Using Member Functionsto Overload Unary Operators

Using Member Functionsto Overload Unary Operators

You may also overload unary operators, such as ++, – –, or the unary – or +. As stated earlier, when a unary operator is overloaded by means of a member function, no object is explicitly passed to the operator function. Instead, the operation is performed on the object that generates the call to the function through the implicitly passed this pointer.For example, here is an expanded version of the previous example program. This version defines the increment operation for objects of type three_d.

```
// Overload a unary operator.
#include <iostream.h>
class three_d
{
      int x, y, z; // 3-D coordinates
public:
      three_d() { x = y = z = 0; }
      three_d(inti, int j, int k)
      {x = i; y = j; z = k; }
      three_d operator+(three_d op2);
      three_d operator=(three_d op2);
      three_d operator++(); // prefix version of ++
      void show() ;
} ;
// Overload +.
three_dthree_d::operator+(three_d op2)
{
   three_d temp;
   temp.x = x + op2.x; // These are integer additions
   temp.y = y + op2.y;// and the + retains its original
   temp.z = z + op2.z; // meaning relative to them.
   return temp;
}
// Overload assignment.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x; // These are integer assignments
```

```
   y = op2.y; // and the = retains its original
   z = op2.z; // meaning relative to them.
   return *this;
}
// Overload the prefix version of ++.
three_dthree_d::operator++()
{
   x++; // increment x, y, and z
   y++;
   z++;
   return *this;
}
// Show X, Y, Z coordinates.
void three_d::show()
{
   cout<< x << ", ";
   cout<< y << ", ";
   cout<< z << "\n";
}

intmain()
{
   three_da(1, 2, 3), b(10, 10, 10), c;
   a.show();
   b.show();

   c = a + b; // add a and b together
   c.show();

   c = a + b + c; // add a, b and c together
   c.show();

   c = b = a; // demonstrate multiple assignment
   c.show();
   b.show();
   ++c;  // increment c
   c.show();

   return 0;
}
```

The output from the program is shown here.

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
2, 3, 4
```

As the last line of the output shows, operator++( ) increments each coordinate in the object and returns the modified object. Again, this is in keeping with the traditional meaning of the ++ operator.As you know, the ++ and – – have both a prefix and a postfix form. For example, both++O;andO++;are valid uses of the increment operator. As the comments in the preceding program state, the operator++( ) function defines the prefix form of ++ relative to the three_d class. However, it is possible to overload the postfix form as well. The prototype for the postfix form of the ++ operator, relative to the three_d class, is shown here:

```
three_dthree_d::operator++(intnotused);
```

The increment and decrement operators have both a prefix and postfix form. The parameternotused is not used by the function, and should be ignored. This parameter is simply a way for the compiler to distinguish between the prefix and postfix forms of the increment operator. (The postfix decrement uses the same approach.)Here is one way to implement a postfix version of ++ relative to the three_d class:

```
// Overload the postfix version of ++.
three_dthree_d::operator++(intnotused)
{
   three_d temp = *this; // save original value

   x++;  // increment x, y, and z
   y++;
   z++;
   return temp; // return original value
}
```

Notice that this function saves the current state of the operand by using the statement

```
three_d temp = *this;
```

and then returns temp. Keep in mind that the traditional meaning of a postfix increment is to first obtain the value of the operand, and then to increment the operand. Therefore,it is necessary to save the current state of the operand and return its original value, before it is incremented, rather than its modified value.

The following version of the original program implements both forms of the ++operator:

```
// Demonstrate prefix and postfix ++.
#include <iostream.h>
class three_d
{
     int x, y, z; // 3-D coordinates
public:
      three_d() { x = y = z = 0; }
      three_d(inti, int j, int k)
      {x = i; y = j; z = k; }

      three_d operator+(three_d op2);
      three_d operator=(three_d op2);
      three_d operator++(); // prefix version of ++
      three_d operator++(intnotused);
      // postfix version of ++
      void show() ;
};

// Overload +.
three_dthree_d::operator+(three_d op2)
{
   three_d temp;

   temp.x = x + op2.x; // These are integer additions
   temp.y = y + op2.y; // and the + retains itsoriginal
```

```
      temp.z = z + op2.z; // meaning relative to them.
      return temp;
   }
   // Overload assignment.
   three_dthree_d::operator=(three_d op2)
   {
      x = op2.x; // These are integer assignments
      y = op2.y; // and the = retains its original
      z = op2.z; // meaning relative to them.
      return *this;
   }

   // Overload the prefix version of ++.
   three_dthree_d::operator++()
   {
      x++;  // increment x, y, and z
      y++;
      z++;
      return *this; // return altered value
   }
   // Overload the postfix version of ++
three_dthree_d::operator++(intnotused)
   {
      three_d temp = *this; // save original value

      x++; // increment x, y, and z
      y++;
      z++;
      return temp; // return original value
   }
   // Show X, Y, Z coordinates.
   void three_d::show( )
   {
      cout<< x << ", ";
      cout<< y << ", ";
      cout<< z << "\n";
   }
   intmain()
   {
   three_da(1, 2, 3), b(10, 10, 10), c;
```

```
a.show();
b.show();

c = a + b; // add a and b together c.show();
c = a + b + c; // add a, b and c together c.show();

c = b = a; // demonstrate multiple assignment
c.show();
b.show();

++c; // prefix increment c.show();
c++; // postfix increment c.show();

a = ++c; // a receives c's value after increment
a.show(); // a and c
c.show(); // are the same 13
a = c++; // a receives c's value prior to increment
a.show(); // a and c c.show(); // now differ

return 0;
}
```

The output is shown here.

```
1, 2, 3
```

As the last four lines show, the prefix increment increases the value of c before its value is assigned to a, and the postfix increment increases c after its value is assigned to a.

Remember that if the ++ precedes its operand, the operator++( ) is called. If it follows its operand, the operator++(intnotused) function is called. This same approach is also used to overload the prefix and postfix decrement operator relative to any class. Youmight want to try defining the decrement operator relative to three_d as an exercise.

Operator Overloading Tips and Restrictions

The action of an overloaded operator, as applied to the class for which it is defined, need not bear any relationship to that operator's

default usage, as applied to C++'s built-in types. For example, the << and >> operators, as applied to cout and cin, have little in common with the same operators applied to integer types. However, to maintain the transparency and readability of your code, an overloaded operator should reflect, when possible, the spirit of the operator's original use. For example, the + relative to three_dis conceptually similar to the + relative to integer types. There would be little benefit in defining the + operator relative to some class in such a way that it acts more the way you would expect the || operator, for instance, to perform. The central concept here is that, while you can give an overloaded operator any meaning you like, for clarity, it is best when its new meaning is related to its original meaning.

There are some restrictions to overloading operators. First, you cannot alter the precedence of any operator. Second, you cannot alter the number of operands required by the operator, although your operator function could choose to ignore an operand. Finally, except for the function call operator (discussed later), operator functions cannot have default arguments.The only operators that you cannot overload are shown here: `(.  ::  .*  ?`). Nonmember binary operator functions have two parameters. Nonmember unary operator functions have one parameter.

Nonmember Operator Functions

You can overload an operator for a class by using a nonmember function, which is often a friend of the class. As you learned earlier, nonmember functions, including friend functions, do not have a this pointer. Therefore, when a friend is used to overload an operator, both operands are passed explicitly when a binary operator is overloaded, and a single operand is passed when a unary operator is overloaded. The only operators that cannot be overloaded using nonmember functions are =, ( ), [ ], and –>.