**类型引用**

|  |
| --- |
| template  class Foo  {  typedef T::SomeType SomeType;  }; |

这段代码在VC++中一点问题也没有，但是GCC并不允许，因为它不知道T::SomeType是什么。你需要改为：

|  |
| --- |
| template  class Foo  {  typedef typename T::SomeType SomeType;  }; |

通过typename T::SomeType告诉GCC，SomeType是一个类型名，而不是其他东西。

当然，这种情况不只是出现在typedef中。例如：

|  |
| --- |
| template  void visit(const Container& cont)  {  for (Container::const\_iterator it = cont.begin(); i  t != cont.end(); ++it) ...  } |

这里的Container::const\_iterator同样需要改为typename Container::const\_iterator。

**基类成员引用**

|  |
| --- |
| template  class Foo :  public Base {  public: void foo()  { base\_func();  m\_base\_member = 0;  }  }; |

这段代码在VC++中同样没有问题，但是GCC中不能通过。因为GCC并不知道base\_func，m\_base\_member是什么。对于这个问题，你可以有两种改法：

改法1：加上域作用符Base::

|  |
| --- |
| template  class Foo :  public Base  {  public: void foo()  { Base::base\_func();  Base::m\_base\_member = 0;  }  }; |

改法2：使用using指示符

|  |
| --- |
| template  class Foo :  public Base  {  public: using Base::base\_func;  using Base::m\_base\_member;  void foo()  { base\_func();  m\_base\_member = 0;  }  }; |

这两种方法各有好处，在class Foo中出现大量的Base::base\_func、m\_base\_member的引用时，使用using是方便的。而如果只有一次，那么方法1显得简短。

**交叉引用许可**

|  |
| --- |
| class SomeClass; template class Foo { public: void foo(SomeClass& a) { a.some\_func(); } void foo2() { SomeClass a; a.some\_func(); } }; class SomeClass { public: void some\_func() { ... } }; |

由于VC++对模板函数的迟编译，因此，一个模板类不只是可以调用一个尚未出现的类成员函数(或者访问其成员变量)，甚至可以定义其实例。这种语法对C++来说确实显得怪异。因为等到编译后面的SomeClass时，他又可以在其函数中定义class Foo的实例，从而出现交叉引用的情况。这在非模板函数的情形下就算你用VC++亦难以做到。

遇到这种情况，该如何移植到GCC中?这个问题有点棘手。我个人认为出现这种情况是不太应该的，这意味着对类与类之间的关系混淆不清。你需要仔细审视一下这两个类正确的关系是什么。如果是移植库(例如WTL就有多处这样的情形)的过程中遇到这种情况，可以把函数的实现体改为放到类定义体外，如下：

|  |
| --- |
| class SomeClass; template class Foo { public: void foo(SomeClass& a); void foo2(); }; class SomeClass { public: void some\_func() { ... } }; template inline void Foo::foo(SomeClass& a) { a.some\_func(); } template inline void Foo::foo2() { SomeClass a; a.some\_func(); } |

void asyncWrite(\_Buffers& \_buffers)

{

ioService().dispatch(boost::bind(&ServiceHandlerType\_::template asyncWriteImpl<\_Buffers>,

shared\_from\_this(),

\_buffers));

}