第16章

名称空间

名称空间（namespace）是 C++ 在 ANSI/ISO 标准下新增加的功能。其目的在于避免大型链接库之间因为使用相同名称而造成冲突。

16.1 因为名称相同而造成的问题
16.2 名称空间的基本语法
16.3 名称空间成员的存取
16.4 使用“using 指令”和“using 声明”以存取名称空间的成员
16.5 标准名称空间
16.6 未命名的名称空间
16.7 常犯的错误
16.8 本章重点
16.9 本章练习

16.1 因为名称相同而造成的问题

在 6.4 节中我们曾经探讨过在 C++ 程序中，各种常量、变量和函数的适用范围（scope）和能够使用的期限（lifetime）两种特性。大家也许也注意到，在本书中，各种程序组件在命名时，我们通常采取

(1) 以大写英文字母开头，紧接着小写英文字母或数字。

(2) 使用 5 个字符以下的简短名称。

两个原则。这两个命名原则除了有简短方便的特性外，也可避免我们撰写的程序与通过 `#include` 预处理指令所包含的链接库之间发生名称冲突的问题。

虽然我们如此小心翼翼，也很难避免链接库之间发生名称冲突的问题。特别是开发大型程序时，常必须引用各种程序包库以提高开发的效率，这个时候，因为名称相同而造成问题的机会就会大增。

提示

绝大部分的程序包库的命名方式都有以下几个特征之一：

1. 全部大写或全部小写英文字母。
2. 常用下标线(underscore) '_' 作为开头或组成单字之间的分隔。〈例如 upper_limit，_oct。〉
3. 名称超过 5 个以上字符。
4. 以常用英文单字作为命名的根据。

以下我们以一个简单的例子来说明这类的问题：

图 16.1.1 中有三个文件，文件名分别是 First.h，Second.h 和 Test.cpp。在 First.h 和 Second.h 中，都定义了 double 常量 `Gain` 且声明了 struct 数据类型 `Member`。但是这两个同名的常量和同名的 struct 数据类型内容并不相同，我们想要在同一个程序中引用。

由于常量和 struct 数据类型的名称相同，我们一次只能选择一个头文件使用，否则在编译时就会发生重复声明 (multiple declaration) 的错误。在图 16.1.1 的例子中是以

```
#include "First.h"
```

来加入头文件 First.h ，而无法同时加入 Second.h。

这个问题限制了我们使用同名常量和同名类 (亦即 class，是面向对象语法的重要基础，将在第 18 章中介绍) 的机会，造成无法正确使用链接库 (library) 的问题。名称空间 (name space) 的设计就是针对这个问题而提出的新语法。

