第 10 章　有关函数的高级主题

第 5 章介绍了有关函数的基本知识。读者了解指针和引用的工作原理后，可以使用函数做更多的工作。本章介绍以下内容：

- 如何重载成员函数？
- 如何重载运算符？
- 如何编写函数以支持动态地为变量分配内存的类？

10.1　重载成员函数

第 5 章介绍了如何通过编写多个名称相同参数不同的函数来实现函数多态（函数重载）。使用相同的方式，也可以对类成员函数进行重载。

程序清单 10.1 中的 Rectangle 类有两个 DrawShape( )函数。其中一个不接受任何参数，它根据类的当前值绘制矩形；另外一个接受两个值：长和宽，并根据这些值绘制矩形，而忽略类的当前值。

程序清单 10.1　重载成员函数

```
1:  //Listing 10.1 Overloading class member functions
2:  #include <iostream>
3:
4:  // Rectangle class declaration
5:  class Rectangle
6:  {
7:    public:
8:      // constructors
9:      Rectangle(int width, int height);
10:     ~Rectangle(){}
11:
12:     // overloaded class function DrawShape
13:     void DrawShape() const;
14:     void DrawShape(int aWidth, int aHeight) const;
15:
16:   private:
17:     int itsWidth;
18:     int itsHeight;
19: };
20:
21: //Constructor implementation
22: Rectangle::Rectangle(int width, int height)
23: {
```

```
24:    itsWidth = width;
25:    itsHeight = height;
26:  }
27:
28:
29:  // Overloaded DrawShape - takes no values
30:  // Draws based on current class member values
31:  void Rectangle::DrawShape() const
32:  {
33:    DrawShape( itsWidth, itsHeight);
34:  }
35:
36:
37:  // overloaded DrawShape - takes two values
38:  // draws shape based on the parameters
39:  void Rectangle::DrawShape(int width, int height) const
40:  {
41:    for (int i = 0; i<height; i++)
42:    {
43:      for (int j = 0; j< width; j++)
44:      {
45:        std::cout << "*";
46:      }
47:    std::cout << std::endl;
48:    }
49:  }
50:
51:  // Driver program to demonstrate overloaded functions
52:  int main()
53:  {
54:    // initialize a rectangle to 30,5
55:    Rectangle theRect(30,5);
56:    std::cout << "DrawShape():" << std::endl;
57:    theRect.DrawShape();
58:    std::cout << "\nDrawShape(40,2):" << std::endl;
59:    theRect.DrawShape(40,2);
60:    return 0;
61:  }
```

输出：

```
DrawShape():
******************************
******************************
******************************
******************************
******************************
DrawShape(40,2):
****************************************
****************************************
```

分析：

程序清单 10.1 是“第 1 周复习”项目的精简版本。为缩短篇幅，删除了检测非法值的代码和一些存取器

函数。主程序被精简成一个简单的程序，而没有使用菜单。

然而，最重要的代码位于第 13 和 14 行，这里重载了 DrawShape( )函数。这些重载类方法的实现位于第 31～49 行。注意，不接受任何参数的 DrawShape( )调用接受两个参数的版本，并将成员变量的当前值传递给它。这样做旨在尽可能地避免两个函数中相同的代码。否则当一个函数发生变化时，确保它们同步将很困难，也容易出错。

第 52～61 行的程序创建了一个 Rectangle 对象，然后调用 DrawShape( )函数，第一次调用时没有传递任何参数，而第二次调用时传递了两个 unsigned short 整数。

编译器根据传入的参数数目和类型来决定调用哪个方法。可以创建第三个名为 DrawShape( )的重载函数，它接受一个尺寸和一个指出该尺寸为长还是宽的枚举值作为参数。

10.2 使用默认值

全局函数可以有一个或多个默认值，类的每个成员函数也可以。声明默认值的规则也相同，如程序清单 10.2 所示。

程序清单 10.2 使用默认值

```
1:  //Listing 10.2 Default values in member functions
2:  #include <iostream>
3:
4:  using namespace std;
5:
6:  // Rectangle class declaration
7:  class Rectangle
8:  {
9:    public:
10:     // constructors
11:     Rectangle(int width, int height);
12:     ~Rectangle(){}
13:     void DrawShape(int aWidth, int aHeight,
14:         bool UseCurrentVals = false) const;
15:
16:   private:
17:     int itsWidth;
18:     int itsHeight;
19: };
20:
21: //Constructor implementation
22: Rectangle::Rectangle(int width, int height):
23: itsWidth(width),      // initializations
24: itsHeight(height)
25: {}                   // empty body
26:
27:
28: // default values used for third parameter
29: void Rectangle::DrawShape(
30: int width,
31: int height,
32: bool UseCurrentValue
```

```
33: } const
34: {
35:    int printWidth;
36:    int printHeight;
37:
38:    if (UseCurrentValue == true)
39:    {
40:       printWidth = itsWidth;      // use current class values
41:       printHeight = itsHeight;
42:    }
43:    else
44:    {
45:       printWidth = width;         // use parameter values
46:       printHeight = height;
47:    }
48:
49:
50:    for (int i = 0; i<printHeight; i++)
51:    {
52:       for (int j = 0; j< printWidth; j++)
53:       {
54:          cout << "*";
55:       }
56:       cout << endl;
57:    }
58: }
59:
60: // Driver program to demonstrate overloaded functions
61: int main()
62: {
63:    // initialize a rectangle to 30,5
64:    Rectangle theRect(30,5);
65:    cout << "DrawShape(0,0,true)..." << endl;
66:    theRect.DrawShape(0,0,true);
67:    cout <<"DrawShape(40,2)..." << endl;
68:    theRect.DrawShape(40,2);
69:    return 0;
70: }
```

输出：

```
DrawShape(0, 0, true)...
******************************
******************************
******************************
******************************
******************************

DrawShape(40,2)...
****************************************************************
****************************************************************
```

分析：

程序清单 10.2 使用了一个带默认参数的函数，而不是对函数 DrawShape( )进行重载。这个函数是在第 13

行声明的，它接受 3 个参数。前两个（aWidth 和 aHeight）是 int 参数；第 3 个（UseCurrentVals）是 bool 参数，默认值为 false。

这个有些笨拙的函数的实现从第 29 行开始。别忘了，在 C++中空白是无关紧要的，因此函数头实际上位于第 29～33 行。

在该方法中，第 38 行计算第 3 个参数 UseCurrentValue 的值。如果它为 true，则分别用成员变量 itsWidth 和 itsHeight 来设置局部变量 printWidth 和 printHeight。

如果 UseCurrentValue 为 false：使用默认值 false 或被用户设置为 false，则用前两个参数来设置 printWidth 和 printHeight。

注意，如果 UseCurrentValue 为 true，则其他两个参数将被忽略。

10.3 在默认值和重载函数之间做出选择

程序清单 10.1 和 10.2 的功能相同，但程序清单 10.1 的重载函数更容易理解，使用起来也更自然。另外，如果需要第三个变体——用户提供长或宽，而不是两个都提供，则很容易对重载函数进行扩展；然而，使用默认值时，增加变体将变得极其复杂。

