# 第 3 周课程简介

读者已经完成了两周的 C++学习。现在，应该熟悉了面向对象编程的一些高级主题，其中包括封装和多态。

## 本周内容简介

本周首先讨论静态函数和友元；第 16 章讨论高级继承；第 17 章将深入介绍流；第 18 章学习如何使用 C++标准新增的特性名称空间；第 19 章介绍模板；第 20 章介绍异常；本书的最后一章（第 21 章）介绍其他书籍不会涉及的一些主题，然后讨论通往 C++高手之路。

# 第 15 章　特殊类和函数

C++提供了几种限制变量和指针作用域及其影响的途径。至此，读者知道如何创建全局变量、局部函数变量、变量指针和类成员变量。

本章将介绍：

- 如何在同一种类型的对象之间共享信息？
- 什么是静态成员变量和静态成员函数？
- 如何使用静态成员变量和静态成员函数？
- 如何创建和使用函数指针和成员函数指针？
- 如何使用函数指针数组？

## 15.1　在同一种类型的对象之间共享数据：静态成员数据

到目前为止，读者可能认为每个对象的数据是其专有的，不在一种类的对象之间共享。例如，如果有 5 个 Cat 对象，每个对象有自己的年龄、体重和其他数据。一个 Cat 对象的年龄不影响另一个对象的年龄。

然而，有时候需要记录用于同一种类型的所有对象的数据。例如，可能想知道在程序中创建了某个类的多少个对象，其中有多少个仍然存在。静态成员变量是同一个类的所有实例共享的变量，它们是全局数据（可供程序所有部分使用）和成员数据（通常只供一个对象使用）的折衷。

可以将静态成员视为属于类的而不是对象。通常，成员数据是每个对象一个，而静态成员数据是每个类一个。程序清单 15.1 声明了一个包含静态数据成员 HowManyCats 的 Cat 类，该变量记录有多少个 Cat 对象；这是通过每次构造对象时都将静态变量 HowManyCats 加 1 以及每次销毁对象时都将其减 1 实现的。

**程序清单 15.1　静态成员数据**

```
 0: //Listing 15.1 static data members
 1:
 2: #include <iostream>
 3: using namespace std;
 4:
 5: class Cat
 6: {
 7:   public:
 8:     Cat(int age):itsAge(age){HowManyCats++; }
 9:     virtual ~Cat() { HowManyCats--; }
10:     virtual int GetAge() { return itsAge; }
11:     virtual void SetAge(int age) { itsAge = age; }
12:     static int HowManyCats;
13:
14:   private:
```

```
15:     int itsAge;
16: };
17:
18: int Cat::HowManyCats = 0;
19:
20: int main()
21: {
22:    const int MaxCats = 5; int i;
23:    Cat *CatHouse[MaxCats];
24:
25:    for (i = 0; i < MaxCats; i++)
26:       CatHouse[i] = new Cat(i);
27:
28:    for (i = 0; i < MaxCats; i++)
29:    {
30:       cout << "There are ";
31:       cout << Cat::HowManyCats;
32:       cout << " cats left!" << endl;
33:       cout << "Deleting the one that is ";
34:       cout << CatHouse[i]->GetAge();
35:       cout << " years old" << endl;
36:       delete CatHouse[i];
37:       CatHouse[i] = 0;
38:    }
39:    return 0;
40: }
```

**输出：**

```
There are 5 cats left!
Deleting the one that is 0 years old
There are 4 cats left!
Deleting the one that is 1 years old
There are 3 cats left!
Deleting the one that is 2 years old
There are 2 cats left!
Deleting the one that is 3 years old
There are 1 cats left!
Deleting the one that is 4 years old
```

**分析：**

第 5～16 行声明了一个简化的 Cat 类。第 12 行将 HowManyCats 声明为一个 int 静态成员变量。

HowManyCats 的声明并未定义一个 int 变量，没有分配存储空间。不同于非静态成员变量，实例化 Cat 对象不会为成员变量 HowManyCats 分配存储空间，因为它不在对象中。因此，第 18 行定义并初始化这个变量。

一种常见的错误是，忘记声明静态成员变量，进而忘记定义它们。千万别让这种情况发生！当然，如果发生了，链接程序将捕获这种错误，显示类似于下面的错误消息：

```
undefined Symbol Cat::HowManyCats
```

对于 itsAge 不必这样做，因为它是非静态成员变量，创建每个 Cat 对象时都会定义它，这是在第 26 行完成的。

在第 8 行，Cat 构造函数将该静态成员变量加 1；在第 9 行，析构函数将其减 1。这样，在任何时刻 HowManyCats 变量都准确记录了有多少个对象被创建且没有被销毁。

第 20～40 行的 main( )函数实例化了 5 个 Cat 对象，并将它们存储到一个数组中。这调用了 Cat 构造函数 5 次，因此 HowManyCats 变量从初始值 0 开始被 5 次加 1。

然后，程序遍历该数组，并在删除当前 Cat 指针前打印 HowManyCats 的值。打印结果表明，起始值为 5（因为创建了 5 个 Cat 对象），然后每循环一次 Cat 对象减少一个。

注意，HowMangCats 变量是公有的，可在 main( )中访问。没有理由这样暴露该成员变量。事实上，只要总是通过 Cat 实例来访问数据，就应将该成员变量及其他成员变量声明为私有的，并且提供一个公有的存取器函数。另一方面，如果你想在没有 Cat 对象的情况下直接访问该数据，有两种选择：将它声明为公有变量，就像程序清单 15.2 中那样；提供一个静态成员函数，这将在本章后面讨论。

