6.1 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 <u>operator overloading</u> 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 care 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.

6.1.1 Overloading Unary Operators

Let's start off by overloading a unary operator. Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33. In the COUNTER example "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; Let's rewrite COUNTER to make this possible. Here's the listing for COUNTPP1:

```
// countpp1.cpp// increment counter variable with ++ operator
#include <iostream.h>
class Counter
    {
private:
int count; //count
public:
Counter() : count(0) { }  //constructor
int ret_count() //return count
{ return count; }
void operator ++ () //increment (prefix)
{++count;}
};
```

```
int main()
{Counter c1, c2; //define and initialize
cout << "\nc1=" << c1.ret_count(); //display
cout << "\nc2=" << c2.ret_count();
++c1; //increment c1
++c2; //increment c2
++c2; //increment c2
cout << "\nc1=" << c1.ret_count(); //display again
cout << "\nc2=" << c2.ret_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 c

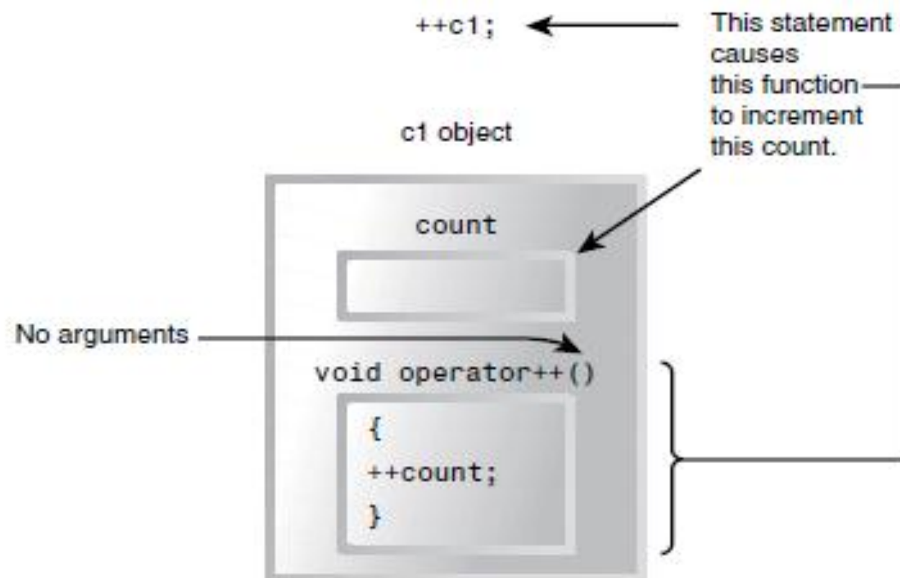
6.2 The operator Keyword

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. We saw "Functions," that the only way the compiler can distinguish between overloaded functions is by looking at the data types and the number of their arguments. In the same way, the only way it can distinguish between overloaded operators is by looking at the data type

of their operands. If the operand is a basic type such as an int, as in ++intvar; then the compiler will use its built-in routine to increment an int. But if the operand is a Counter variable, the compiler will know to use our user-written operator++() instead.

6.2.1 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 1.

6.2.2 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.h>
class Counter
{
private:
int count; //count
public:
Counter() : count(0) //constructor
{ }
int ret_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.ret_count(); //display
cout << "\nc2=" << c2.ret_count();
++c1; //c1=1
c2 = ++c1; //c1=2, c2=2
cout << "\nc1=" << c1.ret_count(); //display again
cout << "\nc2=" << c2.ret_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
```

6.3 Nameless Temporary Objects

In COUNTPP2 we created a temporary object of type Counter, named temp, whose sole purpose was to provide a return value for the ++ operator. This required three statements.

```
Counter temp; // make a temporary Counter object
temp.count = count; // give it same value as this object
return temp; // return it
```

There are more convenient ways to return temporary objects from functions and overloaded operators. Let's examine another approach, as shown in the program COUNTPP3:

```
// countpp3.cpp
// increment counter variable with ++ operator
// uses unnamed temporary object
#include <iostream.h>
class Counter
{
private:
unsigned int count; //count
public:
Counter() : count(0) //constructor no args
{ }
```

```
Counter(int c) : count(c) //constructor, one arg
{ }
int ret_count() //return count
{ return count; }
Counter operator ++ () //increment count
{
++count; // increment count, then return
return Counter(count); // an unnamed temporary object
} // initialized to this count
};
int main()
{
Counter c1, c2; //c1=0, c2=0
cout << "\nc1=" << c1.ret_count(); //display
cout << "\nc2=" << c2.ret_count();
++c1; //c1=1
c2 = ++c1; //c1=2, c2=2
cout << "\nc1=" << c1.ret_count(); //display again
cout << "\nc2=" << c2.ret_count() << endl;
return 0;
}
```

In this program a single statement return Counter (count); does what all three statements did in COUNTPP2. This statement creates an object of type Counter. This object has no name; it won't be around long enough to need one. This unnamed object is initialized to the value provided by the argument count. But wait: Doesn't this require a constructor that takes one argument? It does, and to make this statement work we sneakily inserted just such a constructor into the member function list in COUNTPP3.

```
Counter (int c): count(c) //constructor, one arg
                  { }
```

Once the unnamed object is initialized to the value of count, it can then be returned. The output of this program is the same as that of COUNTPP2. The approaches in both COUNTPP2 and COUNTPP3 involve making a copy of the original object (the object of which the function is a member), and returning the copy.

6.4 Postfix Notation

So far we've shown the increment operator used only in its prefix form. ++c1 What about postfix, where the variable is incremented after its value is used in the expression?

c1++

To make both versions of the increment operator work, we define two overloaded ++ operators, as shown in the POSTFIX program:

```
// postfix.cpp
// overloaded ++ operator in both prefix and postfix
#include <iostream.h>
class Counter
{
private:
unsigned int count; //count
public:
Counter() : count(0) //constructor no args
{ }
Counter(int c) : count(c) //constructor, one arg
{ }
int get_count() const //return count
{ return count; }
Counter operator ++ () //increment count (prefix)
{ //increment count, then return
return Counter(++count); //an unnamed temporary object
} //initialized to this count
Counter operator ++ (int) //increment count (postfix)
{ //return an unnamed temporary
return Counter(count++); //object initialized to this
} //count, then increment count
};
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 (prefix)
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
c2 = c1++; //c1=3, c2=2 (postfix)
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
}
```

Now there are two different declarators for overloading the ++ operator. The one we've seen before, for prefix notation, is

Counter operator ++ ()

The new one, for postfix notation, is

Counter operator ++ (int)

The only difference is the int in the parentheses. This int isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler to create the postfix version of the operator. Here's the output from the program:

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

We saw the first four of these output lines in COUNTPP2 and COUNTPP3. But in the last two lines we see the results of the statement

c2=c1++;

Here, c1 is incremented to 3, but c2 is assigned the value of c1 before it is incremented, so c2 retains the value 2. Of course, you can use this same approach with the decrement operator (--).