CMPT 225 Lab 8: Operator Overloading - Fractions

Start by downloading the zipfile for the lab. It contains code for the Fraction class, a makefile, a set of driver programs and a test script.

Note that the green boxes in this lab and in others are sidebars--longer explanations of incidental things that can be skipped on your first reading. Use those boxes to test and deepen your understanding of C++.

Overloading Arithmetic Operators

C++ allows a programmer to overload operators such as the arithmetic or comparison operators. This allows a programmer to use these operators in a very natural way with objects of classes that they have created. For example:

```
MyClass a;
MyClass b;
// Assign values to objects a and b
MyClass c;
c = a - b; //assign c the result of subtracting b from a
```

In this lab you will overload the arithmetic operators (+, -, *, /) for a **Fraction** class. The **Fraction** class (without the overloaded operators) is provided in the zipfile for the lab.

To write an overloaded operator you need to define the operator in the *.h* file, and write the implementation in the *.cpp* file, just like any other member function. The function name for an overloaded operator is "operator" followed by the symbol for the operator.

Here is the header file entry for the fraction + operator:

```
Fraction operator+(const Fraction & f) const;
```

And here is a stub for the implementation file entry:

```
Fraction Fraction::operator+ (const Fraction& f) const
{
      Fraction result;
      // Calculate result here ...
      return result;
}
```

Memory check

*In the above declaration, **result** is declared as a Fraction, not a Fraction pointer. This means that memory for **result** is allocated with the memory for the function, and that memory will cease to be usable when the function ends (i.e. when **result** goes out of scope).*

*Additionally, the return value is **result**. Is the function returning something that is in memory that will immediately cease to be useable?*

*The answer is yes and no. Yes, it is returning what is in the soon-to-be-gone memory, but it is returning the **value** in that memory, not a pointer or reference to that memory. This has C++'s default **return-by-value** semantics. It returns a copy of what is in the local memory. It does this by using the class's **copy constructor**. A copy constructor is a constructor for a class **Blob** that takes a single argument, which is a **Blob**, and constructs a copy of it (where you can replace **Blob** by any class you like, say **Fraction**). So we can also answer no to the question, in that we are returning a **copy** of what is in the soon-to-be-gone memory.*

*If you are inquisitive, you will have already noticed that **Fraction.h** and **Fraction.cpp** do not contain a copy constructor. So what is happening? What is happening is that C++ is creating a copy constructor for **Fraction**, as it does for any class that does not declare one. This default copy constructor implements a **shallow copy**. This works for **Fraction**. But if you have an object that contains a pointer to dynamic memory, and you want a deep copy, you'll have to write a copy constructor yourself.*

On the other hand, if the declaration were

```
Fraction& Fraction::operator+ (const Fraction& f) const
```

*then we would be in trouble. This declares the function returns a reference to a Fraction, so we would be returning a reference to **result**, not the value of **result**. And that memory we give out the reference to **is** about to become unusable. This type of return is called **return-by-reference**. Whenever you use return-by-reference, ensure that the memory referred to by the return value will still be available after the function ends.*

Implementing an arithmetic operator such as + involves 3 fractions, the one on the left of the operator, the one on the right, and the one that results from adding the two together (the return value). Consider adding two fractions, x and y, and assigning their result to a third fraction, z:

```
z = x + y;
```

The return value is assigned to z, and x is the *receiver* (the one that the overloaded operator "belongs to"), and y is the parameter. That is, the above expression is interpreted exactly like a function call:

```
z = x.operator+(y);
```

Where operator+ is the function name. (Note that you can't normally use '+' in a function name.)

For operators, the receiver is the *left hand operand* of the operator (x in this example).

The receiver

*One conceptualization of object-oriented programming is that a program is a group of interacting objects, and that these objects interact by sending messages to one another. This is a powerful way of thinking about object-oriented programming, because it allows us to think about different variations in the order that messages are sent. These variations can lead to models for parallelism in languages, or definitions of completely new languages that process messages in a different way. In this metaphor, the originator of a message is called the **sender**, and the destination object is called the **receiver**.*

*The conceptual operation of "sending a message" is implemented in C++ and java as "calling a member function." If object **sue** wants to send a message "let's eat dim sum" to object **bob**, this is implemented as a function call **bob.letsEat(dimSum);** somewhere in one of **sue**'s member functions. Then **sue** is the sender, **bob** is the receiver, "let's eat" is the message, and dimSum is the argument.*

Finally, note the use of the **const** keyword in the function header and implementation. Both the receiver and the parameter are declared as constant (the **const** that follows the parameter list declares the receiver is not changed).

