第9章 命名控制

创建名字是编程中最基本的活动，当一个项目中包含大量名字时，名字很容易冲突。 C++允许我们对名字的产生和名字的可见性进行控制，包括名字的存储位置以及名字的连接。

static这个关键字早在人们知道“重载”这个词之前就在C语言中被重载了，在C++中又增加了新的含义。static最基本的含义是指“位置不变的某个东西”（像“静电”），这里则指内存中的物理位置或文件中的可见性。

在这一章里，我们将看到static是怎样控制存储和可见的，还将看到一种通过C++的名字空间特征来控制访问名字的改进方法。我们还可以发现怎样使用C语言中编写并编译过的函数。

9.1 来自C语言中的静态成员

在C和C++中，static都有两种基本的含义，并且这两种含义经常是互相有冲突的：

1) 在固定的地址上分配，也就是说对象是在一个特殊的静态数据区上创建的，而不是每次函数调用时在堆栈上产生的。这也是静态存储的概念。

2) 对一个特定的编译单位来说是本地的（就像我们在后面将要看到的，这在C++中包括类的范围）。这里static控制名字的可见性，所以这个名字在这个单元或类之外是不可见的。这也描述了连接的概念，它决定连接器将看到哪些名字。

本节将着重讨论static的这两个含义，这些都是从C中继承来的。

9.1.1 函数内部的静态变量

通常，在函数体内定义一个变量时，编译器使得每次函数调用时堆栈的指针向下移一个适当的位置，为这些内部变量分配内存。如果这个变量有一个初始化表达式，那么每当程序运行到此处，初始化就被执行。

然而，有时想在两次函数调用之间保留一个变量的值，我们可以通过定义一个全局变量来实现这点，但这样一来，这个变量就不仅仅受这个函数的控制。 C和C++都允许在函数内部创建一个static对象，这个对象将存储在程序的静态数据区中，而不是在堆栈中。这个对象只在函数第一次调用时初始化一次，以后它将在两次函数之间保持它的值。比如，下面的函数每次调用时都返回一个字符串中的下一个字符。

```
//: STATFUN.CPP -- Static vars inside functions
#include <iostream.h>
#include "..\allege.h"

char onechar(const char* string = 0) {
  static const char* s;
  if(string) {
    s = string;
    return *s;
  }
```

```
  else
    allege(s, "un-initialized s");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

main() {
  // Onechar(); // causes allege()
  onechar(a); // Initializes s to a
  char c;
  while((c = onechar()) != 0)
    cout << c << endl;
}
```

static char* s在每次onechar()调用时保留它的值，因为它存放在程序的静态数据区而不是存储在函数的堆栈中。当我们用一个字符指针作参数调用 onechar()时，参数值被赋给s，然后返回字符串的第一个字符。以后每次调用 onechar()都不用带参数，函数将使用缺省参数 0作为string的值，函数就会继续用以前初始化的 s值取字符，直到它到达字符串的结尾标志 —— 空字符为止。这时，字符指针就不会再增加了，这样，指针不会越过字符串的末尾。

但是，如果调用onechar()时没有参数而且s以前也没有初始化，那会怎样呢？我们也许会在s定义时提供一个初始值：

```
static char* s=0；
```

但如果说没有为一个预定义类型的静态变量提供一个初始值的话，编译器也会确保在程序开始时它被初始化为零（转化为适当的类型），所以在onechar()中，函数第一次调用时s将被赋值为零，这样if(!s)后面的程序就会被执行。

上例中s的初始化是很简单的，其实对一个静态对象的初始化（与其他对象的初始化一样）可以是任意的常量表达式，它可以包括常量及在此之前已声明过的变量和函数。

1. 函数体内部的静态对象

用户自定义的静态变量同一般的静态对象的规则是一样的，而且它同样也必须有初始化操作。但是，零赋值只对预定义类型有效，用户自定义类型必须用构造函数来初始化。因此，如果我们在定义一个静态对象时没有指定构造函数参数，这个类就必须有缺省的构造函数。请看下例：

```
//: FUNOBJ.CPP -- Static objects in functions
#include <iostream.h>

class X {
  int i;
public:
  X(int I = 0) : i(I) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};
void f() {
  static X x1(47);
```

```
  static X x2; // Default constructor required
}

main() {
  f();
}
```