**程序清单 15.2　在不通过对象的情况下访问静态成员**

```
 0: //Listing 15.2 static data members
 1:
 2: #include <iostream>
 3: using namespace std;
 4:
 5: class Cat
 6: {
 7:   public:
 8:     Cat(int age):itsAge(age){HowManyCats++; }
 9:     virtual ~Cat() { HowManyCats--; }
10:     virtual int GetAge() { return itsAge; }
11:     virtual void SetAge(int age) { itsAge = age; }
12:     static int HowManyCats;
13:
14:   private:
15:     int itsAge;
16: };
17:
18: int Cat::HowManyCats = 0;
19:
20: void TelepathicFunction();
21:
22: int main()
23: {
24:    const int MaxCats = 5; int i;
25:    Cat *CatHouse[MaxCats];
26:
27:    for (i = 0; i < MaxCats; i++)
28:    {
29:       CatHouse[i] = new Cat(i);
30:       TelepathicFunction();
31:    }
32:
33:    for ( i = 0; i < MaxCats; i++)
34:    {
35:       delete CatHouse[i];
36:       TelepathicFunction();
37:    }
38:    return 0;
39: }
40:
```

```
41: void TelepathicFunction()
42: {
43:    cout << "There are ";
44:    cout << Cat::HowManyCats << " cats alive!" << endl;
45: }
```

**输出：**

```
There are 1 cats alive!
There are 2 cats alive!
There are 3 cats alive!
There are 4 cats alive!
There are 5 cats alive!
There are 4 cats alive!
There are 3 cats alive!
There are 2 cats alive!
There are 1 cats alive!
There are 0 cats alive!
```

**分析：**

除新增了函数 TelepathicFunction( )外，程序清单 15.2 与程序清单 15.1 极其相似。该函数是在第 41～45 行定义的，它既没有创建 Cat 对象，也没有接受 Cat 对象作为参数，但仍能够访问成员变量 HowManyCats。需要重申的是，该成员变量不属于任何对象，而属于类；通过类可访问任何成员函数。如果该变量是公有的，则程序中的任何函数都能够访问它，即使函数没有类的实例。

也可将该成员变量声明为私有而不是公有的。如果这样做，可通过成员函数来访问它，但在这种情况下必须有这个类的对象。程序清单15.3 演示了这种方法。对程序清单15.3 进行分析后，将讨论另一种访问静态成员的方法：通过静态成员函数。

**程序清单 15.3　通过非静态成员函数访问静态成员**

```
 0: //Listing 15.3 private static data members
 1: #include <iostream>
 2: using std::cout;
 3: using std::endl;
 4:
 5: class Cat
 6: {
 7:   public:
 8:     Cat(int age):itsAge(age){HowManyCats++; }
 9:     virtual ~Cat() { HowManyCats--; }
10:     virtual int GetAge() { return itsAge; }
11:     virtual void SetAge(int age) { itsAge = age; }
12:     virtual int GetHowMany() { return HowManyCats; }
13:
14:   private:
15:     int itsAge;
16:     static int HowManyCats;
17: };
18:
19: int Cat::HowManyCats = 0;
20:
21: int main()
22: {
```

```
23:    const int MaxCats = 5; int i;
24:    Cat *CatHouse[MaxCats];
25:
26:    for (i = 0; i < MaxCats; i++)
27:      CatHouse[i] = new Cat(i);
28:
29:    for (i = 0; i < MaxCats; i++)
30:    {
31:      cout << "There are ";
32:      cout << CatHouse[i]->GetHowMany();
33:      cout << " cats left!\n";
34:      cout << "Deleting the one that is ";
35:      cout << CatHouse[i]->GetAge()+2;
36:      cout << " years old" << endl;
37:      delete CatHouse[i];
38:      CatHouse[i] = 0;
39:    }
40:    return 0;
41: }
```

**输出：**

```
There are 5 cats left!
Deleting the one that is 2 years old
There are 4 cats left!
Deleting the one that is 3 years old
There are 3 cats left!
Deleting the one that is 4 years old
There are 2 cats left!
Deleting the one that is 5 years old
There are 1 cats left!
Deleting the one that is 6 years old
```

**分析：**

第 16 行将静态成员变量 HowManyCats 声明为私有的。现在，不能在非成员函数（如前一个程序清单中的 TelepathicFunction()）中直接访问它了。

虽然 HowManyCats 是静态变量，但其作用域仍为整个类。因此类的任何成员函数（如 GetHowMany()）都可以访问它，就像成员函数可以访问任何成员数据一样。然而，Cat 对象外的函数要调用 GetHowMany()，必须有一个 Cat 对象，通过它来调用 GetHowMang()函数。

**应该：**

务必使用静态成员变量在同一个类的所有实例之间共享数据。

要限制对静态成员变量的访问，必须将其声明为保护或私有的。

**不应该：**

不要使用静态成员变量来存储单个对象的数据。静态成员数据由其所属类的所有对象共享。

## 15.2 静态成员函数

静态成员函数类似于静态成员变量：它们不属于某个对象而属于整个类。因此，不通过对象也能调用它

们，如程序清单 15.4 所示。

**程序清单 15.4　静态成员函数**

```
0:  //Listing 15.4 static data members
1:
2:  #include <iostream>
3:
4:  class Cat
5:  {
6:    public:
7:      Cat(int age):itsAge(age){HowManyCats++; }
8:      virtual ~Cat() { HowManyCats--; }
9:      virtual int GetAge() { return itsAge; }
10:     virtual void SetAge(int age) { itsAge = age; }
11:     static int GetHowMany() { return HowManyCats; }
12:   private:
13:     int itsAge;
14:     static int HowManyCats;
15: };
16:
17: int Cat::HowManyCats = 0;
18:
19: void TelepathicFunction();
20:
21: int main()
22: {
23:    const int MaxCats = 5;
24:    Cat *CatHouse[MaxCats]; int i;
25:    for (i = 0; i < MaxCats; i++)
26:    {
27:      CatHouse[i] = new Cat(i);
28:      TelepathicFunction();
29:    }
30:
31:    for ( i = 0; i < MaxCats; i++)
32:    {
33:      delete CatHouse[i];
34:      TelepathicFunction();
35:    }
36:    return 0;
37: }
38:
39: void TelepathicFunction()
40: {
41:    std::cout <<"There are " << Cat::GetHowMany()
42:              <<" cats alive!" << std::endl;
43: }
```

