MTH 4300
Spring 2025
E. Fink

Homework 6

---

rational.cpp

rational.h

test.cpp

expressions.cpp

Recall that calculations involving floating point types (like `double`s) suffer from imprecision, due to their being stored in binary. For example, `0.1 + 0.2 == 0.3` will evaluate to `false`, because `0.1 + 0.2` will evaluate to a value that is a tiny tiny bit greater than `0.3`.

For some purposes, this imprecision is unacceptable. For these situations, an alternative is to create a class for *rational* numbers – i.e, fractions where both numerator and denominator are integers. Then, for example, instead of working with 0.1 and 0.2, you can work with $\frac{1}{10}$ and $\frac{2}{10}$. The advantage of this is that unlike floating point values, integers don't have any imprecision.

Create a class definition for a class called `Rational`. The declaration of the class should be written in `rational.h`, and the implementation should be written in `rational.cpp`. There is test code in the file `test.cpp`. Finally, you will then import the class to reduce three complex rational expressions (as described further down) in `expressions.cpp`.

The objects of the class should have the following *private member variables*:

- `long long num`, representing the *numerator*.
- `long long den`, representing the *denominator* – this should be arranged to always be POSITIVE.

The class definition should also contain the following *member functions*, public except where specified:

- a PRIVATE member function `long long gcf()`, which should return the greatest common factor of `std::abs(num)` and `std::abs(den)` (this should be useful for the next function). This function should use the Euclidean algorithm, implemented non-recursively (via a `while`-loop). In case you are unfamiliar with the Euclidean algorithm, here is some code for the body of the function, which returns the GCD of `x` and `y`, assuming that `x` and `y` are positive:

```
while (y!=0) {
    long long temp = y;
    y = x % y;
    x = temp;
}
return x;
```

- `void reduce()`, which should change `num` and `den` so that $\frac{\texttt{num}}{\texttt{den}}$ is in lowest terms; additionally, if `den` is negative, then both `num` and `den` should be multiplied by -1.
- three constructors: one that receives two long long arguments; one that receives one long long argument `n` setting `num`, and sets `den` to 1; and a default constructor which assigns `0` to `num` and `1` to `den`. Each of these constructors should call `reduce()` before returning. (You are not required to manage the case where `den` is 0 – this would be a good use for exception handling, but we haven't discussed this yet.) You can also use one constructor with default parameters if you wish – in that case, be sure to specify the default parameters in the *declaration*, but then write the constructor itself, with the call to `reduce()`, in the implementation file.
- `Rational& operator+=(const Rational&)` (as well as similar overloads for $-=$, $*=$, $/=$). These methods should update the numerator and denominator of the called-upon object (i.e., `this`). After updating these, the function should call `reduce()`. They should also RETURN `*this`, so that they can be used in compound assignments (`x = y += z;`).

The class definition should also contain the following *non-member friend functions*:

- `Rational operator+(const Rational&, const Rational&)` (as well as overloads for $-$, $*$, $/$). These friends should return new `Rational` values, and `reduce()` them before returning.
- `bool operator==(const Rational&, const Rational&)` (as well as overloads for $!=$, $<$, $>$, $<=$, $>=$). Note that you really only have to implement `==` and `<`: all the rest can be written by using calls to these two. For `<`, you can use cross-multiplication, but be very careful about the signs!

- `std::ostream& operator<<(std::ostream&, const Rational&)`, which prints out `Rational`s in the form `5 / 2`.
- `std::istream& operator>>(std::istream&, Rational&)`, which accepts type Rational numbers of the form `5 / 2`.

Your class implementation should make my client code work in `main()` work. It should be written in the style that we have introduced: the class definition should only provide member function declarations, with member function implementations outside (constructors excepted). Also, access-only member functions should be marked as `const`.

Finally, use this class to write code in `expressions.cpp` which reduces the following expressions:

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7}, \quad \frac{1}{1^3} + \frac{1}{2^3} + \frac{1}{3^3} + \frac{1}{4^3} + \frac{1}{5^3}, \quad 1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{4 + \cfrac{1}{5 + \cfrac{1}{6 + \cfrac{1}{7 + \cfrac{1}{8}}}}}}}$$

Specifications: your submission must

- contain a declaration (in `rational.h`) and definition (in `rational.cpp`) of the class `Rational` that includes all the members listed above, which allow my test code in `test.cpp` to run properly.
- follow the style we have set forth in class, with the class definition containing only declarations of functions, not implementations (aside from constructors), and with functions marked as `const` if appropriate.
- in `expressions.cpp`, find simplified rational representations of the various rational expressions shown above.