Operator Overloading

3.1 Introduction

InC++, operators can be overloaded relative to class types that you define. The principal advantage to overloading operators is that it allows you to seamlesslyintegrate new data types into your programming environment.

Operator overloading allows you to define the meaning of an operator for a particular class. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack. Another class might use the + operator in an entirely different way. When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is defined. Therefore, overloading the + to handle a linked list, for example, does not cause its meaning relative to integers (i.e., addition) to change.

Operator overloading is closely related to function overloading. To overload an operator, you must define what the operation means relative to the class to which it is applied. To do this, you create an operator function, which defines the action of the operator. The general form of an operator function is

```
type classname::operator#(arg-list)
{
operation  relative to the class
}
```

Operators are overloaded using an operator function.Here, the operator that you are overloading is substituted for the #, and type is the type of value returned by the specified operation. Although it can be of any type you choose, the return value is often of the same type as the class for which the operator is being overloaded. This correlation facilitates the use of the overloaded operator in compound expressions.

The specific nature of arg-list is determined by several factors, as you will soon see.

Operator functions can be either members or nonmembers of a class. Nonmember operator functions are often friend functions of the class, however. Although similar, there are some differences between the way a member operator function is overloaded and the way a nonmember operator function is overloaded. Each approach is described here.

3.2 Operator Overloading Using Member Functions

To begin our examination of operator overloading using member functions, we will start with a simple example. The following program creates a class called three_d, which maintains the coordinates of an object in three-dimensional space. This program overloads the + and the = operators relative to the three_d class. Examine it closely:

```
    // Overload operators using member functions.
    #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);
    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;
}
// 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();
return 0;
}
```

This program produces the following output:

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

As you examined the program, you may have been surprised to see that both operatorfunctions have only one parameter each, even though they overload binary operations. Thereason for this apparent contradiction is that when a binary operator is overloaded using a member function, only one argument is explicitly passed to it. The other argument is implicitly passed using the this pointer. Thus, in the line

```
temp.x = x + op2.x;
```

the x refers to this–>x, which is the x associated with the object that invokes the operator function. In all cases, it is the object on the left side of an operation that causes the call to the operator function. The object on the right side is passed to the function.

In general, when you use a member function, no parameters are used when overloading a unary operator, and only one parameter is required when overloading a binary operator. (You cannot overload the ternary ? operator.) In either case, the object that invokes the operator function is implicitly passed via the this pointer.

To understand how operator overloading works, let's examine the preceding program carefully, beginning with the overloaded operator +. When two objects of type three_d are operated on by the + operator, the magnitudes of their respective coordinates are added together, as shown in operator+( ). Notice, however, that this functiondoes not modify the value of either operand. Instead, an object of type three_d, which contains the result of the operation, is returned by the function. To understand whythe + operation does not change the contents of either object, think about the standard arithmetic + operation, as applied like this: 10 + 12. The outcome of this operation is22, but neither 10 nor 12 is changed by it. Although there is no rule that prevents an overloaded operator from altering the value of one of its operands, it is best for the

actions of an overloaded operator to be consistent with its original meaning.

Notice that operator+( ) returns an object of type three_d. Although the function could have returned any valid C++ type, the fact that it returns a three_d object allows the + operator to be used in compound expressions, such as a+b+c. Here, a+b generates a result that is of type three_d. This value can then be added to c. Had any other type of value been generated by a+b, such an expression would not work.

In contrast with the + operator, the assignment operator does, indeed, cause one of its arguments to be modified. (This is, after all, the very essence of assignment.) Since the operator=( ) function is called by the object that occurs on the left side of the assignment, it is this object that is modified by the assignment operation. Most often, the return value of an overloaded assignment operator is the object on the left, after the assignment has been made. (This is in keeping with the traditional action of the = operator.) For example, to allow statements like

```
a = b = c = d;
```

it is necessary for operator=( ) to return the object pointed to by this, which will be the object that occurs on the left side of the assignment statement. This allowsa string of assignments to be made. The assignment operation is one of the most important uses of the this pointer.

```
// This program uses friend operator++() functions.
#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; }

   friend three_d operator+(three_d op1, three_d op2);
   three_d operator=(three_d op2);

   // use a reference to overload the ++
   friend three_d operator++(three_d&op1);
   friend three_d operator++(three_d&op1, intnotused);

   void show() ;
 } ;

// This is now a friend function.
three_d operator+(three_d op1, three_d op2)
{
   three_d temp;
   temp.x = op1.x + op2.x;
   temp.y = op1.y + op2.y;
   temp.z = op1.z + op2.z;
   return temp;
}
// Overload the =.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x;
   y = op2.y;
   z = op2.z;
   return *this;
}
```

```
/* Overload prefix ++ using a friend function.
This requires the use of a reference parameter. */
three_d operator++(three_d&op1)
{
     op1.x++; op1.y++; op1.z++; return op1;
}

/* Overload postfix ++ using a friend function.
This requires the use of a reference parameter. */
three_d operator++(three_d&op1, intnotused)
{
   three_d temp = op1;
   op1.x++; op1.y++; op1.z++;
   return temp;
}
// 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
    a = c++; // a receives c's value prior to increment
    a.show( ); // a and c
    c.show( ); // now differ

    return 0;
}
```

3.8 Overloading the Relational and Logical Operators

Overloading a relational or logical operator, such as ==, <, or && is a straightforward process. However, there is one small distinction. As you know, an overloaded operator function usually returns an object of the class for which it is overloaded. However, an overloaded relational or logical operator typically returns a true or false value. This is in keeping with the normal usage of these operators, and allows them to be used in conditional expression.Here is an example that overloads the = = relative to the three_d class:

```
//overload ==.
bool three_d::operator==(three_d op2)
{
   if((x == op2.x) && (y == op2.y) && (z == op2.z))
   return true;
   else
   return false;
}
```

Once operator==( ) has been implemented, the following fragment is perfectly valid:

```
three_d a, b;
// ...
if(a == b) cout<< "a equals b\n";
```

```
else cout<< "a does not equal b\n";
```

Because == returns a bool result, its outcome can be used to control an if statement. As an exercise, try implementing several of the relational and logical operators relative to the three_d class.