Overview

In this lab you will implement and test a rational number class. You should use the `Complex` number class discussed during the past two lectures to guide you. Many of the member functions you must write here have direct analogs in that example.

A rational number is a ratio of two integers, \( a \) and \( b \):

\[ r = \frac{a}{b} \]

where \( b \neq 0 \). We refer to \( a \) as the numerator and \( b \) as the denominator.

There are many operations on a single rational number \( r \) and on two rational numbers

\[ r_1 = \frac{a_1}{b_1} \quad \text{and} \quad r_2 = \frac{a_2}{b_2}. \]

Here are the operations you need to know about to implement this lab:

\[ -r = \frac{-a}{b} \]
\[ r_1 + r_2 = \frac{a_1 b_2 + a_2 b_1}{b_1 b_2} \]
\[ r_1 \times r_2 = \frac{a_1 a_2}{b_1 b_2} \]
\[ r_1 - r_2 = \frac{a_1 b_2 - a_2 b_1}{b_1 b_2} \]
\[ r_1 == r_2 \text{ if and only if } a_1 b_2 == a_2 b_1 \]

In these definitions, the rational numbers need not be in “lowest terms” (all common factors have been eliminated and \( b > 0 \)), and therefore you do not need to convert a rational number to lowest terms (until the fifth checkpoint, which is extra credit).

General Guidelines

The lab asks you to implement the rational number class in stages, with each checkpoint corresponding to a set of `Rational` functions (members or otherwise). I have written a main program for you with code to test each checkpoint. You can download this from
Initially, all code except that of the first checkpoint is commented out. This main program is an example of the type of code you should be writing to test your own classes.

You will be creating and working on two other files: `Rational.h` and `Rational.cpp`. To complete a checkpoint you must show that your code compiles and show that it works (by uncommenting the appropriate locations in the main program to test it). Finally you must also be careful to demonstrate which member functions are `const` and which are not `const`. (Only member functions may be labeled as `const`.)

**Checkpoints**

There are five checkpoints here, but the fifth is extra credit. Raise your hand to show a TA your work to earn credit for completing a checkpoint. You may not ask for help from the TAs in completing the extra credit checkpoint.

1. Write two constructors and write member functions to access and to set the numerator and denominator of a rational number. (This checkpoint will also require, therefore, that you create the project, create the files `Rational.h` and `Rational.cpp`, and define `private` integer member variables in the `Rational` class.) Here are more details on the six member functions you must write.

   - The first constructor should accept two arguments: the initial numerator and the initial denominator, in that order. There should be default values for these. The default for the numerator should be 0 and the default for the denominator should be 1.
   - The second constructor should be a “copy constructor”. That is, it should initialize a `Rational` object from an existing `Rational` object.
   - The accessor member functions should be just called `numerator` and `denominator`. These should just return the values of the internal numerator and denominator.
   - The setting member functions should be called `set_numerator` and `set_denominator`, and each should accept a single integer argument.

The main function testing/demonstrating the use of these functions demonstrates the various ways the first constructor may be called. Stream operators are the subject of the second checkpoint.
Aside: There are several instances here where code that uses the Rational class can create illegal values by setting the denominator to 0. The best approach to handling this situation is to throw an exception. We aren’t using exceptions in this course, however. The next best option is to use an assert. For this lab, you may simply ignore the possibility and assume that setting the denominator to 0 never happens.

2. Write and test the input and output stream operators and the assignment operator. Recall that the input and output stream operators can not be member functions. The input stream operator should just try to read two integers as the numerator and the denominator and return the stream. Note that no prompting for input is used from inside the operator. The output stream operator should just output the numerator followed by a “:” followed by the denominator. For example

```cpp
Rational x( 5, 2 );
cout << x << endl;
```

should output

```
5:2
```

Make the stream operators friend functions.

3. Write three operators. Each will be implemented in a different form, and it is particularly important that you study the example of the Complex class to help. Here are the operators.

- Implement (binary) `operator-` as a member function.
- Implement `operator+` as a friend function.
- Implement `operator*` as a non-member and a non-friend function. In this case, the operator must access the contents of the Rational objects it works on through their public member functions and constructors. Also, the function prototype should appear in the file `Rational.h` after the class declaration is over. See the ostringstream `operator<<` in the str class for an example.

Each function should return a Rational object.
4. Implement three more operators:

- **operator+=** This should be a member function (although it is not strictly necessary). The operator must accept a single, const Rational object as an argument, and it must return a reference to the current Rational object (*this).

- **unary operator-** This must return a Rational object.

- **operator==** This must return a bool. It can be a member function, a friend function or neither — your choice.

5. **(Extra Credit!)** Implement and test a member function called `lowest_terms` that eliminates common integer factors between the numerator and denominator in a Rational number and makes the denominator positive. Its return type should be void.