如何决定是选择函数重载还是默认值呢？下面是一条经验规则。

在以下情形使用函数重载：

- 没有合理的默认值。
- 需要不同的算法。
- 需要在参数列表中支持不同的类型。

10.4 默认构造函数

构造函数用于创建对象，例如，Rectangle 类的构造函数用于创建有效的矩形对象。构造函数被调用之前，并没有矩形，而只有一块内存区域。构造函数执行完毕后，将有一个完整的、可以使用的矩形对象。这是面向对象编程的重要优点之一：调用程序无需做任何工作来确保对象始于自相容状态。

正如第 6 章讨论的，如果没有显式地为类声明构造函数，编译器将生成一个不接受任何参数、也不执行任何操作的默认构造函数。然而，你可以创建自己的默认构造函数，它不接受任何参数，但根据需要对对象进行设置。

编译器提供的构造函数被称为默认构造函数，但按照惯例，任何不接受参数的构造函数都是默认构造函数。这可能有些令人迷惑，但通常可以根据上下文清楚地判断默认构造函数的含义。

注意，如果你创建了任何构造函数，编译器将不会提供默认构造函数。因此，如果需要一个不接受参数的构造函数，且已经创建了其他构造函数，则必须自己添加默认构造函数。

10.5 重载构造函数

和所有成员函数一样，构造函数也可被重载。重载构造函数的功能非常强大和灵活。

例如，可以声明一个 Rectangle 类，它有两个构造函数：第一个接受长和宽作为参数，创建一个相应大小的矩形；第二个不接受参数，创建一个默认大小的矩形。程序清单 10.3 实现了这种想法。

程序清单 10.3 重载构造函数

```
1: // Listing 10.3 - Overloading constructors
```

```
 2:
 3: #include <iostream>
 4: using namespace std;
 5:
 6: class Rectangle
 7: {
 8:   public:
 9:     Rectangle();
10:     Rectangle(int width, int length);
11:     ~Rectangle() {}
12:     int GetWidth() const { return itsWidth; }
13:     int GetLength() const { return itsLength; }
14:   private:
15:     int itsWidth;
16:     int itsLength;
17: };
18:
19: Rectangle::Rectangle()
20: {
21:   itsWidth = 5;
22:   itsLength = 10;
23: }
24:
25: Rectangle::Rectangle (int width, int length)
26: {
27:   itsWidth = width;
28:   itsLength = length;
29: }
30:
31: int main()
32: {
33:   Rectangle Rect1;
34:   cout << "Rect1 width: " << Rect1.GetWidth() << endl;
35:   cout << "Rect1 length: " << Rect1.GetLength() << endl;
36:
37:   int aWidth, aLength;
38:   cout << "Enter a width: ";
39:   cin >> aWidth;
40:   cout << "\nEnter a length: ";
41:   cin >> aLength;
42:
43:   Rectangle Rect2(aWidth, aLength);
44:   cout << "\nRect2 width: " << Rect2.GetWidth() << endl;
45:   cout << "Rect2 length: " << Rect2.GetLength() << endl;
46:   return 0;
47: }
```

输出：

```
Rect1 width: 5
Rect1 length: 10
Enter a width: 20
```

```
Enter a length: 50

Rect2 width: 20
Rect2 length: 50
```

分析：

Rectangle 类是在第 6～17 行声明的，其中声明了两个构造函数：第 9 行的默认构造函数和第 10 行的第二个构造函数（它接受两个 int 参数）。

第 33 行使用默认构造函数创建了一个矩形，第 34 行和 35 行打印它的长和宽。第 38～41 行提示用户输入宽度和长度，第 43 行调用接受两个参数的构造函数。最后，第 44 和 45 行打印这个矩形的长度和宽度。

和其他重载函数一样，编译器将根据参数的数目和类型选择正确的构造函数。

10.6 初始化对象

到现在为止，我们一直是在构造函数体中设置对象的成员变量。然而，构造函数是分两个阶段调用的：初始化阶段和函数体执行阶段。

大多数变量可在任何一个阶段设置：在初始化阶段进行初始化或者在构造函数体中进行赋值。在初始化阶段初始化成员变量更清晰且效率通常更高。下面的例子演示了如何初始化成员变量：

```
Cat():          // constructor name and parameters
itsAge(5),      // initialization list
itsWeight(8)
{ }             // body of constructor
```

在构造函数的右括号后面加上一个冒号（:），然后是成员变量的名称和一对圆括号。圆括号内是用来初始化成员变量的表达式。如果需要初始化多个成员变量，用逗号将它们分开。

程序清单 10.4 列出程序清单 10.3 中构造函数的定义，这里在初始化部分对成员变量进行初始化，而不是在函数体中进行赋值。

程序清单 10.4 初始化成员变量的代码段

```
1:  // Listing 10.4 - Initializing Member Variables
2:  Rectangle::Rectangle():
3:      itsWidth(5),
4:      itsLength(10)
5:  {
6:  }
7:
8:  Rectangle::Rectangle (int width, int length):
9:      itsWidth(width),
10:     itsLength(length)
11: {
12: }
```

分析：

程序清单 10.4 只是一个代码段，因此没有输出。从第 2 行开始是默认构造函数。正如前面指出的，在标准函数头的后面加上了一个冒号，然后第 3 和 4 行将 itsWidth 和 itsLength 设置为默认值 5 和 10。

从第 8 行开始是第二个构造函数的定义。这个重载版本接受两个参数，第 9 和 10 行使用这些参数来设置类的成员。

有些变量必须始初化而不能赋值，如常量和引用。构造函数体中通常包含其他赋值语句或操作语句，然

而，最好尽可能使用初始化。

10.7　复制构造函数

除提供默认构造函数和析构函数外，编译器还提供一个默认复制构造函数。每当创建对象的拷贝时，都将调用复制构造函数。

正如第 9 章介绍的，按值传递对象时，无论是传入函数还是作为函数的返回值，都将创建该对象的一个临时拷贝。如果该对象是用户定义的对象，将调用相应类的复制构造函数。前一章的程序清单 9.6 说明了这一点。

所有复制构造函数都接受一个参数：指向其所属类的对象的引用。将该引用声明为 const 是个好主意，因为复制构造函数不会修改传入的对象。例如：

```
Cat(const Cat & theCat);
```

Cat 构造函数接受一个指向 Cat 对象的 const 引用。该复制构造函数旨在创建一个 theCat 拷贝。

默认复制构造函数将作为参数传入的对象的每个成员变量复制到新对象的成员变量中。这被称为成员复制（浅复制），这虽然对大多数成员变量来说是可行，但对于指向自由存储区中对象的指针成员变量不可行。

成员（浅）复制只是将对象成员变量的值复制到另外个对象中，两个对象中的指针最后将指向同一个内存块。深层复制将在堆中分配的值复制到新分配的内存中。

如果 Cat 类包含有一个成员变量 itsAge，它是一个指向自由存储区中 int 变量的指针，默认复制构造函数将穿入的 Cat 对象的 itsAge 成员变量的值，复制到新的 Cat 对象的 itsAge 成员变量中。这两个对象的 itsAge 成员变量指向同一个内存块，如图 10.1 所示。

当其中任何一个 Cat 对象不再在作用域中时，这将导致灾难性后果。如果原始 Cat 对象的析构函数释放了这个内存块，而新的 Cat 对象仍指向该内存块，将生成一个迷途指针，程序将处于致命的危险之中。图 10.2 说明了这种问题。

