return** os ;
139. }
140. **int** main()
141. {
142. List<**int**> l ;                            // Integer list
143. cout << l ;
144. **int** i=1;
145. l.add(i) ;
146. i=2;
147. l.add(i) ;
148. i=3;
149. l.add(i) ;
150. cout << "l is " << l << endl ;
151. cout << "head of l is " << l.head() << endl ;
152. List<**int**> m = l.tail() ;
153. List<**int**> o ;
154. o = m;
155. i=4;
156. m.add(i);
157. cout << "m is " << m << endl ;
158. cout << "o is " << o << endl ;
159. List<**char**> clist ;                        // Character list
160. **char** ch;
161. ch='a';
162. clist.add(ch);
163. ch='b';
164. clist.add(ch);
165. cout << clist << endl ;
166. List<string> ls ;                        // string List
167. ls.add(string("hello")) ;
168. ls.add(string("world")) ;
169. cout << "List of strings" << endl ;
170. cout << ls << endl ;
171. List<List<**int**> > listlist ;                // List of lists of integer. Notice that lists of lists are possible
172. listlist.add(o) ;
173. listlist.add(m) ;
174. cout << "List of lists of ints\n" ;
175. cout << listlist << endl ;
176. List<List<List<**int**> > > lllist ;            // List of lists of lists of integer
177. lllist.add(listlist) ;
178. lllist.add(listlist) ;
179. cout << "List of lists of lists of ints\n" ;
180. cout << lllist << "\n" ;
181. List<List<string> > slist ;               // List of list of strings
182. slist.add(ls) ;
183. slist.add(ls) ;
184. cout << "List of lists of strings\n" ;
185. cout << slist << "\n" ;
186. **return** 0 ;
187. }

注意：以上某一个实例好像未通过调试，基于时间本人已忘记，读者发现后可查看相关情况自行更正。