在函数f()内部定义一个静态的X类型的对象，它可以用带参数的构造函数来初始化，也可以用缺省构造函数。程序控制第一次转到对象的定义点时，而且只有第一次时，才需要执行构造函数。

2. 静态对象的析构函数

静态对象的析构函数（包括静态存储的所有对象，不仅仅是上例中的局部静态变量）在程序从main() 块中退出时，或者标准的C库函数exit()被调用时才被调用。多数情况下main()函数的结尾也是调用exit()来结束程序的。这意味着在析构函数内部使用exit()是很危险的，因为这可能陷入一个死循环中。但如果用标准的C库函数abort()来退出程序，静态对象的析构函数并不会被调用。

我们可以用标准C库函数atexit()来指定当程序跳出main()（或调用exit()）时应执行的操作。在这种情况下，在跳出main()或调用exit()之前，用atexit()注册的函数可以在所有对象的析构函数之前被调用。

静态对象的销毁是按它们初始化时相反的顺序进行的。当然只有那些已经被创建的对象才会被销毁。幸运的是，编程系统会记录对象初始化的顺序和那些已被创建的对象。全局对象总是在main()执行之前被创建了，所以最后一条语句只对函数局部的静态对象起作用。如果一个包含静态对象的函数从未被调用过，那么这个对象的构造函数也就不会执行，这样自然也不会执行析构函数。请看下例：

```
//: STATDEST.CPP -- Static object destructors
#include <fstream.h>
ofstream out("statdest.out"); // Trace file

class obj {
  char c; // Identifier
public:
  obj(char C) : c(C) {
    out << "obj::obj() for " << c << endl;
  }
  ~obj() {
    out << "obj::~obj() for " << c << endl;
  }
};

obj A('A'); // Global (static storage)
// Constructor & destructor always called

void f() {
  static obj B('B');
}

void g() {
  static obj C('C');
```

```
}

main() {
  out << "inside main()" << endl;
  f(); // Calls static constructor for B
  // g() not called
  out << "leaving main()" << endl;
}
```

在obj中，字符c的作用就象一个标识符，构造函数和析构函数就可以显示出当前正在操作的对象信息。而A是一个全局的obj类的对象，所以构造函数总是在main()函数之前就被调用。但函数f()的内部的静态obj类对象B和函数g()内部的静态对象C的构造函数只在这些函数被调用时才起作用。

为了说明哪些构造函数与析构函数被调用，在main()中只调用了f()，程序的输出结果为：

```
obj::obj() for A
inside main()
obj::obj() for B
leaving main()
obj::~obj() for B
obj::~obj() for A
```

对象A的构造函数在进入main()函数之前即被调用，而B的构造函数只是因为f()的调用而调用。当从main()函数中退出时，所有被创建的对象的析构函数按创建时相反的顺序被调用。这意味着如果g()被调用，对象B和C的析构函数的调用顺序依赖于g()和f()的调用顺序。

注意跟踪文件ofstream的对象out也是一个静态对象，它的定义（与 extern声明意义相反）应该出现在文件的一开始，在使用 out之前，这一点很重要，否则我们就可能在一个对象初始化之前使用它。

在C++中，全局静态对象的构造函数是在 main()之前调用的，所以我们现在有了一个在进入main()之前执行一段代码的简单的、可移植的方法，并且可以在退出 main()之后用析构函数执行代码。在C中要做到这一点，我们不得不熟悉编译器开发商的汇编语言的开始代码。

9.1.2 控制连接

一般情况下，在文件范围内的所有名字（既不嵌套在类或函数中的名字）对程序中的所有编译单元来说都是可见的。这就是所谓的外部连接，因为在连接时这个名字对连接器来说是可见的，外部的编译单元、全局变量和普通函数都有外部连接。

有时我们可能想限制一个名字的可见性。想让一个变量在文件范围内是可见的，这样这个文件中的所有函数都可以使用它，但不想让这个文件之外的函数看到或访问该变量，或不想这个变量的名字与外部的标识符相冲突。

在文件范围内，一个被明确声明为static的对象或函数的名字对编译单元（用本书的术语来说也就是出现声明的.CPP文件）来说是局部变量；这些名字有内部连接。这意味着我们可以在其他的编译单元中使用同样的名字，而不会发生名字冲突。

