touzani的专栏 αβ

C++ 命名空间namespace

分类： C/ C++ 2007-06-04 16:14 15498人阅读 评论(14) 收藏 举报

c++ iostream string 编译器 list system

目录(?) [+]

命名空间

在C++中，名称（name）可以是符号常量、变量、宏、函数、结构、枚举、类和对象等等。为了避免，在大规模程序的设计中，以及在程序员使用各种各样的C++库时，这些标识符的命名发生冲突，标准C++引入了关键字namespace（命名空间/名字空间/名称空间/名域），可以更好地控制标识符的作用域。

MFC中并没有使用命名空间，但是在.NET框架、MC++和C++/CLI中，都大量使用了命名空间。

1）作用域与命名空间

- **相关概念**

与命名空间相关的概念有：

- 声明域（declaration region）—— 声明标识符的区域。如在函数外面声明的全局变量，它的声明域为声明所在的文件。在函数内声明的局部变量，它的声明域为声明所在的代码块（例如整个函数体或整个复合语句）。
- 潜在作用域（potential scope）—— 从声明点开始，到声明域的末尾的区域。因为C++采用的是先声明后使用的原则，所以在声明点之前的声明域中，标识符是不能用的。即，标识符的潜在作用域，一般会小于其声明域。
- 作用域（scope）—— 标识符对程序可见的范围。标识符在其潜在作用域内，并非在任何地方都是可见的。例如，局部变量可以屏蔽全局变量、嵌套层次中的内层变量可以屏蔽外层变量，从而被屏蔽的全局或外层变量在其倍屏蔽的区域内是不可见的。所以，一个标识符的作用域可能小于其潜在作用域。

- **命名空间**

命名空间（namespace）是一种描述逻辑分组的机制，可以将按某些标准在逻辑上属于同一个集团的声明放在同一个命名空间中。

原来C++标识符的作用域分成三级：代码块（{……}，如复合语句和函数体）、类和全局。现在，在其中的类和全局之间，标准C++又添加了命名空间这一个作用域级别。

命名空间可以是全局的，也可以位于另一个命名空间之中，但是不能位于类和代码块中。所以，在命名空间中声明的名称（标识符），默认具有外部链接特性（除非它引用了常量）。

评论排行

推荐文章

在所有命名空间之外，还存在一个全局命名空间，它对应于文件级的声明域。因此，在命名空间机制中，原来的全局变量，现在被认为位于全局命名空间中。

标准C++库（不包括标准C库）中所包含的所有内容（包括常量、变量、结构、类和函数等）都被定义在命名空间std（standard标准）中了。

2）定义命名空间

有两种形式的命名空间——有名的和无名的。

命名空间的定义格式为：（取自C++标准文档）

named-namespace-definition:

namespace identifier { namespace-body }

unnamed-namespace-definition:

namespace { namespace-body }

namespace-body:

declaration-seq$_{opt}$

即：（自己翻译并改写的）

有名的命名空间：

namespace 命名空间名 {

声明序列$_{可选}$

```
}
```

无名的命名空间：

```
namespace {
    声明序列可选
}
```

命名空间的成员，是在命名空间定义中的花括号内声明了的名称。可以在命名空间的定义内，定义命名空间的成员（内部定义）。也可以只在命名空间的定义内声明成员，而在命名空间的定义之外，定义命名空间的成员（外部定义）。

命名空间成员的外部定义的格式为：

命名空间名::成员名 ……

例如：

```
// out.h
namespace Outer { // 命名空间Outer的定义
    int i; // 命名空间Outer的成员i的内部定义
    namespace Inner { // 子命名空间Inner的内部定义
        void f() { i++; } // 命名空间Inner的成员f()的内部定义，其中的i为Outer::i
        int i;
```

```
        void g() { i++; } // 命名空间Inner的成员g()的内部定义，其中的i为Inner::i

        void h(); // 命名空间Inner的成员h()的声明

    }

    void f(); // 命名空间Outer的成员f()的声明

    // namespace Inner2; // 错误，不能声明子命名空间

}

void Outer::f() {i--;} // 命名空间Outer的成员f()的外部定义

void Outer::Inner::h() {i--;} // 命名空间Inner的成员h()的外部定义

// namespace Outer::Inner2 {/*……*/} // 错误，不能在外部定义子命名空间
```