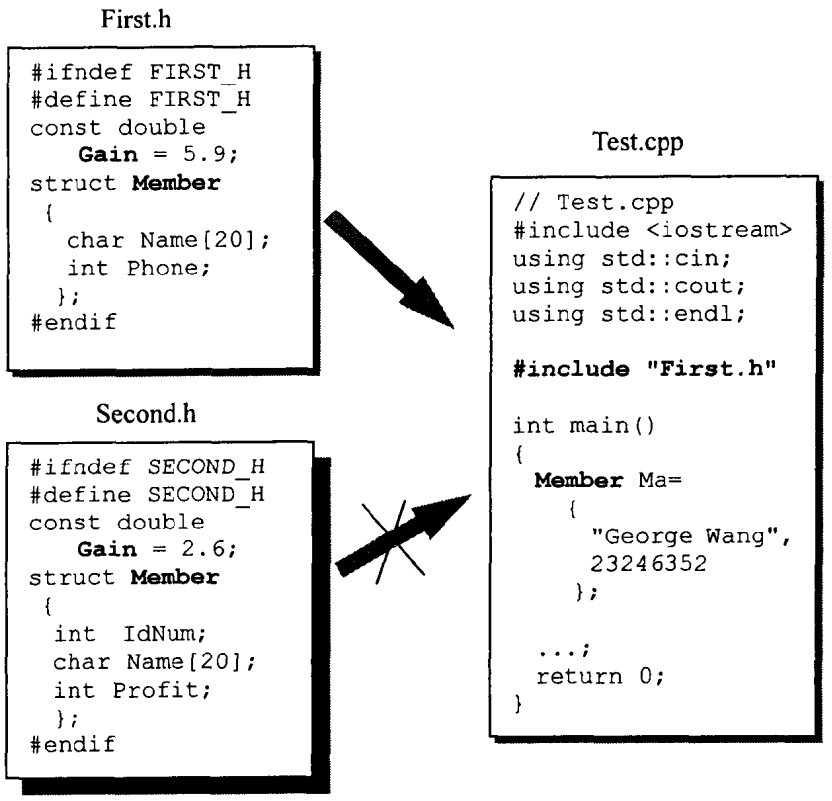

图 16.1.1　因为名称冲突而造成无法引用既有程序的问题

16.2 名称空间的基本语法

名称空间 (name space) 的语法和第 15 章 struct 的语法很类似，但其声明语句之后没有“;”作为结尾。例如：

```
namespace MyNamespace
{
  const double Gain = 5.9;
  struct Member
  {
    char Name[20];
    int  Phone;
  };
double Fnc(double);
}
```

上式声明了 MyNamespace 这个自定的名称空间，其中 namespace 是 C++ 的关键词，而大括号 {} 内包括了常量的定义以及函数和 struct 数据类型的声明 （函数的声明又称为函数的原型）。通常我们称这类的定义和声明为链接库的接口（interface of a library），使用者不需要拥有链接库的原始程序代码就可以知道有哪些重要的常量以及函数的功能和所需的参数是什么等等信息。

名称空间的声明可以用累加的方式完成，不需要全部写在同一次的声明内。以上面的例子而言，我们可以分两次声明，效果完全一样：

```
namespace MyNamespace
{
  const double Gain = 5.9;
}
namespace MyNamespace
{
  struct Member
  {
    char Name[20];
    int  Phone;
  };
double Fnc(double);
}
```

16.3　名称空间成员的存取

名称空间所含的各个项目称为该名称空间的成员（member）。要存取名称空间的成员时，必须要使用该名称空间的名称，加上范围确认运算符（scope resolution operator）“::”，再加上该成员的名称。 例如：

```
MyNamespace::Gain;
MyNamespace::Fnc();
```

名称空间的名称加上范围确认运算符统称为匹配修饰符(qualifier)。“MyNamespace::”就是一个典型的匹配修饰符。

如果要使用名称空间中声明的 struct 数据类型来定义实例（instance）时，则必须写成

```
MyNamespace::Member Ma = {"George Wang", 23246352};
```

也就是说，要在 struct 数据类型的名称前加上匹配修饰符。

嵌套名称空间

名称空间的成员也可以是另一个名称空间，形成嵌套的名称空间 (nested namespace)。例如：

```
namespace NS1
{
    int Size;
    namespace NS2
    {
        const double Ratio;
    }
}
```

嵌套名称空间的内部成员必须套用所有名称空间的名称。例如：

```
int x = NS1::Size;
double y = NS1::NS2::Ratio;
```

不过，嵌套名称空间的用途非常有限，在此不再深究。

名称空间的别名

对于超长的名称空间名称，可以使用别名(alias) 的方式简化。例如，有个名称很长的名称空间：

```
namespace Micro_Soft-Company_Namespace
{
  char Employer[2500][20];
  double Fnc(double);
}
```

可以使用名称空间别名语句 (namespace alias statement) 来为它取个比较精简的代号：

```
namespace MSC = Micro_Soft-Company_Namespace;
```

完成了上述声明之后，就可以使用较简短的方式存取其成员，例如：

```
char Head[20] = MSC::Employer[23];
```

对于嵌套名称空间，可以一次取代两层关系：

```
namespace NS12 = NS1::NS2;
```

此后成员的存取就可以精简成下列的语句：

```
double y = NS12::Ratio;
```

接口与实现的分割

由于名称空间可以分割的特性，我们常常将链接库区分成包括各种常量声明和函数原型的接口部分(interface part)，以及包括算法细节的各种函数定义的实现部分（implementation part)。