**输出：**

```
There are 1 cats alive!
There are 2 cats alive!
There are 3 cats alive!
```

```
There are 4 cats alive!
There are 5 cats alive!
There are 4 cats alive!
There are 3 cats alive!
There are 2 cats alive!
There are 1 cats alive!
There are 0 cats alive!
```

**分析：**

在 Cat 类的声明中，第 14 行将静态成员变量 HowManyCats 声明为私有的。第 11 行将存取器函数 GetHowMany( )声明为公有和静态的。

由于 GetHowMany( )是公有的，因此任何函数都可以访问它；同时由于它是静态的，因此不通过 Cat 对象也可以调用它。因此在第 41 行，函数 TelepathicFunction( )能够访问该公有的静态存取器函数，虽然它没有 Cat 对象可用。然而，需要注意的是，调用 GetHowMany( )时需要对其进行全限定，即在前面加上类名和两个冒号：

```
Cat::GetHowMany()
```

当然，也可以在 main( )函数中，通过可用的 Cat 对象来调用 GetHowMany( )函数，就像调用其他存取器函数一样。

**注意：静态成员函数没有 this 指针。因此，不能将它们声明为 const。另外，由于在成员函数中是通过 this 指针来访问成员数据变量的，因此静态成员函数不能访问非静态成员变量！**

**静态成员函数**

可以像调用其他成员函数一样，通过对象来调用静态成员函数；也可以不通过对象而使用类名进行全限定来调用它们。

**范例：**

```
class Cat
{
   public:
      static int GetHowMany() { return HowManyCats; }
   private:
      static int HowManyCats;
};
int Cat::HowManyCats = 0;
int main()
{
     int howMany;
     Cat theCat;                          // define a cat
     howMany = theCat.GetHowMany();       // access through an object
     howMany = Cat::GetHowMany();         // access without an object
}
```

## 15.3 函数指针

就像数组名是指向数组第一个元素的常量指针一样，函数名也是指向函数的常量指针。可以声明指向函数的指针，并使用该指针来调用相应的函数。这非常有用：它让你能够创建根据用户输入来决定调用哪个函数的程序。

有关函数指针惟一比较棘手的部分是，理解被指向的对象类型。int 指针指向 int 变量，而函数指针必须指向有合适返回类型和特征标的函数。

在下面的声明中：

```
long (* funcPtr) (int);
```

funcPtr 被声明为一个指针(注意变量名前的"*")，它指向接受一个 int 参数且返回类型为 long 的函数。*funcPtr 两边的括号是必不可少的，因为 int 两边的括号结合得更紧密，也就是说，它们的优先级比间接运算符（*）更高。如果没有第一对括号，将声明一个接受 int 参数且返回类型为 long 指针的函数（还记得吗，空格在此没有意义）。

请看下面的两个声明：

```
long * Function (int);
long (* funcPtr) (int);
```

第一行将 Function( )声明为接受一个 int 参数且返回类型为 long 指针的函数；第二行将 funcPtr 声明为函数指针，它指向的函数接受一个 int 参数且返回类型为 long。

函数指针的声明总是包括返回类型和指定参数类型（如果有的话）的括号。程序清单 15.5 演示了如何声明和使用函数指针。

**程序清单 15.5　函数指针**

```
 0: // Listing 15.5 Using function pointers
 1:
 2: #include <iostream>
 3: using namespace std;
 4:
 5: void Square (int&,int&);
 6: void Cube (int&, int&);
 7: void Swap (int&, int &);
 8: void GetVals(int&, int&);
 9: void PrintVals(int, int);
10:
11: int main()
12: {
13:    void (* pFunc) (int &, int &);
14:    bool fQuit = false;
15:
16:    int valOne=1, valTwo=2;
17:    int choice;
18:    while (fQuit == false)
19:    {
20:       cout << (0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: ";
21:       cin >> choice;
22:       switch (choice)
23:       {
24:          case 1: pFunc = GetVals; break;
25:          case 2: pFunc = Square; break;
26:          case 3: pFunc = Cube; break;
27:          case 4: pFunc = Swap; break;
28:         default: fQuit = true; break;
29:       }
30:
31:       if (fQuit == false)
```

```
32:       {
33:          PrintVals(valOne, valTwo);
34:          pFunc(valOne, valTwo);
35:          PrintVals(valOne, valTwo);
36:       }
37:    }
38:    return 0;
39: }
40:
41: void PrintVals(int x, int y)
42: {
43:    cout << "x: " << x << " y: " << y << endl;
44: }
45:
46: void Square (int & rX, int & rY)
47: {
48:    rX *= rX;
49:    rY *= rY;
50: }
51:
52: void Cube (int & rX, int & rY)
53: {
54:    int tmp;
55:
56:    tmp = rX;
57:    rX *= rX;
58:    rX = rX * tmp;
59:
60:    tmp = rY;
61:    rY *= rY;
62:    rY = rY * tmp;
63: }
64:
65: void Swap(int & rX, int & rY)
66: {
67:    int temp;
68:    temp = rX;
69:    rX = rY;
70:    rY = temp;
71: }
72:
73: void GetVals (int & rValOne, int & rValTwo)
74: {
75:    cout << "New value for ValOne: ";
76:    cin >> rValOne;
77:    cout << "New value for ValTwo: ";
78:    cin >> rValTwo;
79: }
```

