第 18 章　创建和使用名称空间

名称空间可用于帮助组织类，更重要的是，可帮助程序员在使用多个库时避免名称冲突。

本章介绍以下内容：

- 如何根据名称解析函数和类？
- 如何创建名称空间？
- 如何使用名称空间？
- 如何使用标准名称空间 std？

18.1　简　　介

名称冲突一度让 C 和 C++开发人员深受困扰。当程序的两个部分（最常见的情况是在不同的库软件包）中出现作用域相同的同名变量时，将发生名称冲突。例如，容器类库很可能声明和实现了 List 类（第 19 章讨论模板时，将更详细地介绍容器类），而在窗口库中也可能有 List 类。假设要为应用程序维护一系列窗口，同时要使用容器类库中的 List 类。你创建一个窗口库的 List 实例来存储窗口，却发现调用的成员函数不可用。这是因为编译器为你创建的是标准库中的容器 List，而你的本意是要创建厂商特定的窗口库中的 List。

名称空间用于减少名称冲突。名称空间在某些方面与类相似，其语法与类极其相似。

在名称空间中声明的东西归名称空间所有。名称空间中的所有内容都是公有的，名称空间还可以嵌套。函数可在名称空间体内定义，也可在名称空间体外定义。如果函数是在名称空间体外定义的，必须在程序中将名称空间导入其全局名称空间中，否则调用函数时必须用名称空间进行全限定。

18.2　根据名称解析函数和类

在编译器分析源代码并建立函数和变量名列表时，也检查名称冲突。那些不能由编译器解决的冲突将留给链接器去解决。

编译器不能检查跨越目标文件或编译单元之间的名称冲突，这是链接器的工作。对于这类情况，编译器甚至不会发出警告。

链接器链接失败并显示下述消息的情况屡见不鲜：

```
identifier multiply defined
```

其中 identifier 是一种被命名的类型。如果在不同的编译单元中定义了作用域相同的相同名称，将看到这种链接器消息。如果在同一个文件重复定义了作用域相同的名称，将导致编译错误。将程序清单 18.1a 和 18.1b 分别进行编译并链接到一起时，链接器将显示错误消息。

程序清单 18.1a　第一个定义 integerValue 的程序清单

```
0: // file first.cpp
1: int integerValue = 0 ;
```

```
2:  int main() {
3:      int integerValue = 0 ;
4:      // . . .
5:      return 0 ;
6:  } ;
```

程序清单 18.1b　第二个定义 integerValue 的程序清单

```
0:  // file second.cpp
1:  int integerValue = 0 ;
2:  // end of second.cpp
```

分析：

笔者的链接器给出如下诊断结果：

```
in second.obj:  integerValue already defined in first.obj。
```

如果这些名称的作用域不同，编译器和链接器将不会显示错误消息。

编译器还可能发出有关标识符隐藏的警告。编译器应该发出警告，指出在 first.cpp（程序清单 18.1a）中，main( )函数中的 integerValue 隐藏了同名的全局变量。

要使用在 main( )外面声明的 integerValue，必须显式地指定要使用的作用域为全局的哪个变量。请看程序清单 18.2a 和 18.2b，它将 10 赋给 main( )外面的 integerValue，而不是在 main( )中声明的 integerValue。

程序清单 18.2a　演示标识符隐藏的第一个程序清单

```
0:  // file first.cpp
1:  int integerValue = 0 ;
2:  int main()
3:  {
4:      int integerValue = 0 ;
5:      ::integerValue = 10 ; //assign to global "integerValue"
6:      // . . .
7:      return 0 ;
8:  } ;
```

程序清单 18.2b　演示标识符隐藏的第二个程序清单

```
0:  // file second.cpp
1:  int integerValue = 0 ;
2:  // end of second.cpp
```

注意：请注意作用域解析运算符::的用法，它表明使用的是全局（而不是局部）变量 integerVaule。

分析：

在函数外部定义的两个全局 int 变量的问题在于，它们的名称和可见性相同，因此将导致链接错误。

18.2.1　变量的可见性

术语可见性（visibility）指的是对象（无论它是变量、类还是函数）的作用域。虽然第 5 章介绍过，这里有必要简要地复习一下。

例如，在函数外部声明和定义的变量的作用域为整个文件（全局），该变量的可见性为其定义位置到文件末尾。在代码块中定义的变量的作用域为代码块（局部），最常见的例子是在函数中定义的变量。程序清单 18.3 说明了变量的作用域。

