Order Matters

Order Matters

When overloading binary operators, remember that in many cases, the orderof the operands does make a difference. For example, while A + B is commutative,A – B is not. (That is, A – B is not the same as B – A!) Therefore, when implementing overloaded versions of the non-commutative operators, you must remember which operand is on the left and which is on the right. For example, in this fragment, subtraction is overloaded relative to the three_d class:

```
// Overload subtraction.
three_dthree_d::operator-(three_d op2)
{
three_d temp;
   temp.x = x - op2.x;
   temp.y = y - op2.y;
   temp.z = z - op2.z;
   return temp;
}
```

Remember, it is the operand on the left that invokes the operator function.The operand on the right is passed explicitly. This is why x – op2.x is the proper order for the subtraction.For example, in the following program, a friend is used instead of a member function to overload the + operation:

```
// Overload + using a friend.
#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);
```

```
   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 assignment.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x; y = op2.y; z = op2.z;
   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;
}
```

As you can see by looking at operator+( ), now both operands are passed to it. The left operand is passed in op1, and the right operand in op2.In many cases, there is no benefit to using a friend function rather than a member function when overloading an operator. However, there is one situation in which a friend function is quite useful: when you want an object of a built-in type to occur on the left side of a binary operator. To understand why, consider the following.

As you know, a pointer to the object that invokes a member operator function is passed in this. In the case of a binary operator, it is the object on the left that invokes the function. This is fine, provided that the object on the left defines the specified operation. For example, assuming some object called Ob, which has integer addition defined for it, then the following is a perfectly valid expression:

```
Ob + 10; // will work
```

Because the object Ob is on the left side of the + operator, it invokes its overloaded operator function, which (presumably) is capable of adding an integer value to some element of Ob. However, this statement won't work:

```
10 + Ob; // won't work
```

The problem with this statement is that the object on the left of the + operator is an integer, a built-in type for which no operation involving an integer and an object of Ob's type is defined.

The solution to the preceding problem is to overload the + using two friend functions. In this case, the operator function is explicitly

passed both arguments, and it is invoked like any other overloaded function, based upon the types of its arguments. One version of the + operator function handles object + integer, and the other handles integer + object. Overloading the + (or any other binary operator) using friend functions allows a built-in type to occur on the left or right side of the operator. The following sample program shows you how to accomplish this:

```
#include <iostream.h>
class CL
{
public:
   int count;
   CL operator=(CL obj);
   friend CL operator+(CL ob, inti);
   friend CL operator+(inti, CL ob);
};


CL CL::operator=(CL obj)
{
   count = obj.count;
   return *this;
}
// This handles ob + int.
CL operator+(CL ob, inti)
{
   CL temp;

   temp.count = ob.count + i;
   return temp;
}
// This handles int + ob.
CL operator+(inti, CL ob)
{
   CL temp;

   temp.count = ob.count + i;
```

```
   return temp;
}
intmain()
{
   CL O;

   O.count = 10;
   cout<<O.count<< " "; // outputs 10

   O = 10 + O; // add object to integer
   cout<<O.count<< " "; // outputs 20

   O = O + 12; // add integer to object
   cout<<O.count; // outputs 32

   return 0;
}
```

As you can see, the operator+( ) function is overloaded twice, to accommodate the two ways in which an integer and an object of type CL can occur in the addition operation.