1. （p340）一个名字空间形成一个作用域，通常的作用域规则也适用于名字空间。因此，名字空间是一个非常基础、非常简单的概念。程序规模越大，用名字空间表达程序的逻辑划分就越有用。全局作用也也是一个名字空间，可以显示地用::来引用。例如：

int f(); //全局函数

int g()

{

int f; //**局部变量；屏蔽了全局函数**

f(); //错误：不能调用一个整型变量

::f(); //正确：调用全局函数

}

**类也是名字空间（见16.2节）**

1. using声明

当我们需要名字空间外频繁使用其名字时，反复使用名字空间进行显示限定很繁琐。

std::vector<std::string> splict(const std::string& s )

{

std::vector<std::string> res;

std::istringstream iss(s);

for(std::string buf; iss>>buf)

res.push\_buck(buf);

return res;

}

我们可以使用using声明来指出这段代码中string表示std::string:

using std::string; //用string表示std::string

std::vector<string> splict(conststring& s )

{

std::vector<string> res;

std::istringstream iss(s);

for(string buf; iss>>buf)

res.push\_buck(buf);

return res;

}

using声明将一个代用名引入了作用域，最好尽量保持代用名的局部性以避免混淆。

当用于一个重载的名字时，using声明会应用于其所有重载版本。例如：

namespace N {

void f(int);

void f(string);

};

void g()

{

using N::f;

f(789); //N::f(int)

f(“Bruce”); //N::f(string)

}

有关在类层次中使用using声明的内容，请见20.3.5节。

1. using指示（p341）

作为替代方法，我们可以使用using指示，要求编译器允许我们在所在作用域中无需使用限定符即可访问某个名字空间的所有名字，例如：

using namespace std; //令来自std的每个名字都可访问

vector<string> splict( const string& s )

{

vector<string> res;

istringstream iss(s);

for(string bug, iss<<buf;)

res.push\_back(buf);

return res;

}

（p342）除了极特殊情况外（例如为了帮助代码转换），不要在头文件中将一个using指示置于全局作用域中，因为你永远也不知道头文件可能在哪里被#include

1. 参数依赖查找（p342）

接受参数类型为用户自定义类型X的函数常常与X定义在相同的名字空间中。因此，如果在使用函数的上下文中找不到函数定义，我们可以在其参数的名字空间中查找它。例如：

namepace Chrono

{

class Date{ /\*…\*/ };

bool operator==( const Date&, const std::string&);

std::string format( const Date& ); //创建字符串

//…

}

void f( Chrono::Date d, int t )

{

std::string s = format( d ); //Chrono::format()

std::string t = format( I ); //错误：作用域中没有format()的定义

}

与使用显示显式限定相比，这种查找规则（称为参数依赖查找，argument-dependent lookup，或简称为ADL）使程序员可以省去很多输入工作，而且它还不像using指示（见14.2.3）那样会污染名字空间。它对于运算符对象（见18.2.5节）和模板参数（见26.3.5节）特别有用，对这些情况显式限定会非常繁琐。

注意，名字空间本身必须处于使用函数的作用不中，且函数必须在函数查找和使用之前。

1. （p343）当一个类成员调用一个命名函数时，编译器会优先选择同一个类的其他成员及其基类而不是基于参数类型查找得到的函数（运算符遵循不同的规则；参见18.2.1节和18.2.5节）。

namespace N

{

struct s { int i };

void f( S );

void g( S );

void h( int );

}

struct Base

{

void f( N::S );

}

struct D : Base

{

void mf();

void g( N::s x )

{

f( x ); //调用 Base::f()

mf( x ); //调用D::mf()

f( 1 ); //错误：没有可用的h( int )

}

}

在C++标准中，关于参数依赖查找的规则都有关联名字空间（associated namespace）的措辞（见iso.3.4.2）。基本上：

* 如果一个参数是一个类成员。关联名字空间即为类本身（包括其基类）和包含类的名字空间。
* 如果一个参数是一个名字空间的成员，则关联空间名字空间即为外层的名字空间。
* 如果一个参数是内置类型，则没有关联名字空间。

参数依赖查找可以帮助我们避免大量乏味、令人分心的代码输入工作，但偶尔也会带来意外的结果。例如在查找函数f()的声明时，并不优先调用所在的namespace中的函数（对于一个class中的f()调用，则会优先查找同一个类中的声明）：

namespace N

{

template<class T>

void f( T, int ); //N::f()

class X {};