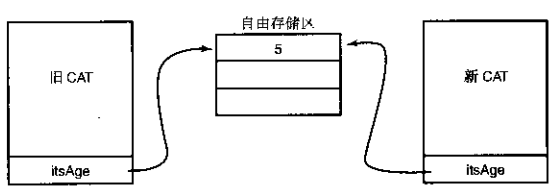

图 10.1　使用默认复制构造函数

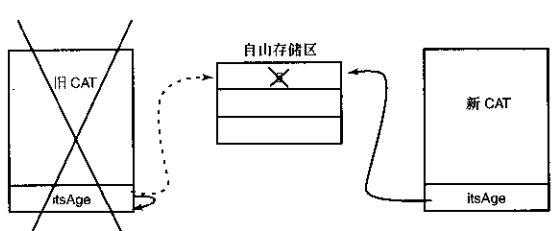

图 10.2　生成迷途指针

这种问题的解决方法是，创建自己的复制构造函数并根据需要分配内存。分配内存后，便可以将原来的值复制到新内存中，这被称为深层复制。程序清单 10.5 演示了如何完成这项工作。

程序清单 10.5　复制构造函数

```
1:  // Listing 10.5 - Copy constructors
2:
3:  #include <iostream>
4:  using namespace std;
5:
6:  class Cat
7:  {
8:    public:
9:      Cat();            // default constructor
10:     Cat (const Cat &); // copy constructor
11:     ~Cat();           // destructor
12:     int GetAge()  const   { return *itsAge; }
13:     int GetWeight() const  { return *itsWeight; }
14:     void SetAge(int age)   { *itsAge = age; }
15:
16:   private:
17:     int *itsAge;
18:     int *itsWeight;
19: };
20:
21: Cat::Cat()
22: {
23:    itsAge = new int;
24:    itsWeight = new int;
25:    *itsAge = 5;
26:    *itsWeight = 9;
27: }
28:
29: Cat::Cat(const Cat & rhs)
30: {
31:    itsAge = new int;
32:    itsWeight = new int;
33:    *itsAge = rhs.GetAge(); // public access
34:    *itsWeight = *(rhs.itsWeight); // private access
35: }
36:
37: Cat::~Cat()
38: {
39:    delete itsAge;
40:    itsAge = 0;
41:    delete itsWeight;
42:    itsWeight = 0;
43: }
44:
45: int main()
46: {
47:    Cat Frisky;
48:    cout << "Frisky's age: " << Frisky.GetAge() << endl;
49:    cout << "Setting Frisky to 6...\n";
50:    Frisky.SetAge(6);
51:    cout << "Creating Boots from Frisky\n";
52:    Cat Boots(Frisky);
```

```
53:    cout << "Frisky's age: " << Frisky.GetAge() << endl;
54:    cout << "Boots' age: " << Boots.GetAge() << endl;
55:    cout << "setting Frisky to 7...\n";
56:    Frisky.SetAge(7);
57:    cout << "Frisky's age: " << Frisky.GetAge() << endl;
58:    cout << "boot's age: " << Boots.GetAge() << endl;
59:    return 0;
60: }
```

输出：

```
Frisky's age: 5
Setting Frisky to 6...
Creating Boots from Frisky
Frisky's age: 6
Boots' age: 6
setting Frisky to 7...
Frisky's age: 7
Boots' age: 6
```

分析：

第 6～19 行声明了 Cat 类。注意，第 9 行声明了一个默认构造函数，第 10 行声明了一个复制构造函数。之所以知道第 10 行是一个复制构造函数，是因为它接受一个指向其所属类的对象的引用（这里为一个 const 引用）。

第 17 和 18 行声明了两个成员变量，其中每个变量都是一个 int 指针。通常，没有理由在类中使用 int 指针，这样做旨在演示如何管理自由存储区中的成员变量。

第 21～27 行的默认构造函数在自由存储中为两个 int 变量分配内存，并给它们赋值。

复制构造函数从第 29 行开始。请注意，参数是 rhs。通常将复制构造函数的参数命名为 rhs，rhs 表示 right hand side（右端）。从第 33 和 34 行的赋值语句可知，作为参数传入的对象位于等号的右边。该复制构造函数的工作原理如下：

第 31 和 32 行在自由存储区中分配内存；然后第 33 和 34 行将原有 Cat 对象中的值赋给新的内存单元。

参数 rhs 是一个作为 cosnt 引用传入复制构造函数的 Cat 对象。作为一个 Cat 对象，rhs 拥有与其他任何 Cat 对象相同的成员变量。

任何 Cat 对象都可以访问其他 Cat 对象的私有成员变量，但良好的做法是尽可能使用公有存取器方法。成员函数 rhs.GetAge( )返回存储在 rhs 的成员变量 itsAge 指向的内存块中的值。在实际的应用程序中，应该采用相同的方式来获取 itsWeight 的值：使用存取器方法。然而，第 34 行表明，同一个类的不同对象可以彼此访问对方的成员，这里直接使用对象 rhs 的私有成员 itsWeight 创建了一个拷贝。

图 10.3 描述了这里发生的情况。现有 Cat 对象的成员变量指向的值被复制到为新 Cat 对象分配的内存中。

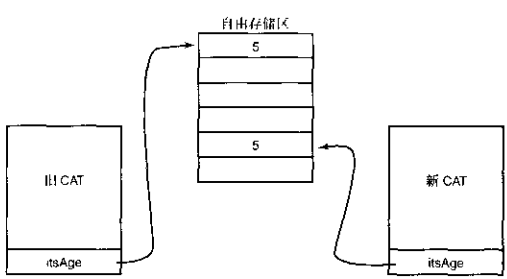

图 10.3　深层复制示意图

第 47 行创建了一个名叫 Frisky 的 Cat 对象，然后打印 Frisky 的年龄，并在第 50 行将其年龄设置为 6。

第 52 行使用复制构造函数和传入的 Frisky 新建了一个名为 Boots 的 Cat 对象。如果将 Frisky 作为参数按值（而不是按引用）传递给函数，编译器将创建相同的复制构造函数调用。

第 53 和 54 行打印这两个 Cat 对象的年龄。毫无疑问，Boots 的年龄和 Frisky 一样，也是 6，而不是默认值 5。第 56 行将 Frisky 的年龄设置为 7，然后再次打印年龄。这回 Frisky 的年龄为 7，但 Boots 的年龄仍为 6，这表明它们存储在不同的内存块中。

当 Cat 对象不再在作用域中时，它们的析构函数将自动被调用。第 37～43 行是 Cat 析构函数的实现。它将 delete 用于两个指针 itsAge 和 itsWeight，将分配的内存还给自由存储区；另外，为安全起见，将指针重新赋为空值。

10.8 运算符重载

C++有很多内置的类型，包括 int、float、char 等。每种类型都有很多内置的运算符，加法运算符（+）和乘法运算符（*）。C++也允许你在自己的类中支持这些运算符。

为全面探索运算符重载，程序清单 10.6 创建了一个新类 Counter。Counter 对象将用于循环计数（感到很奇怪吧！）和其他必须对某个数进行递增、递减或记录的应用程序中。

程序清单 10.6 Counter 类

```
1:  // Listing 10.6 - The Counter class
2:
3:  #include <iostream>
4:  using namespace std;
5:
6:  class Counter
7:  {
8:    public:
9:      Counter();
10:     ~Counter(){}
11:     int GetItsVal()const { return itsVal; }
12:     void SetItsVal(int x) {itsVal = x; }
13:
14:   private:
15:     int itsVal;
16: };
17:
18: Counter::Counter():
19: itsVal(0)
20: {}
21:
22: int main()
23: {
24:    Counter i;
25:    cout << "The value of i is " << i.GetItsVal() << endl;
26:    return 0;
27: }
```

