overload ++ operator

Using a Friend to Overload a Unary Operator

You can also overload a unary operator by using a friend function. However,doing so requires a little extra effort. To begin, think back to the original version of the overloaded ++ operator relative to the three_d class that was implemented as a member function. It is shown here for your convenience:

```
// Overload the prefix form of ++.
three_dthree_d::operator++()
{
   x++; y++; z++;
   return *this;
}
```

As you know, every member function receives as an implicit argument this, which is a pointer to the object that invokes the function. When a unary operator is overloaded by use of a member function, no argument is explicitly declared. The only argument needed in this situation is the implicit pointer to the invoking object. Any changes made to the object's data will affect the object on which the operator function is called. Therefore, in the preceding function, x++ increments the x member of the invoking object.

Unlike member functions, a nonmember function, including a friend, does not receive a this pointer, and therefore cannot access the object on which it was called. Instead, a friend operator function is passed its operand explicitly. For this reason, trying to create a friend operator++( ) function, as shown here, will not work:

```
// THIS WILL NOT WORK
three_d operator++(three_d op1)
{
     op1.x++; op1.y++; op1.z++; return op1;
}
```

This function will not work because only a copy of the object that activated the call to operator++( ) is passed to the function in parameter op1. Thus, the changes inside operator++( ) will not affect the calling object, only the local parameter.

If you want to use a friend function to overload the increment or decrement operators, you must pass the object to the function as a reference parameter. Since a reference parameter is an implicit pointer to the argument, changes to the parameter will affect the argument. Using a reference parameter allows the function to increment or decrement the object used as an operand.

When a friend is used for overloading the increment or decrement operators, the prefix form takes one parameter (which is the operand). The postfix form takes two parameters. The second is an integer, which is not used.Here is the entire three_d program, which uses a friend operator++( ) function. Notice that both the prefix and postfix forms are overloaded.

```
// 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;
}
```

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.