Overloaded Operators

Operator overloading provides a way to manage objects of a class using predefined operators. There are several operators you can overload, but the most common you will encounter is off to the right.

+ - * / % = []
+= -= *= /= %=
< > == <= >= !=
++ -- >> << ()

Consider a class as such:

```
class Integer
{
    public:
       //Constructors and other member functions
       int getX() { return x; }
       int getY() { return y; }
    private:
       int x;
       int y;
};
```

Let's say that in the main function there are two objects of this class, **Obj1** and **Obj2**. If you want to find out if the two objects are equal, you would have to have a long if statement (in main) as such:

```
if((Obj1.getX() == Obj2.getX()) && (Obj1.getY() == Obj2.getY()))
```

It would look nicer if you could simply do `if(Obj1 == Obj2)`. Operator overloading allows this. As you see, the code is easier to read and shorter. The implementation will be done in the class implementation, which allows for further information hiding.

Here are the common types of overloaded operators you will see:

1) **Unary Operators**
2) **Increment/Decrement Operators**
3) **Binary Operators**
4) **Comparison Operators**
5) **Assignment Operator**
6) **Input/Output Operators**
7) **Function Operator**
8) **Bracket Operator**

The syntax will depend on the specific operator to be overloaded. **Some** overloaded operators may be a **friend** of the class or a member of the class (exceptions will be mentioned). If the operator is a **friend**, the overloaded operator will take **one** more object in the formal parameters than if the overloaded operator were a member function of the class. Example: the overloaded plus operator can be written as **(choose one)**:

```
friend Integer operator +(const Integer& Obj1, const Integer& Obj2);
Integer operator +(const Integer& Rhs); //Example for class above
```

Unary/Increment/Decrement Operators

Member Function: The most common unary operator you will come across is the negation operator. As a member function, a unary operator will have no formal parameters as there is only one calling object. An example is: **-Object;** The unary, increment, and decrement operator should return an object of the class. This is to do assignment statements such as: **ReceivingObject = ++Object;** Even though there is a return type (class object), the example with the negation operator is equally valid (it just doesn't do anything with the returned object). Here are examples of the operator declarations (**ClassName** is the return type in these cases):

```
ClassName operator -(); //Negation operator
ClassName operator ++(); //Pre-increment operator
Classname operator ++(int); //Post-increment operator
```

Notice the last example. There needs to be a way to differentiate between the Pre-increment and Post-increment operator. In order to do that, place **int** as an argument. However, the syntax is the same as the other operators: **ReceivingObject = Object++;** or **Object++;**

Friend Function: All of the aforementioned will take one more argument than the member functions. The same syntax as above applies when you invoke these functions. Here are examples:

```
friend ClassName operator -(ClassName &Object);
friend ClassName operator ++(ClassName &Object);
friend Classname operator ++(ClassName &Object, int);
```

Binary/Comparison Operators

Member Function: There are many binary and Comparison operators (assignment operator discussed in the next section). As a member function, a binary/comparison operator will take one argument as you are dealing with a left-hand object (calling object) and a right-hand object. The left-hand object is the calling object. A few examples to invoke these operators are: **AddedObject = LhsObj + RhsObj;** or **LhsObj += RhsObj;** or **LhsObj == RhsObj;** Here are examples of the operator declarations (**ClassName** is the return type in these cases):

```
ClassName operator +(const ClassName &Rhs); //Plus operator
bool operator ==(const ClassName &Rhs); //Equality operator
ClassName& operator +=(const ClassName &Rhs); //Plus/Equals operator
```

Notice how the **+=** operator return a reference. This is similar to what you will see with the **=** operator. This is because you are working with the left-hand side object, manipulating it, and returning ***this** in the function.

Friend Function: All of the aforementioned will take one more argument than the member functions. The same syntax applies when you invoke these functions. **It is standard convention to have the operator += as a member function** (as well as the other conjoined binary operators, such as **-=**, ***=**, etc). However, they do work as a friend function, and are here to show you what is possible. Here are examples:

```
friend ClsNme operator +(const ClsNme &Obj1, const ClsNme &Obj2);
friend bool operator ==(const ClassName &Obj1, const ClassName &Obj2);
friend ClassName& operator +=(ClassName &Obj1, const ClassName &Obj2);
```

Assignment Operator

Member Function Only: One special aspect about the assignment operator is that if you do not explicitly define one, the compiler will automatically create one for you. It just may not behave as you want/expect if you do go that route (see **Rule of Three**). Since this is the case as well as the standard stating as such, the assignment operator must be a member function. This operator will return a reference to the left-hand side object as this is the calling object and you are manipulating it. This also allows for chaining. Here are two examples of the assignment operator:

```
LhsObject = RhsObject;
Object1 = Object2 = Object3; //Chaining
```

Here is the declaration (**ClassName&** is the return type):

```
ClassName& operator =(const ClassName &Rhs);
```

Inside the function, you need to first check to make sure the invoking statement is not **SameObject = SameObject;** The way to check this is:

```
if(this != &Rhs) { //Rest of code }
```

The reason for this is when dealing with dynamically allocated data and arrays, you may delete the dynamically allocated data before performing copy over operations or you may be incrementing a **used** variable when dealing with an array causing a potential segmentation fault (while copying over array data). Since both the left-hand object and the right-hand object are the same, whatever you do to either object affects the same object.

After the function is done with its operations, you will return the reference using the **this** operator, such as: **return *this;** Remember that **this** is a pointer to a particular object and needs to be dereferenced as you are referencing the object and not the address.

Input/Output Operators

Friend Function Only: The input and output operators need to be a **friend** of the class, because the first argument to these functions is an object of the class **istream** or **ostream**. The standard states that binary overloaded functions can only be a member function of the class of the left-hand side object (in this case, the first argument in the formal parameters). Here are the function declarations:

```
friend ostream& operator <<(ostream &input, const ClassName &Obj);
friend istream& operator >>(istream &input, ClassName &Obj);
```

The return type is a reference to an object of the class **ostream** or **istream**. This allows for chaining. This will take the first two arguments in the chain and return a reference for the third argument to be executed, and so forth. Here is an output example (iterations 1-3 only)(parentheses only for clarity):

```
((((cout << Object) << “String”) << Object2) << endl);
(((cout << “String”) << Object2) << endl);
((cout << Object2) << endl); //Pattern continues until semicolon
```

Function/Bracket Operators

Member Function Only: The function and bracket operators must be a member function of the class. In a basic sense, the function operator will act as a class function (functors are not discussed here) and the bracket operator can be used when dealing with an array/vector index. Here are **example** declarations (the function operator can have no arguments and the bracket operator must have one argument):

```
void operator ()(int x, char y); //Return types are whatever you need
int operator [](int x);
```

The way to invoke these overloaded operators are as follows:

```
Object(1,2); //These are examples corresponding to above declarations
Object[1];
```

For the overloaded function operator, you are able to have the same type/amount of arguments as your constructor. The compiler is able to discern which to invoke as the constructor is invoked when declaring/defining an object on the same line.

Example Programs

To see examples of these overloaded operators, look at **Overloading.cpp** for member function overloaded operators and look at **OverloadingFriends.cpp** for friend function overloaded operators.