为了管理上的方便，我们通常进一步将庞大的链接库区分成数个文件。譬如说，可以将上述的“接口部分”储存成一个叫做 MyLib.h 的文件，而将“实现部分”另外储存成一个称为 MyLib.cpp 的文件。实现部分的写法有两种：

(1) 在名称空间主体之中定义函数

这个做法利用名称空间可以分割的特性，在名称空间之内定义函数。例如：

```
// MyLib.cpp
#include "MyLib.h"
```

```
namespace MyNamespace
{
 double Fnc(double x)
  {
      // 函数 Fnc() 的算法细节
  }
}
```

语法上虽然正确，但是不易把接口和实现二者作清楚的分割，不建议采用这种做法。

(2) 在名称空间主体之外定义函数

这种方式在名称空间的主体之外定义函数，让名称空间的主体更为简洁，是比较好的做法。例如：

```
// MyLib.cpp
#include "MyLib.h"
double MyNamespace::Fnc(double x)
  {
      // 函数 Fnc()的算法细节
  }
```

实例介绍

了解名称空间的语法后，让我们再回到 16.1 节的问题，看看名称空间如何解决名称冲突的问题．为了方便对照，我们仍然保留 16.1 节程序中 First.h，Second.h 和 Test.cpp 三个文件的名称：

在文件 First.h 和 Second.h 内，我们分别声明了 NS1 和 NS2 两个名称空间，将各文件内原有的 double 常量 Gain 和 struct 数据类型 Member 包括在其中．此外，我们在各名称空间中增加了一个名叫 Fnc() 的函数．

NS1 和 NS2 两个名称空间中各自的同名函数 Fnc() 的实现部分写在文件 FncLib.cpp 里面，以示范一般链接库的基本组成方式。

文件 Test.cpp 是主程序，在这里面，分别使用名称空间 NS1 和 NS2 中的各成员，包括常量 Gain，struct 数据类型 Member 和函数 Fnc()。

范例程序 文件 First.h

```
// First.h
#ifndef FIRST_H
#define FIRST_H
namespace NS1  // 声明名称空间 NS1
{
  const double Gain = 5.9;
  struct Member
   {
    char Name[20];
    int  Phone;
   };
  double Fnc(double);
}
#endif
```

范例程序 文件 Second.h

```
// Second.h
#ifndef SECOND_H
#define SECOND_H
namespace NS2     // 声明名称空间 NS2
{
  const double Gain = 2.6;
  struct Member
   {
    int     IdNum;
    char    Name[20];
    double  Profit;
   };
  double Fnc(double);
}
#endif
```

范例程序　文件 FncLib.cpp

```
// FncLib.cpp
#include "First.h"    // 引入 NS1 的声明
#include "Second.h"   // 引入 NS2 的声明
// 定义函数 NS1::Fnc()
double NS1::Fnc(double x)
  {return   x * NS1::Gain;}
// 定义函数 NS2::Fnc()
double NS2::Fnc(double x)
  {return   x * NS2::Gain;}
```

范例程序　文件 Test.cpp

```
// Test.cpp
#include <iostream>
using std::cout;
using std::endl;
#include "First.h"   // 引入 NS1 的声明
#include "Second.h"  // 引入 NS2 的声明

// ----主程序----------------------
int main()
{
  NS1::Member Ma={"George Wang", 23246352};
  NS2::Member Mb={34,"Peter White", 12.67};
  cout << "Name of Ma is:  "
       << Ma.Name       << endl;
  cout << "Name of Mb is:  "
       << Mb.Name       << endl;;
  cout << " NS1::Gain is: "
       << NS1::Gain     << endl;
  cout << " NS2::Gain is: "
       << NS2::Gain     << endl;
```

```
    cout << " NS1::Fnc(2.5) is: "
         << NS1::Fnc(2.5) << endl;
    cout << " NS2::Fnc(2.5) is: "
         << NS2::Fnc(2.5) << endl;
}
```

