**C++——模板特化和偏特化**

转载 2010年07月09日 15:46:00

* 标签：
* [c++](http://so.csdn.net/so/search/s.do?q=c++&t=blog) /
* [编译器](http://so.csdn.net/so/search/s.do?q=编译器&t=blog) /
* [class](http://so.csdn.net/so/search/s.do?q=class&t=blog) /
* [button](http://so.csdn.net/so/search/s.do?q=button&t=blog) /
* [vector](http://so.csdn.net/so/search/s.do?q=vector&t=blog) /
* [存储](http://so.csdn.net/so/search/s.do?q=存储&t=blog)

**1.引言**  
C++中的模板分为类模板和函数模板，虽然它引进到C++标准中的时间不是很长，但是却得到了广泛的应用，这一点在STL中有着充分的体现。目前，STL在C++社区中得到了广泛的关注、应用和研究。理解和掌握模板是学习、应用和研究以及扩充STL的基础。而STL模板实例中又充斥着大量的模板特化和偏特化。

**2.模板的定义**  
**(1) 类模板**  
定义一个栈的类模板,它可以用来容纳不同的数据类型  
说明如下:  
template <class T>  
class stack {  
private:  
  list\* top;  
public:  
  stack();  
  stack(const stack&);  
  ~stack();  
  void push(T&);  
  T& pop();  
  //…  
};  
上述定义中，template告诉编译器这是一个模板，尖括号中的<class T >指明模板的参数，可以有一个或多个，具体实现时由用户指定，其中template <class T >中的关键字class可以用关键字typename来代替。  
类模板的使用除了要在声明时指明模板参数外，其余均与普通的类相同，例如：  
stack<int> int\_stack;  
stack<char> ch\_stack;  
stack<string> str\_stack;  
int\_stack.push(10);  
ch\_stack.push(‘z’);  
str\_stack.push(“c++”);  
**(2)函数模板**  
假设现在要定义一个max函数来返回同一类型(这种类型是允许比较的)两个值的最大者.  
template<class T>  
T mymax(const T＆ t1,const T＆ t2)  
{ return t1 < t2 ? t2 : t1; }  
template <class T>的意义与类模板定义中相同。  
模板函数的使用与普通非模板函数使用相同，因为模板函数的参数可以从其传入参数中解析出来。例如：  
int highest = mymax(5,10);  
char c = mymax(‘a’, ’z’);  
mymax(5,10)解析出模板函数参数为int, mymax(‘a’, ’z’)解析出模板函数的参数为char。

**3.模板的特化**  
**(1)类模板特化**  
有时为了需要,针对特定的类型,需要对模板进行特化,也就是特殊处理.例如,stack类模板针对bool类型,因为实际上bool类型只需要一个二进制位，就可以对其进行存储,使用一个字或者一个字节都是浪费存储空间的.  
template <class T>  
class stack {};  
template < >  
class stack<bool> { //…// };  
上述定义中template < >告诉编译器这是一个特化的模板。  
**(2) 函数模板的特化**  
看下面的例子  
main()  
{  
  int highest = mymax(5,10);  
  char c = mymax(‘a’, ’z’);  
  const char\* p1 = “hello”;  
  const char\* p2 = “world”;  
  const char\* p = mymax(p1,p2);  
}  
前面两个mymax都能返回正确的结果.而第三个却不能,因为,此时mymax直接比较两个指针p1 和 p2 而不是其指向的内容.  
针对这种情况,当mymax函数的参数类型为const char\* 时,需要特化。  
template <class T>  
T mymax(const T t1, const T t2)  
{  
   return t1 < t2 ? t2 : t1;  
}  
  
template <>  
const char\* mymax(const char\* t1,const char\* t2)  
{  
   return (strcmp(t1,t2) < 0) ? t2 : t1;  
}  
现在mymax(p1,p2)能够返回正确的结果了。

**4.模板的偏特化**  
模板的偏特化是指需要根据模板的某些但不是全部的参数进行特化  
**(1) 类模板的偏特化**  
例如c++标准库中的类vector的定义  
template <class T, class Allocator>  
class vector { // … // };  
template <class Allocator>  
class vector<bool, Allocator> { //…//};  
这个偏特化的例子中，一个参数被绑定到bool类型，而另一个参数仍未绑定需要由用户指定。  
**(2) 函数模板的偏特化**  
  严格的来说，函数模板并不支持偏特化，但由于可以对函数进行重载，所以可以达到类似于类模板偏特化的效果。  
  template <class T> void f(T);  (a)  
  根据重载规则，对（a）进行重载  
  template < class T> void f(T\*);  (b)  
  如果将（a）称为基模板，那么（b）称为对基模板（a）的重载，而非对（a）的偏特化。C++的标准委员会仍在对下一个版本中是否允许函数模板的偏特化进行讨论。

