Operator Overloading

Operator overloading is one of the most exciting features of object oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. For example, statements like

d3.addobjects(d1, d2);

or the similar but equally obscure

d3 = d1.addobjects(d2);

can be changed to the much more readable

d3 = d1 + d2;

The rather forbidding term **operator overloading** refers to giving the normal C++ operators, such as +, *, <=, and +=, additional meanings when they are applied to user-defined data types.

Normally

a = b + c;

works only with basic types such as **int** and **float**, and attempting to apply it when **a**, **b**, and **c** are objects of a user-defined class will cause complaints from the compiler. However, using overloading, you can make this statement legal even when **a**, **b**, and **c** are user-defined types.

In effect, operator overloading gives you the opportunity to redefine the C++ language. If you find yourself limited by the way the C++ operators work, you can change them to do whatever you want. By using classes to create new kinds of variables, and operator overloading to create new definitions for operators, you can extend C++ to be, in many ways, a new language of your own design.

Overloading Unary Operators

Let's start off by overloading a unary operator. As you may recall from

Chapter 2, unary operators act on only one operand. (An operand is simply a variable acted on by an operator.)
Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33.

In the COUNTER example in Chapter 6, “Objects and Classes,” we created a class Counter to keep track of a count. Objects of that class were incremented by calling a member function:
c1.inc_count();
That did the job, but the listing would have been more readable if we could have used the increment operator ++ instead:
++c1;
All dyed-in-the-wool C++ (and C) programmers would guess immediately that this expression increments c1.
Let’s rewrite COUNTER to make this possible. Here’s the listing for COUNTPP1:

```
// countpp1.cpp
// increment counter variable with ++ operator
#include <iostream>
class Counter
{
private:
unsigned int count; //count
```

```
public:
Counter() : count(0) //constructor
{ }
unsigned int get_count() //return count
{ return count; }
void operator ++ () //increment (prefix)
{
++count;
}
};
////////////////////////////////////////////////////////////////
int main()
{
Counter c1, c2; //define and initialize
cout << “\nc1=” << c1.get_count(); //display
cout << “\nc2=” << c2.get_count();
++c1; //increment c1
++c2; //increment c2
++c2; //increment c2

cout << “\nc1=” << c1.get_count(); //display again
cout << “\nc2=” << c2.get_count() << endl;
return 0;
}
```

In this program we create two objects of class **Counter**: **c1** and **c2**. The counts in the objects are displayed; they are initially 0. Then, using the overloaded ++ operator, we increment **c1** once and **c2** twice, and display the resulting values. Here’s the program’s output:

c1=0 ←□□□ counts are initially 0

c2=0

c1=1 ←□□□ incremented once

c2=2 ←□□□ incremented twice

The statements responsible for these operations are

++**c1;**

++**c2;**

++**c2;**

The ++ operator is applied once to **c1** and twice to **c2**. We use prefix notation in this example; we'll explore postfix later.

The operator Keyword

How do we teach a normal C++ operator to act on a user-defined operand? The keyword operator is used to overload the ++ operator in this declarator:

