第 20 章　作用域与存储类

标志符具有作用域，对象具有存储类，而变量二者都具有。本章将探讨声明与定义的差异，并将讲解如果确定标志符的作用域以及对象的存储类。

20.1　声明与定义

在使用之前，任何标志符都必须被声明或者定义。声明一个名称，就是告诉编译器与这个名称相关联的类型是什么。

定义一个对象或者变量，就是分配空间以及(可能的)赋初值。例如：

```
double x, y, z;
char* p;
int i = 0;
QString message("Hello");
```

定义一个函数，表示在一个 C++语句块中完整地描述它的行为。例如：

```
int max(int a, int b) {
  return a > b ? a : b;
}
```

定义一个类，表示以函数和数据成员声明的顺序指定类的结构，见示例 20.1。此外，类定义还告知编译器它的对象要求多少内存空间。

示例 20.1　src/early-examples/decldef/point.h

```
class Point {                                          1
 public:
    Point(int x, int y, int z);                        2
    int distance(Point other);                         3
    double norm() const {                              4
        return distance(Point(0,0,0));
    }
 private:
    int m_Xcoord, m_Ycoord, m_Zcoord;                  5
};
```

1 类首部。

2 构造函数声明。

3 函数声明。

4 声明与定义。

5 数据成员声明。

示例 20.2 中包含的一些声明不是定义。

示例 20.2　src/early-examples/decldef/point.cpp

```
extern int step;                                       1
class Map;                                             2
int max(int a, int b);                                 3
```

1 对象(变量)声明。

2 (前置)类声明。

3 全局(非成员)函数声明。

每一个不是为定义的声明，都会向编译器传递一个隐含的承诺(链接器会强制这样)：所声明的名称会在程序中某个合适的位置定义。

每一个定义就是一个声明。某个作用域内的任何名称的定义只能出现一次，但是可以存在多个声明。

注意

C++中，变量初始化似乎是“可选的”，但是不管是否指定了初始化，对变量的初始化工作总是会发生。任何具有如下形式的语句：

```
TypeT var;
```

会导致变量 var 的默认初始化。默认初始化表示值是由编译器提供的。对于简单类型(例如，int, double, char)，默认值是未定义的。即赋予 var 的默认值可以是 0，也可能是某个恰好位于内存中的随机垃圾值。对于类对象，其默认值由默认构造函数(如果存在的话)确定，否则编译器会报告错误。所以，强烈建议为所有的变量定义提供精心挑选的初始值，否则，可能会出现无效的结果或者难于定位的、奇怪的运行时错误。有必要重申一下这个原则：所有的对象和变量都应当在创建时(或随后)就正确地初始化。

20.2　标志符的作用域

每一个标志符的作用域都由其所声明的位置确定。程序中标志符的作用域，是它能够被识别并使用的程序区域。如果在标志符的作用域之外使用它的名称，就会导致错误。

在不同的作用域内可以声明并使用同一个名称。同一个名称在不同作用域下的使用规则如下所示。

1. 来自于最近的作用域的名称被首先使用。
2. 如果在最近的作用域内没有定义这个名称，则会使用最近的包含作用域(enclosing scope)中的同一个名称。
3. 如果任何包含作用域内都没有定义这个名称，则编译器会报告错误。

下面探讨 C++中的六种不同的作用域：

1. 块作用域(只用于语句块中)
2. 函数作用域(函数的整个范围)[①]
3. 类作用域(类的整个范围，包括它的成员函数定义)
4. 命名空间作用域(具有类作用域特性的扩展块作用域)
5. 文件作用域(从某个源代码文件声明的开始处到结束处)
6. 全局作用域(与文件作用域类似，但可以扩展至多个源代码文件)

① 只有标记才具有函数作用域。

20.2.1 标志符的默认作用域

下面逐一分析这六种作用域并给出一些示例。

块作用域

在一对大括号(不包括 namespace 语句块)里或者在一个函数参数表中声明的标志符，具有块作用域。块作用域的范围从它的声明开始到右大括号结束。

函数作用域

标记(label)是一种后面跟有一个冒号的标志符。C/C++函数中的标记具有自己的作用域。在整个函数定义中，标记在其声明之前和之后都是可识别的。对于老式的、(正常情况下)应避免使用的、需要标记的 goto 语句，C 和 C++几乎不支持。函数作用域的独特之处在于：标记(声明它的位置)能够出现在它所指向的第一条语句(例如，goto 语句)的后面。示例 20.3 中给出的例子，使用了强烈建议丢弃的 goto 语句以及与之相关的标记。

