Lab 7

Operator Overloading

Operator Overloading in C++

In C++ the overloading principle applies not only to functions, but to operators too. That is, of operators can be extended to work not just with built-in types but also classes. A programmer can provide his or her own operator to a class by overloading the built-in operator to perform some specific computation when the operator is used on objects of that class. Is operator overloading really useful in real world implementations? It certainlly can be, making it very easy to write code that feels natural (we'll see some examples soon). On the other hand, operator overloading, like any advanced C++ feature, makes the language more complicated. In addition, operators tend to have very specific meaning, and most programmers don't expect operators to do a lot of work, so overloading operators can be abused to make code unreadable. But we won't do that.

An Example of Operator Overloading

Complex a(1.2,1.3); //this class is used to represent complex numbers
Complex b(2.1,3); //notice the construction taking 2 parameters for the real and imaginary part
Complex c = a+b; //for this to work the addition operator must be overloaded
The addition without having overloaded operator + could look like this:

Complex c = a.Add(b);
This piece of code is not as readable as the first example though--we're dealing with numbers, so doing addition should be natural. (In contrast to cases when programmers abuse this technique, when the concept represented by the class is not related to the operator--ike using + and - to add and remove elements from a data structure. In this cases operator overloading is a bad idea, creating confusion.) In order to allow operations like Complex c = a+b, in above code we overload the "+" operator. The overloading syntax is quite simple, similar to function overloading, the keyword operator must be followed by the operator we want to overload:

```
class Complex
{
public:
	Complex(double re,double im)
		:real(re),imag(im)
		{};
	Complex operator+(const Complex& other);
	Complex operator=(const Complex& other);
private:
	double real;
	double imag;
};
Complex Complex::operator+(const Complex&  other)
{
   double result_real = real + other.real;
```

```
    double result_imaginary = imag + other.imag;
    return Complex( result_real, result_imaginary );}
```

The assignment operator can be overloaded similarly. Notice that we did not have to call any accessor functions in order to get the real and imaginary parts from the parameter other since the overloaded operator is a member of the class and has full access to all private data. Alternatively, we could have defined the addition operator globally and called a member to do the actual work. In that case, we'd also have to make the method a friend of the class, or use an accessor method to get at the private data:

```
friend Complex operator+(Complex);
Complex operator+(const Complex  &num1, const Complex &num2)
{
    double result_real = num1.real + num2.real;
    double result_imaginary = num1.imag + num2.imag;
    return Complex( result_real, result_imaginary );
}
```

Why would you do this? when the operator is a class member, the first object in the expression must be of that particular type. It's as if you were writing:

```
Complex a( 1, 2 );
Complex a( 2, 2 );
Complex c = a.operator=( b );
```

when it's a global function, the implicit or user-defined conversion can allow the operator to act even if the first operand is not exactly of the same type:

```
Complex c = 2+b;      //if the integer 2  can be converted by the Complex class, this expression is valid
```

By the way, the number of operands to a function is fixed; that is, a binary operator takes two operands, a unary only one, and you can't change it. The same is true for the precedence of operators too; for example the multiplication operator is called before addition. There are some operators that need the first operand to be assignable, such as : operator=, operator(), operator[] and operator->, so their use is restricted just as member functions(non-static), they can't be overloaded globally. The operator=, operator& and operator, (sequencing) have already defined meanings by default for all objects, but their meanings can be changed by overloading or erased by making them private. Another intuitive meaning of the "+" operator from the STL string class which is overloaded to do concatenation:

```
string prefix("de");
string word("composed");
string composed = prefix+word;
```

Using "+" to concatenate is also allowed in Java, but note that this is not extensible to other classes, and it's not a user defined behavior. Almost all operators can be overloaded in C++:

```
      +     -     *     /     %     ^     &     |
    ~     !     ,     =          =
    ++    --    <<    >>    ==    !=    &&    ||
    +=    -=    /=    %=    ^=    & =   |=    *=
    <<=   >>=   [ ]   ( )   ->    ->*   new   delete
```

The only operators that can't be overloaded are the operators for scope resolution (::), member selection (.), and member selection through a pointer to a function(.*). Overloading assumes you specify a behavior for an operator that acts on a user defined type and it can't be used just with general pointers. The standard behavior of operators for built-in (primitive) types cannot be changed by overloading, that is, you can't overload operator+(int,int).
The logic(boolean) operators have by the default a short-circuiting way of acting in expressions with multiple boolean operations. This means that the expression:

if(a && b && c)
will not evaluate all three operations and will stop after a false one is found. This behavior does not apply to operators that are overloaded by the programmer.
Even the simplest C++ application, like a "hello world" program, is using overloaded operators. This is due to the use of this technique almost everywhere in the standard library (STL). Actually the most basic operations in C++ are done with overloaded operators, the IO(input/output) operators are overloaded versions of shift operators(<<, >>). Their use comes naturally to many beginning programmers, but their implementation is not straightforward. However a general format for overloading the input/output operators must be known by any C++ developer. We will apply this general form to manage the input/output for our Complex class:

```
friend ostream &operator<<(ostream &out, Complex c)     //output
{
      out<<"real part: "<<real<<"\n";
      out<<"imag part: "<<imag<<"\n";
      return out;
}
friend istream &operator>>(istream &in, Complex &c)     //input
{       cout<<"enter real part:\n";
      in>>c.real;
      cout<<"enter imag part: \n";
      in>>c.imag;
      return in;}
```

Notice the use of the friend keyword in order to access the private members in the above implementations. The main distinction between them is that the operator>> may encounter unexpected errors for incorrect input, which will make it fail sometimes because we haven't handled the errors correctly. A important trick that can be seen in this general way of overloading IO is the returning reference for istream/ostream which is needed in order to use them in a recursive manner:

```
Complex a(2,3);
Complex b(5.3,6);
cout<<a<<b;
```

Note: Before doing your lab exercises run the examples.

Exercises will be provided by the instructors in the lab.