内部连接的一个好处是这个名字可以放在一个头文件中而不用担心连接时发生冲突。那些通常放在头文件里的名字，像常量、内联函数（ inline function），在缺省情况下都是内部连接的（当然常量只有在C++中缺省情况下是内部连接的，在C中它缺省为外部连接）。注意连接只引用那些在连接/装载期间有地址的成员，因此类声明和局部变量并没有连接。

• 冲突问题

下面例子说明了static的两个含义怎样彼此交叉的。所有的全局对象都是隐含为静态存储类，所以如果我们定义（在文件范围）

```
int a=0;
```

则a被存储在程序的静态数据区，在进入main()函数之前，a即已初始化了。另外，a对全局都是可见的，包括所有的编译单元。用可见性术语，static（只在编译单元内可见）的反义是extern，它表示这个名字对所有的编译单元都是可见的。所以上面的定义和下面的定义是相同的。

```
extern int a=0;
```

但如果这样定义：

```
static int a=0;
```

我们只不过改变了a的可见性，现在a成了一个内部连接。但存储类型没有改变——对象总是驻留在静态数据区，而不管是static还是extern。

一旦进入局部变量，static就不会再改变变量的可见性（这时 extern是没有意义的），而只是改变变量的存储类型。

对函数名，static和extern只会改变它的可见性，所以如果说：

```
extern void f()；
```

它和没有修饰时的声明是一样的：

```
void f()；
```

如果定义：

```
static void f()；
```

它意味着f()只在本编译单元内是可见的，这有时称作文件静态。

9.1.3 其他的存储类型指定符

我们会看到static和extern用得很普遍。另外还有两个存储类型指定符，这两种用得较少。一个是auto,人们几乎不用它，因为它告诉编译器这是一个局部变量，实际上编译器总是可以从变量定义时的上下文中判断出这是一个局部变量。所以 auto是多余的。还有一个是register，它也是局部变量，但它告诉编译器这个特殊的变量要经常用到，所以编译器应该尽可能地让它保存在寄存器中。它用于优化代码。各种编译器对这种类型的变量处理方式也不尽相同，它们有时会忽略这种存储类型的指定。一般，如果要用到这个变量的地址，register指定符通常都会被忽略。应该避免用register类型，因为编译器在优化代码方面通常比我们做得更好。

9.2 名字空间

虽然名字可以在类中被嵌套，但全局函数、全局变量以及类的名字还是在同一个名字空间中。虽然static关键字可以使变量和函数内部连接（使它们的文件静态），但在一个大项目中，如果对全局的名字空间缺乏控制就会引起很多问题。为了解决这些问题，开发商常常使用冗长、难懂的名字，以使冲突减少，但这样我们不得不一个一个地敲这些名字（typedef常常用来简化这些名字）。这不是一个很好的解决方法。

我们可以用C++的名字空间特征（我们的编译器可能还没有实现这一特征，请查阅技术文档），把一个全局名字空间分成多个可管理的小空间。名字空间的关键字，像 class,struct,enum和union一样，把它们的成员的名字放到了不同的空间中去 ,尽管其他的关键字有其他的目的，

但namespace唯一的目的是产生一个新的名字空间。

9.2.1 产生一个名字空间

名字空间的产生与一个类的产生非常相似：

```
namespace MyLib {
  // Declarations
}
```

这就产生了一个新的名字空间，其中包含了各种声明。 namespace与class、struct、union和enum有着明显的区别：

1) namespace只能在全局范畴定义，但它们之间可以互相嵌套。

2) 在namespace定义的结尾，右大括号的后面不必要跟一个分号。

3) 一个namespace可以在多个头文件中用一个标识符来定义，就好象重复定义一个类一样。

```
//: HEADER1.H
namespace MyLib {
  extern int X;
  void f();
  // ...
}
//: HEADER2.H
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
  extern int Y;
  void g();
  // ...
}
```

4) 一个namespace的名字可以用另一个名字来作它的别名，这样我们就不必敲打那些开发商提供的冗长的名字了。

```
namespace BobsSuperDuperLibrary {
  class widget { /* ... */ };
  class poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
```