**输出：**

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
```

```
New value for ValOne: 2
New value for ValTwo: 3
x: 2 y: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y: 64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

**分析：**

第 5～8 行声明了 4 个函数，它们的返回类型和特征标都相同，都返回 void 且接受两个 int 引用参数。

第 13 行将 pFunc 声明为一个函数指针，指向返回值为 void 且接受两个 int 引用参数的函数。由于特征标匹配，因此 pFunc 指向前面介绍的任何一个函数。程序不断让用户选择要调用哪个函数，然后让指针 pFunc 指向用户选择的函数。第 33～35 行打印两个 int 变量的当前值，并调用用户指定的函数，然后再次打印这些值。

**函数指针**

除用函数指针名代替函数名外，通过函数指针调用函数和直接调用函数的方法相同。

给函数指针赋值的方法是，将不带括号的函数名赋给它。函数名是指向函数的常量指针。函数指针的用法与函数名一样。函数指针必须在返回类型和特征标方面与赋给它的函数一致。

**范例：**

```
long (*pFuncOne) (int, int);
long SomeFunction (int, int);
pFuncOne = SomeFunction;
pFuncOne(5,7);
```

**警告：**函数指针非常危险，对此必须心中有数。可能在本想调用函数时不小心给函数指针赋值了；也可能在本想给函数指针赋值时不小心调用了函数。

### 15.3.1　为什么使用函数指针

一般而言，不应使用函数指针。函数指针可追溯到面向对象编程面世之前的 C 语言时代。提供它们旨在支持只备面向对象的某些优点的编程风格；然而，如果编写的是高度动态的、需要基于运行阶段的决策执行不同操作的程序，函数指针可提供可行的解决方案。

即使不使用函数指针，无疑也能编写像程序清单 15.5 那样的程序，但使用函数指针可使程序的意图和用法更明显：从列表中选择一个函数，然后调用它。

程序清单 15.6 使用了程序清单 15.5 中的函数原型和定义，但程序主体没有使用函数指针。注意这两个程序清单之间的差别。

**程序清单 15.6　在不使用函数指针的情况下重写程序清单 15.5**

```
0:  // Listing 15.6 Without function pointers
1:
2:  #include <iostream>
3:  using namespace std;
```

```
 4:
 5: void Square (int&,int&);
 6: void Cube (int&, int&);
 7: void Swap (int&, int &);
 8: void GetVals(int&, int&);
 9: void PrintVals(int, int);
10:
11: int main()
12: {
13:    bool fQuit = false;
14:    int valOne=1, valTwo=2;
15:    int choice;
16:    while (fQuit == false)
17:    {
18:       cout << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: ";
19:       cin >> choice;
20:       switch (choice)
21:       {
22:         case 1:
23:           PrintVals(valOne, valTwo);
24:           GetVals(valOne, valTwo);
25:           PrintVals(valOne, valTwo);
26:           break;
27:
28:         case 2:
29:           PrintVals(valOne, valTwo);
30:           Square(valOne,valTwo);
31:           PrintVals(valOne, valTwo);
32:           break;
33:
34:         case 3:
35:           PrintVals(valOne, valTwo);
36:           Cube(valOne, valTwo);
37:           PrintVals(valOne, valTwo);
38:           break;
39:
40:         case 4:
41:           PrintVals(valOne, valTwo);
42:           Swap(valOne, valTwo);
43:           PrintVals(valOne, valTwo);
44:           break;
45:
46:        default :
47:           fQuit = true;
48:           break;
49:       }
50:    }
51:    return 0;
52: }
53:
54: void PrintVals(int x, int y)
55: {
```

```
56:    cout << "x: " << x << " y: " << y << endl;
57: }
58:
59: void Square (int & rX, int & rY)
60: {
61:    rX *= rX;
62:    rY *= rY;
63: }
64:
65: void Cube (int & rX, int & rY)
66: {
67:    int tmp;
68:
69:    tmp = rX;
70:    rX *= rX;
71:    rX = rX * tmp;
72:
73:    tmp = rY;
74:    rY *= rY;
75:    rY = rY * tmp;
76: }
77:
78: void Swap(int & rX, int & rY)
79: {
80:    int temp;
81:    temp = rX;
82:    rX = rY;
83:    rY = temp;
84: }
85:
86: void GetVals (int & rValOne, int & rValTwo)
87: {
88:    cout << "New value for ValOne: ";
89:    cin >> rValOne;
90:    cout << "New value for ValTwo: ";
91:    cin >> rValTwo;
92: }
```

**输出：**

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y: 64
```

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

**分析：**

一种很有吸引力的做法是，在 while 循环的开头和末尾分别调用 PrintVals( )，而不是在每个 case 语句中调用。然而，这样在退出的情况下也会调用 PrintVals( )，但不符合设计方案。

如果不使用函数指针，除代码更长以及重复调用来完成同样的工作外，整体的清晰度也将有一定程度的降低。然而，这是一种人为的情形，旨在说明函数指针的工作原理。在现实情形下，使用函数指针的优点将更明显：使用函数指针可以减少重复的代码，使程序更清晰，还可根据运行阶段的情况决定调用哪个函数。

**提示：**面向对象编程通常应该让你能够避免创建或传递函数指针，而是通过所需的对象调用所需的函数或通过类调用所需的静态成员函数。如果你需要一个函数指针数组，应自问是否实际上需要的是一个合适的对象数组。

**快捷调用方式**

不需要对函数指针解除引用，虽然也可以这么做。因此，如果 pFunc 是一个函数指针，指向接受一个 int 参数且返回类型为 long 的函数，将合适的函数赋给该指针后，便可以用下面任何一种方式来调用该函数：