示例 20.3 src/goto/goto.cpp

```
[ . . . . ]
int look() {
    int i=0;
    for (i=0; i<10; ++i) {
        if (i == rand() % 20)
            goto found;                    1
    }
    return -1;
found:                                     2
    return i;
}
[ . . . . ]
```

1 最好使用 break 或者 continue。

2 goto 充当了标记的前置声明。

关于标记的一种相关的但是危险性小一些的用法，是将其用在 switch 语句块中。switch 语句是一种具有计算功能的 goto 语句，且由于它的动作都位于一个语句块之内，所以不会导致像 goto 语句那样的有效性问题。示例 20.4 是使用 switch 语句的一个例子。

示例 20.4 src/switch/switchdemo.cpp

```
#include <QTextStream>
#include "usermanager.h"

QTextStream cout(stdout);
QTextStream cin(stdin);
enum  Choices {LOAD = 1, ADD, CHANGE, CHECK, SAVE, LIST, QUIT};    1

// Function Prototypes
void addUsers(UserManager&);
void changePassword(UserManager&);
Choices menu();
```

```
//etc.

int main() {
  // some code omitted
  while (1) {
    switch (menu()) {                                    2
    case LOAD:
      cout << "Reading from file ...\n"
           << um.loadList() << " loaded to list"
           << endl;
      break;
    case ADD:
      cout << "Adding users to the list ..." << endl;
      addUsers(um);
      break;
    case SAVE:
      cout << "Saving changes ...\n"
           << um.saveList() << " users in file" << endl;
      break;
    case CHANGE:
      cout << "Changing password ..." << endl;
      changePassword(um);
      break;
    case CHECK:
      cout << "Checking a userid/pw combo ..." << endl;
      checkUser(um);
      break;
    case LIST:
      cout << "Listing users and passwords ...\n";
      um.listUsers();
      break;
    case QUIT:
      cout << "Exiting the program ..." << endl;
      return 0;
    default:
      cout << "Invalid choice! " << endl;
    }
  }
}
```

1 枚举见第 19 章的探讨。

2 menu()获得来自于用户的一个值。

正如 19.2.2 节中所讲，case 标记(default 标记除外)与常规的标记不同，因为它要求是整型常量。case 标记的作用域为整个 switch 语句，所以可以称其具有 switch 作用域。

有时，标记被用来解决各种兼容性问题。例如，标记被用来阻止 C++编译器对某些类定义中的“signals:”和“slots:”声明给出错误信息(参见 8.3 节)。

注意

尽管 goto 是 C++语言的一部分，但绝对不要使用它。

命名空间作用域

在某个命名空间里声明的标志符，具有命名空间作用域。这种标志符可以用在声明之后、命名空间定义里面的任何地方。命名空间定义是开放的，可以扩展。同一个命名空间的后续定义，只会向其添加项目，如示例 20.5 中所示。如果在命名空间里面试图重新定义某个项目，则会导致编译错误。

示例 20.5 src/namespace/openspace/opendemo.txt

```
//File:  a.h"
#ifndef _A_H_
#define _A_H_

#include <iostream>
namespace A {
  using namespace std;
  void f() { cout << "f from A\n"; }
  void g() { cout << "g from A\n"; }
}
#endif

//File:  new-a.h
#ifndef NEW_A_H_
#define NEW_A_H_
#include <iostream>

namespace A {
  //void k() { h(); }                                    1
  //void g() { cout << "Redefine g()/n"; }               2
  void h() {
    cout << "h from newA\n";
    g();
  }
}
#endif

File: opendemo.cpp
#include "a.h"
#include "new-a.h"

int main() {
  using namespace A;
  f();
  h();
}

/*Run

openspace> ./a.out
f from A
h from newA
g from A
openspace>

*/
```

1 错误!

2 错误!

