Programming Notes 17: Operator Overloading

As usual, more examples of the notes are in the example program.

1. Operators

First off, what exactly is an operator? You've probably heard me mention them before, what with the "assignment operator" or the "insertion operator" or whatever else. Well, an operator is basically a certain character (or two) that the language uses to describe an operation on one or more pieces of data. For example, some of the operators in C++ are:

+, -, *, /, %, ++, --, ==, !=, >, <, >=, <=, &&, ||, !, &, |, ^, ~, >>, <<, =, +=, -=, *=, and /=

Most of these operators you've seen before: for example "+" will add two values, "=" will assign a value to another value, and ">" will test if one value is greater than another. All these operators are automatically defined for some types, usually the primitive types (int, float, char, etc). However, if you were to try to use one on an ADT that you have created, you'll notice you can't: you'll get compiler errors. Operator overloading is a way to use these operators on your own data types.

Why do you want to do this, you may ask? Well, technically operator overloading does nothing more than what you can do with normal member functions, but using them will very often make your code cleaner, easier to understand, and easier to create.

2. Operator Overloading

So, how does one go about overloading an operator? It's probably more straightforward than you think: an operator overload is prototyped in much the same way a member function is. The basic syntax is:

<return type> operator<operator>(<parameters>);

e.g.

bool operator==(const type& comp);

However, calling an overloaded operator is much simpler than calling a member function: you simply use the operator. There are a few things to remember here: first, the left hand object is the calling object, meaning that it contains the operator function that will be called, and the right hand object is the parameter. In other words:

result = obj1 == obj2;

Is equivalent to

result = obj1.operator==(obj2);

Implementing an overloaded operator is also not much different than any other member function, for example a simple equality compare operator:

```
bool type::operator==(const type& comp)
{
    return x == comp.x && y = comp.y;
}
```

Which tests if the data members "x" and "y" are the same in both the calling object and the comparison object passed to the function. Pretty simple, right? You can apply this same logic to most of the other operators. For example, a "+" operator:

Prototype:

```
class type
{
    type operator+(const type& add);
    // …
}
```

Implementation:

```
type type::operator+(const type& add)
{
    return type(x + add.x, y + add.y);
}
```

Which returns a new object of type "type" using the "x" and "y" values of the calling object and parameter added together. Note that this uses a parametrized constructor to create the object to return, so that would have to exist as well.

3. Return Types and Chaining

Before we get into the details of how to use operators, you should know what return types are used with overloaded operators. Technically, the return type is no different than any other function. However, overloaded operators almost always return the type, or a reference to the type it uses. For example, the assignment operator, used to copy values from one object to another of the same type, is almost always prototyped as:

```
type& operator=(const type& src);
```

First, we return by reference because we are returning an object that already exists, the calling object. When you create a new object, for example in the "+" operator, you cannot return by reference. The reason we don't just return "void" is for "chaining," which means using multiple operators in a row, as in:

```
obj1 = obj2 = obj3;
```

Because the assignment operator is right-associative, this statement is equivalent to:

```
obj1.operator=(obj2.operator=(obj3));
```

Here you can see why the operator has to return the same type as the object it's using: the inner assignment returns another object of the same type so that the object 1's assignment operator can use that. This works the same way with operators such as "+," "-," "++," and many others.

It's very straightforward to return your result in operators such as "+" and "-," as you are creating a new object that contains your values added together. However, in operators that modify the calling object (for example the assignment operator or "+="), you must somehow return the calling object. The keyword "this" gives you a pointer to the calling object, but you must return an actual object (or reference), so you can dereference "this" to get your actual calling object. For example:

```
type& type::operator=(const type& src)
{
    x = src.x;
    return *this;
}
```

4. Friend vs. Member Operators

As of yet, I've only talked about overloading operators as member functions. However, you can also implement them as free functions, if you make them a friend of the class(es) you want them to manipulate. The syntax of a friend operator is very similar, except that because the operator is technically a normal, free, function, there will be no calling object, and hence you must pass both objects in as parameters. For example:

```
friend ostream& operator<<(ostream& out, const type& source);
```

As with all other friend functions, the operator is implemented like a free function, and like operator members, the name is still "operator" plus the actual operator. For example:

```
ostream& operator<<(ostream& out, const type& source)
{
    out << source.x << source.y << endl;
}
```

It may seem that there is no point to make an operator a friend over a member, or vice versa, and this is true for most operators. It doesn't matter how you implement your operators, except for a few special cases. These are that the assignment operator must always be a member, and the insertion/extraction operator must be friends, as well as any other operators that would require you to modify a different class. What I mean by this, you will learn in the last two sections.

5. Assignment Operator vs. Copy Constructor

You might think that the assignment operator and the copy constructor do much the same thing, and you'd be right: they both copy data from one instance of a type into another. However, there are

a couple of key differences. First, you should remember that the constructor of an object is only called once, when the object is instantiated. This means that the copy constructor will always start with a blank slate, and never needs to clean up data before copying in new data.

Additionally, because the copy constructor is only called when you're creating a new object, you know that your parameter, the source object, cannot be the same object as the calling object. On the other hand, with the assignment operator, you have no such guarantee. This may sound confusing, but look at the code:

```
type object1(object1);          //This makes no sense, and will not compile.

object2 = object2;              //This is totally valid and will compile.
```

This means that within your assignment operator, you should always make sure the source object is not the same as the calling object. Of course, the only way to know for sure if two things are exactly the same is to compare their addresses, and the keyword "this" conveniently provides us with a pointer to the current calling object. Note that to do this, you must pass your source object by reference, or else a copy will be made when it is passed to the function. This allows you to easily test, for example:

```
type& type::operator=(const type& source)
{
    if(this != &source)
    {
        // do copy
    }
}
```

Finally, although counterintuitive, if you use the assignment operator while creating an object, your program will actually call the copy constructor. The assignment operator is only called when you have two objects, already instantiated, set equal to each other.

```
type object2 = object1;          // Will call the copy constructor

type object2;
object2 = object1;               // Will call the assignment operator
```

6. Insertion/Extraction Operators

The final thing we're going to cover is how to overload the insertion and extraction operators in your own class. First, take a look at this:

```
cout << object1;
```

Earlier, I said that the left hand object is the one that calls the function, so would that make "cout" the calling object? Actually, yes. However, we can't modify the type of "cout," so we need another way to describe the operator. This is why it must be a friend rather than a member of our ADT.

Other than friendship, the insertion/extraction operators are exceedingly simple to implement. The insertion operator should take a parameter of type "ostream&," as this is what describes an output stream. It should return the same type, for chaining. On the other hand, the extraction operator should take a parameter of type "istream&," as this is what describes an input stream. Again, it should return the same type for chaining.

Using the types as described will allow you to simply use "cout" and "cin" with your ADT, but will also work for any input/output streams, including file streams.

For example:

```
ostream& operator<<(ostream& out, const type& source)
{
    out << source.x << " " << source.y << endl;
    return out;
}
```

Is called with

```
cout << object1;                    //Assuming object1 is of type "type"
```

7. Brackets/parenthesis operators

Finally, a quick word on the bracket/parenthesis operators: they are usually used to look up a piece of data within a data structure (if your class is a data structure), and take any parameters you want. They are declared as follows:

```
char operator[](int index);
char operator()(int index);

char type::operator[](int index)
{
    return data[index];
}

char type::operator()(int index)
{
    return data[index];
}
```

You get the idea.