```
pFunc(x);
```

或者

```
(*pFunc)(x);
```

这两种形式等效，前者是后者的简化版本。

### 15.3.2 函数指针数组

就像可以声明 int 指向数组一样，也可以声明函数指针数组，其中的指针指向返回特定类型和具有特定特征标的函数。程序清单 15.7 再次重写了程序清单 15.5，这次使用数组一次性调用选择的所有函数。

**程序清单 15.7 使用函数指针数组**

```
 0: // Listing 15.7
 1: //demonstrates use of an array of pointers to functions
 2:
 3: #include <iostream>
 4: using namespace std;
 5:
 6: void Square(int&,int&);
 7: void Cube(int&, int&);
 8: void Swap(int&, int &);
 9: void GetVals(int&, int&);
10: void PrintVals(int, int);
11:
12: int main()
13: {
14:    int valOne=1, valTwo=2;
15:    int choice, i;
16:    const MaxArray = 5;
17:    void (*pFuncArray[MaxArray])(int&, int&);
18:
19:    for (i=0; i < MaxArray; i++)
20:    {
21:       cout << "(1)Change Values (2)Square (3)Cube (4)Swap: ";
22:       cin >> choice;
```

```
23:       switch (choice)
24:       {
25:         case 1:   pFuncArray[i] = GetVals; break;
26:         case 2:   pFuncArray[i] = Square; break;
27:         case 3:   pFuncArray[i] = Cube; break;
28:         case 4:   pFuncArray[i] = Swap; break;
29:         default:  pFuncArray[i] = 0;
30:       }
31:     }
32:
33:     for (i=0; i < MaxArray; i++)
34:     {
35:       if ( pFuncArray[i] == 0 )
36:         continue;
37:       pFuncArray[i](valOne,valTwo);
38:       PrintVals(valOne,valTwo);
39:     }
40:     return 0;
41: }
42:
43: void PrintVals(int x, int y)
44: {
45:    cout << "x: " << x << " y: " << y << endl;
46: }
47:
48: void Square (int & rX, int & rY)
49: {
50:    rX *= rX;
51:    rY *= rY;
52: }
53:
54: void Cube (int & rX, int & rY)
55: {
56:    int tmp;
57:
58:    tmp = rX;
59:    rX *= rX;
60:    rX = rX * tmp;
61:
62:    tmp = rY;
63:    rY *= rY;
64:    rY = rY * tmp;
65: }
66:
67: void Swap(int & rX, int & rY)
68: {
69:    int temp;
70:    temp = rX;
71:    rX = rY;
72:    rY = temp;
73: }
74:
```

```
75: void GetVals (int & rValOne, int & rValTwo)
76: {
77:    cout << "New value for ValOne: ";
78:    cin >> rValOne;
79:    cout << "New value for ValTwo: ";
80:    cin >> rValTwo;
81: }
```

**输出：**

```
(1)Change Values (2)Square (3)Cube (4)Swap: 1
(1)Change Values (2)Square (3)Cube (4)Swap: 2
(1)Change Values (2)Square (3)Cube (4)Swap: 3
(1)Change Values (2)Square (3)Cube (4)Swap: 4
(1)Change Values (2)Square (3)Cube (4)Swap: 2
New Value for ValOne: 2
New Value for ValTwo: 3
x: 2 y: 3
x: 4 y: 9
x: 64 y: 729
x: 729 y: 64
x: 531441 y:4096
```

**分析：**

第 17 行将 pFuncArray 声明为包含 5 个函数指针的数组，这些指针指向返回类型为 void 且接受两个 int 引用参数的函数。

第 19～31 行要求用户选择要调用的函数，并将相应函数的地址赋给每个数组元素。第 33～39 行依次调用这些函数，每次调用后都打印结果。

### 15.3.3 将函数指针传递给其他函数

函数指针（和函数指针数组）可被传递给其他函数，后者可执行操作，然后使用传入的指针调用相应的函数。

例如，可以对程序清单 15.5 进行改进，将指向用户选择的函数的指针传递给另一个（在 main( )外面的）函数，后者打印值，调用指针指向的函数，然后再次打印值。程序清单 15.8 演示了这种做法。

**程序清单 15.8 将函数指针作为参数传递给函数**

```
 0: // Listing 15.8 Without function pointers
 1:
 2: #include <iostream>
 3: using namespace std;
 4:
 5: void Square(int&,int&);
 6: void Cube(int&, int&);
 7: void Swap(int&, int &);
 8: void GetVals(int&, int&);
 9: void PrintVals(void (*)(int&, int&),int&, int&);
10:
11: int main()
12: {
13:    int valOne=1, valTwo=2;
14:    int choice;
15:    bool fQuit = false;
```