输出：

```
The value of i is 0
```

分析：

实际上，这是一个没有什么实用价值的类。它是在第 6 到 16 行定义的，惟一的一个成员变量是 int 变量。

在第 9 行声明并在第 19 行实现的默认构造函数将惟一的成员变量 itsVal 初始化为 0。

不像真正的 int 变量，不能对 Counter 对象执行递增、递减、相加、赋值或其他运算。以此为代价使得打印它的值要困难得多。

10.8.1　编写一个递增函数

运算符重载可恢复这个类中已被精简掉的很多功能。例如，有两种方法可以让 Counter 对象能够递增。第一种方法是编写一个递增方法，如程序清单 10.7 所示。

程序清单 10.7　添加一个递增方法

```
1:  // Listing 10.7 - The Counter class
2:
3:  #include <iostream>
4:  using namespace std;
5:
6:  class Counter
7:  {
8:    public:
9:      Counter();
10:     ~Counter(){}
11:     int GetItsVal()const { return itsVal; }
12:     void SetItsVal(int x) {itsVal = x; }
13:     void Increment() { ++itsVal; }
14:
15:   private:
16:     int itsVal;
17: };
18:
19: Counter::Counter():
20: itsVal(0)
21: {}
22:
23: int main()
24: {
25:   Counter i;
26:   cout << "The value of i is " << i.GetItsVal() << endl;
27:   i.Increment();
28:   cout << "The value of i is " << i.GetItsVal() << endl;
29:   return 0;
30: }
```

输出：

```
The value of i is 0
The value of i is 1
```

分析：

程序清单 10.7 添加了一个 Increment 函数，它是在第 13 行定义的。虽然这样做可行，但使用起来却很麻烦。程序总需增加++运算符，当然这是可以实现的。

10.8.2　重载前缀运算符

通过声明下述形式的函数，可以重载前缀运算符：

```
returnType operator op ()
```

其中 *op* 是要重载的运算符。因此，可以采用下面的语法来重载++运算符：

```
void operator++()
```

该语句指出，要重载运算符++，它不返回任何值，因此返回类型为 void。程序清单 10.8 演示了这种解决方案。

程序清单 10.8　重载运算符++

```
1:  // Listing 10.8 - The Counter class
2:  // prefix increment operator
3:
4:  #include <iostream>
5:  using namespace std;
6:
7:  class Counter
8:  {
9:   public:
10:    Counter();
11:    ~Counter(){}
12:    int GetItsVal()const { return itsVal; }
13:    void SetItsVal(int x) {itsVal = x; }
14:    void Increment() { ++itsVal; }
15:    void operator++ () { ++itsVal; }
16:
17:  private:
18:    int itsVal;
19: };
20:
21: Counter::Counter():
22: itsVal(0)
23: {}
24:
25: int main()
26: {
27:    Counter i;
28:    cout << "The value of i is " << i.GetItsVal() << endl;
29:    i.Increment();
30:    cout << "The value of i is " << i.GetItsVal() << endl;
31:    ++i;
32:    cout << "The value of i is " << i.GetItsVal() << endl;
33:    return 0;
34: }
```

输出：

```
The value of i is 0
The value of i is 1
The value of i is 2
```

分析：

第 15 行重载了运算符++，从中可知，该重载的运算符只是将私有成员 itsVal 的值递增。第 31 行使用了这个重载的运算符，其语法与 int 等内置类型的语法非常接近。

此时，读者可能考虑为创建的 Counter 添加其他功能，如检测 Counter 是否超过最大可能取值。然而，前面编写的递增运算符存在一个明显的缺陷。如果将 Counter 对象放在赋值语句的右边，将不管用。例如：

```
Counter a = ++i;
```

上述代码企图创建一个新的 Counter 对象 a，然后将 *i* 递增并将结果赋给 a。内置的复制构造函数将处理这种赋值，但当前的递增运算符不返回一个 Counter 对象，其返回类型为 void。不能将空值（void）赋给任何东西，包括 Counter 对象（你不能无中生有）。

10.8.3　运算符重载函数的返回类型

显然，需要返回一个 Counter 对象，以便可以将其赋给另一个 Counter 对象。应返回哪一个对象呢？一种方法是创建一个临时对象并将其返回。程序清单 10.9 演示了这种方法。

程序清单 10.9　返回一个临时对象

```
// Listing 10.9 - operator++ returns a temporary object

#include <iostream>

using namespace std;

class Counter
{
  public:
    Counter();
    ~Counter(){}
    int GetItsVal()const { return itsVal; }
    void SetItsVal(int x) { itsVal = x; }
    void Increment() { ++itsVal; }
    Counter operator++ ();

  private:
    int itsVal;

};

Counter::Counter():
itsVal(0)
{}

Counter Counter::operator++()
{
   ++itsVal;
   Counter temp;
   temp.SetItsVal(itsVal);
   return temp;
}

int main()
{
   Counter i;
   cout << "The value of i is " << i.GetItsVal() << endl;
   i.Increment();
   cout << "The value of i is " << i.GetItsVal() << endl;
   ++i;
```

```
41:    cout << "The value of i is " << i.GetItsVal() << endl;
42:    Counter a = ++i;
43:    cout << "The value of a: " << a.GetItsVal();
44:    cout << " and i: " << i.GetItsVal() << endl;
45:    return 0;
46: }
```

输出：

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

分析：

在这个版本中，operator++是在第 15 行声明和第 26～32 行定义的。这个版本被声明为返回一个 Counter 对象。第 29 行创建了一个临时变量 temp，它的值被设置为与当前对象相同。在第 42 行，返回的临时变量立刻被赋给 a。

10.8.4 返回无名临时对象

实际上，没有必要对第 29 行创建的临时对象进行命名。如果 Counter 有一个接受一个参数的构造函数，只需将该构造函数的结果作为递增运算符的返回值即可。程序清单 10.10 演示了这种想法。

程序清单 10.10 返回一个无名临时对象

```
1: // Listing 10.10 - operator++ returns a nameless temporary object
2:
3: #include <iostream>
4:
5: using namespace std;
6:
7: class Counter
8: {
9:   public:
10:    Counter();
11:    Counter(int val);
12:    ~Counter(){}
13:    int GetItsVal()const { return itsVal; }
14:    void SetItsVal(int x) {itsVal = x; }
15:    void Increment() { ++itsVal; }
16:    Counter operator++ ();
17:
18:  private:
19:    int itsVal;
20: };
21:
22: Counter::Counter():
23: itsVal(0)
24: {}
25:
26: Counter::Counter(int val):
27: itsVal(val)
28: {}
```

```
29:
30: Counter Counter::operator++()
31: {
32:    ++itsVal;
33:    return Counter (itsVal);
34: }
35:
36: int main()
37: {
38:    Counter i;
39:    cout << "The value of i is " << i.GetItsVal() << endl;
40:    i.Increment();
41:    cout << "The value of i is " << i.GetItsVal() << endl;
42:    ++i;
43:    cout << "The value of i is " << i.GetItsVal() << endl;
44:    Counter a = ++i;
45:    cout << "The value of a: " << a.GetItsVal();
46:    cout << " and i: " << i.GetItsVal() << endl;
47:    return 0;
48: }
```