注意：

不能在命名空间的定义中声明（另一个嵌套的）子命名空间，只能在命名空间的定义中定义子命名空间。

也不能直接使用“命名空间名::成员名 ……”定义方式，为命名空间添加新成员，而必须先在命名空间的定义中添加新成员的声明。

另外，命名空间是开放的，即可以随时把新的成员名称加入到已有的命名空间之中去。方法是，多次声明和定义同一命名空间，每次添加自己的新成员和名称。例如：

```
namespace A {
```

```
        int i;
        void f();
    } // 现在A有成员i和f()
    namespace A {
        int j;
        void g();
    } // 现在A有成员i、f()、j和g()
```

还可以用多种方法，来组合现有的命名空间，让它们为我所用。例如：

```
    namespace My_lib {
        using namespace His_string;
        using namespace Her_vector;
        using Your_list::List;
        void my_f(String &, List &);
    }
    ……
    using namespace My_lib;
    ……
```

```
Vector<String> vs[5];

List<int> li[10];

my_f(vs[2], li[5]);
```

3）使用命名空间

- **作用域解析运算符（::）**

对命名空间中成员的引用，需要使用命名空间的作用域解析运算符::。例如：

```
// out1.cpp

#include "out.h"

#include <iostream>

int main ( ) {

    Outer::i = 0;

    Outer::f(); // Outer::i = -1;

    Outer::Inner::f(); // Outer::i = 0;

    Outer::Inner::i = 0;

    Outer::Inner::g(); // Inner::i = 1;

    Outer::Inner::h(); // Inner::i = 0;
```

```
    std::cout << "Hello, World!" << std::endl;

    std::cout << "Outer::i = " << Outer::i << ",  Inner::i = " << Outer::Inner::i << std::endl;

}
```

- **using指令（using namespace）**

为了省去每次调用Inner成员和标准库的函数和对象时，都要添加Outer::Inner::和sta::的麻烦，可以使用标准C++的using编译指令来简化对命名空间中的名称的使用。格式为：

using namespace 命名空间名[::命名空间名……];

在这条语句之后，就可以直接使用该命名空间中的标识符，而不必写前面的命名空间定位部分。因为using指令，使所指定的整个命名空间中的所有成员都直接可用。例如：

```
// out2.cpp

#include "out.h"

#include <iostream>

// using namespace Outer; // 编译错误，因为变量i和函数f()有名称冲突

using namespace Outer::Inner;

using namespace std;

int main ( ) {
```

```
        Outer::i = 0;

        Outer::f(); // Outer::i = -1;

        f(); // Inner::f()，Outer::i = 0;

        i = 0; // Inner::i

        g(); // Inner::g()，Inner::i = 1;

        h(); // Inner::h()，Inner::i = 0;

        cout << "Hello, World!" << endl;

        cout << "Outer::i = " << Outer::i << ",  Inner::i = " << i << endl;

    }
```

又例如：（.NET框架）

```
    using namespace System::Drawing::Imaging;

    using namespace System::Window::Forms::Design::Behavior;
```

- **using声明（using）**

除了可以使用using编译指令（组合关键字using namespace）外，还可以使用using声明来简化对命名空间中的名称的使用。格式为：

using 命名空间名::[命名空间名::……]成员名;