```
16:
17:    void (*pFunc)(int&, int&);
18:
19:    while (fQuit == false)
20:    {
21:      cout << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: ";
22:      cin >> choice;
23:      switch (choice)
24:      {
25:        case 1:  pFunc = GetVals; break;
26:        case 2:  pFunc = Square; break;
27:        case 3:  pFunc = Cube; break;
28:        case 4:  pFunc = Swap; break;
29:        default:  fQuit = true; break;
30:      }
31:
32:      if (fQuit == false)
33:        PrintVals ( pFunc, valOne, valTwo);
34:    }
35:
36:    return 0;
37: }
38:
39: void PrintVals( void (*pFunc)(int&, int&),int& x, int& y)
40: {
41:    cout << "x: " << x << " y: " << y << endl;
42:    pFunc(x,y);
43:    cout << "x: " << x << " y: " << y << endl;
44: }
45:
46: void Square (int & rX, int & rY)
47: {
48:    rX *= rX;
49:    rY *= rY;
50: }
51:
52: void Cube (int & rX, int & rY)
53: {
54:    int tmp;
55:
56:    tmp = rX;
57:    rX *= rX;
58:    rX = rX * tmp;
59:
60:    tmp = rY;
61:    rY *= rY;
62:    rY = rY * tmp;
63: }
64:
65: void Swap(int & rX, int & rY)
66: {
67:    int temp;
```

```
68:    temp = rX;
69:    rX = rY;
70:    rY = temp;
71: }
72:
73: void GetVals (int & rValOne, int & rValTwo)
74: {
75:    cout << "New value for ValOne: ";
76:    cin >> rValOne;
77:    cout << "New value for ValTwo: ";
78:    cin >> rValTwo;
79: }
```

**输出：**

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
x: 2 y: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y:64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

**分析：**

第 17 行将 pFunc 声明为一个函数指针，指向返回类型为 void 且接受两个 int 引用参数的函数。第 9 行将 PrintVals 声明为接受 3 个参数的函数。第一个参数是这样的函数指针：指向返回类型为 void 且接受两个 int 引用参数的函数；第二个参数和第三个参数都是 int 引用。在第 19 和 20 行，程序提示用户指定要调用哪个函数，然后第 33 行调用 PrintVals()，并将函数指针 pFunc 作为第一个参数。

找一位 C++程序员问他下面的声明是什么意思？

```
void PrintVals(void (*)(int&, int&), int&, int&);
```

这种声明不常用，每次需要它时你都可能需要查阅书籍，但在确实需要这种结构时（这种情况很少见），它确实可以减少程序的代码量。

### 15.3.4 将 typedef 用于函数指针

结构 void（*）（int&，int&）有点繁琐。可以使用 typedef 来简化，方法是声明一种这样的类型（程序清单 15.9 中，名为 VPF）：指向返回类型为 void 且接受两个引用参数的函数的指针。程序清单 15.9 使用这种 typedef 语句重写了程序清单 15.8。

**程序清单 15.9 使用 typedef 提高函数指针的可读性**

```
0: // Listing 15.9.
1: // Using typedef to make pointers to functions more readable
2:
3: #include <iostream>
```

```
 4: using namespace std;
 5:
 6: void Square(int&,int&);
 7: void Cube(int&, int&);
 8: void Swap(int&, int &);
 9: void GetVals(int&, int&);
10: typedef void (*VPF) (int&, int&) ;
11: void PrintVals(VPF,int&, int&);
12:
13: int main()
14: {
15:    int valOne=1, valTwo=2;
16:    int choice;
17:    bool fQuit = false;
18:
19:    VPF pFunc;
20:
21:    while (fQuit == false)
22:    {
23:       cout << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: ";
24:       cin >> choice;
25:       switch (choice)
26:       {
27:         case 1:  pFunc = GetVals; break;
28:         case 2:  pFunc = Square; break;
29:         case 3:  pFunc = Cube; break;
30:         case 4:  pFunc = Swap; break;
31:         default:  fQuit = true; break;
32:       }
33:
34:       if (fQuit == false)
35:          PrintVals ( pFunc, valOne, valTwo);
36:    }
37:    return 0;
38: }
39:
40: void PrintVals( VPF pFunc,int& x, int& y)
41: {
42:    cout << "x: " << x << " y: " << y << endl;
43:    pFunc(x,y);
44:    cout << "x: " << x << " y: " << y << endl;
45: }
46:
47: void Square (int & rX, int & rY)
48: {
49:    rX *= rX;
50:    rY *= rY;
51: }
52:
53: void Cube (int & rX, int & rY)
54: {
55:    int tmp;
```

```
56:
57:    tmp = rX;
58:    rX *= rX;
59:    rX = rX * tmp;
60:
61:    tmp = rY;
62:    rY *= rY;
63:    rY = rY * tmp;
64: }
65:
66: void Swap(int & rX, int & rY)
67: {
68:    int temp;
69:    temp = rX;
70:    rX = rY;
71:    rY = temp;
72: }
73:
74: void GetVals (int & rValOne, int & rValTwo)
75: {
76:    cout << "New value for ValOne: ";
77:    cin >> rValOne;
78:    cout << "New value for ValTwo: ";
79:    cin >> rValTwo;
80: }
```

**输出：**

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
x: 2 y: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y: 64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

**分析：**

第 10 行使用 typedef 将 VPF 声明为这样的类型：指向返回类型为 void 且接受两个 int 引用参数的函数的指针。

第 11 行将函数 PrintVals( )声明为接受 3 个参数：一个 VPF 和两个 int 引用。第 19 行将 pFunc 声明为 VPF 类型。

定义类型 VPF 后，后面声明 pFunc 和 PrintVals( )的代码都清晰得多。正如读者看到的，输出与前一个程序清单相同。别忘了，typedef 的主要功能是替换。在这个例子中，通过使用 typedef，使得代码更容易理解。

## 15.4　成员函数指针

到现在为止，创建的所有函数指针都指向常规的非成员函数。也可以创建指向类成员函数的指针。这是一种非常高级且不常用的技巧，应尽可能避免使用。然而，理解这种技巧至关重要，因为确实有些人会使用它。

要创建成员函数指针，可使用与创建函数指针相同的语法，但需要包含类名和作用域运算符（::）。因此，如果 pFunc 指向类 Shape 的一个这样的成员函数：接受两个 int 参数且返回类型为 void，则 pFunc 的声明如下：