5) 我们不能像类那样去创建一个名字空间的实例。

1. 未命名的名字空间

每个编译单元都可包含一个未命名的名字空间 —— 在namespace关键字后面没有标识符。

```
namespace {
  class arm  { /* ... */ };
  class leg  { /* ... */ };
  class head { /* ... */ };
  class robot {
```

```
    arm Arm[4];
    leg Leg[16];
    head Head[3];
    // ...
  } Xanthan;
  int i, j, k;
}
```

在编译单元内，这个空间中的名字自动而无限制地有效。每个编译单元要确保只有一个未命名的名字空间。如果把一个局部名字放在一个未命名的名字空间中，无需加上 static说明就可以让它们作内部连接。

2. 友元

可以在一个名字空间的类定义之内插入一个 friend 声明：

```
namespace me {
  class us {
    //...
    friend you();
  };
}
```

这样函数you()就成了名字空间me的一个成员。

9.2.2 使用名字空间

可以用两种方法在一个名字空间引用同一个名字：一种是用范围分解运算符，还有一种是用using关键字。

1. 范围分解

名字空间中的任何命名都可以用范围分解运算符明确指定，就像引用一个类中的名字一样：

```
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void foo();
}

int X::Y::i = 9;

class X::Z {
  int u, v, w;
public:
  Z(int I);
  int g();
};
```

```
X::Z::Z(int I) { u = v = w = I; }
int X::Z::g() { return u = v = w = 0; }

void X::foo() {
  X::Z a(1);
  a.g();
}
```

到目前为止，名字空间看上去很像一个类。

2. using指令

用using 关键字可以让我们立即输入整个名字空间，摆脱输入一个名字空间中标识符的烦恼。这种using和namespace关键字的搭配使用叫作using 指令。using 关键字在当前范围内直接声明了名字空间中的所有的名字，所以可以很方便地使用这些无限制的名字：

```
namespace math {
  enum sign { positive, negative };
  class integer {
    int i;
    sign s;
  public:
    integer(int I = 0)
      : i(I),
        s(i >= 0 ? positive : negative)
    {}
    sign Sign() { return s; }
    void Sign(sign S) { s = S; }
    // ...
  };
  integer A, B, C;
  integer divide(integer, integer);
  // ...
}
```

现在可以在函数内部声明math中的所有名字，但允许这些名字嵌套在函数中。

```
void arithmetic() {
  using namespace math;
  integer X;
  X.Sign(positive);
}
```

如果不用using指令，这个名字空间的所有名字都需要完全限定。

using 指令有一个缺点，那就是看起来不那么直观， using指令引入名字可见性的范围是在创建using的地方。但我们可以使来自using 指令的名字暂时无效，就像它们已经被声明为这个范围的全局名一样。

```
void q() {
  using namespace math;
  integer A; // Hides math::A;
```

```
    A.Sign(negative);
    math::A.Sign(positive);
  }
```

如果有第二个名字空间

```
  namespace calculation {
    class integer {};
    integer divide(integer, integer);
    // ...
  }
```

这个名字空间也用using指令来引入，就可能产生冲突。这种二义性出现在名字的使用时，而不是在using指令使用时。

```
  void s() {
    using namespace math;
    using namespace calculation;
    // Everything's ok until:
    divide(1, 2); // Ambiguity
  }
```

这样，即使永远不产生二义性，写using指令引入带名字冲突的名字空间也是可能的。

3. using声明

可以用using声明一次性引入名字到当前范围内。这种方法不像 using指令那样把那些名字当成当前范围的全局名来看待，而是在当前范围之内进行一个声明，这就意味着在这个范围内它可以废弃来自using指令的名字。

```
  namespace U {
    void f();
    void g();
  }

  namespace V {
    void f();
    void g();
  }

  void func() {
    using namespace U; // Using directive
    using V::f; // Using declaration
    f(); // Calls V::f();
    U::f(); // Must fully qualify to call
  }
```

using声明给出了标识符的完整的名字，但没有了类型方面的信息。也就是说，如果名字空间中包含了一组用相同名字重载的函数， using声明就声明了这个重载的集合内的所有函数。

可以把using声明放在任何一般的声明可以出现的地方。 using声明与普通声明只有一点不同：using声明可以引起一个函数用相同的参数类型来重载（这在一般的重载中是不允许的）。