}

namespace N2

{

N::X x;

void f( N::x, unsigned int );

void g()

{

f( x, 1 ); //调用N::f( X, int )

}

}

对于g()中的f()调用，选择N2::f()似乎很明显，但结果并不是这样。编译器会应用重载解析规则，并查找到最佳匹配：对于f( x, 1 ),N::f()是最佳匹配，因为对于参数1，类型int比之unsigned根伟匹配。也存在相反的例子，编译器选择了调用者名字空间的函数，但程序员期待的确实使用一个已知名字空间中更好的函数

1. 名字空间是开放的（p344）

名字空间是开放的；即，你可以从多个分离的名字空间中想一个名字空间添加名字。例如：

namespace A

{

int f(); //现在A包含成员f()

}

namespace A

{

int g(); //现在A有两个成员，f()和g()

}

1. （p349）显式限定、using声明和using指示间的权衡必须具体情况具体分析。基本原则是：
2. 如果多个名字确实有相同的限定，则对此名字空间使用using指示。
3. 若果名字空间中的特定名字经常使用某个限定，则在此名字使用using声明
4. 如果一个限定对某个名字来说并不常用，则在此名字出现的地方使用显式限定，使之更清晰。
5. 不要对与用户程序处于相同名字空间中的名字使用显式限定。
6. （p349）接口和名字

很明显，我们为Parser设计的名字空间定义非Parser呈现给用户的理想接口。取而代之的是，Parser设计了一组声明，以便能方便地编写各个语法分析器函数。呈献给用户的Parser接口则要简单的多：

namespace Parser

{ //用户接口

double expr( bool );

}

我们看到，Parser的名字空间做了两件事：

1. 为实现语法分析器的函数提供公共环境
2. 为用户提供语法分析器的外部接口

因此，驱动程序代码和main()函数看到的接口应该是我们确定能最好地表达这些函数的公共环境的接口。即：

namespace Parser

{ //实现者接口

double prim( bool );

double term( bool );

double expr( bool );

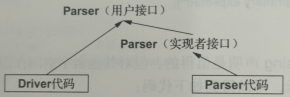
using namespace Lexer; //使用词法分析器提供的所有特征

using Error::error;

using Table::table;

}

接口和代码之间的关系可图示如下：



我们可以为用户接口和实现者接口起不同的名字，但（由于名字空间是开放的；见14.2.5节）不必这样。没有独立的名字不会导致混淆，因为程序的物理布局（见15.3.3节）自然地提供了独立的名字空间。假如我们界定使用一个独立的实现者名字空间，对于用户而言语法分析器的设计也没有什么不同：

namespace Parser

{ //用户接口

double expr( bool );

}

namespace Parsar\_impl

{ //实现者接口

using namespace Parser;

double prim( bool );

double term( bool );

double expr( bool );

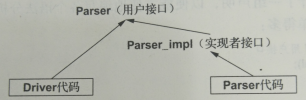
using namespace Lexer; //使用词法分析器提供的所有特征

using Error::error;

using Table::table;

}

接口和代码的关系可图示如下：



对于大规模程序，我倾向于引入\_impl接口。

1. （p351）组合使用名字空间

using声明将名字添加到局部作用域中，而using只是则不会，它只是简单地令名字在其所在作用域中可访问。例如：

namespace X

{

int i, j, k;

}

int k;

void f1()

{

int i = 0;

using namespace X; //令来自X的名字访问

i++; //局部i

j++; //X::j

k++; //错误：X的k还是全局的k？

::k++; //全局k

X::k++; //X的k

}

void f2()

{

int i = 0;

using X::i; //错误：i在f2()中声明了两次

using X::j;

using X::k; //隐藏了全局k

i++;

j++; //X::j

k++; //X::k

}

一个局部声明的名字（普通声明或用using声明）会隐藏同名的非局部声明，而且在声明点上改名字任何不合法的重载都会被检测出来。

1. 名字空间别名（p352）

如果用户为其名字空间起了一个较短的名字，不同名字空间的名字很可能冲突，但是长名字在实际代码中可能并不实用。通过为名字空间的长名字起一个别名，我们就可以解决这一两难境地：

namespace American\_Telephone\_and\_Telegraph

{ //太长

//…

}

namespace ATT = American\_Telephone\_and\_Telegraph;

ATT::String s3 = “Grieg”;

ATT::String s4 = “Nielsen”;