输出：

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

分析：

第 11 行声明了一个新的构造函数，它接受一个 int 参数。其实现位于第 26～28 行。它将 itsVal 初始化为传入的值。

operator++的实现被简化了。第 32 行对 itsVal 进行递增。第 33 行创建一个临时 Counter 对象，并将其初始化为 itsVal 的值，然后将其作为 operator++的结果返回。

这种方法更精致，但提出了一个问题：为什么要创建临时对象？别忘了，每个临时对象都必须创建，然后销毁，这些操作的开销都可能很高。另外，对象 *i* 已经存在且有正确的值，为什么不将它返回呢？这种问题可以使用 this 指针来解决。

10.8.5　使用 this 指针

this 指针被传递给所有成员函数，包括诸如 operator++()等重载运算符。在前面创建的程序清单中，this 指针指向 *i*，如果对其解除引用，将返回对象 *i*，而 *i* 的成员变量 itsVal 中有正确的值。程序清单 10.11 对 this 指针解除引用，并将结果返回，以避免创建不必要的临时对象。

程序清单 10.11　返回 this 指针

```
1: // Listing 10.11 - Returning the dereferenced this pointer
2:
3: #include <iostream>
4:
5: using namespace std;
6:
7: class Counter
8: {
```

```
 9:   public:
10:     Counter();
11:     ~Counter(){}
12:     int GetItsVal()const { return itsVal; }
13:     void SetItsVal(int x) {itsVal = x; }
14:     void Increment() { ++itsVal; }
15:     const Counter& operator++ ();
16:
17:   private:
18:     int itsVal;
19: };
20:
21: Counter::Counter():
22: itsVal(0)
23: {};
24:
25: const Counter& Counter::operator++()
26: {
27:   ++itsVal;
28:   return *this;
29: }
30:
31: int main()
32: {
33:   Counter i;
34:   cout << "The value of i is " << i.GetItsVal() << endl;
35:   i.Increment();
36:   cout << "The value of i is " << i.GetItsVal() << endl;
37:   ++i;
38:   cout << "The value of i is " << i.GetItsVal() << endl;
39:   Counter a = ++i;
40:   cout << "The value of a: " << a.GetItsVal();
41:   cout << " and i: " << i.GetItsVal() << endl;
42:   return 0;
43: }
```

输出：

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

分析：

第 26～30 行的 operator++实现被修改成对 this 指针解除引用以返回当前对象。这提供了一个 Counter 对象，以便赋给 a。正如前面讨论过的，如果 Counter 对象分配了内存，则覆盖复制构造函数至关重要。在这个例子中，默认复制构造函数是可行的。

注意，返回的值是一个 Counter 引用，因而避免了创建额外的临时对象。由于使用返回的 Counter 对象的函数不应修改它，因此将该引用声明为 const。

返回的 Counter 必须是 const，否则将可以对其执行修改操作。例如，如果返回的值不是常量，则可能这样编写第 39 行：

```
Counter a = ++++i;
```

这行代码的意思是，将递增运算符（++）用于++i 的结果。这将导致对 i 使用递增运算符两次，这样的操作是应该禁止的。

请尝试这样做：将声明和实现（第 15 行和 25 行）中的返回值修改为非 const，将第 39 行修改为如上所示。在调试器中，在第 39 行设置一个断点，然后步进执行。读者将发现执行了递增运算符两次，对返回值（现在不是 const 的）进行了递增。

为禁止这种情况发生，将返回值声明为 const 的。如果将第 15 行和 25 行的返回值恢复为 const 的，但保留第 39 行不变，编译器将指出不能对 const 对象应用递增运算符。

10.8.6　重载后缀运算符

至此，你重载了前缀运算符。如果要重载后缀递增运算符，该如何办呢？编译器将遇到一个问题：如何区分前缀和后缀？根据约定，在声明后缀运算符函数时，提供一个 int 参数。该参数的值被忽略，它只是一个信号，指出这是后缀运算符。

10.8.7　前缀和后缀之间的差别

编写后缀运算符函数之前，必须理解前缀运算符和后缀运算符之间的差别。第 4 章对此做了详细介绍（参见程序清单 4.3）。

这里回顾一下，前缀运算符的意思是，先递增后使用；而后缀的意思是，先使用后递增。

前缀运算符只需递增然后返回对象本身，而后缀运算符必须返回递增前的值。为此，必须创建一个临时对象来存储原来的值，然后将原始对象的值递增，再返回临时对象。

再次复习一遍。请看下面这行代码：

```
a = x++;
```

如果 x 为 5，则执行这条语句后，a 为 5，但 x 为 6。也就是说，先返回 x 的值并将其赋给 a，然后再将 x 递增。如果 x 是一个对象，则其后缀递增运算符必须将原来的值存储到一个临时对象中，将 x 递增到 6，然后返回临时对象，以便将原来的其值赋给 a。

注意，由于返回的是临时对象，因此必须按值而不是按引用返回它，因为函数返回后，临时对象将不在作用域中。

程序清单 10.12 演示了前缀和后缀运算符的用法。

程序清单 10.12　前缀和后缀运算符

```
1:  // Listing 10.12 - Prefix and Postfix operator overloading
2:
3:  #include <iostream>
4:
5:  using namespace std;
6:
7:  class Counter
8:  {
9:    public:
10:     Counter();
11:     ~Counter(){}
12:     int GetItsVal()const { return itsVal; }
13:     void SetItsVal(int x) {itsVal = x; }
14:     const Counter& operator++ ();   // prefix
15:     const Counter operator++ (int); // postfix
16:
17:   private:
```

```
18:     int itsVal;
19: };
20:
21: Counter::Counter():
22: itsVal(0)
23: {}
24:
25: const Counter& Counter::operator++()
26: {
27:   ++itsVal;
28:   return *this;
29: }
30:
31: const Counter Counter::operator++(int theFlag)
32: {
33:   Counter temp(*this);
34:   ++itsVal;
35:   return temp;
36: }
37:
38: int main()
39: {
40:   Counter i;
41:   cout << "The value of i is " << i.GetItsVal() << endl;
42:   i++;
43:   cout << "The value of i is " << i.GetItsVal() << endl;
44:   ++i;
45:   cout << "The value of i is " << i.GetItsVal() << endl;
46:   Counter a = ++i;
47:   cout << "The value of a: " << a.GetItsVal();
48:   cout << " and i: " << i.GetItsVal() << endl;
49:   a = i++;
50:   cout << "The value of a: " << a.GetItsVal();
51:   cout << " and i: " << i.GetItsVal() << endl;
52:   return 0;
53: }
```

输出：

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
The value of a: 3 and i: 4
```

分析：

后缀运算符是在第 15 行声明的，其实现位于第 31～36 行。前缀运算符是在第 14 行声明的。

在第 32 行，传递给后缀运算符的参数用作一个信号，指出这是后缀运算符，其值没有被使用。

10.8.8 重载双目数学运算符

递增运算符（++）是单目运算符，只作用于一个对象。加法运算符（+）是双目运算符，它接受两个对象（一个为当前对象，另一个为其他类对象）。显然，重载诸如加法（+）、减法（-）、乘法（*）、除法（/）和求模（%）等运算符的方式不同于重载前缀和后缀运算符。来考虑如何为 Counter 重载运算符+。

这里的目标是，能够声明两个 Counter 变量，然后将它们相加，如下例所示：

