CSci 1113, Fall 2014
Lab Exercise 10 (Week 11): Operator Overloading

Operator Overloading

Operator "overloading" is one way in which C++ supports the object-oriented principle of a*d hoc polymorphism*. *Polymorphism* is derived from the Greek words "*poly"* ("many") and "*morph*" ("form"). *Ad hoc polymorphism* refers to the notion that the actual *operation* performed for any *operator* is entirely dependent on the form (*type*) of its *operands*. For example, the division operator / results in different operations for floating-point operands vs. integer operands.

Operator overloading is useful when constructing complex classes. At a minimum, most user-defined classes should include overloaded stream *input* and *output* operators (>> and <<) to support debugging and file I/O. In this lab exercise, we continue to develop our understanding of class construction by extending the work from the previous lab by including overloaded operators.

Warm-up

Complete the following paper/pencil exercises. First, do these individually. Then you and your partner should share your answers and make any needed corrections, making sure you both understand the solutions. Then you both should discuss your answers with one of your TA.

1) Consider the following class declaration:

```
class Point
{ public:
      Point( );
      Point(int xval, int yval);
      friend Point operator +(Point, Point);
  private:
      int  xloc;
      int  yloc;

} ;
```

a. Write the function definition for the overloaded + operator that will produce the sum of the x values and the sum of the y values of two `Point` objects.

b. Rewrite the `Point` class interface to make the overloaded + operator a public member function of the class.

c. Provide the function definition for the overloaded + *member* operator of the `Point` class.

d. Describe what the overloaded output operator (e.g., <<) return-type must be and why.
(***Hint***: consider how the following statement is evaluated:

```
cin >> some_object >> some_other_object;
```

e. In order to overload the output operator for a class we make it a non-member *friend* function of the class and not a member function. Explain why.

f. Overload the output operator for the point class to output a point object as follows:

x,*y*

where *x* and *y* are the coordinate values for the point, separated by a comma.

Stretch

1) **Complex Number Class**

Extend the `Complex` class that you completed in last week's Lab Exercises. Recall that a *complex number* is of the form a + b*i* where a and b are real numbers and $i^2 = -1$. For example, 2.4 + 5.2*i* and 5.73 - 6.9*i* are complex numbers. a is referred to as the *real* part of the complex number and b*i* the *imaginary* part.

Make the following changes to the `Complex` class definition:

a) Add a constructor that will take two `double` arguments and initialize the *real* and *imaginary* components respectively.

b) Replace the `input` and `output` methods with overloaded >> and << operators, respectively, that are *friend* functions of the class. The output should be `realValue + imaginaryValue i` e.g., :

9.3 + 5.7`i` 12.4 - 8.4`i`

Note, if the i`maginaryValue` component is zero, simply output the `realValue` as a 'regular' floating-point value. Also, if the `imaginaryValue` is negative, replace the '+' symbol with '-'. You may input the complex value in any format you wish.

c) Provide an overloaded addition operator that will return the sum of two `Complex` objects. For complex numbers a + b*i* and c + d*i*, addition is defined as

(a+c) + (b+d)*i*.

4) Provide an overloaded *negation* operator (unary -) that will negate both the real and imaginary parts of a complex number. For example if `z` is the complex number 3 - 2i, then `-z` should return a complex number -3 + 2i.

Put the class interface in a file named `Complex.h` and the implementation in a file named `Complex.cpp`. Be sure to include the compiler directives `#ifndef`, `#define` and `#endif` to support separate compilation.

Thoroughly test your `Complex` number class by writing a test program that constructs various `Complex` values and displays the results until you are convinced that it is operating correctly. Include the following statements in your test program and show the results to a TA:

```
Complex c1,c2,c3;
cout << "Enter two complex values: ";
cin >> c1 >> c2;
c3 = c1+c2;
cout << "The sum is: " << c3 << endl;
cout << "and negating the sum yields: " << -c3 << endl;
```

Workout

1) **Quadratic Polynomials**

Write a program that will output the sum of two quadratic polynomials. Your program must do the following:

1. Define an abstract data type, `Poly` with three private data members `a`, `b` and `c` (type `double)` to represent the coefficients of a quadratic polynomial in the form: $ax^2 + bx + c$
2. Include a constructor in the `Poly` class to initialize all private data members with caller-supplied values (in addition to the default constructor!)
3. Overload the *addition* operator to return the sum of two `Poly` objects.
4. Overload the << (output) operator to output `Poly` objects in the following format, e.g.,:

   ```
   ax^2 + bx + c
   ```

   Where *a*, *b* and *c* are the coefficients of the `Poly` object. Do not display the *a* or *b* terms if they have zero coefficients. Moreover, if any coefficient is negative it should be precede by a minus sign, and not a plus sign.
5. In your `main()` function, declare and initialize two `Poly` objects, `q1` and `q2`, to represent the following polynomials: $3x^2 + 4x – 2$ and $-4x + 10$. Also declare a third, un-initialized `Poly` object named `sum`.
6. Output the sum of the two polynomials to the console using the following C++ code exactly as it appears:

   ```
   sum = q1 + q2;
   cout << q1 << " : q1\n";
   cout << q2 << " : q2\n";
   cout << sum << " : q1+q2\n";
   ```

Put the class interface in a file named `Poly.h` and the implementation in a file named `Poly.cpp`. Be sure to include the compiler directives `#ifndef`, `#define` and `#endif` to support separate compilation.

2) **More Polynomial Methods**

Extend the `Poly` class to support the *evaluation* of quadratic polynomials as follows:

1. Add a member function named `eval` that takes a single `double` argument representing the '*x*' value and returns the result (type `double`) of evaluating the polynomial at *x*.
2. Add a `void` member function named `roots` that will take two *reference* arguments of type `Complex`, then compute and provide *both* roots of a quadratic polynomial object. Recall that the roots of a quadratic polynomial are given by the quadratic equation:

   $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   The expression under the radical is referred to as the *discriminant*. If the *discriminant* is negative, the roots must be expressed as imaginary values using Complex numbers as follows:

   $$x_1 = \frac{-b}{2a} + \frac{\sqrt{|b^2 - 4ac|}}{2a}i \quad , \quad x_2 = \frac{-b}{2a} - \frac{\sqrt{|b^2 - 4ac|}}{2a}i$$

   The `roots` method should first determine if the roots are real or imaginary based on the value of the *discriminant*. If the roots are *real*, return them as complex values with the imaginary component set to zero, if the roots are *imaginary*, return them in complex form using the absolute value of the *discriminant*.

3. Overload the >> (input) operator to input a `Poly` object in the following format:

   `(a,b,c)`

   Where *a*, *b* and *c* are the data values of the `poly` object. The input will include the parentheses and commas as shown.

Now, write a main program that will evaluate the quadratic polynomial $2x^2 - 6x + 5$ at the integer values 0 through 10. Include the following C++ code:

```
Poly  inpoly;
cout << "Input a quadratic polynomial: ";
cin >> inpoly;
for( int i=0; i<= 10; i++)
    cout << "f(" << i << ") is: "
         << inpoly.eval(i) << endl;
Complex c1,c2;
inpoly.roots(c1,c2);
cout << "The roots of f(x) are "
     << c1 << "\t" << c2 << endl;
```

Check

Individually write down (a) one important thing you learned from this lab exercise, and (b) one question you still have. Then you and your partner should share and discuss what you both have written.

Challenge

1) **Polynomial Evaluation**

The current `Poly::eval` method takes a floating-point value for *x* and returns $f(x)$ as a double. Provide an overloaded `Poly::eval` method that will take a complex number as its argument and return a complex result.

In order to do this, you will need to add overloaded operators to the `Complex` class to support *multiplication* and *subtraction*. If this is done properly, the code for the floating-point `Poly::eval` method should be usable with very little (if any) modification.

2) **Further Overloaded Operators**

Add the following additional operators to your `Complex` number class:

- If you wrote your overloaded + operator as a friend function, write it as a non-member, non-friend function. (Hint: you will need to use accessor member functions in writing this.)
- Write an overloaded == operator that checks if two complex number objects have the same real and imaginary values.
- Write an overloaded += operator as a member function of the class. The statement `c1 += c2` should add `c1` and `c2` together and store the result in `c1`.