类作用域

在某个类定义里声明的标志符，具有类作用域。类作用域的范围是类定义里的任何地方[①]，或者成员函数体里的任何地方[②]。

文件作用域

如果某个标志符的声明没有位于大括号对中，且被声明为静态的，则它就具有文件作用域。文件作用域的范围从它的声明开始到文件结束。关键字 static 使其他源文件看不到这个标志符，从而使其作用域被限制在声明它的文件内。文件作用域变量不能被声明成 extern 类型，其他文件也无法访问它。

由于文件作用域变量不会被输出，所以它不会扩展到(扰乱)全局命名空间。这种变量常用于 C 程序中，因为 C 语言中没有提供隐藏特性的实现方式，比如针对类成员的 private 声明。

注意

为了与 C 语言具有后向兼容性，C++中支持文件作用域，但是只要有可能，就应使用命名空间或者静态类成员。

全局作用域

如果某个标志符的声明没有位于大括号对中，且没有被声明为静态的，则它就具有全局作用域。这种标志符的作用域从声明处开始，到源文件的结尾处结束，但是通过 extern 关键字，它可以扩展至其他的源文件。可以利用 extern 声明来访问在其他源文件中定义的全局化标志符。

C++中，对变量采用全局作用域并不是必需的。通常而言，只有类和命名空间才应在全局作用域下定义。如果确实需要“全局”变量，则可以通过使用 public static 类成员或者命名空间成员来获得类似的功能。由于编译器一次只能处理一个源文件，所以只有链接器(或者模板编译器)才能够严格区分全局作用域与文件作用域，如示例 20.6 所示。

示例 20.6　全局作用域与文件作用域的比较

```
In file 1:

int g1;         // global
int g2;         // global
static int g3; // keyword static limits g3 to file scope
(etc.)


In file 2:

int g1;            // linker error!
```

① 包括位于所引用成员声明之前的 inline 函数定义。

② 要记住的是，非静态成员的作用域不包括静态成员函数的函数体。

```
extern int g2;     // OK, share variable space
static int g3;     // okay, 2 different variable spaces
(etc.)
```

位于命名空间中的标志符，可以通过使用作用域解析运算符“NamespaceName::”而成为全局标志符。也可以不用作用域解析运算符而用关键字 using，即可使命名空间中的标志符在其他作用域内可用。

命名空间变量和静态类成员具有静态存储类，能够在全局范围内被访问。它们就好像全局变量，但没有扩大(扰乱)全局命名空间。更多细节，请参见 20.4 节。

20.2.2 文件作用域，块作用域与“operator::”的比较

前面已经看到并使用过作用域解析运算符，它用来通过“ClassName::”扩展类的作用域或者访问它的成员。类似的语法“NamespaceName::”被用来访问命名空间中的每一个符号。C++还具有(一元)文件作用域解析运算符“::”，它提供从所包含的作用域内访问全局对象、命名空间对象或者文件作用域对象的能力。下面的练习将各种作用域用于这个运算符。

20.2.2.1 练习：文件作用域，块作用域与“operator::”的比较

1. 确定示例 20.7 中各个变量的作用域。
2. 假设你是计算机，预测这个程序的输出结果。

示例 20.7 src/early-examples/scopex.cpp

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

long x = 17;
float y = 7.3;                                          1
static int z = 11;                                      2

class Thing {
    int m_Num;                                          3
public:
    static int s_Count;                                 4
    Thing(int n = 0) : m_Num(n) {++s_Count;}
    ~Thing() {--s_Count;}
    int getNum() { return m_Num; }
};

int Thing::s_Count = 0;
Thing t(11);
int fn(char c, int x) {                                 5
    int z = 5;                                          6
    double y = 6.933;
    { char y;                                           7
    Thing z(4);                                         8
    y = c + 3;
    ::y += 0.3;                                         9
    cout << y << endl;                                  10
    }
```

```
    cout << Thing::s_Count
         << endl;                                    11
    y /= 3.0;                                        12
    ::z++;                                           13
    cout << y << endl;
    return x + z;
}