Overloading the << Operator for ostream receivers

This tutorial covers the << operator for ostream receivers. (Recall that cout is an ostream.) Please read this carefully and follow along by running the given code. You should modify test_cout.cpp and use it as the driver program for doing so.

The driver program can be built by running "make test_cout", and run by "./test_cout"

It is often useful to overload the ostream << operator, so that you can print an object like this:

```
Fraction f(2,3);
cout << f; //which should print something like 2/3
```

Achieving this is a little more complicated than the process shown previously. There are a couple of reasons for this. The first is that the **<<** operator is itself an overloaded operator (overloaded by the **ostream** class). In theory we could get access to **iostream.h** and mess around with it so that **ostream** recognized **Fraction**s but this is not a good idea.

Instead we will write a 2-operand *function* that overloads the operator. This is in contrast to writing a 1-operand *member function*, as we did for the operator + above. Such a 2-operand operator overload declaration looks like this:

```
void operator<<(ostream os, Fraction f);
```

Whatever function body is given to that operator<< will be used whenever C++ finds an expression x << y where x is an **ostream** and y is a **Fraction**. The function is global--it is not a member of any class. There is no receiver for it. Just two arguments.

We can make this declaration fancier by passing references to the arguments, and declaring that the **Fraction** is a constant.

```
void operator<<(ostream & os, const Fraction & f);
```

Lastly, we're going to make this function a *friend* of **Fraction**. This means that the function can access all of the members of **Fraction**, including private and protected members (variables and functions).

To make this version of operator<< a friend of **Fraction**, we declare it as a friend inside **Fraction**'s class body in its header (.h) file:

```
friend void operator<<(ostream & os, const Fraction & f);
```

Note that the function is **not a member** of the Fraction class, so when you implement it in the .cpp file you should not precede it with **Fraction::**.

A reasonable implementation would look like this:

```
void operator<<(ostream & os, const Fraction & f){
        os << f.numerator << "/" << f.denominator;
}
```

As noted above this is not a member of the **Fraction** class, but because the function is a friend of the class, it can access its private members. Note that in **cout << f;** the first parameter (**ostream & os**) is matched to the left operand, and the second parameter (**Fraction & f**) is matched to the right operand.

Try outputting a fraction using **cout**, it should work OK. Unfortunately, it really isn't good enough as you can see if you put this line of code in your test program:

```
Fraction f(3, 4);
cout << f << " is a fraction!";
```

You will get a compilation error. Why? (That's a rhetorical question.)

The **<<** is a *binary* operator (it takes two operands) and the **cout** statement above is read from left to right so is actually equivalent to:

```
(cout << f) << " is a fraction!";
```

Our operator **<<** function is void (it doesn't return anything) so that is equivalent to:

```
void << " is a fraction!"; //illegal operands compilation errors
```

So we need to return an *ostream* object to be used as the left operand for the **<<** operator. So the final version should look like this:

```
ostream & operator<<(ostream & os, const Fraction & f){
        os << f.numerator << "/" << f.denominator;
        return os;
}
```

Don't forget to change the return type in the header file as well.

Now test it to make sure that it works!

Fraction Class

The code for the fraction class that you should complete is provided in the zipfile (Fraction.h, Fraction.cpp). The *.cpp* file includes a reminder (in comments) of how you would go about implementing the arithmetic operators. Be sure that the arithmetic operators make objects that respect the class invariants defined at the top of Fraction.cpp (fractions are to be kept in lowest terms, with the denominator positive).

Complete the +,-,/,* operators. You can build the test executables by typing "make." They can be automatically tested by following the instructions below.

Testing

There is also a test script (again test.py), which you can run. If you have correctly built executables that solve this lab you will see:

```
uname@hostname: ~$ make
g++ -Wall -c test_add.cpp
g++ -Wall -c Fraction.cpp
g++ -Wall -o test_add test_add.o Fraction.o
g++ -Wall -c test_sub.cpp
g++ -Wall -o test_sub test_sub.o Fraction.o
g++ -Wall -c test_mult.cpp
g++ -Wall -o test_mult test_mult.o Fraction.o
g++ -Wall -c test_div.cpp
g++ -Wall -o test_div test_div.o Fraction.o
g++ -Wall -c test_cout.cpp
g++ -Wall -o test_cout test_cout.o Fraction.o

uname@hostname: ~$ ./test.py
Running test add... passed
Running test sub... passed
Running test mult... passed
Running test div... passed
Passed 4 of 4 tests.
```

If you have passed at least 3 of 4 tests, please ask a TA to take a look at this output, and receive your mark for this lab.