Lab 4:
Be Rational

Due date: March 10

Assignment

In this lab, you will design a simple C++ class for representing rational numbers. A *rational number* is any number that can be represented as a ratio (division) of two integers (the *numerator* and the *denominator*). 5 ($\frac{5}{1}$), one third ($\frac{1}{3}$), zero ($\frac{0}{1}$), and 2.5 ($\frac{5}{2}$) are rational numbers; π, e, and $\sqrt{2}$ are irrational numbers. By convention, only the numerator can be negative.

You will have two tasks in this assignment: design and code the `Rational` class, and then test your class with a tester main method.

1. Design and implement a class `Rational`. Put the class declaration in a .h file with include guards as shown in class, and the implementation in a .cpp file that has #included the .h file. The class should have the following member variables and methods:

   (a) Two private member variables for storing the numerator and denominator components of the number, both `int`s.

   (b) Two constructors:

      i. a default constructor that performs no action, and initializes the numerator to be 0 and the denominator to be 1.
      `Rational num; // the number 0 (a.k.a.  0/1)`

      ii. a constructor taking two int parameters representing the numerator and denominator components of the rational number. Example:
      `Rational num2(1, 3); // this is the number 1/3 = 0.3333.....`
      This constructor should call the `Normalize` method outlined below to reduce the number to lowest terms.

   (c) Two sets of accessors and mutators:

      i. Methods to access and mutate the numerator of the number: `GetNumerator` and `SetNumerator`;

      ii. Methods to get and set the denominator of the number: `GetDenominator` and `SetDenominator`;

      iii. Reminder: *access* is the same as *get*, and *mutate* is the same as *set*. You need to be comfortable with both sets of terms.

   (d) These simple methods:

      i. `void Normalize()`
      This **PRIVATE METHOD** puts the rational number into "normal form." In normal form, a rational number has a positive denominator, and only has a negative numerator if the number is negative. Additionally, the numerator and denominator are reduced until they have no factor in common. You ***must*** call the `Normalize` method *any time the numerator or denominator is changed.*

      To normalize, perform two tasks: if the number should be negative (how can you tell?), make the numerator negative and the denominator positive. Then reduce the numerator and denominator by finding their greatest common divisor (remember Homework 2? you may add a `gcd` private function to your class) and dividing each by the gcd (if it is not 1).

      The net effect is demonstrated here:
      ```
      Rational num1(5, -20); // normalized to -1/4
      Rational num2(-10, -2); // normalized to 5/1
      ```

ii. `boolean Equals(const Rational &other)`
This public method returns true if two rational numbers are equal. You can assume each number is normalized.
```
Rational num1(5, -10);
Rational num2(-13 - 2, 2 * 15);
bool isEqual = num1.Equals(num2); // == true
```

iii. `Rational Add(const Rational &other)`
This public method creates and returns a new `Rational` object representing the sum of the current rational number with the parameter `other`. Don't overcomplicate this; figure out how to add two numbers $\frac{a}{b} + \frac{c}{d}$, find their new numerator and denominator, then construct and return a `Rational` object with those values. Remember, the constructor will normalize the new number for you.

iv. `std::string ToString()`
This public method returns a String representation of the `Rational`. If the denominator is 0, just return the numerator. Otherwise, return the number formatted as `n/d` where `n` is the numerator and `d` the denominator. Review the `ToString` method of the `Card` class from lecture.
```
Rational num1(5, -10);
cout << num1.ToString(); // prints "-1/2"
Rational num2(8, 2);
cout << num2.ToString(); // prints "4"
```

(e) **IMPORTANT:** any function that does not modify the numerator/denominator member variables must be declared `const`. Ask me if you aren't sure which functions this refers to. To clarify, the **function** is const, not its **return type**.

2. Add a second file to your project containing a main function. Include your .h file for the `Rational` class.

   (a) Construct two rational numbers, one using the default constructor and one using the two-parameter constructor. Call these variables `r1` and `r2`, and let `r2` be the number $\frac{5}{12}$. **Declare r2 to be const.**

   (b) Demonstrate your `ToString()` method by printing out the value of `r2.ToString()`.

   (c) Use the `SetNumerator` and `SetDenominator` methods to change `r1` to have a numerator of 48 and a denominator of 36.

      i. While you're at it, try using SetNumerator on r2, and make sure you understand why the compiler gives an error.

   (d) Show that your mutator methods worked, by printing the value of `GetNumerator` and `GetDenominator` for `r1`.

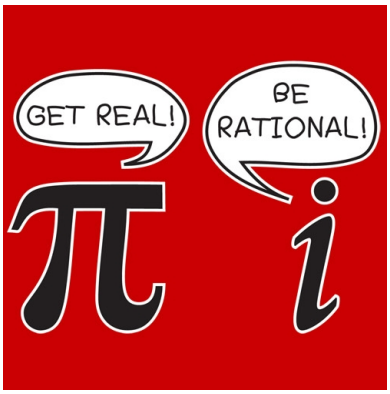
   (e) Print the result of the `.Equals()` method to see if `r1` is equal to `r2`.

   (f) In a single statement, output the result of `r1.Add(r2)`. (Hint: do not store the result in a variable; just figure out how to print the result of the function call.)

Deliverables

Hand in:

1. A printed copy of your code, **printed from Visual Studio or your IDE when possible.** If you cannot print from your editor, copy your code into Notepad or another program with a fixed-width (monospace) font and print from there. Print **all .h and .cpp files**.
2. The output of your code, as described in step 2 above.
3. **Note**: your code must declare exactly the methods described in this spec, and **nothing more**.


GET REAL!
BE RATIONAL!
π
i