注意，关键字using后面并没有跟关键字namespace，而且最后必须为命名空间的成员名（而在using编译指令的最后，必须为命名空间名）。

与using指令不同的是，using声明只是把命名空间的特定成员的名称，添加该声明所在的区域中，使得该成员可以不需要采用，（多级）命名空间的作用域解析运算符来定位，而直接被使用。但是该命名空间的其他成员，仍然需要作用域解析运算符来定位。例如：

```
// out3.cpp

#include "out.h"

#include <iostream>

using namespace Outer; // 注意，此处无::Inner

using namespace std;

// using Inner::f; // 编译错误，因为函数f()有名称冲突

using Inner::g; // 此处省去Outer::，是因为Outer已经被前面的using指令作用过了

using Inner::h;

int main ( ) {

    i = 0; // Outer::i

    f(); // Outer::f()，Outer::i = -1;

    Inner::f(); // Outer::i = 0;

    Inner::i = 0;
```

```
        g(); // Inner::g()，Inner::i = 1;

        h(); // Inner::h()，Inner::i = 0;

        cout << "Hello, World!" << endl;

        cout << "Outer::i = " << i << ",  Inner::i = " << Inner::i << endl;

    }
```

- **using指令与using声明的比较**

可见，using编译指令和using声明，都可以简化对命名空间中名称的访问。

using指令使用后，可以一劳永逸，对整个命名空间的所有成员都有效，非常方便。而using声明，则必须对命名空间的不同成员名称，一个一个地去声明，非常麻烦。

但是，一般来说，使用using声明会更安全。因为，using声明只导入指定的名称，如果该名称与局部名称发生冲突，编译器会报错。而using指令导入整个命名空间中的所有成员的名称，包括那些可能根本用不到的名称，如果其中有名称与局部名称发生冲突，则编译器并不会发出任何警告信息，而只是用局部名去自动覆盖命名空间中的同名成员。特别是命名空间的开放性，使得一个命名空间的成员，可能分散在多个地方，程序员难以准确知道，别人到底为该命名空间添加了哪些名称。

虽然使用命名空间的方法，有多种可供选择。但是不能贪图方便，一味使用using 指令，这样就完全背离了设计命名空间的初衷，也失去了命名空间应该具有的防止名称冲突的功能。

一般情况下，对偶尔使用的命名空间成员，应该使用命名空间的作用域解析运算符来直接给名称定位。而对一

个大命名空间中的经常要使用的少数几个成员，提倡使用using声明，而不应该使用using编译指令。只有需要反复使用同一个命名空间的许多数成员时，使用using编译指令，才被认为是可取的。

例如，如果一个程序（如上面的outi.cpp）只使用一两次cout，而且也不使用std命名空间中的其他成员，则可以使用命名空间的作用域解析运算符来直接定位。如：

```
#include <iostream>

……

std::cout << "Hello, World!" << std::endl;

std::cout << "Outer::i = " << Outer::i << ",  Inner::i = " << Outer::Inner::i << std::endl;
```

又例如，如果一个程序要反复使用std命名空间中的cin、cout和cerr（如上面的outi.cpp），而不怎么使用其他std命名空间中的其他成员，则应该使用using 声明而不是using指令。如：

```
#include <iostream>

……

using std::cout;

cout << "Hello, World!" << endl;

cout << "Outer::i = " << Outer::i << ",  Inner::i = " << Outer::Inner::i << endl;
```

4）命名空间的名称

- **命名空间别名**

标准C++引入命名空间，主要是为了避免成员的名称冲突。若果用户都给自己的命名空间取简短的名称，那么

这些（往往同是全局级的）命名空间本身，也可能发生名称冲突。如果为了避免冲突，而为命名空间取很长的名称，则使用起来就会不方便。这是一个典型的两难问题。

标准C++为此提供了一种解决方案——命名空间别名，格式为：

namespace 别名 = 命名空间名;

例如：（AT&T美国电话电报公司）

```
namespace American_Telephone_and_Telegraph { // 命名空间名太长
    class String {
        String(const char*);
        // ……
    }
}
American_Telephone_and_Telegraph::String s1 // 使用不方便
    = new American_Telephone_and_Telegraph::String("Grieg");
namespace ATT = American_Telephone_and_Telegraph; // 定义别名
ATT::String s2 = new ATT::String("Bush"); // 使用方便
ATT::String s3 = new ATT::String("Nielsen");
```