操作结果

```
Name of Ma is:  George Wang
Name of Mb is:  Peter White
NS1::Gain is: 5.9
NS2::Gain is: 2.6
NS1::Fnc(2.5) is: 14.75
NS2::Fnc(2.5) is: 6.5
```

讨 论

以 Borland C++ Command Line Compiler 的使用方式为例，本程序在 DOS 下的编译指令为：

```
bcc32 Test FncLib
```

16.4　使用“using 指令”和“using 声明”以存取名称空间的成员

从 16.3 节的使用经验，我们可以很明显地感觉到使用名称空间虽然可以避免名称冲突的问题，其代价却是更庞杂的语句．为了避免在存取时使用诸如

```
double x = MyNamespace::Ratio;
```

的冗长成员名称，可以在程序中加入“using 指令”（using directive）加以简化．在 using 指令之后，所有的成员都不再需要匹配修饰符 (qualifier) 了．例如：

```
using namespace MyNamespace;
double x = Ratio;
```

也就是说，使用了“using 指令”之后，名称空间内所定义的所有成员就已完全开放，可以直接存取，不需再用范围确认运算符“::”了．

事实上，如果我们曾经在程序中声明了名称空间 MyNamespace，则上述 using 指令相当于在指令的地方加入了下列四个声明语句:

```
const double Gain = 5.9;
struct Member
  {
    char Name[20];
    int  Phone;
  };
double Fnc(double);
```

名称空间成员的适用范围

不管是使用“using 指令”还是“using 声明”，当名称空间成员的名称和局部变量（local variables）相同时，采用的是局部变量．这个规则和局部变量的优先权高于全局变量 (global variables) 的道理是一样的．此外，如果 using 指令或 using 声明写在区块 (block，亦即以大括号 {} 围起来的单元) 之内，则其作用范围就只到那个区块结尾的地方．

例如以下的完整程序 Priority.cpp：

范例程序　文件 Priority.cpp

```
// Priority.cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
namespace NS     // 声明名称空间 NS
{
   int M = 10;                  // (1)
   float x =1.0,  y = 2.0;      // (2)
   char C = 'G';                // (3)
 }
// ----主程序-----------------------
int main()
{
    NS::M = 6;                  //  更动(1)中的 M
    {
      float y = 7.5;            // (4)定义局部变量 y
      using namespace NS;       //  using 指令
      x = 5.6;                  //  更动(2) 中的 x
      cout << y;                //  输出(4) 中的 y (=7.5)
    }
}
// C = 'P';      错误！此处已无法取用 (3) 中的 C
```

实例介绍

了解名称空间的语法后，让我们再回到 16.1 节和 16.3 节的问题，看看 using 指令如何在实际问题上使用。

为了方便对照，我们仍然保留 16.2 节问题内的 First.h，FncLib.cpp 和 Test.cpp 三个文件的名称：

- 文件 First.h 完全没有改变，这个文件声明了名称空间 NS1。
- 在文件 FncLib.cpp 中，我们只保留了函数 NS1::Fnc()的实现。
- 文件 Test.cpp 也只包括了名称空间 NS1 成员的使用。在这个文件里面示范了 using 指令的使用方式。

范例程序　文件 First.h

```
// First.h
#ifndef FIRST_H
#define FIRST_H
namespace NS1  // 声明名称空间 NS1
{
  const double Gain = 5.9;
  struct Member
   {
    char Name[20];
    int  Phone;
   };
  double Fnc(double);
}
#endif
```

范例程序　文件 FncLib.cpp

```
// FncLib.cpp
#include "First.h"     // 引入 NS1 的声明
// 定义函数 NS1::Fnc()
double NS1::Fnc(double x)
  {return   x * NS1::Gain;}
```