当然这种不确定性要到使用时才表现出来，而不是在声明时。

using声明也可以出现在一个名字空间内，其作用与在其他地方时一样：

```
namespace Q {
  using U::f;
  using V::g;
  // ...
}

void m() {
  using namespace Q;
  f(); // calls U::f();
  g(); // calls V::g();
}
```

一个using 声明是一个别名，它允许我们在不同的名字空间声明同样的函数。如果我们不想由于引入不同名字空间的函数而导致重复定义一个函数时，可以用 using声明，它不会引起任何不确定性和重复。

9.3 C++中的静态成员

有时需要为某个类的所有对象分配一个单一的存储空间。在 C语言中，可以用全局变量，但这样很不安全。全局数据可以被任何人修改，而且，在一个项目中，它很容易与其他的名字相冲突。如果可以把一个数据当成全局变量那样去存储，但又被隐藏在类的内部，并且清楚地与这个类相联系，这种处理方法当然是最理想的了。

这一点可以用类的静态数据成员来实现。类的静态成员拥有一块单独的存储区，而不管我们创建了多少个该类的对象。所有这些对象的静态数据成员都共享这一块静态存储空间，这就为这些对象提供了一种互相通信的方法。但静态数据属于类，它的名字只在类的范围内有效，并且可以是public（公有的）、private（私有的）或者protected（保护的）。

9.3.1 定义静态数据成员的存储

因为类的静态数据成员有着单一的存储空间而不管产生了多少个对象，所以存储空间必须定义在一个单一的地方。当然以前有些编译器会分配存储空间，但现在编译器不会分配存储空间。如果一个静态数据成员被声明但没有定义时，连接器会报告一个错误。

定义必须出现在类的外部（不允许内联）而且只能定义一次，因此它通常放在一个类的实现文件中。这种规定常常让人感到很麻烦，但实际上它是很合理的。比方说：

```
class A {
  static int i;
public:
  //...
};
```

之后，在定义文件中，

```
int A::i=1;
```

如果定义了一个普通的全局变量，可以写：

```
int i=1;
```

在这里，类名和范围分解运算符用于指定了i的范围。

有些人对A::i是私有的这点感到疑惑不解，还有一些事似乎被公开地处理。这不是破坏了类结构的保护性吗？有两个原因可以保证它绝对的安全。第一，这些变量的初始化唯一合法是在定义时。事实上，如果静态数据成员是一个带构造函数的对象时，可以调用构造函数来代替“=”操作符。第二，一旦这些数据被定义了，终端用户就不能再定义它——否则连接器会报告错误。而且这个类的创建被迫产生这个定义，否则这些代码在测试时无法连接。这就保证了定义只出现一次并且它是由类的构造者来控制的。

一个静态成员的初始化表达式是在一个类的范围内，请看下例：

```
//: STATINIT.CPP -- Scope of static initializer
#include <iostream.h>

int x = 100;

class withStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "withStatic::x = " << x << endl;
    cout << "withStatic::y = " << y << endl;
  }
};

int withStatic::x = 1;
int withStatic::y = x + 1;
// WithStatic::x NOT ::x

main() {
  withStatic WS;
  WS.print();
}
```

这里，withStatic::限定符把withStatic的范围扩展到全部定义。

1. 静态数组的初始化

我们不仅可以产生静态常量对象，而且可以产生静态数组对象，包括常量数组与非常量数组，下面是初始化一个静态数组的例子：

```
//: STATARRY.CPP -- Initializing static arrays

class Values {
  static const int size;
  static const float table[4];
  static char letters[10];
};
```

```
const int Values::size = 100;

const float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

main() {}
```

对所有的静态数据成员，我们必须提供一个单一的外部定义。这些定义必须有内部连接，所以可以放在头文件中。初始化静态数组的方法与其他集合类型的初始化一样，但不能用自动计数。除此之外，在类定义结束时，编译器必须知道足够的类信息来创建对象，包括所有成员的精确大小。

2. 类中的编译时常量

在第7章中，我们介绍了用枚举型来创建一个编译时常量（一种被计算出来的常量表达式，如数组大小）的方法，它作为一个类的局部常量。尽管这种方法的使用很普遍，但常被称作“ enum hack ”，因为它使枚举丧失了本来的作用。