程序清单 18.3　变量的作用域

```
0:  // Listing 18.3
```

```
 1: int globalScopeInt = 5 ;
 2: void f()
 3: {
 4:     int localScopeInt = 10 ;
 5: }
 6: int main()
 7: {
 8:     int localScopeInt = 15 ;
 9:     {
10:         int anotherLocal = 20 ;
11:         int localScopeInt = 30 ;
12:     }
13:     return 0 ;
14: }
```

分析：

第 1 个 int 变量 globalScopeInt（第 1 行）在函数 f( )和 main( )中可见。下一个 int 变量 localScopeInt（第 4 行）在函数 f( )中可见，该变量的作用域为局部，这意味着它只在定义它的代码块内可见。

在函数 main( )中，不能访问 f( )中定义的 localScopeInt。函数 f( )返回时，localScopInt 将不再在作用域中。第三个 int 变量名也为 localScopeInt，是在 main( )函数（第 8 行）中定义的，该变量的作用域为代码块。

注意，在 main( )中定义的 localScopeInt 不会与 f( )中定义的 localScopInt 发生冲突。接下来的两个 int 变量（anotherLocal 和 localScopInt，第 10 和 11 行）的作用域都为代码块。到达右大括号后（第 12 行），这两个变量就将不再可见。

注意，该 localScopeInt 隐藏了左大括号前定义的 localScopeInt（程序中定义的第二个 localScopeInt）。到达右大括号后，第二个 localScopeInt 将可见。对在大括号内定义的 localScopInt 任何修改都不会影响外部的 localScopeInt。

18.2.2 链接性

名称的链接性可以为内部（internal）或外部（external）。这两个术语指的是名称在多个编译单元中还是一个编译单元中可用。链接性为内部的名称只在定义它的编译单元中可用。例如，链接性为内部的变量可被同一个编译单元中的不同函数共享。链接性为外部的名称对其他编译单元来说也是可用的。程序清单 18.4a 和 18.4b 说明了内部链接性和外部链接性。

程序清单 18.4a　内部链接性和外部链接性

```
0: // file: first.cpp
1: int externalInt = 5 ;
2: const int j = 10 ;
3: int main()
4: {
5:     return 0 ;
6: }
```

程序清单 18.4b　内部链接性和外部链接性

```
0: // file: second.cpp
1: extern int externalInt ;
2: int anExternalInt = 10 ;
3: const int j = 10 ;
```

分析：

在 first.cpp（程序清单 18.4a）中，第 1 行定义的 externalInt 变量的链接性为外部。虽然它是在 first.cpp

中定义的，但在 second.cpp 中也可以访问它。在两个文件定义的两个 j 为 const，const 变量的链接性默认为内部。可以显式声明 const 变量的链接性以覆盖其默认链接性，如程序清单 18.5a 和 18.5b 所示。

程序清单 18.5a　覆盖 const 变量的默认链接性

```
0:  // file: first.cpp
1:  extern const int j = 10 ;
```

程序清单 18.5b　覆盖 const 变量的默认链接性

```
0:  // file: second.cpp
1:  extern const int j ;
2:  #include <iostream>
3:  int main()
4:  {
5:      std::cout << "j is " << j << std::endl ;
6:      return 0 ;
7:  }
```

第 5 行调用 cout 时，指定了名称空间 std。编译并运行这个范例时，将显示如下内容：

```
j is 10
```

18.2.3　静态全局变量

标准委员会建议不要使用静态全局变量。声明静态全局变量的方法如下：

```
static int staticInt = 10 ;
int main()
{
      //...
}
```

建议不使用 static 来限制外部变量的作用域，将来这样做可能是非法的。应使用名称空间而不是 static。当然，要使用名称空间，需要知道如何创建。

应该：

应使用名称空间而不是静态全局变量

不应该：

不要将作用域为整个文件的变量使用关键字 static。

18.3　创建名称空间

名称空间的声明语法类似于结构和类：首先使用关键字 namespace，然后是可选的名称空间名和左大括号。名称空间声明以右大括号结束，但没有分号。

例如：

```
namespace Window
{
      void move( int x, int y) ;

      class List
      {
            // ...
```

```
    }
}
```

