现在，有下面的代码：

```
namespace lx1
{
    class Point3d
    {
    public:
        Point3d (double dx, double dy, double dz)
            : m_dX(dx), m_dY(dy), m_dZ(dz)
        {}

        double getX() const { return m_dX; };
        double getY() const { return m_dY; };
        double getZ() const { return m_dZ; };

    private:
        double m_dX;
        double m_dY;
        double m_dZ;
    };

    void TestPoint(const Point3d &pt)
    {
        cout << "Output from lx1::TestPoint()." << endl;
    }
}

namespace lx2
{
    void TestPoint(const lx1::Point3d &pt)
    {
        cout << "Output from lx2::TestPoint()." << endl;
    }

    void ShowPoint3d(const lx1::Point3d &pt)
```

```
  {
    TestPoint(pt);

    cout << "X: " << pt.getX() << endl;
    cout << "Y: " << pt.getY() << endl;
    cout << "Z: " << pt.getZ() << endl;
  }
}
```

你能发现代码中有什么问题吗？

上面的代码看上去没有什么问题，却不能通过编译，会得到一个“'lx2::TestPoint' : ambiguous call to overloaded function”的错误。也就是说编译器不能确定在 ShowPoint3d()函数中调用的是哪个 TestPoint()函数。

也许你会非常不解，为什么会出现这样的编译错误。在命名空间 lx2 中只有一个函数 TestPoint()，为什么编译器会不能确定调用哪个 TestPoint()函数呢？虽然在命名空间 lx1 中有一个跟 lx2 中参数列表相同的 TestPoint()函数，可是在命名空间 lx2 中并没有用 using namespace lx1;这样的语句，编译器应该不会去命名空间 lx1 中去匹配 TestPoint()函数呀。

事实上，出现编译错误的原因就是在命名空间 lx1 和 lx2 里面都有一个函数列表相同的 TestPoint()函数。

在 C++中有这样一个名字查找规则——**如果在声明函数的参数时使用了一个类，那么在查找匹配的函数名字时，编译器会在包含参数类型的名字空间中也进行查找**。

在上面的代码中，命名空间 lx2 中的 TestPoint()函数参数是 lx1::Point3d。按照上面的规则，编译器在查找匹配的函数名字时，也会去包含参数 Point3d 的名字空间（也就是 lx1）中进行匹配查找。而在命名空间 lx1 中也有一个参数列表跟命名空间 lx2 中一样的 TestPoint()函数，所以会出现上面的编译错误。

这是 C++中一条非常容易被忽视的名字查找规则，因此要格外重视。

这个是我学习 C++以后一直不知道的一个方面知识，最近想回顾一下 C++的基础知识，所以在看一本书。这本书中讲了这个关于名字查找这个部分我觉得很新奇。我就稍微总结一下。

什么是名字查找？

以我的感觉应该算是这样的吧，比如你调用了一个函数，编译器是怎么去寻找这个函数的定义的吧。这个就是所谓的名字查找的过程。先写一个程序来看看这个名字查找的运行过程吧!!!

[cpp] view plain copy print?

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
namespace test
{
    class A
    {
    public:
        A():str_("")
        {
        }
        A(std::string str):str_(str)
        {
        }
        void setstr(std::string const &str)
        {
        this->str_ = str;
        }

        std::string getstr() const
        {
        return this->str_;
        }
    private:
        std::string str_;
    };
}
std::istream& operator>>(std::istream& in,test::A& thiz)
{
    std::string str_tmp;
    if(in >> str_tmp)
    {
    thiz.setstr(str_tmp);
    }
    else
    {
    thiz.setstr("wrong");
    }
    return in;
}
```

```
std::ostream& operator<<(std::ostream& out,test::A const& thiz)
{
    out<<thiz.getstr()<<std::endl;
    return out;
}
int main()
{
    using namespace test;
    using namespace std;
    vector<A> token;
    copy(istream_iterator<A>(cin),istream_iterator<A>(),back_inserter(token))
;
    copy(token.begin(),token.end(),ostream_iterator<A>(cout," "));
    return 0;
}
```

这个程序的 class A 这个类定义在 namespace test 中，而<< >>这两个操作符则定义在全局作用域中，运行结果发生编译提出了
很多莫名其妙的错误，最为主要的就是这句：
“error: no match for 'operator>>' in
'*((std::istream_iterator<test::A>*)this)->std::istream_iterator<test::A>::_M_stream >>
((std::istream_iterator<test::A>*)this)->std::istream_iterator<test::A>::_M_value'
”

大概可以理解为没有匹配的>>这个函数的意思吧。这就是疑问所在了，为什么我定义的流操作符在不能识别，却说没有找到匹配的函数呢。

这就是 C++ 编译器中的名字查找的功能。
C++名字查找有两个方法
一个是 OL(ordinary name lookup) 普通查找规则

一个是 ADL(argument-depentment lookup)依赖于实参的名字查找

这两个查找规则就是 C++的查找规则，如果经过这两个规则还是没有找过的话那编译器就会报出没有匹配函数这样的错误。OL 这个规则是从相邻的作用域开始进行查找，如果没有找过的话，那就到更加大的一个作用域去进行查找，OL 有这样的一个规则(OL terminates as soon as the name is found),也就是说当编译器在这个作用域中找到了与要找的函数名相同的时候，这就不再会到更加大的作用域中去寻找。也就说停止在这个地方了。如果存在有重

载的问题，那编译器会在找到的这个作用域内的进行考虑，到底哪个函数才是最为匹配的，但是它不会往外层进行查找了。这就有可能会形成错误。

ADL 规则的意思就是和字面意思差不多，编译器根据实参的类型，去包含着这些类型的名字空间中去查找我们所要的函数定义或者名字。如果是类的话，那就可能包含了它的本身还是他所有基类的名字空间，如果是模板类的话，那就是定义原型模板的名字空间和所有模板实参的名字空间。

分析上述程序的名字查找的过程，copy1(copy 第一次调用)会调用>>这个操作符，编译器会把当作 istream 的成员函数，然后去 istream 的类中去寻找这个函数的名字，但由于 istream 只是对内定的数据类型是重载的，所有不是最佳的匹配，但是编译器还是在类的作用域中找到了这个名字，所以他不会去全局中去找。

然而编译器使用 ADL 规则也只是找到了名字空间 std 和 test 两个，但是我们自定义的这个流操作符却是定义在全局作用域中，所以最后会报错。

如果将自己定义的<< >>包含在 test 中的话，那就可以通过编译运行的。

所以将相关操作符的声明和主要类型放在同一个名字空间中非常重要的。不然编译器找不到他们的....