为了用一种更好的方法来完成同样的事，我们可以在一个类中使用一个静态常量（我们的编译器可能还不支持这一技巧，请查阅随机文档）。因为它既是常量（它不会改变）又是静态的（整个类中只有唯一的一个定义点），所以一个类中的静态常量可被用作一个编译时常量，如下例：

```
class X {
  static const int size;
  int array[size];
public:
  // ...
};

const int X::size = 100; // definition
```

如果我们在类中的一个常量表达式用到静态常量，那么这个静态常量的定义应该出现在这个类的任何实例或类的成员函数定义之前（也许在头文件中）。和一个内部数据类型的全局常量一样，它并不会为常量分配存储空间，又由于它是内部连接的，所以也不会产生冲突。

这种方法的另一个好处是任何预定义类型都可以作为一个静态常量成员，而用 enum时，只能使用整型值。

9.3.2 嵌套类和局部类

可以很容易地把一个静态数据成员放在一个嵌套内中。这样的成员的定义显然是上节中情况的扩展——我们只须用另一种级别的指定。然而在局部类（在函数内部定义的类）中不能有

静态数据成员。如下例：

```
//: LOCAL.CPP -- Static members & local classes
#include <iostream.h>

// Nested class CAN have static data members:
class outer {
  class inner {
    static int i; // OK
  };
};

int outer::inner::i = 47;

// Local class cannot have static data members:
void f() {
  class foo {
  public:
//! static int i;  // Error
    // (how would you define i?)
  } x;
}

main() {}
```

我们可以看到一个局部类中有静态成员的直接问题。为了定义它，怎样才能在文件范围描述一个数据呢？实际上局部类很少使用。

9.3.3 静态成员函数

像静态数据成员一样，我们也可以创建一个静态成员函数，它为类的全体服务而不是为一个类的部分对象服务。这样就不需要定义一个全局函数，减少了全局或局部名字空间的占用，把这个函数移到了类的内部。当产生一个静态成员函数时，也就表达了与一个特定类的联系。

静态成员函数不能访问一般的数据成员，它只能访问静态数据成员，也只能调用其他的静态成员函数。通常，当前对象的地址（this）是被隐含地传递到被调用的函数的。但一个静态成员函数没有this，所以它无法访问一般的成员函数。这样使用静态成员函数在速度上可以比全局函数有少许的增长，它不仅没有传递 this所需的额外的花费，而且还有使函数在类内的好处。

用static关键字指定了一个类的所有对象占有相同的一块存储空间，函数可以并行使用它，这意味着一个局部变量只有一个拷贝，函数每次调用都使用它。

下面是一个静态数据成员和静态成员函数如何在一起使用的例子：

```
//: SFUNC.CPP -- Static member functions

class X {
  int i;
  static int j;
```

```
public:
  X(int I = 0) : i(I) {
     // Non-static member function can access
     // Static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Error: static member function
    // Cannot access non-static member data
    return ++j;
  }
  static int f() {
    //! val(); // Error: static member function
    // Cannot access non-static member function
    return incr(); // OK -- calls static
  }
};

int X::j = 0;

main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Only works with static members
}
```

因为静态成员函数没有this指针，所以它不能访问非静态的数据成员，也不能调用非静态的成员函数，这些函数要用到this指针)。

注意在main()中一个静态成员可以用点或箭头来选取，把那个函数与一个对象联系起来，但也可以不与对象相连（因为一个静态成员是与一个类相联，而不是与一个特定的对象相连），用类的名字和范围分解运算符。

这是一个有趣的特点：因为静态成员对象的初始化方法，我们可以把上述类的一个静态数据成员放到那个类的内部。下面是一个例子，它把构造函数变成私有的，这样 egg类只有一个唯一的对象存在，我们可以访问那个对象，但不能产生任何新的egg对象。

```
//: SELFMEM.CPP -- Static member of same type
// Ensures only one object of this type exists.
// Also referred to as a "singleton" pattern.
#include <iostream.h>

class egg {
  static egg E;
  int i;
```

```
  egg(int I) : i(I) {}
public:
  static egg* instance() { return &E; }
  int val() { return i; }
};

egg egg::E(47);

main() {
//!  egg x(1); // error -- can't create an egg
  // You can access the single instance:
  cout << egg::instance()->val() << endl;
}
```