```
Counter varOne, varTwo, varThree;
VarThree = VarOne + VarTwo;
```

同样，可以编写函数 Add()，它接受一个 Counter 参数，将该 Counter 对象和当前对象的值相加，然后返回一个值为相加结果的 Counter 对象。程序清单 10.13 演示了这种方法。

程序清单 10.13　Add()函数

```
1:  // Listing 10.13 - Add function
2:
3:  #include <iostream>
4:
5:  using namespace std;
6:
7:  class Counter
8:  {
9:    public:
10:     Counter();
11:     Counter(int initialValue);
12:     ~Counter();
13:     int GetItsVal()const { return itsVal; }
14:     void SetItsVal(int x) { itsVal = x; }
15:     Counter Add(const Counter &);
16:
17:   private:
18:     int itsVal;
19: };
20:
21: Counter::Counter(int initialValue):
22: itsVal(initialValue)
23: {}
24:
25: Counter::Counter():
26: itsVal(0)
27: {}
28:
29: Counter Counter::Add(const Counter & rhs)
30: {
31:   return Counter(itsVal+ rhs.GetItsVal());
32: }
33:
34: int main()
35: {
36:   Counter varOne(2), varTwo(4), varThree;
37:   varThree = varOne.Add(varTwo);
38:   cout << "varOne: " << varOne.GetItsVal()<< endl;
39:   cout << "varTwo: " << varTwo.GetItsVal() << endl;
40:   cout << "varThree: " << varThree.GetItsVal() << endl;
41:
42:  return 0;
43: }
```

输出：

```
varOne: 2
varTwo: 4
varThree: 6
```

分析：

函数 Add( )是在第 15 行声明的。它接受一个 const Counter 引用参数，这是要将其与当前对象相加的数值。它返回一个 Counter 对象，如第 37 行所示，这个 Counter 对象将被赋给赋值语句左边的变量。也就是说，VarOne 是当前对象，VarTwo 是函数 Add( )的参数，结果将被赋给 VarThree。

为创建 varThree 而又不给它赋初值，需要一个默认构造函数。该默认构造函数将 itsVal 初始化为 0，如第 25～27 行所示。由于 varOne 和 varTwo 必须被初始化为非零值，因此创建了另一个构造函数，如第 21～23 行所示。这个问题的另一种解决方法是，为第 11 行声明的构造函数的参数指定默认值 0。

第 29～32 行是函数 Add( )本身。它管用，但使用起来不自然。

重载加法运算符（operator+）

通过重载加法运算符，可使 Counter 类使用起来更自然。前面介绍过，要重载运算符，可使用如下结构：

```
returnType operator op ()
```

程序清单 10.14 演示了如何重载加法运算符。

程序清单 10.14　operator+

```
 1: // Listing 10.14 - Overload operator plus (+)
 2:
 3: #include <iostream>
 4:
 5: using namespace std;
 6:
 7: class Counter
 8: {
 9:   public:
10:     Counter();
11:     Counter(int initialValue);
12:     ~Counter();
13:     int GetItsVal()const { return itsVal; }
14:     void SetItsVal(int x) {itsVal = x; }
15:     Counter operator+ (const Counter &);
16:   private:
17:     int itsVal;
18: };
19:
20: Counter::Counter(int initialValue):
21: itsVal(initialValue)
22: {}
23:
24: Counter::Counter():
25: itsVal(0)
26: {}
27:
28: Counter Counter::operator+ (const Counter & rhs)
29: {
```

```
30:    return Counter(itsVal + rhs.GetItsVal());
31: }
32:
33: int main()
34: {
35:    Counter varOne(2), varTwo(4), varThree;
36:    varThree = varOne + varTwo;
37:    cout << "varOne: " << varOne.GetItsVal()<< endl;
38:    cout << "varTwo: " << varTwo.GetItsVal() << endl;
39:    cout << "varThree: " << varThree.GetItsVal() << endl;
40:
41:    return 0;
42: }
```

输出：

```
varOne: 2
varTwo: 4
varThree: 6
```

分析：

operator+是在第 15 行声明，在第 28～31 行定义的。

如果将其与前一个程序清单中 Add( )函数的声明和定义进行比较，将发现它们几乎相同。然而，它们使用的语法却大相径庭。下面的语句：

```
VarThree = VarOne + VarTwo;
```

比下面的语句更自然：

```
VarThree = VarOne.Add(VarTwo);
```

变化并不大，但足以使程序更容易使用和理解。

第 36 行使用了该运算符：

```
36:    varThree = varOne + VarTwo;
```

编译器将上述代码转换为：

```
VarThree = varOne.Operator + (VarTwo);
```

当然，也可直接编写成这样，编译器将乐意接受。

将对左边的操作数调用 operator+方法，并将右边的操作数作为参数传递给它。

10.8.9　运算符重载中存在的问题

正如本章描述的，重载的运算符可以是成员函数，也可以是非成员函数；后者将在第 15 章讨论友元函数时介绍。

必须是类成员的运算符包括赋值运算符（=）、下标运算符（[]）、函数调用运算符（()）和间接运算符（->）。运算符[]将在第 13 章介绍数组时讨论；重载运算符->将在第 15 章讨论智能指针（smart pointer）时介绍。

10.8.10　对运算符重载的限制

用于内置类型（如 int）的运算符不能重载，运算符的优先级和目数（单目还是双目）不能改变。不能创建新的运算符，因此不能把**声明为“乘方”运算符。

运算符的目数指的是运算符使用多少项。有些 C++运算符是单目的，只使用一项（如 myValue++）；有些运算符是双目的，使用两项，如（a+b）；只有一个运算符是三目的，使用三项。运算符（?）是 C++中惟一一个三目运算符，因而通常被称为三目运算符（a > b ? x : y）。

10.8.11 重载什么

运算符重载是新 C++程序员最容易滥用的功能之一。为一些较为模棱两可的运算符创建令人感兴趣的新用法很有吸引力，但这样做总是会导致代码令人迷惑和难以理解。

当然，将相加运算符（+）重载为相减，将相乘运算符重载为相加很有趣，但任何专业程序员都不会这样做。这种奇怪但怪异的运算符用法隐含更大的风险：使用+表示将一系列字母拼接起来或将字符串分拆。也许充分的理由考虑这些用法，但谨慎从事的理由更充分。别忘了，重载运算符旨在让程序更容易使用和理解。

应该：

当运算符重载能使程序清晰明了时，应该这样做。

务必从重载的运算符返回一个其所属类的对象。

不应该：

不要创建不直观的运算符行为。

不要将前缀和后缀运算符混为一谈，尤其是进行重载时。

10.8.12 赋值运算符

如果你没有创建，编译器提供的第 4 个也是最后一个函数是赋值运算符（operator=()）。每当给对象赋值时都将调用这个运算符。例如：

```
Cat catOne(5,7);
Cat catTwo(3,4);
// ... other code here
catTwo = catOne;
```

上述代码创建 catOne，并将其初始化为 itsAge 和 itsWeight 分别等于 5 和 7。然后创建 catTwo，并将 3 和 4 赋给其成员变量。

然后，将 catOne 的值赋给 catTwo。这提出了两个问题：如果 itsAge 是指针将如何？catTwo 的原始值将发生什么变化？

前面介绍复制构造函数时，已经讨论如何处理其值存储在自由存储区中的成员变量。这里出现的问题与图 10.1 和图 10.2 演示的相同。

C++程序员区分浅复制（成员复制）和深层复制。浅复制只是复制成员，两个对象中的指针将指向自由存储区中的同一个区域；而深层复制分配必要的内存，如图 10.3 所示。

然而，对于赋值运算符，还存在另一个问题。对象 catTwo 已经存在且已经分配了内存。为避免内存泄漏，必须将该内存块释放。但如果将 catTwo 赋给它自己情况将如何呢？