名称 Window 惟一地标识了该名称空间。对于已命名的名称空间，可以有多个实例。这些实例可以位于同一个文件中，也可以位于多个编译单元中，编译器将把它们合并成一个名称空间。C++标准库名称空间 std 是具有这种特征的典范，这是合理的，因为标准库是一个逻辑功能组，但它过于庞大和复杂，无法放在单个文件中。

名称空间的基本思想是，将相关的项目组合到一个特定的（已命名）区域。程序清单 18.6a 和 18.6b 是一个跨越多个头文件的简单名称空间。

程序清单 18.6a　将相关的项目组合到一起

```
0: // header1.h
1: namespace Window
2: {
3:     void move( int x, int y) ;
4: }
```

程序清单 18.6b　将相关的项目组合到一起

```
0: // header2.h
1: namespace Window
2: {
3:     void resize( int x, int y ) ;
4: }
```

分析：

正如读者看到的，名称空间 Window 分布在两个头文件中。编译器将函数 move( )和 resize( )都视为名称空间 Window 的组成部分。

18.3.1　声明和定义类型

在名称空间中，可以声明和定义类型和函数。当然，这是一个设计和维护问题。优秀的设计要求将接口和实现分开。不仅对于类应遵循这条原则，对于名称空间也应如此。下面是一个凌乱、糟糕的名称空间：

```
namespace Window {
        // . . . other declarations and variable definitions.
        void move( int x, int y) ; // declarations
        void resize( int x, int y ) ;
        // . . . other declarations and variable definitions.

        void move( int x, int y )
        {
              if( x < MAX_SCREEN_X  && x > 0 )
                    if( y < MAX_SCREEN_Y  && y > 0 )
                          platform.move( x, y ) ; // specific routine
        }

        void resize( int x, int y )
        {
              if( x < MAX_SIZE_X  && x > 0 )
                    if( y < MAX_SIZE_Y  && y > 0 )
                          platform.resize( x, y ) ; // specific routine
        }
        // . . . definitions continue
```

```
}
```

从中可知，名称空间很快就变得凌乱不堪！这个例子只有大约 20 行；可以想见，如果该名称空间增长 4 倍，将会是什么样的。

18.3.2　在名称空间外定义函数

应在名称空间体外定义名称空间中的函数。这样做可以将函数的声明和定义分开，并避免名称空间体凌乱不堪。通过将函数定义同名称空间分离，可以将名称空间及其包含的声明放在头文件中，而将定义放在实现文件中。程序清单 18.7a 和 18.7b 演示了这种分离。

程序清单 18.7a　在名称空间中声明函数头

```
0:  // file header.h
1:  namespace Window {
2:      void move( int x, int y) ;
3:      // other declarations ...
4:  }
```

程序清单 18.7b　将实现放在源代码文件中

```
0:  // file impl.cpp
1:  void Window::move( int x, int y )
2:  {
3:      // code to move the window
4:  }
```

18.3.3　添加新成员

要添加新成员，只能在名称空间体内进行；不能使用限定符语法来定义新成员。采用这种定义方式将导致编译错误。下面的例子说明了这种错误：

```
namespace Window
{
      // lots of declarations
}
//...some code
int Window::newIntegerInNamespace ; // sorry, can't do this
```

最后一行代码是非法的，编译器将显示一条错误消息。要更正这种错误，将声明移到称空间体内即可。

添加新成员时，不应指定访问限定符（如 public 或 private），名称空间中的所有成员都是公有的。下面的代码不能通过编译，因为不能将名称空间成员指定为私有的：

```
namespace Window
{
      private:
            void move( int x, int y ) ;
}
```

18.3.4　嵌套名称空间

可以将名称空间嵌套在另一个名称空间中，原因是名称空间的定义也是声明。要访问被嵌套的名称空间，必须使用包含它的名称空间来限定。如果嵌套了名称空间，必须依次限定每个名称空间。例如，下面的代码将一个已命名的名称空间嵌套在另一个名称空间中：

```
namespace Window
{
      namespace Pane
```

```
    {
        void size( int x, int y ) ;
    }
}
```

