# Lab 10

## Polymorphism

Skills Required

* Polymorphism
* Overloaded operators
* Use of overloaded stream operators in file output
* Create and use classes, Exception Handling, Read and write Files, Create Functions, Include Headers and other files, Loops (while, for), Conditional (if, switch), Datatypes, assignment, etc.

**Assignment**

Overloaded operators are functions that redefine how operators (like +, -, \*, /, ==, <, >, <<, >>, and more) work. (Alternatively, you can view operators as functions that have a special calling syntax. A+B calls the operator + function for that data type.) 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. A friend object or function is a non-member that nevertheless can directly access private data. While this is sometimes a useful way to solve a problem, it should be used sparingly, because it breaks encapsulation. More on this below.

**Fraction Class**

1. Unlike most labs, this lab provides you with a little bit of starter code and a finished main program. If you have implemented everything correctly, the main program provided should run without error.
2. For this lab, a Fraction class will consist of two integers that contain a numerator and denominator. These should be private variables.
3. Next, implement a default constructor. This should just set the numerator and denominator equal to 0.
4. After that, you will want to write the overloaded input and output stream operators. Include these lines in your Fraction class header.   
    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 your class. 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. You can read the slash as a character variable, and ignore it. There’s no need to store it as part of the fraction. 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.
5. 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 output a Fraction to cout or to a file, too, and by this point what you read in should be the same as what you write back out. You may want to test your code by reading a Fraction from the keyboard (cin) and writing it to the screen (cout).
6. 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 parameter (rhs) should be passed as a const reference. The result should be passed back as a separate object. It should be passed as a const object, one that cannot be assigned to. That’s to prevent this from being legal:  
   (A + B) = C; // this should NOT work. If the sum A+B returns a const Fraction, it won’t.   
     
   Do this for the operator+, operator-, operator\*, and operator/ functions. They will all be declared about the same, but will require different implementations. The basic pattern inside the function definitions are:
   1. Declare a local Fraction object, to hold the result.
   2. Set the values of the result’s Numerator and Denominator as appropriate.
   3. Return the result.
7. 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.
8. Finally, implement the overloaded == operator:  
    bool operator==(const 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.
9. 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. Or, as your program will read it, 2 fractions with an operator between them.

<https://www.learncpp.com/cpp-tutorial/96-overloading-the-comparison-operators/>

**Submit your assignment**

1. Update all files, clean up the Debug directory as needed, and either commit/sync to GitHub or zip up your project folder and upload it to Canvas.

**The Big Picture: Getting by with (a little) help from your Friends**

Use of friend functions allows non-member code to reach into a class and directly access private data. Friendship does not inherit—something that’s a Friend to a parent class is not automatically a Friend to child classes, or vice-versa. The risk is that not only are we allowing outside code to modify private data, we must now check and possibly update that code if we make a change to the private section, breaking the public/private distinction that makes objects so useful. Thus, declaring a function a friend is usually only done if it’s really necessary to get the job done, or if going through member functions adds an unacceptably large overhead.

Another risk is that by allowing direct access, we’re bypassing any protections in our setter methods. For example, our input operator could happily read in a fraction of 0 / 0 and then call reduce() on itself. If we’d added code to a SetDenominator() method to ensure the denominator was never equal to 0, we’d avoid that.

The point here is not “don’t use friend functions.” Sometimes they really are the most efficient way to solve a problem. But be aware of the risk and extra burden that’s being taken on. Just as you’d think twice, and maybe a third time, before deciding to give a friend the spare keys to your car, you should think carefully before declaring a function as a friend.