**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>
class Counter
{
private:
unsigned int count; //count
public:
Counter() : count(0) //constructor no args
{ }
Counter(int c) : count(c) //constructor, one arg
{ }
unsigned 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. The designers of C++ are fond of recycling existing operators and keywords to play multiple roles, and int is the one they chose to indicate postfix. (Well, can you think of a better syntax?) 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 (--).

## Overloading Binary Operators

Binary operators can be overloaded just as easily as unary operators. We'll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators.

### Arithmetic Operators

In the ENGLCON program in Chapter 6 we showed how two English Distance objects could be added using a member function add_dist():

**dist3.add_dist(dist1, dist2);**

By overloading the + operator we can reduce this dense-looking

expression to **dist3 = dist1 + dist2;**

Here's the listing for ENGLPLUS, which does just that:

```
// englplus.cpp
// overloaded '+' operator adds two Distances
#include <iostream>
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ } //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "\'-" << inches << '\"'; }
Distance operator + ( Distance ) const; //add 2 distances
};
//--------------------------------------------------------
//add this distance to d2
Distance Distance::operator + (Distance d2) const //return sum
{
```

```
int f = feet + d2.feet; //add the feet
float i = inches + d2.inches; //add the inches
if(i >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
i -= 12.0; //by 12.0 and
f++; //increase feet by 1
} //return a temporary Distance
return Distance(f,i); //initialized to sum
}
//////////////////////////////////////////////////////////////

int main()
{
Distance dist1, dist3, dist4; //define distances
dist1.getdist(); //get dist1 from user
Distance dist2(11, 6.25); //define, initialize dist2
dist3 = dist1 + dist2; //single '+' operator
dist4 = dist1 + dist2 + dist3; //multiple '+' operators
//display all lengths
cout << "dist1 = "; dist1.showdist(); cout << endl;
cout << "dist2 = "; dist2.showdist(); cout << endl;
cout << "dist3 = "; dist3.showdist(); cout << endl;
cout << "dist4 = "; dist4.showdist(); cout << endl;
return 0;
}
```

To show that the result of an addition can be used in another addition as well as in an assignment, another addition is performed in main(). We add

dist1, dist2, and dist3 to obtain dist4 (which should be double the value of dist3), in the statement dist4 = dist1 + dist2 + dist3;

Here's the output from the program:

```
Enter feet: 10
Enter inches: 6.5
dist1 = 10'-6.5" ←□□□ from user
dist2 = 11'-6.25" ←□□□ initialized in program
dist3 = 22'-0.75" ←□□□ dist1+dist2
dist4 = 44'-1.5" ←□□□ dist1+dist2+dist3
```

In class Distance the declaration for the operator+() function looks like this:

Distance operator + ( Distance );