要在名称空间 Window 外部访问函数 size( )，必须用这两个名称空间的名称限定这个函数，如下所示：

```
Window::Pane::size( 10, 20 ) ;
```

18.4 使用名称空间

来看一个使用名称空间的例子以及作用域解析运算符的相关用法。在这个例子中，首先在名称空间 Window 中声明了所有需要的类型和函数，然后定义了已声明的成员函数。这些成员函数都是在名称空间外定义的，并使用作用域解析运算符显式地标识了函数名。程序清单 18.8 演示了如何使用名称空间。

程序清单 18.8 使用名称空间

```
 0: #include <iostream>
 1: // Using a Namespace
 2:
 3: namespace Window
 4: {
 5:    const int MAX_X = 30 ;
 6:    const int MAX_Y = 40 ;
 7:    class Pane
 8:    {
 9:       public:
10:          Pane() ;
11:          ~Pane() ;
12:          void size( int x, int y ) ;
13:          void move( int x, int y ) ;
14:          void show() ;
15:       private:
16:          static int count ;
17:          int x ;
18:          int y ;
19:    };
20: }
21:
22: int Window::Pane::count = 0 ;
23: Window::Pane::Pane() : x(0), y(0) { }
24: Window::Pane::~Pane() { }
25:
26: void Window::Pane::size( int x, int y )
27: {
28:    if( x < Window::MAX_X  &&  x > 0 )
29:       Pane::x = x ;
30:    if( y < Window::MAX_Y  &&  y > 0 )
31:       Pane::y = y ;
32: }
33: void Window::Pane::move( int x, int y )
34: {
35:    if( x < Window::MAX_X  &&  x > 0 )
36:       Pane::x = x ;
```

```
37:     if( y < Window::MAX_Y  &&  y > 0 )
38:         Pane::y = y ;
39: }
40: void Window::Pane::show()
41: {
42:     std::cout << "x " << Pane::x ;
43:     std::cout << " y " << Pane::y << std::endl ;
44: }
45:
46: int main()
47: {
48:     Window::Pane pane ;
49:
50:     pane.move( 20, 20 ) ;
51:     pane.show() ;
52:
53:     return 0 ;
54: }
```

输出：

```
x 20 y 20
```

分析：

第 7～19 行声明的类 Pane 被嵌套在第 3～20 行的名称空间 Window 中，因此必须使用 Window::来限定名称 Pane。

第 16 行在 Pane 中声明的静态变量 count 像通常那样被定义。在第 26～32 行的函数 Pane::size( )中，MAX_X 和 MAX_Y 都被全限定，这是因为 Pane 在作用域内；如果不这样做，编译器将显示错误消息。这一点也适用于函数 Pane::move( )。

另外有趣的一点是，在两个函数定义内都使用了 Pane::x 和 Pane::y。这是为什么呢？如果将函数 Pane::move( )写成下面这样将遇到麻烦：

```
void Window::Pane::move( int x, int y )
{
      if( x < Window::MAX_X  &&  x > 0 )
          x = x ;
      if( y < Window::MAX_Y  &&  y > 0 )
          y = y ;
      Platform::move( x, y ) ;
}
```

你知道问题所在吗？编译器可能不会给出答案，有些编译器甚至根本不会显示错误消息。

问题的根源在于函数的参数。参数 x 和 y 隐藏了在 Pane 类中声明的私有实例变量 x 和 y。下面的语句将 x 和 y 分别赋给它们自身：

```
x = x;
y = y;
```

18.5　关键字 using

关键字 using 用于 using 编译指令和 using 声明。关键字 using 的语法决定是编译指令还是声明。

18.5.1　using 编译指令

using 编译指令将名称空间中声明的所有名称导入到当前作用域中，这样引用这些名称时无需用其所属的

名称空间进行限定。下面的例子演示了 using 编译指令的用法：

```
namespace Window
{
    int value1 = 20 ;
    int value2 = 40 ;
}
. . .
Window::value1 = 10 ;

using namespace Window ;
value2 = 30 ;
```

using 编译指令的作用域从其声明之处开始到当前作用域结束。注意，引用 value1 时必须限定它，但引用 value2 时不需要限定，因为编译指令将名称空间中的所有名称导入到当前作用域中。