E的初始化出现在类的声明完成后，所以编译器已有足够的信息为对象分配空间并调用构造函数。

9.4 静态初始化的依赖因素

在一个指定的编译单元中，静态对象的初始化顺序严格按照对象在该单元中定义出现的顺序。而清除的顺序则与初始化的顺序正好相反。

当然在多个编译单元之间没有严格的初始化顺序，也没有办法来指定这种顺序。这可能会引起不小问题。下面的例子如果一个文件包含上述情况就会立即引起灾难（它会暂停操作系统的运行，中止复杂的进程）。

```
// first file
#include <fstream.h>
ofstream out("out.txt");
```

另一个文件在它的初始表达式之一中用到了 out 对象：

```
// second file
#include <fstream.h>
extern ofstream out;
class oof {
public:
  oof() { out << "barf"; }
} OOF;
```

这个程序可能运行，也可能不运行。如果在建立可执行文件时第一个文件先初始化，那么就不会有问题，但如果第二个文件先初始化，oof的构造函数依赖out的存在，而此时out还没有创建，于是引起混乱。这只是一个互相依赖的静态对象初始化的问题，因为当我们进入 main()时，所有静态对象的构造函数都已经被调用了。

在ARM [1]中可以看到一个更丧气的例子，在一个文件中：

```
extern int y;
int x = y + 1;
```

[1] The Annotated C++ Reference Manual,Bjarne Stroustrup和Margaret Ellis著，1990年，pp.20-21。

在另一个文件中

```
extern int x;
int y = x + 1;
```

对所有的静态对象，连接装载系统在程序员指定的动态初始化发生前保证一个静态成员初始化为零。在前一个例子中，fstream out 对象的存储空间赋零并没有特殊的意思，所以它在构造函数调用前确实是未定义的。然而，对内部数据类型，初始化为零是有意义的，所以如果文件按上面的顺序被初始化，y开始被初始化为零，所以x变成1，而后y被动态初始化为2。然而，如果初始化的顺序颠倒过来，x被静态初始化为零,y被初始化为1，而后x被初始化为2。

程序员必须意识到这些，因为他们可能会在编程时遇到互相依赖的静态变量的初始化问题，程序可能在一个平台上工作正常，把它移到另一个编译环境时，突然莫名其妙地不工作了。

怎么办

有三种方法来处理这一问题：

1) 不用它，避免初始化时的互相依赖。这是最好的解决方法。

2) 如果实在要用，就把那些关键的静态对象的定义放在一个文件中，这样我们只要让它们在文件中顺序正确就可以保证它们正确的初始化。

3) 如果我们确信把静态对象放在几个编译单元中是不可避免的（比方在编写一个库时，我们无法控制那些使用该库的程序员）这时我们可用由 Jerry Schwarz在创建iostream库（因为cin,cout和cerr的定义是在不同的文件中）时提供的一种技术。

这一技术要求在库头文件中加上一个额外的类。这个类负责库中静态对象的动态初始化。下面是一个简单的例子：

```
//: DEPEND.H -- Static initialization technique
#ifndef DEPEND_H_
#define DEPEND_H_
#include <iostream.h>
extern int x; // Delarations, not definitions
extern int y;

class initializer {
  static int init_count;
public:
  initializer() {
    cout << "initializer()" << endl;
    // Initialize first time only
    if(init_count++ == 0) {
      cout << "performing initialization"
           << endl;
      x = 100;
      y = 200;
    }
  }
~initializer() {
  cout << "~initializer()" << endl;
```

```
    // Clean up last time only
    if(--init_count == 0) {
      cout << "performing cleanup" << endl;
      // Any necessary cleanup here
    }
  }
};

// The following creates one object in each
// file where DEPEND.H is included, but that
// object is only visible within that file:
static initializer init;

#endif // DEPEND_H_
```

x、y的声明只是表明这些对象的存在，并没有为它们分配存储空间。然而 initializer init 的定义为每个包含此头文件的文件分配那些对象的空间，因为名字是 static的（这里控制可见性而不是指定存储类型，因为缺省时是在文件范围内）它只在本编译单元可见，所以连接器不会报告一个多重定义错误。

