C++ Foundation, Assignment 3

In this assignment you'll write a program that performs calculations on rational numbers. Next to it, you'll have to make UML class diagrams of the classes in your program.

Part 1: Rational number class

Design and program a class `rational` that can represent *rational numbers*. Rational numbers can be written as a fraction where both the numerator and the denominator are integers. Some examples of rational numbers are: 1/1, 45/31, -4/5, 125/3, -42/11, 0/1, 85/1 and so on.

The class `rational` should have the following *public* functions:

- Default constructor that initialises the number with a value of 1/1.
- Constructor that takes two integers (numerator and denominator).
- Constructor that takes only one integer (the denominator is assumed to be 1).
- Constructor that takes a `std::string` argument that contains a valid representation of a rational number in decimal base. (e.g. "-34/2" or "(56/77)" - brackets are optional).
- `num` and `den` for obtaining the values of a numerator and a denominator.
- `set_num` and `set_den` for setting them.
- `to_str` that returns a `std::string` representation of a `rational` instance.
- `to_double` that returns a double value of a `rational` instance (e.g. 3/5 == 0.6).
- `is_positive` and `is_negative` that returns a `bool` value.
- `is_inf` and `is_nan` that return `true` if a `rational` instance contains infinity of *Not a Number* respectively.

Additionally you'll need to implement a `gcd` (member) function that calculates the *greatest common divisor* of two integers.

A sign of a number should be only kept in its numerator. If a `rational` is constructed with numbers `4` and `-5` (a rational number -4/5), it should store it with a numerator of `-4` and a denominator of `5`.

Fractions should be always stored in the simplest form. For instance, a `rational` initialised with:

- `45` and `5` should be stored as `9/1`.
- `385` and `33` should be stored as `35/3`.
- `22` and `-4` should be stored as `-11/2`.

Fractions must be also simplified after changing a numerator or a denominator with the `set_num`/ `set_den` functions. Use the `gcd` function to simplify fractions. (Simplified form is the class invariant).

If both the numerator and the denominator are `0`, the `rational` instance represents a NaN value.

If only the denominator is `0`, the `rational` instance represents infinity. The sign of the infinity depends on the sign of the numerator.

Part 2: Operations on rational numbers

Write a class `rational_calc` that implements the following operations:

- `add`, `subtract`, `multiply`, `divide` for performing those operations on `rational` numbers. Each of those functions takes two `rational` objects and returns a `rational` object.
- `pow` that performs exponentiation and takes a `rational` object and an integer exponent. It returns a `rational` object.
- (optional) `sqrt` that calculates square root of a `rational` number. If performing this operation is not possible because either the numerator's or the denominator's roots are not integers or the `rational` number passed to this function is *negative*, *infinity* or *NaN*, the function should return a `rational` instance representing NaN.

Add a function `calculate` to your `rational_calc` class. This function takes a `std::string` argument that contains an algebraic expression on rational numbers. It performs the calculation if it's possible and returns its result as a `rational` object:

```
rational rational_calc::calculate(std::string expresssion);
```

Assume valid input. Numbers are always separated by spaces from operators. All numbers are formatted as rational numbers, except for exponents — those are non-negative integers. A square root expressions begins with the characters `sqrt` followed by a rational number. Some examples of valid expressions:

```
3/4 + 67/-3
7/5 * 45/9
-143/57 / 32/9
10/8 - 0/3
4/5 - 9/-4
5/6 ^ 2
64/-8 ^ 3
4/0 + 5/3
6/4 * 0/0
sqrt 144/9
sqrt -36/25
sqrt 36/6
-3/11 ^ 0
```

The results of those expressions are:

```
-259/12
7/1
-429/608
5/4
61/20
25/36
-512/1
1/0 (+inf)
0/0 (NaN)
```

```
4/1
0/0 (NaN)
0/0 (NaN)
1/1
```

Part 3: file processing

Add a class `rational_processor` that can process text files that contain algebraic expressions on rational numbers. This class should have one public function `process` that takes a `std::string` file name. It should:

- open this file,
- read the expressions line-by-line, and while doing so:
  - calculate the results of the expressions using the `rational_calc` class,
  - write the results of those calculations, one line per result, to an output file.
- return `true` on success or `false` if a file with the given name couldn't be opened.

The `process` function should accept a file name (`file_name`) with any extension (e.g.: `.in`, `.txt`) and write its output to a file named `file_name.out`.

This class should either have a data member of type `rational_calc` or use it as a local variable in the `process` function.

Part 4: program

Finish your program by completing the `main` function. Your program should ask a user if she wants to perform calculations on hand-entered expressions or process a file. After getting the user choice, the program should proceed by either accepting rational expressions from the standard input and executing them, or by asking for an input file name and processing it.

Part 5: UML diagrams

Document your full program using UML class diagrams. Only include the classes that you programmed. Draw associations between the classes. **Deliver your diagram as a PDF file.**

Things to think about if you are bored (optional)

1. Make parsing in `rational_calc::calculate` more robust. Do not assume valid text input, and if the input is invalid return a `rational` representing NaN and set a `bool` flag in the `rational_calc` object that can be checked with a public function. This flag will inform a caller that the input was invalid. Do not forget to reset the flag every time `calculate` is called.
2. Make use of this new flag in the `rational_processor` class. When a line in a file contains an invalid expression, write `invalid` to the output file.

3. Make `rational_calc::pow` accept negative exponents. Remember to check that results are rational.
4. Let `rational_calc::calculate` accept expressions with brackets and multiple operations:

   ```
   (5/4 + 6/-5) * (-8/3 / 5/6) + (sqrt 4/9)
   ```

   result:

   ```
   38/75
   ```

5. Add a constructor that accepts a `double` to the `rational` class. A `double` can be *NaN* or *inf* — there are functions for checking it. Calling such a constructor can give a result like:

   ```
   auto r = rational(-0.865);
   std::cout << r.to_str(); // prints: -173/200
   ```