范例程序 文件 Test.cpp

```
// Test.cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include "First.h"      // 引入 NS1 的声明
// 名称空间 NS1 的 using 指令
using namespace NS1;

// ----主程序-----------------------
int main()
{
  Member Ma={"George Wang", 23246352};
  Member Mb={"Peter White", 34536767};
  cout << "Name of Ma is        "
       << Ma.Name  << endl;
  cout << "Phone number of Mb is "
       << Mb.Phone << endl;;
  cout << "Gain is: "
       << Gain     << endl;
  cout << "Fnc(2.5) is: "
       << Fnc(2.5) << endl;
}
```

操作结果

```
Name of Ma is          George Wang
Phone number of Mb is 34536767
Gain is: 5.9
Fnc(2.5) is: 14.75
```

讨 论

1. 以 Borland C++ Command Line Compiler 的使用方式为例，本程序在 DOS 下的编译指令仍然为：

```
bcc32 Test FncLib
```

2. 从文件 Test.cpp 的内容，可见 using 指令

```
using namespace NS1;
```

的确可以简化名称空间成员的使用。例如，可以直接使用以下的语句：

```
Member Ma = { "George Wang",  23246352 };
cout  <<  Gain     << endl;
cout  <<  Fnc(2.5) << endl;
```

而不需使用

```
NS1::Member Ma = { "George Wang",  23246352 };
cout  <<  NS1::Gain     << endl;
cout  <<  NS1::Fnc(2.5) << endl;
```

却也因为开放了所有的声明，而使得名称空间 NS2 的成员因为名称冲突的问题而无法使用。

使用“using 指令”虽然可以解决程序冗长的问题，却也因为开放了所有的声明，而违背了最初使用名称空间的目的，名称冲突的问题再度浮现。

所幸，除了“using 指令”外，还有“using 声明”（using declaration），可以选择性地声明名称空间中的成员。例如：

```
namespace NS1  // 声明名称空间 NS1
{
  const double Gain = 5.9;
  double A, B, C;
  void Fnc1(double);
```

```
    void Fnc2(double);
    void Fnc3(double);
    void Fnc4(double);
}
namespace NS2  // 声明名称空间 NS2
{
  const double Gain = 5.9;
  double A, B, E;
  void Fnc3(double);
  void Fnc4(double);
  void Fnc5(double);
  void Fnc6(double);
}
```

在 **NS1** 和 **NS2** 这两个名称空间中， 变量 A，B 和函数 Fnc3()、Fnc4()同名．如果我们使用以下的 using 声明:

```
using NS1::A;
using NS1::Fnc2;
using NS1::Fnc3;
using NS2::B;
using NS1::Fnc5;
```

就可以选择性地直接取用 **NS1** 中的变量 A 和函数 Fnc2()、 Fnc3()，以及 **NS2** 中的变量 B 和函数 Fnc5()，而不用担心其它成员间的同名问题；这就是最新 ANSI/ISO C++ 标准的精神所在。

提示

使用“using 声明”可以选择性地直接取用名称空间中的成员，避免“using 指令”全面开放，导致失控的危险． 但是，它对于 16.1 节所述的问题仍然没有帮助，因为我们想要

取用的是不同名称空间中的同名成员，所以在这种情况下还是必须逐一在该成员的名称前加上匹配修饰符来区别.

16.5　标准名称空间

C++ 的标准链接库(the C++ Standard Library)将所有的链接库成员都含括在简称为 **std** 的名称空间里面，包括第 12 章的数据流链接库 <fstream> 和第 13 章的输出格式链接库 <iomanip>，以及第 14 章的计时链接库 <ctime> 等等都是。

能够以单一名称空间 **std** 包含这些链接库的原因，是因为名称空间具有可以用累加的方式逐步加入的特性，如 16.2 节所述。

按照 16.3 和 16.4 两节的讨论，使用 cout 和 endl 有下列三种 using 语句的语法可以选用:

(1) 使用“using 声明”

```
#include <iostream>
using std::cout;
using std::endl;
cout << 3.5 << endl;
```

(2) 使用“using 指令”

```
#include <iostream>
using namespace std;
cout << 3.5 << endl;
```

(3) 使用匹配修饰符 (qualifier)

```
#include <iostream>
std::cout << 3.5 << std::endl;
```