int main() {
    int x, y = 10;
    char ch = 'B';                                   14
    x = fn(ch, y);
    cout << x << endl;
    cout << ::y << endl;                             15
    cout << ::x / 2 << endl;
    cout << ::z << endl;
}
```

1 作用域：_____
2 作用域：_____
3 作用域：_____
4 作用域：_____
5 作用域：_____
6 作用域：_____
7 作用域：_____
8 作用域：_____
9 作用域：_____
10 作用域：_____
11 作用域：_____
12 作用域：_____
13 作用域：_____
14 作用域：_____
15 作用域：_____

20.3　存储类

只要创建了对象，就会为它分配如下四种可能的位置空间，这些位置被称为存储类。

注意

作用域指的是标志符可以访问的代码区域。存储类指的是内存中的某个位置。

1. 静态区域——全局变量、静态局部变量、静态数据成员被保存在静态存储区域中。静态对象的生命周期从加载它的对象模块开始，到程序终止时结束。
   静态区域常用于指针、简单类型和字符串常量，较少用于复杂对象。

2. 程序栈(自动存储类 auto[①])——函数参数、局部变量、返回值以及其他的临时对象都保存于栈中。栈存储是在执行对象定义时自动分配的。位于这个存储类中的对象，对函数或者语句块而言是局部的[②]。
   对于局部(块作用域)变量，其生命周期由包围所执行代码的括号确定。
3. 堆存储或者自由存储(动态存储)——通过 new 关键字创建的对象。堆对象的生命周期完全由 new 和 delete 关键字的使用确定。
   通常而言，堆对象的分配和释放应当小心地置于所封装的类里面。
4. 另一个存储类是 C 语言遗留下来的，被称为寄存器(register)。它是自动存储类的一种特殊形式，由数量相对较小的、最快速的可用内存组成，通常位于 CPU 中。
   这种类型的存储类，可以通过在变量声明中使用关键字 register 来请求它。大多数 C++编译器都会忽略这个关键字，而将这种变量置于栈中，但是对寄存器内存可能具有更高的访问优先级。为某个对象请求这种存储类，意味着不能用取址运算符(&)获得它的地址。

20.3.1 全局变量，静态变量与 QObject

通常而言，存在如下两个理由来证明有必要赋予某个对象全局作用域：

1. 对象的生命周期需要与程序的执行时间相同。
2. 需要在程序的多个位置访问这个对象。

在 C++中，应当尽可能地少使用全局作用域，而应采用其他的机制。但是，下面情况依然可使用全局作用域标志符：

- 类名称
- 命名空间名称
- 全局指针 qApp 指向正在运行的 QApplication 对象。

如果将全局对象变成静态类成员或者命名空间成员，就可以避免扩大全局命名空间的规模，但依然可以使这个对象能被多个源代码模块访问。

静态变量与 QObject

当创建 QObject 或者其他感兴趣的类[③]时，要注意的是，在 qApp(单一的 QApplication)被销毁之后，即 main()完成之后，不应当再调用它们的析构函数。

静态 QObject(以及其他的复杂类)在 qApp 被销毁之后如果依旧存在，则可能会存在代码清理问题。这是因为，当 qApp 被销毁时，它会带走许多其他的对象。

通常而言，需要对所有复杂对象的析构顺序进行控制。一种途径是确保在栈(或者堆)上分配的每一个 QObject 都是某个已经位于栈上的 QObject 的子对象或者孙对象等。

QApplication(或者它的派生实例)是所有这些对象的“最根本”栈对象，所以应尽量使其成为“最后一个存在的 QObject”。这使得 qApp 成为了程序中任何孤立 QObject “最后的祖先”的一个好选择。

① 可选关键字 auto 几乎从不使用。
② 或者为另一个对象的成员。
③ 这里的“感兴趣的类”，指的是其析构函数需执行某些重要的清理工作的任何类。

20.3.1.1　全局变量与 const

const 全局变量的作用域与常规的全局变量的作用域稍有不同。

默认情况下，被声明成 const 的全局对象具有文件作用域。与在所有语句块之外声明的静态对象不同，如果在初始化时将全局 const 变量声明成 extern 类型，则可以将它扩展到其他的文件中。例如，在一个文件中，可以具有如示例 20.8 所示的代码。

示例 20.8　src/const/globals/chunk1.cpp

```
const int NN = 10;       // file scope
const int MM = 44;       // file scope
extern const int QQ = 7; // can be accessed from other files

int main() {
// NN = 12;          // error
   int array[NN];    // okay
// QQ++;             // error
   double darray[QQ];
   return 0;
}
```

在另一个文件中，可以具有如示例 20.9 所示的代码。

示例 20.9　src/const/globals/chunk2.cpp