下面是一个包含x、y和init_count定义的文件：

```
//: DEPDEFS.CPP -- Definitions for DEPEND.H
#include "depend.h"
// Static initialization will force
// all these values to zero:
int x;
int y;
int initializer::init_count;
```

（当然，一个文件的init静态实例也放在这个文件中）假设库的使用者产生了两个其他的文件：

```
//: DEPEND.CPP -- Static initialization
#include "depend.h"
```

和

```
//: DEPEND2.CPP -- Static initialization
#include "depend.h"

main() {
  cout << "inside main()" << endl;
  cout << "leaving main()" << endl;
}
```

现在哪个编译单元先初始化都没有关系。当第一次包含 DEPEND.H的编译单元被初始化时，init_count为零，这时初始化就已经完成了（这是由于内部类型的全局变量在动态初始化之前都被设置为零）。对其余的编译单元，初始化会跳过去。清除按相反的顺序，且 ~initializer()可确

保它只发生一次。

这个例子用内部类型作为全局静态对象，这种方法也可以用于类，但其对象必须用initializer动态初始化。一种方法就是创建一个没有构造函数和析构函数的类，但用不同的名字的成员函数来初始化和清除这个类。当然更常用的做法是在 initializer()函数中，设定指向对象的指针，并在堆中动态创建它们。这要用到两个C++的关键字new和delete，第12章中介绍。

9.5 转换连接指定

如果C++中编写一个程序需要用到C库，那该怎么办呢？如果这样声明一个C函数：

```
float f(int a,char b);
```

C++的编译器就会将这个名字变成像 _f_int_int之类的东西以支持函数重载（和类型安全连接）。然而，C编译器编译的库一般不做这样的转换，所以它的内部名为 _f。这样，连接器将无法解决我们C++对f（）的调用。

C++中提供了一个连接转换指定，它是通过重载 extern关键字来实现的。extern后跟一个字符串来指定我们想声明的函数的连接类型，后面是函数声明。

```
extern "C" float f(int a,char b);
```

这就告诉编译器 f()是C连接，这样就不会转换函数名。标准的连接类型指定符有“ C ”和“ C++ ”两种，但编译器开发商可选择用同样的方法支持其他语言。

如果我们有一组转换连接的声明，可以把它们放在花括号内：

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

或在头文件中：

```
extern "C" {
#include "myheader.h"
}
```

多数C++编译器开发商在他们的头文件中处理转换连接指定，包括 C和C++，所以我们不用担心它们。

虽然标准的C++只支持“ C ”和“ C++ ”两种连接转换指定，但用同样的方法可以实现对其他语言的支持。

9.6 小结

static关键字很容易使人糊涂，因为它有时控制存储分配，而有时控制一个名字的可见性和连接。

随着C++名字空间的引入，我们有了更好的、更灵活的方法来控制一个大项目中名字的增长。

在类的内部使用static是在全程序中控制名字的另一种方法。这些名字不会与全局名冲突，并且可见性和访问也限制在程序内部，使我们在维护我们的代码时能有更多的控制。

9.7 练习

1. 创建一个带整型数组的类。在类内部用未标识的枚举变量来设置数组的长度。增加一个

const int 变量，并在构造函数初始化表达式表中初始化。增加一个 static int 成员变量并用特定值来初始化。增加一个内联（inline）构造函数和一个内联（inline）型的print()函数来显示数组中的全部值，并在这两函数内调用静态成员函数。

2. STATDEST.CPP中，在main()内用不同的顺序调用f()、g()来检验构造函数与析构函数的调用顺序，我们的编译器能正确地编译它们吗？

3. 在STATDEST.CPP中，把out的定义变为一个extern声明，并把实际定义放到A（它的构造函数obj传送信息给out）的定义之后，测试我们的机器是怎样进行缺省错误处理的。当我们运行程序时确保没有其他重要程序在运行，否则我们的机器会出现错误。

4. 创建一个类，它的析构函数显示一条信息，然后调用 exit()。创建这个类的一个全局静态对象，看看会发生什么？

5. 修改第7章的VOLATILE.CPP，使comm::isr()作为一个中断服务例程来运行。