CS11 Intro C++ Lab 5: Operator Overloading

This assignment will give you an opportunity to practice C++ operator overloading best-practices (and other C++ best practices) in a reasonably simple context. You will need to implement a simple `Rational` class that represents rational numbers as *numerator / denominator*. The goal will be to implement most of the basic arithmetic operators, as well as the stream-output operator.

Your `Rational` class should be implemented in the files `rational.h` and `rational.cpp`. The class should also satisfy the following requirements:

- Represent positive and negative rational numbers as two `int` values, numerator and denominator. The denominator should never be negative or zero; the numerator may be any whole number.

- Support initialization of the forms `Rational{}` (initializes to 0/1), `Rational{n}` (initializes to `n`/1), and `Rational{n, d}` (initializes to `n`/`d`). If `d` is 0, throw a `std::invalid_argument` exception with a suitable error message. If `d` is negative, invert the sign of both `n` and `d` so that `d` is positive (for example, `Rational(-3, -5)` = 3/5).

  Don't forget that functions can specify default arguments; that is a very easy way to implement this set of initialization call-patterns. Additionally, the one-argument call-pattern will allow the compiler to implicitly convert integers into `Rational` numbers.

- Provide a member function `num()` that returns the numerator, and another member function `denom()` that returns the denominator.

- Provide a member function `reciprocal()` that returns a `Rational` object containing the reciprocal of the number (but leaves the number itself unchanged). (This function should also throw if the reciprocal has a 0 for the denominator, but that will happen automatically if your constructor is implemented correctly.)

- Provide a member function `reduce()` that reduces the rational number such that the greatest common divisor of the numerator and denominator is 1. Note that this function should mutate the `Rational` object that it is invoked on.

  You may find it helpful to implement a `gcd(a, b)` helper function; if you do this, make sure to put it in the `private` section of your class.

  Also, if the numerator is 0 and the denominator is not 1, your `reduce()`

function should modify the number to 0/1.

- Support addition, subtraction, multiplication and division with both the simple arithmetic and compound assignment operators.

  All of these operations should generate a result that is fully reduced; for example, 2/5 + 6/10 = 1/1.

  Recall that compound-assignment operators `+=`, `-=`, `*=` and `/=` are best to implement as member operator-overloads, and the simple arithmetic operators `+`, `-`, `*` and `/` are best to implement as non-member operator overloads.

  Additionally, think about how you might reuse your work. For example, you might implement `-=` in terms of `+=`, and `/=` in terms of `*=` using the `reciprocal()` operation.

- Support stream-output using the << operator. If the denominator is 1, only output the numerator; if the denominator is not 1, output "*numerator/denominator*".

- Make sure to follow `const`-correctness guidelines and use `const` and references everywhere that is appropriate.

- Comment your class thoroughly in the Doxygen style, including a class-level description, and a comment for every data-member and member-function. (Comments may be brief if the function's purpose is obvious, but you must still comment all of these things.)

  You do not need to create a `Doxyfile` or run Doxygen this time, as long as you follow the Doxygen format properly.

- Make sure to comment illegal arguments, and exceptions that may be thrown and the circumstances that would cause them to be thrown.

Testing

A test suite for your class is provided here; make sure to fix any test failures you encounter.

Build Automation

Now that you know how to use `make` to automate your build process, create a `Makefile` for your project that builds and tests your `Rational` code.

- The `all` target should build the test binary, but not run it.
- The `test` target should run the test binary.
- The `clean` target should delete `.o` files and binaries. **As always, make sure to back up your code before testing your `clean` target for the first time!**
- Make sure all files are compiled with the `-Wall` and `-Werror` arguments (these will go in your `CXXFLAGS` variable) so that any malformed but otherwise legal code is identified by the compiler.

Submitting Your Work

Once you have completed the above tasks, submit your work through csman. Make sure to submit these files:

- `Makefile`
- `rational.h`
- `rational.cpp`

You do not need to submit the test code; we will test your program with a fresh copy of these files.

Assignment Feedback Survey

Please also complete and submit a feedback survey with your submission, telling us about your experience with this assignment. Doing so will help us to improve CS11 Intro C++ in the future.