名字空间别名还允许用户引用“库”以及通过单一声明定义使用的具体是哪个库。例如：

namespace Lib = Foundation\_library\_v2r11;

//…

Lib::set s;

Lib::String s5 = “Sibelius”;

这极大地简化了替换库版本的任务。你可以使用Lib而不是使用Foundation\_library\_v2r11，这样就能通过修改别名Lib的初始化语句并重新编译来实现将库版本更新到“v3r02”。重新编译会捕获源码级的不兼容。但另一方面，过度使用（任何种类的）别名会导致混乱。

1. 组合名字空间(p352)

我们通常需要组合已有接口来构造新的接口。例如：

namespace His\_string

{

class String { /\*…\*/ };

String operator+( const String&, const String& );

String operator+( const String&, const char\* );

void fill( char );

//…

}

namespace Her\_vector

{

template<class T>

class Vector{ /\*…\*/ };

//…

}

namespace My\_lib

{

using namespace His\_string;

using namespace Her\_vector;

void my\_fct( String& );

}

有了如上的声明，我们就可以用My\_lib来编写程序了：

void f()

{

My\_lib::String s = “Byron”; //寻找My\_lib::His\_string::String

//…

}

using namespace My\_lib;

void g( Vector<String>& vs )

{

//…

my\_fct( vs[5] );

//…

}

如果显示一个显式限定的名字（如本例中的My\_lib::String）并未声明在所限定的名字空间中，编译器就会在using指示提及的名字空间（如His\_string）中寻找它。

只有当我们需要定义某些实体时，才真正的需要了解一个实体的真正名字空间。

void My\_lib::fill( char c ) { /\*…\*/ } //错误：My\_lib中并未声明fill()

void His\_string::fill( char c ){ /\*…\*/ } //正确：fill()在His\_string中声明

void My\_lib::my\_fct( String& v ){/…/}//正确：String为My\_lib::String，表示His\_string::String

理想情况下，一个名字空间应该：

1. 表达一组逻辑相关的特性；
2. 不会让用户访问不相关的特性；
3. 不会给用户增加符号表示上的严重负担

结合#include机制（见15.2.2节），本节和下一节中介绍的名字空间组合技巧可为实现这三点要求提供强有力的支持。

1. 组合与选择（p353）

组合机制（使用using指示）与选择机制（using声明）的结合满足了现实世界中大多数应用实例对灵活性的需求。使用这些机制，我们在访问各种特性时可解决他们的组合所引起的名字冲突和二义性。例如：

namespace His\_lib

{  
 class String{ /\*…\*/ };

template<class T>

class Vector{ /\*…\*/ };

//…

}

namespace Her\_lib

{

template<class T>

class Vector{ /\*…\*/ };

class String{ /\*…\*/ };

//…

}

namespace My\_lib

{

using namespace His\_lib; // His\_lib中所有的实体

using namespace Her\_lib; // Her\_lib中所有的实体

using His\_lib::String; //解决潜在冲突：使用His\_lib中的版本

using Her\_lib::Vector; //解决潜在冲突：使用Her\_lib中的版本

template<class T>

class List{ /\*…\*/ }; //其他内容

}

当编译器在一个名字空间中进行查找时，在其中显式地声明名字（包括使用using声明声明的名字）较之通过using指示变为可见的名字优先级更高（见14.4.1节）。因此，My\_lib的使用者会看到String和Vector的名字冲突顺利解决，分别使用了His\_String和Her\_lib::Vector。而List则迷人解析为My\_lib::List，而不管His\_lib或Her\_lib是否提供了List。

当我将一个名字空间纳入一个新的名字空间中时，我通常倾向于不改变它的名字空间。这样我就不必对同一个实体记忆两个不同的名字了。但有时起一个新的名字时必须的或者更好的。例如：

namespace Lib2

{

using namespace His\_lib; //His\_lib中所有的实体

using namespace Her\_lib; //Her\_lib中所有的实体

using His\_lib::String; //解决潜在冲突：使用His\_lib中的版本

using Her\_lib::Vector; //解决潜在冲突：使用Her\_lib中的版本

using Her\_string = Her\_lib::String; //重命名

template <class T>

using His\_vec = His\_lib::Vector<T>;

template<class T>

class List { /\*…\*/ }; //其他内容

//…

}

C++语言并未提供重命名的机制，但对于类型和模板，我们可以通过使用using引入别名来实现重命名（见3.4.5节和6.5节）