using 编译指令可用于任何作用域级别中。这让你能够在代码块作用域中使用它，离开该代码块后，名称空间中所有的名称将不在作用域中。下面的例子说明了这一点：

```
namespace Window
{
    int value1 = 20 ;
    int value2 = 40 ;
}
//. . .
void f()
{
    {
       using namespace Window ;
       value2 = 30 ;
    }
    value2 = 20 ; //error!
}
```

f()中的最后一行代码（value2 = 20;）是一种错误，因为 value2 没有定义。该名称在前一个代码块中是可用的，因为 using 编译指令将其导入到了该代码块中。离开该代码块后，名称空间 Window 中的名称便不在作用域中。要使该行代码合法，必须对 value2 进行全限定：

```
Window::value2 = 20 ;
```

在局部作用域中声明的变量将隐藏被导入到该作用域的名称空间中的同名变量。这类似于局部变量将隐藏同名的全局变量。即使在局部变量后导入名称空间，该局部变量也将隐藏名称空间中的同名变量。请看下面的例子：

```
namespace Window
{
    int value1 = 20 ;
    int value2 = 10 ;
}
//. . .
void f()
{
    int value2 = 10 ;
    using namespace Window ;
    std::cout << value2 << std::endl ;
}
```

该函数的输出为 10 而不是 40。名称空间 Window 中的 value2 被 f()中的 value2 隐藏了。要使用名称空间

中的名称，必须使用名称空间名来限定它。

使用被定义为全局又在名称空间中定义了的名称可能导致歧义。这种歧义只在使用了该名称时才浮出水面，仅仅导入名称空间不会导致歧义。下面的代码段说明了这一点：

```
namespace Window
{
    int value1 = 20 ;
}
//. . .
using namespace Window ;
int value1 = 10 ;
void f()
{
    value1 = 10 ;
}
```

歧义发生在函数 f()中。Using 编译指令将 Window::value1 导入到了全局名称空间中；由于定义了一个全局变量 value1，因此在 f()中使用 value1 将导致错误。如果删除 f()中的这行代码便不会出现错误。

18.5.2 using 声明

using 声明类似于 using 编译指令，只是 using 声明提供更细致的控制。具体地说，using 声明用于将名称空间中的一个名称导入到当前作用域中，然后便可以只使用其名称来引用该对象。下面的例子演示了 using 声明的用法：

```
namespace Window
{
    int value1 = 20 ;
    int value2 = 40 ;
    int value3 = 60 ;
}
//. . .
using Window::value2 ;    //bring value2 into current scope
Window::value1 = 10 ;     //value1 must be qualified
value2 = 30 ;
Window::value3 = 10 ;     //value3 must be qualified
```

using 声明将指定的名称添加到当前作用域中，这种声明不影响名称空间中的其他名称。在前面的例子中，引用 value2 时不需要限定，但引用 value1 和 value3 需要。使用 using 声明可以更细致地控制将名称空间中的哪些名称导入到当前作用域中；而 using 编译指令将名称空间中所有的名称都导入到当前作用域中。

将名称导入作用域中后，该名称将一直可见，直到到达当前作用域末尾，这与其他声明相同。using 声明可用于全局名称空间中或任何局部作用域中。

将名称空间中的名称导入到已经声明了同名变量的局部作用域中将导致错误，反之亦然。下面的例子说明了这一点：

```
namespace Window
{
    int value1 = 20 ;
    int value2 = 40 ;
}
//. . .
void f()
{
    int value2 = 10 ;
```

```
    using Window::value2 ; // multiple declaration
    std::cout << value2 << std::endl  ;
}
```

f()函数中的第二行代码将导致编译错误，因为名称 vaule2 已经定义了。如果在声明局部变量 value2 之前，使用 using 声明导入该名称也将导致同样的错误。

使用 using 声明导入到局部作用域中的名称将隐藏该作用域外声明的其他同名变量，下面的代码段说明了这一点：

```
namespace Window
{
    int value1 = 20 ;
    int value2 = 40 ;
}
int value2 = 10 ;
//. . .
void f()
{
    using Window::value2 ;
    std::cout << value2 << std::endl  ;
}
```

f()中的 using 声明隐藏了在全局名称空间中定义的 value2。

正如前面指出的，using 声明让你能够更细致地控制名称空间中名称的导入。using 编译指令将名称空间中所有的名称都导入到当前作用域中。using 声明优于 using 编译指令，因为 using 编译指令违背名称空间机制的初衷。声明更明确，因为它明确地指定了要导入到作用域中的名称。using 声明不会污染全局名称空间，而 using 编译指令会（当然，除非要导入名称空间中所有的名称）。使用 using 声明可将名称隐藏、全局名称空间污染和歧义降低到易于控制的水平。