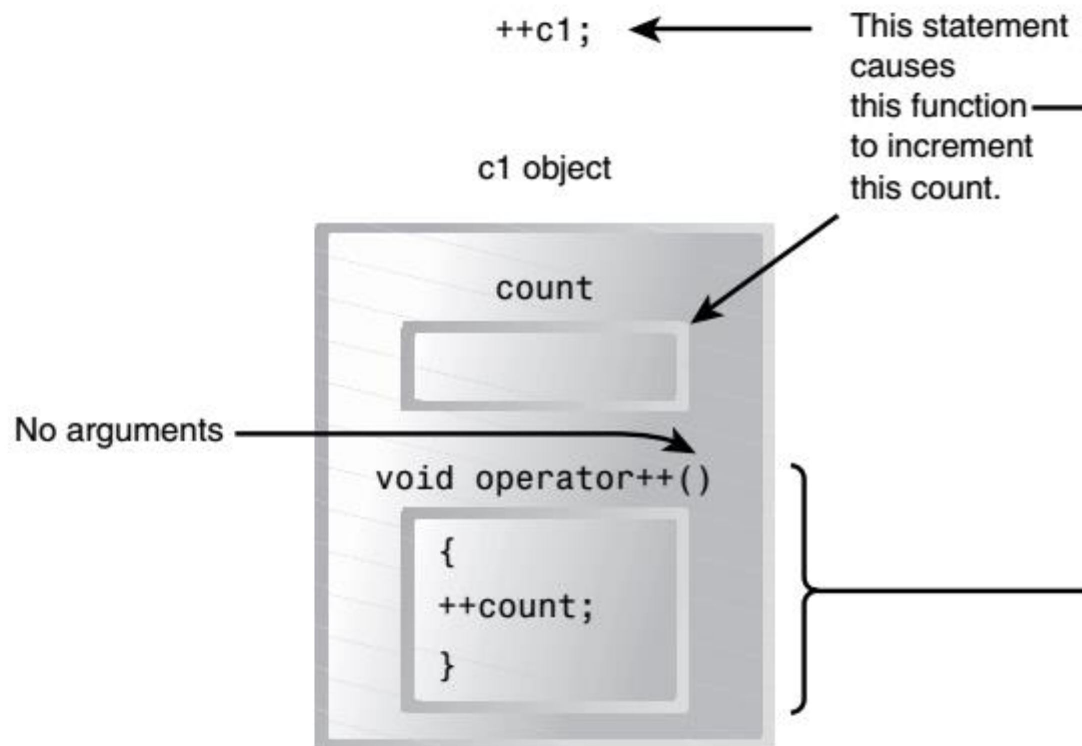
void operator ++ ()

The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here). This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand (the variable operated on by the ++) is of type Counter.

Operator Arguments

In main() the ++ operator is applied to a specific object, as in the expression ++c1. Yet operator++() takes no arguments. What does this operator increment? It increments the count data in the object of which it is a member. Since member functions can always access the particular

object for which they've been invoked, this operator requires no arguments. This is shown in Figure 8.1.



Operator Return Values

The operator++() function in the COUNTPP1 program has a subtle defect. You will discover it if you use a statement like this in main():

c1 = ++c2;

The compiler will complain. Why? Because we have defined the ++ operator to have a return type of void in the operator++() function, while in the assignment statement it is being asked to return a variable of type Counter. That is, the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1. So as defined in COUNTPP1, we can't use ++ to increment Counter objects in assignments; it must always stand alone with its operand. of course the normal ++ operator, applied to basic data types such as int, would not have this problem.

To make it possible to use our homemade operator++() in assignment expressions, we must provide a way for it to return a value. The next program, COUNTPP2, does just that.

```
// countpp2.cpp
// increment counter variable with ++ operator, return value
#include <iostream>
class Counter
{
private:
unsigned int count; //count
public:
Counter() : count(0) //constructor
{ }
unsigned int get_count() //return count
{ return count; }
Counter operator ++ () //increment count
{
++count; //increment count
Counter temp; //make a temporary Counter
temp.count = count; //give it same value as this obj
return temp; //return the copy
}
};
int main()
{
Counter c1, c2; //c1=0, c2=0
cout << "\nc1=" << c1.get_count(); //display
```

```
cout << "\nc2=" << c2.get_count();
++c1; //c1=1
c2 = ++c1; //c1=2, c2=2
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
}
```

Here the operator++() function creates a new object of type Counter, called temp, to use as a return value. It increments the count data in its own object as before, then creates the new temp object and assigns count in the new object the same value as in its own object. Finally, it returns the temp object. This has the desired effect. Expressions like
++c1 now return a value, so they can be used in other expressions, such as
c2 = ++c1;
as shown in main(), where the value returned from c1++ is assigned to c2.
The output from this program is

c1=0
c2=0
c1=2
c2=2