```
void(Shape::*pFunc) (int, int);
```

成员函数指针的用法和函数指针相同，只是需要通过相应类的对象来调用。程序清单 15.10 演示了成员函数指针的用法。

**程序清单 15.10　成员函数指针**

```
0:  //Listing 15.10 Pointers to member functions using virtual methods
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  class Mammal
6:  {
7:    public:
8:       Mammal():itsAge(1) { }
9:       virtual ~Mammal() { }
10:      virtual void Speak() const = 0;
11:      virtual void Move() const = 0;
12:   protected:
13:      int itsAge;
14: };
15:
16: class Dog : public Mammal
17: {
18:   public:
19:     void Speak()const { cout << "Woof!" << endl; }
20:     void Move() const { cout << "Walking to heel..." << endl; }
21: };
22:
23:
24: class Cat : public Mammal
25: {
26:   public:
27:     void Speak()const { cout << "Meow!" << endl; }
28:     void Move() const { cout << "slinking..." << endl; }
29: };
30:
31:
32: class Horse : public Mammal
33: {
34:   public:
```

```
35:     void Speak()const { cout << "Winnie!" << endl; }
36:     void Move() const { cout << "Galloping..." << endl; }
37: };
38:
39:
40: int main()
41: {
42:    void (Mammal::*pFunc)() const =0;
43:    Mammal* ptr =0;
44:    int Animal;
45:    int Method;
46:    bool fQuit = false;
47:
48:    while (fQuit == false)
49:    {
50:      cout << "(0)Quit (1)dog (2)cat (3)horse: ";
51:      cin >> Animal;
52:      switch (Animal)
53:      {
54:        case 1:  ptr = new Dog; break;
55:        case 2:  ptr = new Cat; break;
56:        case 3:  ptr = new Horse; break;
57:        default:  fQuit = true; break;
58:      }
59:      if (fQuit == false)
60:      {
61:        cout << "(1)Speak  (2)Move: ";
62:        cin >> Method;
63:        switch (Method)
64:        {
65:          case 1:  pFunc = Mammal::Speak; break;
66:          default: pFunc = Mammal::Move; break;
67:        }
68:
69:         (ptr->*pFunc)();
70:        delete ptr;
71:      }
72:    }
73:    return 0;
74: }
```

**输出：**

```
(0)Quit (1)dog (2)cat (3)horse: 1
(1)Speak (2)Move: 1
Woof!
(0)Quit (1)dog (2)cat (3)horse: 2
(1)Speak (2)Move: 1
Meow!
(0)Quit (1)dog (2)cat (3)horse: 3
(1)Speak (2)Move: 2
Galloping
(0)Quit (1)dog (2)cat (3)horse: 0
```

**分析：**

第 5～14 行声明了抽象类 Mammal，它有两个纯虚函数：Speak( )和 Move( )。从 Mammal 派生了 3 个子类：Dog、Cat 和 Horse，每个子类都覆盖了 Speak( )和 Move( )。

main( )从第 40 行开始。第 50 行询问用户要创建哪种类型的动物。第 54～56 行在自由存储区中创建用户指定的子类对象，并让 ptr 指向它。

第 61 行提示用户选择要调用的函数。第 65～66 行将指针 pFunc 指向用户选择的方法(Speak( )或 Move( ))。第 69 行通过创建的对象调用用户选择的方法，这里通过指针 ptr 来访问对象，通过指针 pFunc 来访问函数。

最后，第 71 行对 ptr 指针调用 delete，将为对象分配的内存归还给自由存储区。注意，没有理由对 pFunc 调用 delete，因为它是一个指向代码的指针，而不是指向自由存储区中对象的指针。事实上试图这样做将导致编译错误。

### 成员函数指针数组

像函数指针一样，成员函数指针也可存储在数组中。可以用不同成员函数的地址来初始化数组，且可使用下标表示法来调用这些函数。程序清单 15.11 演示了这种技巧。

**程序清单 15.11　成员函数指针数组**

```
 0: //Listing 15.11 Array of pointers to member functions
 1: #include <iostream>
 2: using std::cout;
 3: using std::endl;
 4:
 5: class Dog
 6: {
 7:   public:
 8:     void Speak()const { cout << "Woof!" << endl; }
 9:     void Move() const { cout << "Walking to heel..." << endl; }
10:     void Eat() const { cout << "Gobbling food..." << endl; }
11:     void Growl() const { cout << "Grrrrr" << endl; }
12:     void Whimper() const { cout << "Whining noises..." << endl; }
13:     void RollOver() const { cout << "Rolling over..." << endl; }
14:     void PlayDead() const
          { cout << "The end of Little Caesar?" << endl; }
15: };
16:
17: typedef void (Dog::*PDF)()const ;
18: int main()
19: {
20:    const int MaxFuncs = 7;
21:    PDF DogFunctions[MaxFuncs] =
22:       {Dog::Speak,
23:        Dog::Move,
24:        Dog::Eat,
25:        Dog::Growl,
26:        Dog::Whimper,
27:        Dog::RollOver,
28:        Dog::PlayDead };
29:
30:    Dog* pDog =0;
31:    int Method;
```

