**Lab 6**

Overloaded operators are functions the redefine how operators (like +, -, \*, /, ==, <, >, <<, >>, and more) work. Often these work great when dealing with classes that have more complicated math methods to them (like adding complex numbers, for example). Today we will write a Fraction class that will add, subtract, multiply, divide, and determine equality of two fractions.

In order to implement overloaded input stream (>>) and overloaded output stream (<<) operators, you need to allow the istream and ostream classes to be friend objects. Since the << and >> don’t belong to the Fraction class, you have to write the functions outside of the class and allow them to be friends. We will talk more about this in class and more will be below.

**Directions**

1. Open Visual Studio 2010 and create a new project and name it Lab6.
2. Open up Blackboard and download the files and save them in your project folder (usually under My Documents, then under Visual Studio 2010, Projects, Lab6, then Lab6 if you have that folder).
3. In Visual Studio, click on Project, Add Existing Item, and add the files to your project.
4. Edit the comments at the top of the files with your name and the date and at least on sentence of what this program achieves.
5. Start implementing the missing parts.

**Fraction Class**

1. For this lab, a Fraction class will consist of two integers that contain a numerator and denominator. These should be private variables.
2. Next, implement a default constructor. This should just set the numerator and denominator equal to 0.
3. After that, you will want to write the overloaded input and output stream operators. These have the following format:  
    friend ostream& operator<< (ostream& output, const Fraction& fract);  
    friend istream& operator>> (istream& input, Fraction& fract);  
     
   The basic idea of streams are that all of the items in a stream flow together like a stream of water, and you can throw in or remove more things from the stream as you want. For the input stream, you take out the data you like, store it how you wish, then keep the stream going by return input;. For the output stream, you add in the data you want to output, then return the stream by calling return output;.   
     
   Since these functions are really overloading istream’s and ostream’s versions of the functions, you have to make them friends of those classes. To do this, you just add the word “friend” in front of the declaration. You do not add it to the definition, otherwise your compiler will throw an error.  
     
   To write the input (operator>>), you will want to pull out an integer, a slash, and an integer from the stream. The slash you can just ignore. The stream above is called input, and works just like cin and fin have worked in the past. So to read in an integer out of the stream:  
    input >> fract.numerator; // Read one integer out of the stream  
   Remember when you’ve pulled out the data you want, return input;.  
     
   Similarly, the output stream should be similar, only you add data to the output stream. Print fractions by just printing the numerator, a slash, then a denominator.
4. At this point, you may wish to test if your program works. Instead of reading in individual data, you can now make a Fraction object, and just read that in, like this:  
    Fraction f;  
    fin >> f;  
   Since you’ve told it how to read in a Fraction object, this will now build a Fraction object by pulling out the data it needs. You can also cout or fout a fraction object too, and by this point what you read in should be the same as what you write back out.
5. At this point, what’s left is writing the functions to add, subtract, multiply, or divide Fraction objects. If you have two Fractions named f1 and f2 and you add them (f1 + f2), then what really is happening is that f2 is passed as a parameter into f1’s operator+ function. So for example:  
    const Fraction operator+(Fraction rhs);  
   In this instance, rhs is the “right hand side” of the equation. You can then do the math to add numerators and denominators, and return back a Fraction object with the correct result. Note: do NOT modify either the current object or rhs. The result should be passed back as a separate object.  
     
   Do this for the operator+, operator-, operator\*, and operator/ functions. They will all be declared about the same, but will require different implementations.  
     
   All operators should reduce or simplify the fractions after performing the math. For example, 40/100 should reduce down to 2/5. You can do this by just calling reduce() on the Fraction object. For your convenience, we have provided this for you.
6. Finally, implement the overloaded == operator:  
    bool operator==(Fraction rhs);  
   Since == returns whether something is equal (true) or not equal (false), you should return true or false based on whether the numerators and denominators are equal.
7. Finally, the reduce function has been provided for you, but it doesn’t account for a couple of issues regarding negative fractions. You need to implement some more to that function to deal with the negative numbers. Do this by making the numerator negative. For example:
   1. 4/5 should be 4/5 (correct)
   2. -4/5 should be -4/5 (correct)
   3. 4/-5 should be -4/5 (swap negative)
   4. -4/-5 should be 4/5 (double negative is positive)

**File Structure**

The main() function to read in and write out data files has been provided for you. For reference, each line of the input file contains an integer, a slash, an integer, an operator, an integer, a slash, and an integer.

**Grading and Turning In**

Turn in just Fraction.h and Fraction.cpp on Blackboard by the end of class. Lab6.cpp shouldn’t change, so it will be graded with the same version provided on Blackboard. You will be graded using the items listed in the syllabus (algorithm, functional/structural composition, proper input/output, documentation, programming style, and compilation). If you have not completed the lab, turn it in anyway and get partial credit.