18.6 名称空间别名

名称空间别名用于给已命名的名称空间提供另一个名称。别名提供了引用名称空间的简捷方式。尤其是当名称空间名很长时，创建别名可避免冗长的重复输入。请看下面的代码：

```
namespace the_software_company
{
    int value ;
    // . . .
}
the_software_company::value = 10 ;
. . .
namespace TSC = the_software_company ;
TSC::value = 20 ;
```

当然，这样做的一个缺点是，别名可能与已有的名称冲突。在这种情况下，编译器将发现这种冲突，你可以通过给别名重命名来解决这种冲突。

18.7 未命名的名称空间

未命名的名称空间是没有名称的名称空间。未命名空间的一种常见用途是，防止目标文件或其他编译单

元中的全局数据发生名称冲突。每个编译单元都有独一无二的未命名名称空间。在未命名的名称空间（每个编译单元一个）中定义的所有名称都可以不加显式限定的情况下进行引用。程序清单 18.9a 和 18.9b 提供了一个这样的例子：在两个独立文件中包含两个未命名的名称空间。

程序清单 18.9a　一个未命名的名称空间

```
0:  // file: one.cpp
1:  namespace
2:  {
3:      int value ;
4:      char p( char *p ) ;
5:      //. . .
6:  }
```

程序清单 18.9b　另一个未命名的名称空间

```
0:  // file: two.cpp
1:  namespace
2:  {
3:      int value ;
4:      char p( char *p ) ;
5:      //. . .
6:  }
7:  int main()
8:  {
9:      char c = p( char * ptr ) ;
10: }
```

分析：

将这两个程序清单编译成一个可执行文件时，每个文件中的名称、值和函数 p()都是不同的。在编译单元中引用（未命名的名称空间中的）名称时，不用进行限定。在上面的例子中，通过在每个文件中调用函数 p()演示了这种用法。

这种用法隐含着包含一条导入未命名的名称空间的 using 编译指令。因此，不能访问另一个编译单元中未命名的名称空间中的成员。

未命名的名称空间的行为与链接性为外部的静态对象相同。请看下面的例子：

```
static int value=10;
```

别忘了，标准委员会已经摒弃了关键字 static 的这种用法。现在有名称空间，可用来替换上述静态声明代码。另一种看待未命名名称空间的方法是，将其视为链接性为内部的全局变量。

18.8　标准名称空间 std

名称空间的典范是 C++标准库。标准库全部位于名为 std 的名称空间中。所有的函数、类、对象和模板都是在名称空间 std 中声明的。

读者见过下面这样的代码：

```
#include <iostream>
using namespace std;
```

以这种方式使用 using 编译指令时，将导入指定名称空间中的所有名称。

使用 using 编译指令来导入标准库是一种糟糕的做法。为什么呢？因为这样做将导入头文件中的所有名称，污染了应用程序的全局名称空间。请切记，所有的头文件都使用名称空间功能，如果包含多个标准头文

件并指定 using 编译指令，则头文件中声明的所有东西都将被导入到全局名称空间中。

读者可能注意到了，本书的大部分范例都违反了这种规则；这样做并非要提倡违背该规则，而是旨在简化范例。应使用 using 声明而不是 using 编译指令，如程序清单 18.10 所示。

程序清单 18.10　使用 std 名称空间的正确方式

```
 0:  #include <iostream>
 1:  using std::cin ;
 2:  using std::cout ;
 3:  using std::endl ;
 4:  int main()
 5:  {
 6:      int value = 0 ;
 7:      cout << "So, how many eggs did you say you wanted?" << endl ;
 8:      cin >> value ;
 9:      cout << value << " eggs, sunny-side up!" << endl ;
10:      return( 0 ) ;
11:  }
```

