PIC 10B: Homework 2

Due February 6

The `zip` file you extracted to find this `pdf` also includes three files called `rational.hpp`, `rational.cpp`, and `main.cpp`. In this assignment, you will edit and submit `rational.hpp` and `rational.cpp`. You are strongly encouraged to edit `main.cpp` to test your code. However, you should not turn in `main.cpp`.

Make sure you follow these instructions:

- Upload `rational.hpp` and `rational.cpp` to **Gradescope** before the deadline.
- Name the files exactly as just stated, with no extra spaces and no capital letters.
- Do **not** upload `main.cpp` to Gradescope.
- Do **not** enclose the files in a folder or zip them. You should be submitting **exactly two files** and they should have extensions `.hpp` and `.cpp`.
- When working on your homework assignment:
  - do **not** change `num` or `den` to be `public`;
  - do **not** use the `friend` keyword more than the 3 times it is already used in `rational.hpp`;
  - **violating either of these conditions will result in a score of 0 points.**
- Be sure that your code builds and runs with my files on **Visual Studio 2022**.

Grading: We will use test cases to check your operators one by one.

The rationals and doubles

The *rational numbers* are a fancy name for the fractions you learn about in elementary school. They consist of a *numerator* and *denominator*, and we often write them with the numerator above the denominator separated by a horizontal line, e.g., $\frac{1}{6}$.

Consider the following code:

```
int main() {
    double sixth = 1.0 / 6.0;
    std::cout << std::boolalpha;
    std::cout << (sixth + sixth + sixth + sixth + sixth + sixth == 1.0);
    std::cout << std::endl;
    return 0;
}
```

It prints `false` to the console. This gives proof that `double`s are not very good at storing rational numbers. Of course, we know that $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = 1$, but a `double` is unable to accurately store $\frac{1}{6}$. As a consequence, `sixth + sixth + sixth + sixth + sixth + sixth` actually gives a number a little smaller than the `double` 1.0. The goal of this assignment is to fix this issue. You will write a class called `Rational` and, upon successfully completing the assignment, executing the following code will give `true`:

```
#include <iostream>
#include "rational.hpp"

int main() {
    Rational sixth(1, 6);
    std::cout << std::boolalpha;
    std::cout << (sixth + sixth + sixth + sixth + sixth + sixth == Rational(1));
    std::cout << std::endl;
    return 0;
}
```

Rationals representation is not unique

It is important to note that a rational does not have a unique representation:

$$\frac{1}{2} = \frac{2}{4} = \frac{3}{6} = \frac{4}{8} = \frac{5}{10} = \frac{6}{12};$$

$$\frac{-1}{3} = \frac{1}{-3} = \frac{-222}{666} = \frac{222}{-666}.$$

Throughout this assignment, we will store rationals in a convenient *simplified form*:

- A denominator should always be strictly positive.

- A numerator and denominator should always have a highest common factor of 1.

So, to store the fraction $\frac{6}{12}$, we would store a numerator of 1 and a denominator of 2. To store the fraction $\frac{222}{-666}$, we would store a numerator of -1 and a denominator of 3.

`rational.hpp` and `rational.cpp`

The file `rational.hpp` contains a partial class interface for a class called `Rational`. An instance of a `Rational` has the following *private* member variables:

- `num`: an `int` for storing the numerator of a rational;
- `den`: an `int` for storing the denominator of a rational.

You are not allowed to change these member variables to be `public`. Doing so will result in 0 points.

The file `rational.cpp` contains a few definitions already. The default constructor creates $0 = \frac{0}{1}$, the rational with numerator 0 and denominator 1. Note that any integer can be regarded as a rational with denominator equal to 1, so there also is a constructor based on this idea. Finally, a user can specify a `_num` and `_den` in order to construct a `Rational` of their choice. This constructor calls two helper private member functions `make_den_pos` and `simplify` in order to store the rational in its simplified form. The `operator>>` is also defined. Its signature is: `std::istream& operator>>(std::istream&, Rational&)`.

Tasks

1. Define `std::ostream& operator<<(std::ostream& out, const Rational& r)`. So
   ```
   std::cout << Rational(8) << ' ' << Rational(7, 11) << std::endl;
   std::cout << Rational(-8) << ' ' << Rational(-7, 11) << std::endl;
   ```
   should print
   ```
   8 7/11
   -8 -7/11
   ```
   Don't use `std::cout` in the function definition. (This operator should work just as well for file streams.)

2. Define `operator+=`, `operator-=`, `operator*=`, and `operator/=`. It may help to see the formulas:
   $$\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}, \quad \frac{a}{b} - \frac{c}{d} = \frac{ad-bc}{bd}, \quad \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}, \quad \frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}.$$

Make sure to follow the guidance given in lecture and the code snippets to ensure the correct signatures. Are these operators better as free functions or member functions? How should references and `const` be used? Remember:

- We are storing the rationals in their simplified form.
- `operator/=` should say that an assertion failed in the case of division by zero.

3. Define `operator++` (pre++ and post++) and `operator--` (pre−− and post−−). Follow the guidance given in lecture to ensure the correct function signatures.

4. Define the **unary** operators `operator+` and `operator-`. Follow the guidance given in lecture to ensure the correct function signatures.

5. Define the **binary** operators `operator+`, `operator-`, `operator*`, and `operator/`. Follow the guidance given in lecture to ensure the correct function signatures.

   - Do not change `num` or `den` to be public.
   - Do not use the `friend` keyword more than the 3 times it is already used in `rational.hpp`.
   - Doing so will result in 0 points.

6. Define `operator==` and `operator<`. Follow the guidance given in lecture to ensure the correct function signatures.

   - **Remember:** the entire point of writing this class is to overcome the imprecision of doubles. Do not use any doubles while defining these functions!

7. Define `operator!=`, `operator>`, `operator<=`, and `operator>=`. Follow the guidance given in lecture to ensure the correct function signatures.

   - Do not change `num` or `den` to be public.
   - Do not use the `friend` keyword more than the 3 times it is already used in `rational.hpp`.

8. Define `operator double()`. Follow the guidance given in lecture to ensure the correct function signature.

   Although doubles cannot perform arithmetic perfectly, they are still incredibly useful. It would make sense for someone to want to convert a rational to a `double`, which approximates the rational. Your operator definition should ensure that the following code prints `true` twice:

```
double d = static_cast<double>(Rational(1, 2));
std::cout << std::boolalpha;
```

```
std::cout << (0.5 == d) << std::endl;
std::cout << (0.5 == static_cast<double>(Rational(1, 2))) << std::endl;
```

However, it should also ensure that the following code prints `3/2` withot errors:

```
Rational r(1, 2);
std::cout << (1 + r) << std::endl;
```

In other words, we don't want to allow for *implicit* conversions into `double`!