**5.模板特化时的匹配规则**  
**(1) 类模板的匹配规则**  
最优化的优于次特化的，即模板参数最精确匹配的具有最高的优先权  
例子：  
template <class T> class vector{//…//}; // (a)  普通型  
template <class T> class vector<T\*>{//…//};  // (b) 对指针类型特化  
template <>   class vector <void\*>{//…//};  // (c) 对void\*进行特化  
每个类型都可以用作普通型（a）的参数，但只有指针类型才能用作（b）的参数，而只有void\*才能作为(c)的参数  
**(2) 函数模板的匹配规则**  
非模板函数具有最高的优先权。如果不存在匹配的非模板函数的话，那么最匹配的和最特化的函数具有高优先权  
例子：  
template <class T> void f(T);  // (d)  
template <class T> void f(int, T, double); // (e)  
template <class T> void f(T\*);  // (f)  
template <> void f<int> (int) ; // (g)  
void f(double);  // (h)  
bool b;  
int i;  
double d;  
f(b); // 以 T = bool 调用 （d）  
f(i,42,d) // 以 T = int 调用（e）  
f(&i) ; // 以 T = int\* 调用（f）  
f(d);  //  调用（g）***我怎么觉得是调用（h）***  
**参考文献**  
[1] Bjarne Stroustrup, The C++ Programming Language (Special Edition), Addison Wesley,2000  
[2] Nicolai M.Josuttis, The C++ Standard Library – A Tutorial and Reference ,Addison Wesley,1999  
[3] Stanley Lippman and Josée Lajoie ,C++ Primier, 3rd Edition ,Addison Wesley Longman ,1998

转载自：<http://blog.csdn.net/zhang810413/archive/2007/12/18/1948603.aspx>

**补充：**

Partial Template Specialization能够让你在模板(Template)的所有可能的实体中特化出一组子集.  
1.**模板的特化(template specialization):**  
    例如,定义如下的一个模板:  
    template<class Window, class Controller>  
    class Widget  
    {  
      ... 泛化实现代码 ...  
    };  
    然后你可以像下面那样明确地加以特化:  
    template<>    //注意:template后面的尖括号中不带任何内容;  
    class Widget<ModalDialog, MyController>  
    {  
      ... 特化实现代码 ...  
    };  
    其中ModalDialog和MyController是你自己另外定义的类;有了这个Widget的特化定义之后,如果你以后定义了Widget<ModalDialog, MyController>对象时,编译器就会使用上述的特化定义,如果定义了其它泛型对象,那么编译器就是用原本的泛化定义;这就是模板的特化.  
  
2.**Partial Template Specialization(模板偏特化)**  
    模板特化是通过"给模板中的所有模板参数一个具体的类"的方式来实现的.而模板偏特化则是通过"给模板中的部分模板参数以具体的类,而留下剩余的模板参数仍然使用原来的泛化定义"的方式来实现的;  
    比如,就上面的Widget类模板的定义来说,有时候想针对任意的Window来搭配一个特定的MyController类特化Widget,这个时候就需要使用模板偏特化机制了.下面的Widget类模板就是Widget的偏特化定义:  
    template<class Window>                        //仍然使用原来的泛化定义;  
    class Widget<Window, MyController>            //MyController是具体的类,是特化定义;  
    {  
      ... 偏特化实现代码 ...  
    };  
    这就是一个偏特化定义;一个MyController类可以搭配任意一种Window.  
    通常在一个类模板的偏特化定义中,你只会特化某些模板参数而留下其它泛化参数.当你在程序中具体实现上述类模板的时,编译器会试着找出最匹配的模板定义.这个寻找过程十分复杂精细,允许你以富有创意的方式来进行偏特化.例如,假设你有一个Button类模板,它有一个模板参数,那么,你不但可以拿任意的Window搭配特定的MyController来特化Widget,还可以拿任意Button搭配特定的MyController来偏特化Widget:  
    template<class ButtonArg>  
    class Widget<Button<ButtonArg>, MyController>    //使用任意Button搭配具体的类MyContorller  
    {  
      ... 偏特化实现代码 ...  
    };  
    模板的偏特化能力很强大.当你实例化一个模板时,编译器会把目前存在的偏特化模板和全特化模板做比较,并找出其中最合适、最匹配的实现.这样,灵活性就很大.但是不幸的是,模板的偏特化机制不能用在函数身上,不论成员函数还是非成员函数.  
  
注意:  
1.虽然你可以全特化类模板中的成员函数,但是你不能偏特化他们;  
2.你不能偏特化命名空间级别(namespace-level)的函数(non-member).最接近"命名空间级别模板函数"的偏特化机制就是函数重载,那就意味着你对"函数参数"(而非返回值类型或内部所用类型)有很精致的特化能力;  
3.特化或全特化时,template后面的尖括号中不带任何内容;  
  
总结:  
模板特化/全特化是指给每一个模板参数一个具体的类型,以具体实现这个模板,而且template后面的尖括号中不带任何内容;  
模板偏特化是指只给部分模板参数一个具体的类型,来实现这个模板;

转载自：<http://hi.baidu.com/klcdyx2008/blog/item/5adbf77b79f316f90bd1873c.html>