输出：

```
So, how many eggs did you say you wanted?
4
4 eggs, sunny-side up!
```

分析：

正如读者看到的，使用了名称空间中的 3 个名称，它们是在第 1～3 行声明的。

另一种方法是，对使用的名称进行权限定，如程序清单 18.11 所示。

程序清单 18.11　对名称进行全限定

```
0:  #include <iostream>
1:  int main()
2:  {
3:      int value = 0 ;
4:      std::cout << "How many eggs did you want?" << std::endl ;
5:      std::cin >> value ;
6:      std::cout << value << " eggs, sunny-side up!" << std::endl ;
7:      return( 0 ) ;
8:  }
```

输出：

```
How many eggs did you want?
4
4 eggs, sunny-side up!
```

分析：

对名称进行全限定的方法可能适合于小程序，对于代码量很大的程序可能非常繁琐。想象一下，必须在使用的每个标准库中的名称前加上前缀 std::有多繁琐！

18.9　小　　结

本章介绍了贯穿本书一直在使用的功能。

创建名称空间与声明类极其相似，但有两个不同的地方需要注意。首先，名称空间的右括号后没有分号；其次，名称空间是开放的，而类是封闭的。这意味可以在其他文件或同一文件的其他部分继续定义同一个名称空间。

可以声明的任何东西都可放在名称空间中。如果为可重用的库设计类，应使用名称空间功能。在名称空间中声明的函数应在名称空间体外定义，这样可将实现与接口分开并避免名称空间凌乱不堪。

using 编译指令用于将名称空间中所有的名称导入到当前作用域中，这相当于将指定的名称空间中所有的名称加入到全局名称空间中。通常使用 using 编译指令是一种糟糕的做法，尤其是对于标准库而言。相反，应使用 using 声明。

using 声明用于将名称空间中特定的名称导入到当前作用域中，这让你只需使用名称便可引用相应的对象。

从本质上说，名称空间别名类似于 typedef。名称空间别名让你能够为已命名的名称空间创建另一个名称。使用名称很长或嵌套的名称空间时，这很有用。

每个文件都可以包含一个未命名的名称空间。顾名思义，未命名的名称空间就是没有名称的名称空间。未命名的名称空间让你在使用其中的名称时无需进行限定，它使其中的名称的作用域为当前编译单元。未命名的名称空间与使用关键字 static 声明全局变量相似。

18.10　问　与　答

问：必须使用名称空间吗？

答：不，编写一些简单程序时完全可以忽略名称空间，但务必使用旧的标准库（如#include <string.h>）而不是新库（如#include <cstring>）。鉴于本章介绍过的原因，建议使用名称空间。

问：C++是惟一一种使用名称空间的语言吗？

答：不。其他语言也使用名称空间来组织和分开值，如 Visual Basic 7（.NET）和 C#等。还有一些语言有类似的概念，如 Java 中的包。

问：什么是未命名的名称空间？为什么需要未命名的名称空间？

答：未命名的名称空间是没有名称的名称空间，用于封装一系列声明以避免潜在的名称冲突。未命名的名称空间中的名称只能在声明该名称空间的编译单元中使用。

18.11　作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

18.11.1　测验

1. 如果函数 MyFunc( )位于名称空间 Inner 中，而后者又被嵌套在名称空间 Outer 中，如何访问该函数？
2. 请看下面的代码：

```
int x = 4;
int main()
{
    for( y = 1; y < 10; y++)
    {
        cout << y << ":" << endl;
        {
            int x = 10 * y;
```

```
                cout << "X = " << x << endl
            }
        }
        // *** HERE ***
    }
```

当程序执行到“HERE”处时，X 的值为多少？

3. 在不使用关键字 using 的情况下，能否使用在名称空间中定义的名称？
4. 常规名称空间和未命名名称空间之间的主要区别是什么？
5. 包含关键字 using 的语句有哪两种？它们之间的区别何在？
6. 什么是未命名的名称空间？为何需要这种名称空间？
7. 什么是标准名称空间？

18.11.2 练习

1. 查错：下面的程序有何错误？

```
0:  #include <iostream>
1:  int main()
2:  {
3:      cout << "Hello world!" << endl;
4:      return 0;
5:  }
```

2. 指出 3 种更正练习 1 中问题的方法。
3. 声明一个名为 MyStuff 的名称空间，该名称空间中包含一个名为 MyClass 的类。