[**C++ 模板里面的typename**](http://www.cnblogs.com/xuxm2007/archive/2011/12/11/2283907.html)

最近写个小东西,遇到模板的一个报错,挺有意思的,错误代码如下

[复制代码](javascript:void(0);)

 1 #include<iostream>  
 2 #include<set>  
 3 using namespace std;  
 4   
 5 template <class T>  
 6 void Set2Str(set<T> inset)  
 7 {  
 8     //typename T::iterator itset;  
 9     set<T>::iterator itset;  
10 }  
11   
12   
13 int main (int argc, char const\* argv[])  
14 {  
15     set<int> a;  
16     Set2Str<int>(a);  
17   
18     return 0;  
19 }

[复制代码](javascript:void(0);)

 报错信息如下

1.cpp: In function `void Set2Str(std::set<T, std::less<\_Key>, std::allocator<\_CharT> >)':  
1.cpp:9: 错误：expected `;' before "itset"  
1.cpp: In function `void Set2Str(std::set<T, std::less<\_Key>, std::allocator<\_CharT> >) [with T = int]':  
1.cpp:16:   instantiated from here  
1.cpp:9: 错误：dependent-name ` std::set<T,std::less<\_Key>,std::allocator<\_CharT> >::iterator' is parsed as a non-type, but instantiation yields a type  
1.cpp:9: 附注：say `typename  std::set<T,std::less<\_Key>,std::allocator<\_CharT> >::iterator' if a type is meant

大家注意这一句"XXX,is parsed as a non-type, but instantiation yields a type" ,这个就是错误的原因,其实下面已经给出了解决方法,"say `typename' XXX if a type is meant".就是添加typename.就如同上面的注释一样.

具体解释请参见下面的网址

http://www.michael-hammer.at/blog/type\_or\_non-type/

http://blog.csdn.net/dick\_china/article/details/4522253

问题：在下面的 template declarations（模板声明）中 class 和 typename 有什么不同？

template<class T> class Widget; // uses "class"  
template<typename T> class Widget; // uses "typename"

　　答案：没什么不同。在声明一个 template type parameter（模板类型参数）的时候，class 和 typename 意味着完全相同的东西。一些程序员更喜欢在所有的时间都用 class，因为它更容易输入。其他人（包括我本人）更喜欢 typename，因为它暗示着这个参数不必要是一个 class type（类类型）。少数开发者在任何类型都被允许的时候使用 typename，而把 class 保留给仅接受 user-defined types（用户定义类型）的场合。但是从 C++ 的观点看，class 和 typename 在声明一个 template parameter（模板参数）时意味着完全相同的东西。

　　然而，C++ 并不总是把 class 和 typename 视为等同的东西。有时你必须使用 typename。为了理解这一点，我们不得不讨论你会在一个 template（模板）中涉及到的两种名字。

　　假设我们有一个函数的模板，它能取得一个 STL-compatible container（STL 兼容容器）中持有的能赋值给 ints 的对象。进一步假设这个函数只是简单地打印它的第二个元素的值。它是一个用糊涂的方法实现的糊涂的函数，而且就像我下面写的，它甚至不能编译，但是请将这 些事先放在一边——有一种方法能发现我的愚蠢：

template<typename C> // print 2nd element in  
void print2nd(const C& container) // container;  
{  
　// this is not valid C++!  
　if (container.size() >= 2) {  
　　C::const\_iterator iter(container.begin()); // get iterator to 1st element  
　　++iter; // move iter to 2nd element  
　　int value = \*iter; // copy that element to an int  
　　std::cout << value; // print the int  
　}  
}