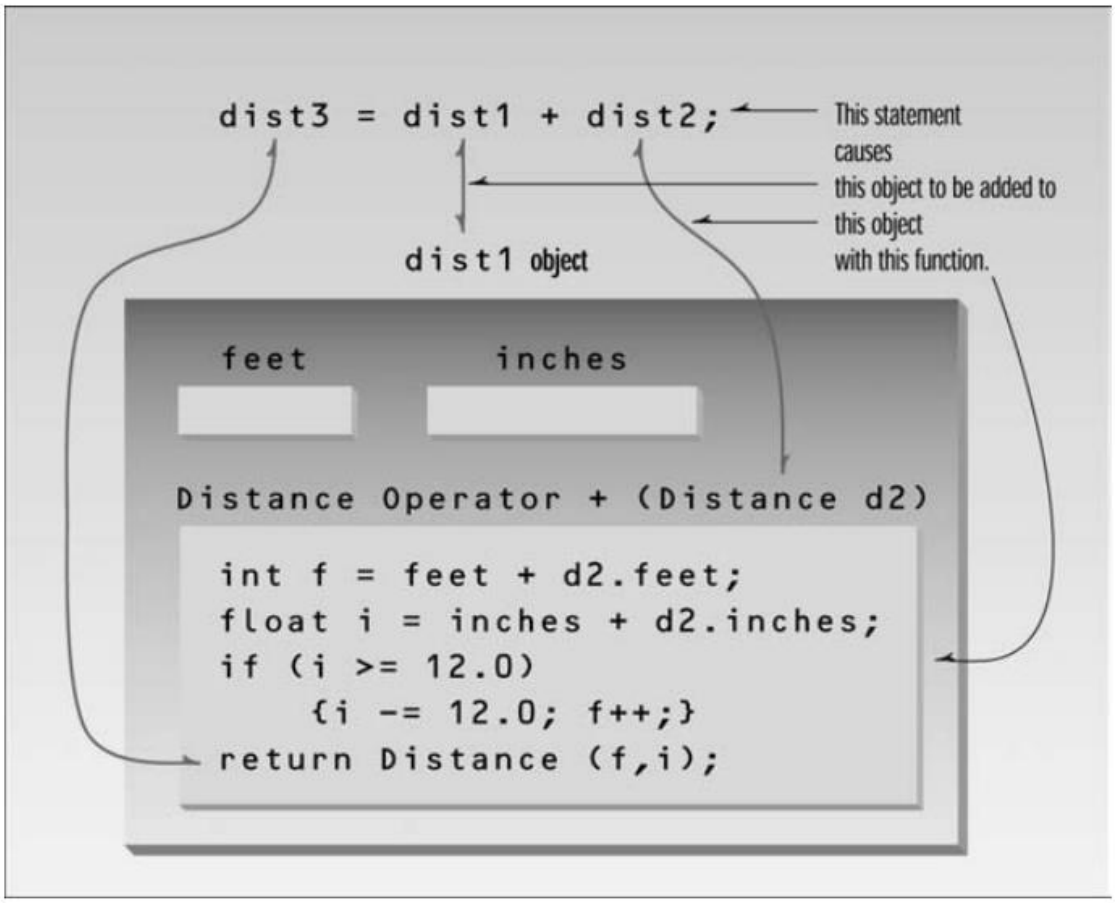
This function has a return type of Distance, and takes one argument of type Distance.

In expressions like

dist3 = dist1 + dist2;

it's important to understand how the return value and arguments of the operator relate to the objects. When the compiler sees this expression it looks at the argument types, and finding only type Distance, it realizes it must use the Distance member function operator+(). But what does this function use as its argument—dist1 or dist2? And doesn't it need two arguments, since there are two numbers to be added? Here's the key: The argument on the left side of the operator (dist1 in this case) is the object of which the operator is a member. The object on the right side of the operator (dist2) must be furnished as an argument to the operator. The

operator returns a value, which can be assigned or used in other ways; in this case it is assigned to dist3. Figure 8.2 shows how this looks.




In the operator+() function, the left operand is accessed directly—since this is the object of which the operator is a member—using feet and inches. The right operand is accessed as the function's argument, as d2.feet and d2.inches.

We can generalize and say that an overloaded operator always requires one less argument than its number of operands, since one operand is the object of which the operator is a member.

That's why unary operators require no arguments. (This rule does not apply to friend functions and operators, C++ features we'll discuss in Chapter 11.)

To calculate the return value of operator+() in ENGLPLUS, we first add the feet and inches from the two operands (adjusting for a carry if necessary). The resulting values, f and i, are then used to initialize a nameless Distance object, which is returned in the statement

return Distance(f, i);

This is similar to the arrangement used in COUNTPP3, except that the constructor takes two arguments instead of one. The statement

dist3 = dist1 + dist2;

in main() then assigns the value of the nameless Distance object to dist3. Compare this intuitively obvious statement with the use of a function call to perform the same task, as in the ENGLCON example in Chapter 6. Similar functions could be created to overload other operators in the Distance class, so you could subtract, multiply, and divide objects of this class in natural-looking ways.