```
extern const int NN = 22;      // a different constant
extern const int MM;           // error
// declare global constant - storage allocated elsewhere
extern const int QQ;     // external declaration

void newFunction() {
   int x = QQ + NN;
}
```

示例 20.9 中有一个 const int 变量 NN，它与示例 20.8 中同名的 const 变量彼此不相干。示例 20.9 中可以共享 const int QQ 的使用，因为存在 extern 修饰符。示例 20.9 不能通过用 extern 修饰符声明 MM 来访问具有文件作用域的 const MM。

20.3.2　练习：存储类

给出示例 20.10 中创建/定义每一个对象时的作用域/存储类。如果存在名称冲突，描述所发生的错误。

示例 20.10　src/storage/storage.cpp

```
#include <QString>


int i;                                    1
static int j;                             2
extern int k;                             3
const int l=10;                           4
extern const int m=20;                    5
```

```
class Point                                                    6
{
    public:
    QString name;                                              7
    QString toString() const;
    private:
    static int count;
    int x, y;                                                  8
};

int Point::count = 0;                                          9

QString Point::toString() const {
    return QString("(%1,%2)").arg(x).arg(y);
                                    /* Scope: _________   Storage class: ________
*/
}

int main(int argc, char** argv)                                10
{
    int j;                                                     11
    register int d;
    int* ip = 0;                                               12
    ip = new int(4);                                           13
    Point p;                                                   14
    Point* p2 = new Point();                                   15
}
```

1 作用域：_____ 存储类：_____
2 作用域：_____ 存储类：_____
3 作用域：_____ 存储类：_____
4 作用域：_____ 存储类：_____
5 作用域：_____ 存储类：_____
6 作用域：_____ 存储类：_____
7 作用域：_____ 存储类：_____
8 作用域：_____ 存储类：_____
9 作用域：_____ 存储类：_____
10 argc 和 argv 的作用域/存储类：_____
11 作用域：_____ 存储类：_____
12 作用域：_____ 存储类：_____
13 作用域：_____ 存储类：_____
14 作用域：_____ 存储类：_____
15 作用域：_____ 存储类：_____

20.4 命名空间

在 C 和 C++中，存在一个包含如下两类信息的全局作用域：

- 所有全局函数和变量的名称
- 通常对全部程序都可用的类名称和类型名称

类是分组一个共同的标题(类名称)下的名称(成员)的一种方式，但是有时希望对名称具有更高级的分组形式。

命名空间机制提供了将全局作用域分组成各个命名的子作用域的途径。这有助于避免命名冲突，当开发使用模块的程序时，有可能出现命名冲突。定义命名空间的语法如下：

```
namespace namespaceName { decl1, decl2, ...}
```

任何合法的标志符都可以用于可选的 *namespaceName*(命名空间名称)。示例 20.11 和示例 20.12 在不同的文件中定义了两个独立的命名空间，每一个都包含相同名称的函数。

示例 20.11　src/namespace/a.h

```
#include <iostream>
namespace A {
    using namespace std;
    void f() {
        cout << "f from A\n";
    }

    void g() {
        cout << "g from A\n";
    }
}
```

示例 20.12　src/namespace/b.h

```
#include <iostream>

namespace B {
    using namespace std;
    void f() {
        cout << "f from B\n";
    }

    void g() {
        cout << "g from B\n";
    }
}
```

示例 20.13 中包含两个头文件，并使用作用域解析来调用这两个文件中声明的函数。

示例 20.13　src/namespace/namespace1.cc

```
#include "a.h"
#include "b.h"

int main() {

    A::f();
    B::g();
}
```

输出如下所示。

```
f from A
g from B
```

using 关键字使得不必使用作用域解析即可引用命名空间中的各个成员。它的语句具有如下两种形式。

1. using 指令

   ```
   using namespace namespaceName
   ```

   将整个命名空间导入到当前作用域中。

2. using 声明

   ```
   using namespaceName::identifier
   ```

   将某个特定的标志符从命名空间导入到当前作用域中。

必须仔细地多加练习，以确保当标志符出现在多个命名空间中时不会导致歧义。示例 20.14 是一个具有歧义的函数调用的例子。

示例 20.14 src/namespace/namespace2.cc

```
#include "a.h"
#include "b.h"

int main() {
    using A::f;                          1
    f();
    using namespace B;                   2
    g();                                 3
    f();                                 4
}
```

1 声明——将“A::f()”带入作用域。

2 将命名空间 B 中的全部名称带入作用域。

3 可以这样做。

4 出现歧义。

输出如下所示。

