CS256 — Exercise 5

February 3, 2016

Due: Friday, February 12, 2016 before midnight (60 points)

1 Preparing the Project

I highly recommend that you work with a partner for this exercise.

1. Go to https://codebank.xyz and you should see a project in the CS256 group named `CS256-EX5`. Fork this project for your own user.
2. Next, clone the repository so you have a local copy:

   ```
   $ git clone https://codebank.xyz/username/CS256-EX5.git
   ```

3. The repository should have three files: `rat.cpp`, `rat.h`, and `main.cpp`.

2 The `rat` class

In this exercise, we will create a very robust class for rational numbers (fractions) called `rat`. Once you're done, you will see that C++'s features such as implicit conversion and operator overloading can make a class act very much like a built-in type such as `int`, which is why we are naming our class in a similar manner, so users will think of it like a primitive numeric type.

You have been provided three files: the class header, class implementation, and a test driver. The class header and implementation will start mostly empty. The test driver has a class that is marked a friend of `rat` so it can access private members as needed. I recommend looking through the tester class to see what kinds of tests it is performing as you write your functions.

1. Compile and run the provided files. You should see the first test has passed already as long as all of the files exist as expected.
2. Next, add two instance variables to `rat`:
   (a) An `int` named `num` for the numerator of the fraction.
   (b) An `int` named `den` for the denominator of the fraction.
3. Add the following constructors (provide declaration in header file and implementation in cpp file):
   (a) A default constructor that initializes the `rat` to $\frac{0}{1}$.
   (b) A constructor that takes an `int` named `n` and makes the fraction $\frac{n}{1}$.
   (c) A constructor that takes two `int`s named `n` and `d` and creates the fraction $\frac{n}{d}$. If the denominator provided is 0, set the entire fraction to $\frac{0}{1}$.

Note: use the initializer list technique for making the constructors discussed in class on Monday. For the third constructor, in the constructor body you can include an `if` statement that checks for invalid denominator after the initialization.

4. After adding them, uncomment the second test function call from `main.cpp`. If your constructors are correct, your tests should all pass.

5. Add getters for each instance variable named `getNum` and `getDen`. Make sure to make them `const` since they won't be used to modify the object calling them.

6. Uncomment the next test function to test the getters.

7. As a helper function, we will include a function that reduces a fraction. Create this as a private member of `rat` and use the following implementation:

```
void rat::reduce()
{
    int g = gcd(num, den);
    num /= g;
    den /= g;
    if (den < 0)
    {
        num = -num;
        den = -den;
    }
}
```

   This function will take a fraction like $\frac{10}{2}$ and set it to $\frac{5}{1}$ by dividing by their GCD. It will also make sure the negative sign is only on the numerator, so a fraction like $\frac{6}{-9}$ becomes $\frac{-2}{3}$.

8. Now, go back to the third constructor and add a call to `reduce()` at the end of the body so that any fraction we make is properly reduced.

9. Uncomment the next test function call from `main.cpp` to test your constructors again.

10. Next, create a `reciprocal()` function with the following prototype:

```
rat reciprocal() const;
```

    Given a fraction $\frac{a}{b}$, this function should return $\frac{b}{a}$ properly reduced.

11. Uncomment the next function call in `main.cpp` to test it.

12. Now, we will add overloaded arithmetic operators so we can perform arithmetic with our fractions. These operators should take a `const` reference variable of type `rat` as a parameter which corresponds to the second operand while `this` corresponds to the first. They will return a `rat` object that is the result of the operation. The prototype for the `+` operator should be the following:

```
rat operator+(const rat& right) const;
```

    From this, you can determine the proper prototypes for the `-`, `*`, and `/` operators.

    Implementing addition can be a bit tricky, write down on a piece of paper the expression $\frac{a}{b} + \frac{c}{d}$ and do the algebra to see what the result should look like. If you have any questions about this, let me know.

    Once you have implemented addition, use it to implement the subtraction so you don't need to repeat similar code. For division, use your multiplication and `reciprocal()` function.

    Make sure you call `reduce()` at the end before returning the answer for all four operators so we always get the most reduced form as the result.

13. After implementing them, uncomment the next test function and test your program again.
14. Now we will implement the relational operators for `rat`: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

    These operators should take a `const` reference of type `rat` as a parameter as the second operand, return a `bool` as the result, and be `const` because they aren't going to modify the object calling them.

    The prototype for `==` should look like:

    ```
    bool operator==(const rat& right) const;
    ```

    From this, you should determine what the other prototypes should look like.

    Hint: Implement `==` and `<` first. It is possible to implement all of the other operators in terms of these. For example, to implement `>` given `==` and `<`, you could use the logic that `>` is the same as *not* `==` *and not* `<`.
15. Uncomment the next function to test your relational operators.
16. Finally, we will create a function `str()` that will act like Java's `toString()`. Given fraction $\frac{n}{d}$, it should return a string that looks like `n/d` that is completely reduced.

    It should have the following prototype:

    ```
    std::string str() const;
    ```

    Hint: to implement `str()`, you can use a `std::stringstream` from the `sstream` header (already included for you in `rat.h`).
17. Uncomment the final test function to test your `str()` function.
18. There are still more things to add to this class if you would like to improve it, but only the above are graded. Here are examples of things to add:
    (a) Implicit conversion to `double`
    (b) Implicit conversion from `double`
    (c) Implement `>>` and `<<` operators for use with `istream` and `ostream` respectively
    (d) Implement combined assignment operators, e.g., `r1 += r2;`
    (e) Refactor to use `long` or `long long` to allow for larger range of fractions
    (f) Add additional functions for other mathematical operations that might be performed on fractions

3 Submission

Once you have completed the program, you can use `git add`, `git commit`, and `git push` to push the changes to https://codebank.xyz. You can make as many commits and push as many times as desired until the deadline.