```
catTwo = catTwo;
```

没有人会故意这样做，但当引用和解除引用的指针掩盖了自己给自己赋值的情况时，就有可能发生这样的事情。

如果不认真处理这种问题，catTwo 将释放分配给自己的内存。然后，从赋值运算符右边指向的内存中复制值时，将出现大问题：值不见了。

为防止这种情况发生，赋值运算符必须检查右边是否是对象本身，这是通过检查 this 指针实现的。程序清单 10.15 是一个重载了赋值运算符的类。

程序清单 10.15 赋值运算符

```
1:  // Listing 10.15 - Copy constructors
2:
3:  #include <iostream>
```

```
 4:
 5: using namespace std;
 6:
 7: class Cat
 8: {
 9:   public:
10:     Cat();            // default constructor
11:     // copy constructor and destructor elided!
12:     int GetAge() const { return *itsAge; }
13:     int GetWeight() const { return *itsWeight; }
14:     void SetAge(int age) { *itsAge = age; }
15:     Cat & operator=(const Cat &);
16:
17:   private:
18:     int *itsAge;
19:     int *itsWeight;
20: };
21:
22: Cat::Cat()
23: {
24:    itsAge = new int;
25:    itsWeight = new int;
26:    *itsAge = 5;
27:    *itsWeight = 9;
28: }
29:
30:
31: Cat & Cat::operator=(const Cat & rhs)
32: {
33:    if (this == &rhs)
34:       return *this;
35:    *itsAge = rhs.GetAge();
36:    *itsWeight = rhs.GetWeight();
37:    return *this;
38: }
39:
40:
41: int main()
42: {
43:    Cat Frisky;
44:    cout << "Frisky's age: " <<Frisky.GetAge() <<endl;
45:    cout << "Setting Frisky to 6...\n";
46:    Frisky.SetAge(6);
47:    Cat Whiskers;
48:    cout << "Whiskers's age: " << Whiskers.GetAge() << endl;
49:    cout << "copying Frisky to Whiskers...\n";
50:    Whiskers = Frisky;
51:    cout << "Whiskers' age: " << Whiskers.GetAge() << endl;
52:    return 0;
53: }
```

输出：

```
frisky's age: 5
```

```
Setting frisky to 6...
Whiskers' age: 5
copying frisky to whiskers...
whiskers' age: 6
```

分析：

程序清单 10.15 再次用到了 Cat 类，但为节省篇幅，删除了复制构造函数和析构函数。该程序清单新增的内容是，第 15 行中的赋值运算符方法声明，该方法将用于重载赋值运算符。第 31~38 行定义了这个的方法。

第 33 行检测当前对象（目标 Cat 对象）是否与源 Cat 对象相同，这是通过检查右边的 Cat 对象（rhs）的地址是否与存储在 this 指针中的地址相同实现的。如果它们相同，则什么也不用做，因为左边和右边的对象是同一个对象。因此，第 34 行返回当前对象。

如果右边的对象和左边的对象不是同一个对象，则第 35 和 36 行复制成员，然后返回 this 指针指向的对象。

在上程序中，第 50 行使用赋值运算符将名为 Frisky 的 Cat 对象赋给名为 Whiskers 的 Cat 对象。对于该程序清单中的其他代码，读者应该很熟悉。

该程序清单做了这样的假设：如果两个对象的地址相同，则它们必然是同一个对象。当然，也可以重载相等运算符（==），让你能够决定对象相等意味着什么。

10.9 处理数据类型转换

读者知道如何将一个对象赋给同一种类型的另一个对象后，来看另一种情形。如果试图将一个内置类型（如 int 或 unsigned short）的变量赋给一个用户定义的类（如本章前面创建的 Counter）对象，将发生什么情况呢？程序清单 10.16 试图将一个 int 值赋给一个 Counter 对象。

注意：程序清单 10.16 不能通过编译！

程序清单 10.16 试图将一个 int 值赋给一个 Counter 对象

```
1:  // Listing 10.16 - This code won't compile!
2:
3:  #include <iostream>
4:
5:  using namespace std;
6:
7:  class Counter
8:  {
9:    public:
10:     Counter();
11:     ~Counter(){}
12:     int GetItsVal()const { return itsVal; }
13:     void SetItsVal(int x) {itsVal = x; }
14:   private:
15:     int itsVal;
16: };
17:
18: Counter::Counter():
19: itsVal(0)
20: {}
```

```
21:
22: int main()
23: {
24:    int theInt = 5;
25:    Counter theCtr = theInt;
26:    cout << "theCtr: " << theCtr.GetItsVal() << endl;
27:    return 0;
28: }
```

输出：

```
Compiler error! Unable to convert int to Counter
```

分析：

在第 7～16 行声明的 Counter 类只有一个默认构造函数，它没有声明将任何内置类型转换为 Counter 对象的方法。在 main()中，第 24 行声明了一个 int 变量，然后将其赋给一个 Counter 对象，然而这种赋值将导致错误。除非你告诉编译器，否则它不知道应该将 int 变量的值赋给成员变量 itsVal。

程序清单 10.17 通过创建一个转换构造函数修复了这种错误，该构造函数接受一个 int 参数，并创建一个 Counter 对象。

程序清单 10.17　将 int 转换为 Counter

```
 1: // Listing 10.17 - Constructor as conversion operator
 2:
 3: #include <iostream>
 4:
 5: using namespace std;
 6:
 7: class Counter
 8: {
 9:   public:
10:     Counter();
11:     Counter(int val);
12:     ~Counter(){}
13:     int GetItsVal()const { return itsVal; }
14:     void SetItsVal(int x) { itsVal = x; }
15:   private:
16:     int itsVal;
17: };
18:
19: Counter::Counter():
20: itsVal(0)
21: {}
22:
23: Counter::Counter(int val):
24: itsVal(val)
25: {}
26:
27: int main()
28: {
29:    int theInt = 5;
30:    Counter theCtr = theInt;
31:    cout << "theCtr: " << theCtr.GetItsVal() << endl;
```

```
32:    return 0;
33: }
```

输出：

```
theCtr: 5
```

分析：

最重要的修改是第 11 行和第 23～25 行。第 11 行重载构造函数，使其接受一个 int 参数；第 23～25 行实现这个构造函数。这个构造函数的作用是，使用一个 int 变量创建一个 Counter 对象。

经过上述修改后，编译器就能调用接受一个 int 参数的构造函数。具体情况如下：

第 1 步：创建一个名为 theCtr 的 Counter 对象。

这就像代码 int x = 5；将创建一个 int 变量 x，然后将其初始化为 5 一样。在这里，创建一个 Counter 对象 theCtr，并用 int 变量 theInt 对其进行初始化。

第 2 步：将 theInt 的值赋给 theCtr。

然而，theInt 是一个 Int 变量，并不是 Counter 对象！因此，首先必须将其转换为一个 Counter 对象。编译器将自动尝试进行某种转换，但你必须告诉它如何转换，这是通过创建一个转换构造函数来实现的。在这个例子中，需要为 Counter 创建一个只接受一个 int 参数的构造函数：