```
32:    bool fQuit = false;
33:
34:    while (!fQuit)
35:    {
36:       cout << "(0)Quit (1)Speak (2)Move (3)Eat (4)Growl";
37:       cout << " (5)Whimper (6)Roll Over (7)Play Dead: ";
38:       std::cin >> Method;
39:       if (Method <= 0 || Method >= 8)
40:       {
41:          fQuit = true;
42:       }
43:       else
44:       {
45:          pDog = new Dog;
46:          (pDog->*DogFunctions[Method-1])();
47:          delete pDog;
48:       }
49:    }
50:    return 0;
51: }
```

**输出：**

```
(0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play Dead: 1
Woof!
(0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play Dead: 4
Grrr
(0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play Dead: 7
The end of Little Caesar?
(0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over(7)Play Dead: 0
```

**分析：**

第 5～15 行声明了 Dog 类，其中 7 个成员函数的返回类型和特征标都相同。第 17 行使用 typedef 将 PDF 声明为这样的类型：指向 Dog 类的不接受任何参数且没有返回值的 const 成员函数（Dog 的 7 个成员函数的特征标）的指针。

第 21～28 行将数组 DogFunction 声明为能存储 7 个成员函数指针的数组，并用这些函数的地址对其进行初始化。

第 36～37 行提示用户选择一个方法。除非选择了 Quit，否则将在堆上创建一个 Dog 对象，然后第 46 行使用数组调用正确的方法。这里有一行代码很适合用来考验公司的 C++程序员高手，看他是否知道其含义：

```
(pDog->*DogFunctions[Method-1])();
```

你将能够告诉这位高手，这行代码使用存储在一个数组中、下标为 Mothod - 1 的指针元素来调用它指向的方法。

同样，应尽可能避免使用这种技巧。如果不得不使用，应该做清晰地说明，并考虑另一种完成所需任务的方式。

**应该：**

*应通过类的对象来调用成员函数指针。*

*应使用 typedef 来使成员函数指针的声明更容易理解。*

**不应该：**

*在简单的解决方案可行的情况下不要使用成员函数指针。*

声明函数指针时别忘了括号，否则声明的将是返回指针的函数。

## 15.5 小　　结

本章介绍了如何在类中创建静态成员变量。每个类而不是每个对象有一个静态成员变量的实例。如果静态成员被声明为公有的，在没有对象的情况下，可以使用全限定名称来访问它。

静态成员变量的用途之一是，可用作供类的实例共享的计数器。由于静态成员变量不是对象的一部分，因此其声明并不会分配内存，另外，静态成员变量必须在类声明外定义和初始化。

静态成员函数和静态成员变量一样，也是类的一部分。在没有相应类的对象时，也可以访问静态成员函数，同时静态成员函数可用来访问静态成员数据。由于静态成员函数没有 this 指针，因此不能用来访问非静态成员数据。

由于静态成员函数没有 this 指针，因此也不能被定义为 const。成员函数声明中的 const 表示 this 指针是 const。

本章还介绍了 C++中的一个复杂主题：如何声明和使用函数指针和成员函数指针。读者学习了如何创建这些指针的数组、如何将它们传递给函数以及如何调用存储在数组中的指针指向的函数。读者知道，使用函数指针并不是什么好主意，面向对象技术应该能够让你避免在几乎任何情况下这样做。

## 15.6 问　与　答

**问**：在可以使用全局数据的情况下为何还要使用静态数据？

**答**：静态数据的作用域为其所属的类。因此，只有通过类的对象、通过使用类名的显式调用（如果静态数据是公有的）或通过静态成员函数，才能访问静态数据。静态数据在访问方面的限制和强类型特征，使其比个局数据更安全。

**问**：在可以使用全局函数的情况下为何还要使用静态成员函数？

**答**：静态成员函数的作用域为其所属的类，只有通过类的对象或显式全限定（如 ClassName::FunctionName()）才能调用它们。

**问**：使用大量函数指针和成员函数指针的情况常见吗？

**答**：不，这些指针有专门用途，不是常用的结构。很多复杂的功能强大的程序并不使用它们。然而，有时候使用它们是惟一的解决方案。

## 15.7 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。

### 15.7.1 测验

1. 静态成员变量可以是私有的吗？
2. 编写一个名为 itsStatic、类型为 int 的静态成员变量的声明。
3. 编写一个名为 SomeFunction、返回类型为 int 且不接受任何参数的静态函数的声明。
4. 编写一个这样的函数指针的声明：它指向返回类型为 long 且接受一个参数的函数。
5. 修改练习 4 中的函数指针声明，使其成为指向 Car 类的成员函数的指针。

6. 编写一个包含 10 个元素的函数指针数组的声明，其中的函数指针符合练习 5 中的定义。

### 15.7.2 练习

1. 编写一个小程序，它声明一个这样的类：包含一个成员变量和一个静态成员变量，其构造函数初始化成员变量并将静态成员变量加 1，析构函数将静态成员变量减 1。

2. 在练习 1 编写的代码的基础上，编写一个小程序，它创建 3 个对象，并显示它们的成员变量和静态成员变量。然后销毁每个对象，并显示这样做对静态成员变量的影响。

3. 修改练习 2 中的程序，使用静态成员函数来访问静态成员变量，并将静态成员变量声明为私有的。

4. 编写一个成员函数指针来访问练习 3 的程序中的非静态成员数据，并使用该指针来打印该成员变量的值。

5. 在前面的练习中创建的类中添加两个成员变量。添加获得这些成员变量的值的存取器函数，并让所有成员函数的返回类型和特征标相同。使用成员函数指针来访问这些函数。