　　我突出了这个函数中的两个 local variables（局部变量），iter 和 value。iter 的类型是 C::const\_iterator，一个依赖于 template parameter（模板参数）C 的类型。一个 template（模板）中的依赖于一个 template parameter（模板参数）的名字被称为 dependent names（依赖名字）。当一个 dependent names（依赖名字）嵌套在一个 class（类）的内部时，我称它为 nested dependent name（嵌套依赖名字）。C::const\_iterator 是一个 nested dependent name（嵌套依赖名字）。实际上，它是一个 nested dependent type name（嵌套依赖类型名），也就是说，一个涉及到一个 type（类型）的 nested dependent name（嵌套依赖名字）。

print2nd 中的另一个 local variable（局部变量）value 具有 int 类型。int 是一个不依赖于任何 template parameter（模板参数）的名字。这样的名字以 non-dependent names（非依赖名字）闻名。（我想不通为什么他们不称它为 independent names（无依赖名字）。如果，像我一样，你发现术语 "non-dependent" 是一个令人厌恶的东西，你就和我产生了共鸣，但是 "non-dependent" 就是这类名字的术语，所以，像我一样，转转眼睛放弃你的自我主张。）

　　nested dependent name（嵌套依赖名字）会导致解析困难。例如，假设我们更加愚蠢地以这种方法开始 print2nd：

template<typename C>  
void print2nd(const C& container)  
{  
　C::const\_iterator \* x;  
　...  
}

　　这看上去好像是我们将 x 声明为一个指向 C::const\_iterator 的 local variable（局部变量）。但是它看上去如此仅仅是因为我们知道 C::const\_iterator 是一个 type（类型）。但是如果 C::const\_iterator 不是一个 type（类型）呢？如果 C 有一个 static data member（静态数据成员）碰巧就叫做 const\_iterator 呢？再如果 x 碰巧是一个 global variable（全局变量）的名字呢？在这种情况下，上面的代码就不是声明一个 local variable（局部变量），而是成为 C::const\_iterator 乘以 x！当然，这听起来有些愚蠢，但它是可能的，而编写 C++ 解析器的人必须考虑所有可能的输入，甚至是愚蠢的。

　　直到 C 成为已知之前，没有任何办法知道 C::const\_iterator 到底是不是一个 type（类型），而当 template（模板）print2nd 被解析的时候，C 还不是已知的。C++ 有一条规则解决这个歧义：如果解析器在一个 template（模板）中遇到一个 nested dependent name（嵌套依赖名字），它假定那个名字不是一个 type（类型），除非你用其它方式告诉它。缺省情况下，nested dependent name（嵌套依赖名字）不是 types（类型）。（对于这条规则有一个例外，我待会儿告诉你。）

记住这个，再看看 print2nd 的开头：

template<typename C>  
void print2nd(const C& container)  
{  
　if (container.size() >= 2) {  
　　C::const\_iterator iter(container.begin()); // this name is assumed to  
　　... // not be a type

　　这为什么不是合法的 C++ 现在应该很清楚了。iter 的 declaration（声明）仅仅在 C::const\_iterator 是一个 type（类型）时才有意义，但是我们没有告诉 C++ 它是，而 C++ 就假定它不是。要想转变这个形势，我们必须告诉 C++ C::const\_iterator 是一个 type（类型）。我们将 typename 放在紧挨着它的前面来做到这一点：

template<typename C> // this is valid C++  
void print2nd(const C& container)  
{  
if (container.size() >= 2) {  
typename C::const\_iterator iter(container.begin());  
...  
}  
}

　　通用的规则很简单：在你涉及到一个在 template（模板）中的 nested dependent type name（嵌套依赖类型名）的任何时候，你必须把单词 typename 放在紧挨着它的前面。（重申一下，我待会儿要描述一个例外。）

　　typename 应该仅仅被用于标识 nested dependent type name（嵌套依赖类型名）；其它名字不应该用它。例如，这是一个取得一个 container（容器）和这个 container（容器）中的一个 iterator（迭代器）的 function template（函数模板）：

template<typename C> // typename allowed (as is "class")  
void f(const C& container, // typename not allowed  
typename C::iterator iter); // typename required

　　C 不是一个 nested dependent type name（嵌套依赖类型名）（它不是嵌套在依赖于一个 template parameter（模板参数）的什么东西内部的），所以在声明 container 时它不必被 typename 前置，但是 C::iterator 是一个 nested dependent type name（嵌套依赖类型名），所以它必需被 typename 前置。