```
f from A
g from B
f from A
```

提示

为了确保各种命名空间的名称是唯一的，有时程序员需要使用非常长的命名空间名称。利用类似下面的命令，就可以很轻易地为长的命名空间名称定义一个别名：

```
namespace xyz = verylongcomplicatednamespacename;
```

20.4.1 匿名命名空间

没有名称的命名空间，是一个匿名命名空间，它与静态全局标志符或者静态文件作用域标志符相似。匿名命名空间的作用域从它的声明点开始，到文件结尾处结束[①]。

① 除非这个匿名命名空间出现在另一个命名空间中(C++语言允许这样)。如果是这样，则匿名命名空间的作用域会被进一步由包含它的命名空间的方括号所限制。

示例 20.15 表明，使用匿名命名空间就可以不使用静态全局声明。

示例 20.15　src/namespace/anonymouse.h

```
namespace {
    const int MAXSIZE = 256;
}

void f1() {
    int s[MAXSIZE];
}
```

20.4.2　开放的命名空间

任何命名空间定义都是开放的，这意味着可以通过声明另一个名称相同但具有新项目的命名空间，向已有的命名空间增加成员。新项目被添加到命名空间中的顺序，是编译器遇到这些命名空间声明的顺序。

类与命名空间相似，但是类不是开放的，因为它必须充当对象创建的模式。

using 指令不会扩充使用它的作用域，它会将名称从指定的命名空间导入到当前的作用域中。

局部定义的名称的优先级，会被来自于命名空间的名称的优先级高(但这种名称依然可通过作用域解析运算符访问)。

20.4.3　命名空间，静态对象与 extern

在命名空间里面声明的对象隐含为静态的，这意味着在整个程序中它只能被创建一次。静态对象的初始化只能存在于一个 C++模块中。为了声明一个静态(全局或者命名空间)对象而不定义它，必须使用关键字 extern[①]。示例 20.16 展示了如何声明命名空间变量。

示例 20.16　src/qstd/qstd.h

```
[ . . . . ]
namespace qstd {

     // declared but not defined:
     extern QTextStream cout;
     extern QTextStream cin;
     extern QTextStream cerr;

     // function declarations:
     bool yes(QString yesNoQuestion);
     bool more(QString prompt);

     int promptInt(int base = 10);
     double promptDouble();
     void promptOutputFile(QFile& outfile);
     void promptInputFile(QFile& infile);
};
[ . . . . ]
```

① 即使位于命名空间里面也需要使用这个关键字!

函数和类可以在命名空间的头文件中声明或者定义。但是，如果在头文件中没有定义，则命名空间中的每一个顶级对象(它对命名空间函数不是局部的)都必须在.cpp 文件中定义，如示例 20.17 所示。

示例 20.17 src/qstd/qstd.cpp

```
[ . . . . ]
QTextStream qstd::cout(stdout, QIODevice::WriteOnly);
QTextStream qstd::cin(stdin, QIODevice::ReadOnly);
QTextStream qstd::cerr(stderr, QIODevice::WriteOnly);

/* Namespace members are like static class members */
bool qstd::yes(QString question) {
    QString ans;
    cout << QString(" %1 [y/n]? ").arg(question);
    cout.flush();
    ans = cin.readLine();
    return (ans.startsWith("Y", Qt::CaseInsensitive));
}
```

20.5 复习题

1. 什么是作用域？什么类型的“事物”具有作用域？
2. 什么是存储类？什么类型的“事物”具有存储类？
3. 何时初始化静态对象？要注意同时考虑全局对象和块作用域对象。
4. const 是如何充当作用域修饰符的？
5. extern 的含义是什么？
6. 根据所用场合的不同，关键字 static 具有多种含义。
   a. 解释 static 如何能用作作用域修饰符。
   b. 解释 static 如何能用作存储类修饰符。
   c. 给出关键字 static 的另一种用法。
7. 定义在某个命名空间中的对象的存储类是什么？
8. 对于使用关键字 register 声明的指向某个对象的指针，其特殊之处在哪里？
9. 类与命名空间有什么不同？
10. 如果希望在命名空间的头文件中声明某个对象而不定义它，则必须做什么？