- **无名命名空间**

标准C++引入命名空间，除了可以避免成员的名称发生冲突之外，还可以使代码保持局部性，从而保护代码不被他人非法使用。如果你的目的主要是后者，而且又为替命名空间取一个好听、有意义、且与别人的命名空间不重名的名称而烦恼的话，标准C++还允许你定义一个无名命名空间。你可以在当前编译单元中（无名命名空间之外），直接使用无名命名空间中的成员名称，但是在当前编译单元之外，它又是不可见的。

无名命名空间的定义格式为：

```
namespace {
    声明序列可选
}
```

实际上，上面的定义等价于：（标准C++中有一个隐含的使用指令）

```
namespace $$$ {
    声明序列可选
}
using namespace $$$;
```

例如：

```
namespace {
    int i;
```

```
    void f() {/*……*/}
}
int main() {
    i = 0; // 可直接使用无名命名空间中的成员i
    f(); // 可直接使用无名命名空间中的成员f()
}
```

上一篇 C++ 关键字 explicit, export, mutable
下一篇 C++ 关键字 typeid, typename

特权级3（调用门）

自己对杨一夫创业心得的感悟。

可恶的Java数组下标越界检查

C#正则表达式编程（四）：正则表达式

Android 进阶 - Activity服务启动分析

图像处理实用资源

查看评论

11楼 The_lastest 2014-08-24 15:30发表

写得不错，易懂，感觉比我看的书都写得棒一样。

10楼 jluhongfeng 2013-09-19 11:26发表

学习，领教，膜拜啦

9楼 timewalker08 2011-12-24 16:25发表

::v(::前面没有命名空间的名字)是引用全局命名空间的变量或函数对吧。
那假如：

```
int i = 0; //全局命名空间
void fun()
{
int i = 1;
cout << ::i << endl; //这里应该打印0对吧
}
```

8楼 XING2005112117 2011-04-13 10:16发表

请问，如果我要引用外来库（这个库有源码有解决方案），里面的类和MFC有冲突，用了同一个名字，我想用命名空间来区别，但是这个库写的时候没有显式的用namespace指定命名空间，我在网上看到默认的命名空间是项目名，怎么样才能在另外的项目里面用上它的命名空间里来使用里面的类呢，谢谢了！

7楼 smart84 2010-09-29 10:09发表

// out2.cpp

#include "out.h"

#include <iostream>

// using namespace Outer; // 编译错误，因为变量i和函数f()有名称冲突

using namespace Outer::Inner;

using namespace std;

//////////////////////////////////////

上面这段中,我在Dev-C++ 4.9.9中试了一下.编译不会出错的.
但在使用Outer::i,与Outer::Inner::i时,要明确指出就可以了.
也就是说没有冲突发生.

Re: firo_baidu 2011-09-01 16:54发表

回复smart84：恩，理应如此。

6楼 longshen123 2009-10-16 09:41发表

不错... 学习了

5楼 点墨 2009-08-03 16:55发表

不错。

4楼 111 2008-05-26 16:27发表

3楼强$$$是自己定义的名

3楼 postbox0782 2008-05-08 12:22发表

博主,你的这篇文章写的很棒,但我在VS2008中测试后,感觉有二处不对.
1/无论using namespace 还是 using ,好象都不报错,而是用局部的覆盖
2/using namespace $$$; , 没有这种用法
期待博主回复

Re: firo_baidu 2011-09-01 17:07发表

回复postbox0782：问题2，在C++ primer 中命名空间中有讲。。。
我有一个问题：为什么不能在函数外引用命名空间的成员：

```
#include <iostream>
namespace TEST
{
int test;
}
TEST::test=1;//Syntax err ,I really don't know why?
int main()
{
TEST::test=1;//Ok,no problem.
return 0;
}
```

Re: firo_baidu 2011-09-01 17:57发表

回复firo_baidu：是因为：全局域，不能执行赋值操作！！

2楼 boxxer 2008-04-10 20:51发表

确实是篇好文章，在几个地方都找到了这篇文章，但是其中都有乱码，原来这里才是最根本的出处。

1楼 ollydbg23 2007-12-20 15:02发表

这篇文章写的太棒了，基本上吧名字空间非常全面简练的总结了一下，而且写的通俗易懂，不错！加油！

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap