# Namespaces

## 14.2 Namespaces

A namespace is a (named) scope. You can access members defined earlier in a namespace from later declarations, but you cannot (without special effort) refer to members from outside the namespace. For example:

class Glyph { /\* ... \*/ };

class Line { /\* ... \*/ };

namespace Text\_lib {

class Glyph { /\* ... \*/ };

class Word { /\* ... \*/ }; // sequence of Glyphs

class Line { /\* ... \*/ }; // sequence of Words

class Text { /\* ... \*/ }; // sequence of Lines

File∗ open(const char∗); // open text file

Word operator+(const Line&, const Line&); // concatenate

}

Glyph glyph(Line& ln, int i); // ln[i]

Here, the Word and Line in the declaration of Te xt\_lib::operator+() refer to Te xt\_lib::Word and Text\_lib::Line. That local name lookup is not affected by the global Line. Conversely, the Glyph and Line in the declaration of the global glyph() refer to the global ::Glyph and ::Line. That (nonlocal) lookup is not affected by Te xt\_lib’s Glyph and Line.

To refer to members of a namespace, we can use its fully qualified name. For example, if we want a glyph() that uses definitions from Te xt\_lib, we can write:

Text\_lib::Glyph glyph(Text\_lib::Line& ln, int i); // ln[i]

Other ways of referring to members from outside their namespace are using-declarations (§14.2.2),

using-directives (§14.2.3), and argument-dependent lookup (§14.2.4).

### 14.2.1 Explicit Qualification

A member can be declared within a namespace definition and defined later using the *namespace-name* **::** *member-name* notation.

Members of a namespace must be introduced using this notation:

**namespace namespace-name {**

**//** *declaration and definitions*

**}**

For example:

**namespace Parser {**

**double expr(bool); //** *declaration*

**double term(bool);**

**double prim(bool);**

**}**

**double val = Parser::expr(); //** *use*

**double Parser::expr(bool b) //** *definition*

**{**

**//** *...*

**}**

We cannot declare a new member of a namespace outside a namespace definition using the qualifier syntax (§iso.7.3.1.2). The idea is to catch errors such as misspellings and type mismatches, and also to make it reasonably easy to find all names in a namespace declaration. For example:

**void Parser::logical(bool); //** *error : no logical() in Parser*

**double Parser::trem(bool); //** *error : no trem() in Parser (misspelling)*

**double Parser::prim(int); //** *error : Parser ::prim() takes a bool argument (wrong type)*

A namespace is a scope. The usual scope rules hold for namespaces. Thus, ‘‘namespace’’ is a very

fundamental and relatively simple concept. The larger a program is, the more useful namespaces

are to express logical separations of its parts. The global scope is a namespace and can be explicitly referred to using **::**. For example:

**int f(); //** *global function*

**int g()**

**{**

**int f; //** *local var iable; hides the global function*

**f(); //** *error : we can’t call an int*

**::f(); //** *OK: call the global function*

**}**

Classes are namespaces (§16.2)

### 14.2.2 using-Declarations

When a name is frequently used outside its namespace, it can be a bother to repeatedly qualify it with its namespace name. Consider:

**#include<string>**

**#include<vector>**

**#include<sstream>**

**std::vector<std::string> split(const std::string& s)**

**//** *split s into its whitespace-separated substrings*

**{**

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

**std::istringstream iss(s);**

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

**res.push\_back(buf);**

**return res;**

**}**

The repeated qualification **std** is tedious and distracting. In particular, we repeat **std::string** four times in this small example. To alleviate that we can use a **using**-declaration to say that in this code **string** means **std::string**:

**using std::string; //** *use ‘‘str ing’’ to mean ‘‘std::str ing’’*

**std::vector<string> split(const string& s)**

**//** *split s into its whitespace-separated substrings*

**{**

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

**std::istringstream iss(s);**

**for (string buf; iss>>buf;)**

**res.push\_back(buf);**

**return res;**

**}**

A **using**-declaration introduces a synonym into a scope. It is usually a good idea to keep local synonyms as local as possible to avoid confusion.

When used for an overloaded name, a **using**-declaration applies to all the overloaded versions. For example:

**namespace N {**

**void f(int);**

**void f(string);**

**};**

**void g()**

**{**

**using N::f;**

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

**f("Bruce"); //** *N::f(string)*

**}**

For the use of **using**-declarations within class hierarchies, see §20.3.5.

### 14.2.3 using-Directives

Alternatively, we can use a **using**-directive to request that every name from a namespace be accessible in our scope without qualification. For example:

**using namespace std; //** *make every name from std accessible*

**vector<string> split(const string& s)**

**//** *split s into its whitespace-separated substrings*

**{**

**vector<string> res;**

**istringstream iss(s);**

**for (string buf; iss>>buf;)**

**res.push\_back(buf);**

**return res;**

**}**

A **using**-directive makes names from a namespace available almost as if they had been declared outside their namespace (see also §14.4). Using a **using**-directive to make names from a frequently used and well-known library available without qualification is a popular technique for simplifying code. This is the technique used to access standard-library facilities throughout this book. The standard-library facilities are defined in namespace **std**.

Within a function, a **using**-directive can be safely used as a notational convenience, but care should be taken with global **using**-directives because overuse can lead to exactly the name clashes that namespaces were introduced to avoid. For example:

**namespace Graph\_lib {**

**class Shape { /\*** *...* **\*/ };**

**class Line : Shape { /\*** *...* **\*/ };**

**class Poly\_line: Shape { /\*** *...* **\*/ }; //** *connected sequence of lines*

**class Text : Shape { /\*** *...* **\*/ }; //** *text label*

**Shape operator+(const Shape&, const Shape&); //** *compose*

**Graph\_reader open(const char**∗**); //** *open file of Shapes*

**}**

**namespace Text\_lib {**

**class Glyph { /\*** *...* **\*/ };**

**class Word { /\*** *...* **\*/ }; //** *sequence of Glyphs*

**class Line { /\*** *...* **\*/ }; //** *sequence of Words*

**class Text { /\*** *...* **\*/ }; //** *sequence of Lines*

**File**∗ **open(const char**∗**); //** *open text file*

**Word operator+(const Line&, const Line&); //** *concatenate*

**}**

**using namespace Graph\_lib;**

**using namespace Text\_lib;**

**Glyph gl; //** *Te xt\_lib::Glyph*

**vector<Shape**∗**> vs; //** *Graph\_lib::Shape*

So far, so good. In particular, we can use names that do not clash, such as **Glyph** and **Shape**. Howev er, name clashes now occur as soon as we use one of the names that clash – exactly as if we had not used namespaces. For example:

**Te xt txt; //** *error : ambiguous*

**File**∗ **fp = open("my\_precious\_data"); //** *error : ambiguous*

Consequently, we must be careful with **using**-directives in the global scope. In particular, don’t place a **using**-directive in the global scope in a header file except in very specialized circumstances (e.g., to aid transition) because you never know where a header might be **#include**d.

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 );

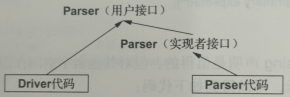
using namespace Lexer; //使用词法分析器提供的所有特征

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 );

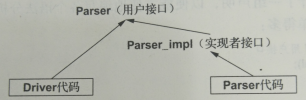
using namespace Lexer; //使用词法分析器提供的所有特征

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>; //重命名

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）

1. 版本控制(p356)

对很多类型的接口而言，最苛刻的测试就是应对一系列的新版本。考虑一个广为使用的接口。如一个ISO C++标准头文件。经过一段时间，标准委员会会定义新的版本，例如C++98头文件的C++11版本。新版本可能增加了函数、重命令了类、删除了私有扩展（本不该包含的内容）、改变类型、修改了模板。现实应用中可能有书一行代码使用了旧版本头文件，而新版本的实现者不可能看到或者修改这些代码，这就给实现者的生活增加了很多“乐趣”。不消说，破坏这些旧代码会引起强烈的不满，后果与不能提供更好的新版本一样严重。除少数情况外，到目前为止已介绍的名字空间特性已能处理这类问题，但当涉及的代码量非常庞大是，“少数情况”仍意味着大量代码。为此，C++提供了一种在两个版本间进行选择的机制，可以简单明确地保证用户看到其中一个特定版本，这就是内联名字空间（inline namespace）：

namespace Popular

{

inline namespace V3\_2

{ //V3\_2提供了Popular的默认含义

double f( double );

int f( int );

template<class T>

class C{ /\*…\*/ };