"typename must precede nested dependent type names"（“typename 必须前置于嵌套依赖类型名”）规则的例外是 typename 不必前置于在一个 list of base classes（基类列表）中的或者在一个 member initialization list（成员初始化列表）中作为一个 base classes identifier（基类标识符）的 nested dependent type name（嵌套依赖类型名）。例如：

template<typename T>  
class Derived: public Base<T>::Nested {  
　// base class list: typename not  
　public: // allowed  
　　explicit Derived(int x)  
　　: Base<T>::Nested(x) // base class identifier in mem  
　　{  
　　　// init. list: typename not allowed  
　  
　　　typename Base<T>::Nested temp; // use of nested dependent type  
　　　... // name not in a base class list or  
　　} // as a base class identifier in a  
　　... // mem. init. list: typename required  
};

　　这样的矛盾很令人讨厌，但是一旦你在经历中获得一点经验，你几乎不会在意它。

　　让我们来看最后一个 typename 的例子，因为它在你看到的真实代码中具有代表性。假设我们在写一个取得一个 iterator（迭代器）的 function template（函数模板），而且我们要做一个 iterator（迭代器）指向的 object（对象）的局部拷贝 temp，我们可以这样做：

template<typename IterT>  
void workWithIterator(IterT iter)  
{  
　typename std::iterator\_traits<IterT>::value\_type temp(\*iter);  
　...  
}

　　不要让 std::iterator\_traits<IterT>::value\_type 吓倒你。那仅仅是一个 standard traits class（标准特性类）的使用，用 C++ 的说法就是 "the type of thing pointed to by objects of type IterT"（“被类型为 IterT 的对象所指向的东西的类型”）。这个语句声明了一个与 IterT objects 所指向的东西类型相同的 local variable（局部变量）(temp)，而且用 iter 所指向的 object（对象）对 temp 进行了初始化。如果 IterT 是 vector<int>::iterator，temp 就是 int 类型。如果 IterT 是 list<string>::iterator，temp 就是 string 类型。因为 std::iterator\_traits<IterT>::value\_type 是一个 nested dependent type name（嵌套依赖类型名）（value\_type 嵌套在 iterator\_traits<IterT> 内部，而且 IterT 是一个 template parameter（模板参数）），我们必须让它被 typename 前置。

如果你觉得读 std::iterator\_traits<IterT>::value\_type 令人讨厌，就想象那个与它相同的东西来代表它。如果你像大多数程序员，对多次输入它感到恐惧，那么你就需要创建一个 typedef。对于像 value\_type 这样的 traits member names（特性成员名），一个通用的惯例是 typedef name 与 traits member name 相同，所以这样的一个 local typedef 通常定义成这样：

template<typename IterT>  
void workWithIterator(IterT iter)  
{  
　typedef typename std::iterator\_traits<IterT>::value\_type value\_type;  
　value\_type temp(\*iter);  
　...  
}

　　很多程序员最初发现 "typedef typename" 并列不太和谐，但它是涉及 nested dependent type names（嵌套依赖类型名）规则的一个合理的附带结果。你会相当快地习惯它。你毕竟有着强大的动机。你输入 typename std::iterator\_traits<IterT>::value\_type 需要多少时间？

　　作为结束语，我应该提及编译器与编译器之间对围绕 typename 的规则的执行情况的不同。一些编译器接受必需 typename 时它却缺失的代码；一些编译器接受不许 typename 时它却存在的代码；还有少数的（通常是老旧的）会拒绝 typename 出现在它必需出现的地方。这就意味着 typename 和 nested dependent type names（嵌套依赖类型名）的交互作用会导致一些轻微的可移植性问题。

　　Things to Remember

　　·在声明 template parameters（模板参数）时，class 和 typename 是可互换的。

　　·用 typename 去标识 nested dependent type names（嵌套依赖类型名），在 base class lists（基类列表）中或在一个 member initialization list（成员初始化列表）中作为一个 base class identifier（基类标识符）时除外。