```
class Counter
{
   Counter ( int x);
// ..
};
```

这个构造函数根据一个 int 值来创建一个 Counter 对象。它是通过创建一个临时无名 Counter 对象来实现的。为方便说明，假设根据 int 值创建的临时 Counter 对象名为 wasInt。

第 3 步：将 wasShort 赋给 theCtr，这与下述代码等价：

```
theCtr = wasInt;
```

在这一步中，用 wasInt（运行构造函数时创建的临时对象）替换赋值运算符右边的变量。也就是说，编译器为你创建一个临时 Counter 对象后，用它来初始化 theCtr。

为理解这一点，必须知道所有运算符重载的工作原理都相同：使用关键字 operator 来声明重载的运算符。对于双目运算符（如=或+），右边的变量将成为参数。这种工作是由编译器完成的。因此：

```
a = b;
```

将变成：

```
a.operator=(b);
```

然而，如果试图像下面这样进行反向赋值，将发生什么情况呢？

```
1: Counter theCtr(5);
2: int theInt = theCtr;
3: cout << "theInt : " << theInt  << endl;
```

同样，这将导致编译错误。虽然编译器知道如何根据一个 int 变量来创建一个 Counter 对象，但它并不知道如何进行相反的操作。

10.10 转换运算符

为解决将类对象转换为其他类型的问题，C++提供了转换运算符，可以将它们加入到你的类中。这让你的类能够指定如何执行到内置类型的隐式转换。程序清单 10.18 演示了这一点。然而，需要注意的一点是，转换运算符没有指定返回值，虽然它们确实返回一个转换后的值。

程序清单 10.18　将 Counter 转换为 unsigned short

```
1:  // Listing 10.18 - Conversion Operators
2:  #include <iostream>
3:
4:  class Counter
5:  {
6:    public:
7:      Counter();
8:      Counter(int val);
9:      ~Counter(){}
10:     int GetItsVal()const { return itsVal; }
11:     void SetItsVal(int x) {itsVal = x; }
12:     operator unsigned int();
13:   private:
14:     int itsVal;
15: };
16:
17: Counter::Counter():
18: itsVal(0)
19: {}
20:
21: Counter::Counter(int val):
22: itsVal(val)
23: {}
24:
25: Counter::operator unsigned int ()
26: {
27:   return ( int (itsVal) );
28: }
29:
30: int main()
31: {
32:   Counter ctr(5);
33:   int theInt = ctr;
34:   std::cout << "theInt: " << theInt << std::endl;
35:   return 0;
36: }
```

输出：

```
theShort: 5
```

分析：

第 12 行声明了转换运算符。注意，该声明以关键字 operator 打头，且没有返回值。该函数的实现位于第 25～28 行。第 27 行返回将 itsVal 转换为 int 的值。

现在，编译器知道如何在 int 和 Counter 对象之间进行转换了，且将在它们之间相互赋值。

10.11　小　　结

本章介绍了如何重载类的成员函数，还介绍了如何给函数提供默认值以及何时使用默认值和何时使用重载。

通过重载类构造函数，可以创建灵活的类，这种类可以根据其他对象创建自己的对象。对象的初始化发生在构造函数的初始化阶段，与在构造函数体中进行赋值相比，初始化的效率更高。

如果你没有创建复制构造函数和赋值运算符，编译器将为你提供，但它们执行成员复制。在成员数据为指向自由存储区中内存的指针的类中，必须覆盖编译器提供的这些方法，以便能够给成员变量分配内存。

几乎所有的 C++运算符能够被重载，但必须小心以防创建用法不直观的运算符重载函数。不能改变运算符的目数，也不能创建新的运算符。

this 指针指向当前对象，它是所有成员函数的一个隐含参数。重载运算符常常对 this 指针解除引用，并返回其结果，以便重载运算符能够用于表达式中。

转换运算符能够创建这样的类，即可用于需要其他类型对象的表达式中。它们没有遵循所有函数都返回一个显式值这条规则；与构造函数和析构函数一样，它们没有返回类型。

10.12 问 与 答

问：为什么在能够重载函数时还使用默认值？

答：维护一个函数比维护两个更容易；同时，与学习两个函数体相比，理解一个带默认参数的函数通常更容易。另外，修改一个函数而忘记修改另一个函数是导致错误的常见原因。

问：鉴于重载函数存在的问题，为什么不总是使用默认值呢？

答：重载函数提供了默认变量不能提供的功能，如通过改变参数的类型而不仅仅是数目来改变参数列表，为不同的参数类型组合提供不同的实现。

问：编写类构造函数时，如何决定什么应放在初始化部分中，什么应放在构造函数体中？

答：一条简单的经验规则是，尽可能在初始化阶段多做些工作，即初始化所有的成员变量。有些工作（如计算和打印语句）必须放在构造函数体中。

问：重载的函数可以有默认参数吗？

答：可以。任何重载的函数都可以有默认参数，只要遵循常规函数的默认参数规则。

问：为什么有些成员函数在类声明中定义而有些不是呢？

答：在声明中定义成员函数的实现使其成为内联的。一般而言，仅当函数特别简单时才这样做。注意，即使成员函数是在类声明外定义的，也可以使用关键字 inline 来使其成为内联的。

10.13 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。

10.13.1 测验

1. 重载成员函数时，它们必须在哪些方面有所不同？
2. 声明和定义之间有何区别？
3. 什么时候调用复制构造函数？
4. 什么时候调用析构函数？
5. 复制构造函数和赋值运算符（=）之间有何不同？
6. 什么是 this 指针？
7. 如何区分重载的前缀和后缀递增运算符？
8. 可以为 short int 重载 operator+吗？
9. 在 C++中，重载 operator++使其对类中成员变量的值递减合法吗？

10. 在转换运算符的声明中必须有什么样的返回值？

10.13.2　练习

1. 编写一个 SimpleCircle 类声明，它只有一个成员变量：itsRadius。提供一个默认构造函数、一个析构函数和 itsRadias 的存取器函数。

2. 为练习 1 中创建的类的默认构造函数编写实现，将 itsRadius 初始化为 5。请在构造函数的初始化部分而不是函数体中完成这项工作。

3. 给练习 1 中创建的类添加另一个构造函数，该函数接受一个值作为参数，并将它赋给 itsRadius。

4. 为 SimpleCircle 类创建一个前缀递增运算符和一个后缀递增运算符，对 itsRadius 进行递增。

5. 修改 SimpleCircle 类，将 itsRadius 存储在自由存储区中，并相应地修改原来的方法。

6. 给 SimpleCircle 提供一个复制构造函数。

7. 给 SimpleCircle 提供一个赋值运算符。

8. 编写一个创建两个 SimpleCircle 对象的程序，其中一个是使用默认构造函数的，另一个被实例化为 9。对每个对象调用递增运算符函数，然后打印它们的值。最后，将第二个对象赋给第一个，并打印它们的值。

9. 查错：下述赋值运算符的实现有什么错误？

```
SQUARE SQUARE ::operator=(const SQUARE & rhs)
{
   itsSide = new int;
   *itsSide = rhs.GetSide();
   return *this;
}
```

10. 查错：下述 operator+的实现有什么错误？

```
VeryShort  VeryShort::operator+ (const VeryShort& rhs)
{
     itsVal += rhs.GetItsVal();
     return *this;
}
```