}

namespace V3\_0

{

//…

}

namespace V2\_4\_2

{

double f( double );

template<class T>

class C{ /\*…\*/ };

}

}

在本例中，Popular包含三个名字空间，每一个都定义了一个Popular版本。inline指出V3\_2是Popular的默认含义。因此我们可以编写如下代码：

using namespace Popular；

void f()

{

f(1); //Popular::V3\_2::f(int)

V3\_0::f(1); //Popular::V3\_0::f(double)

V2\_4\_2::f(1); ////Popular:: V2\_4\_2::f(double)

}

template<class T>

Popular::C<T>{ /\*…\*/ }

这种inline namespace方法是侵入式的。即，为了改变默认版本（子名字空间），必须修改头文件源码。而且，简单地使用这种方法处理版本问题需要复制大量代码（不同版本中的共同代码）。但是，使用#include技巧可以将赋值降低到最低。例如：

//文件 V3\_common.h：

//大量声明…

//文件V3\_2.h中

namespace V3\_2 //V3\_2提供了Popular的默认语义

{

double f( double );

int f( int );

template<class T>

class C{ /\*…\*/ };

#include”V3\_common”

}

//文件V3\_0.h：

namespace V3\_0

{

#include”V3\_common”

}

//文件Popular.h:

namespace Popular

{

inline

#include “V3\_2.h”

#include “V3\_0.h”

#include “V2\_4\_2.h”

}

我不推荐复杂地使用头文件，除非真的必要。这个例子多次违反了不要包含头文件到非局部作用局的原则和语法构造不要垮文件边界（使用inline）的原则，请参考15.2.2节。

大多数情况下，我们可以用侵入性不太强的方法来实现版本控制。我能想到的唯一一个完全不可能用其他方法实现的例子是显式使用名字空间的模板（如，Popular::C<T>）的特例化。但是，很多时候“大多数情况可解决”并不足够好。而且，组合多种技术而得的方案是否完全正确就不那么显然了。

1. 名字空间嵌套（p358）

void h();

namespace X

{

void g();

//…

namespace Y

{

void f();

void ff();

}

//…

}

对这段代码会应用通常的作用域和限定规则：

void X::Y::ff()

{

f(); g(); h();

}

void X::g()

{

f() //错误：X中无f()

Y::f(); //正确

}

void h()

{

f(); //错误：无全局f()

Y::f(); //错误：无全局Y

X::f(); //错误：X中无f()

X::Y::f(); //正确

}

关于标准库中的名字空间嵌套的例子，请见chrono（35.2节）和rel\_ops（见35.5.3节）。

1. 无名名字空间

有时将一组声明封装在一个名字空间中只是为了防止名字冲突，即，目的是保持代码的局部性而非为用户提供一个借口。例如：

#include “header.h”

namespace Mine

{

int a;

void f(){}

int g() {}

}

由于我们不希望在局部环境之外的代码看到名字Mine，那么创建这么一个局部名字就变成了一个麻烦，可能意外地与其他名字冲突。既然如此，我们可以简单地为名字空间命名：

#include “header.h”

namespace

{

int a;

void f(){}

int g() {}

}

显然，必须提供某种方法实现从无名名字空间之外访问其成员。为此，每个无名名字空间都有一个隐含的using指示。前面这个声明等价于：

namespace $$$

{

int a;

void f(){}

int g() {}

}

using namespace $$$;

其中$$$是此名字空间所在作用域中的一个独一无二的名字。特别是，不同编译单元中的无名名字空间是不同的。如我们所期望，我们无法从另一个编译单元中为一个无名字空间的成员命名。

建议：不要将using指示放在头文件中；14.2.3节