Rational Thinking

Fundamental number types like `int` and `double` are limited in both the range of values they can hold, and the precision with which they can represent fractional values – $\frac{1}{10}$, for example, has no exact floating point representation. This situation can be improved in two steps – firstly creating a representation for arbitrarily large integers; then using these integers to implement rational numbers (fractions). This has, of course, been done many times and some languages (such as Python) support exact integers automatically. In the time-honoured tradition of reinventing the wheel, we're asking you to implement firstly large integers, then rationals. You'll be doing this in C++, and learning about its operator overloading mechanism along the way.

This is also an étude about modularity, encapsulation, reuse, and algorithm choice. Your submission should consist of exactly four files – `Integer.h`, `Integer.cpp`, `Rational.h`, and `Rational.cpp`, implementing classes to represent large integers and rational numbers respectively. Both classes should be contained within the `cosc326` namespace. You should consider your choice of algorithms and data structures carefully. While we are not expecting elaborate optimisations or convoluted algorithms, we do expect your implementation to be reasonably efficient.

Task

Your `Integer` class should implement the following functions:

- The following constructors:
  - A default constructor that creates `Integer` with a value of 0
  - A copy constructor that duplicates the provided `Integer`
  - One that takes a `std::string` of digits (possibly with a leading `+` or `-`)
- A destructor
- The assignment operator: `=`
- The unary operators: `+` and `-`
- The binary arithmetic operators: `+`, `-`, `*`, `/`, and `%`
- The compound assignment operators: `+=`, `-=`, `*=`, `/=`, and `%=`
- The comparison operators: `==`, `!=`,`<`,`<=`,`>`, and `>=`
- The streaming insertion and extraction operators: `<<` and `>>`
- A function, `gcd(a,b)`, that returns the greatest common divisor of two `Integer`s

The various operators should have the same semantics as C++ `int`s. There are quite a lot of operators, but many can be defined in terms of one another. For example, if you have already defined `==`, then `!=` should be trivial. You should use this to maximise encapsulation and reuse by minimising the number of functions that are explicitly defined, and using non-member functions where practical. You may, of course define additional methods to help implement the required functionality, but this should not be exposed to users of the class. Your implementation must not use any libraries that permit arbitrary precision arithmetic, or which implement rational arithmetic, but may use ordinary `int` types and operations.

You should then use your `Integer` class to create a `Rational` class with the following functions:

- The following constructors:
  - A default constructor that creates `Rational` with a value of 0
  - A copy constructor that duplicates the provided `Rational`
  - Constructors taking one, two, or three `Integer`s with the following results:

```
Rational r1(a);       // r1 = a
Rational r2(a, b);    // r2 = a/b
Rational r3(a, b, c); // r3 = a + b/c
```

  - A constructor that takes a `std::string` parameters.
- A destructor
- The assignment operator: `=`
- The unary operators: `+` and `-`
- The binary arithmetic operators: `+`, `-`, `*`, and `/`
- The compound assignment operators: `+=`, `-=`, `*=`, and `/=`
- The comparison operators: `==`, `!=`,`<`,`<=`,`>`, and `>=`
- The streaming insertion and extraction operators: `<<` and `>>`

When your class inserts a rational into a stream, it should use the following format:

1. Values which are whole integers should be output as such.
2. Values which are in the range $(0, 1)$ should be output as n/d, where n is the numerator and d the denominator in their simplest form.
3. Values which are greater than one and not whole numbers should be output as $w\,.\,n/d$, where w is the whole integer part, and n/d is a fraction less than 1.
4. Negative values should have a leading minus sign.

Input to the `std::string` constructor or the stream extraction operator should be less strict, allowing for improper fractions, leading zeroes or plus signs, etc.

For example, the following are valid outputs for a `Rational`:

- `1`
- `2/3`
- `-3.1/2`

The following are invalid for output, but should be accepted as input:

- `0.1/3`
- `4/3`
- `-3.2/4`
- `+15.32/2`

Relates to Objectives

1.2 2.2 2.10 3.1 3.4 3.5 3.6 4.4 4.5

(Pair 2)