CSCI 152, Performance and Data Structures, Lab Exercise 3

Guidelines for Lab Exercises:

- This is a lab exercise. The purpose of lab exercises is to teach you important programming skills and features of C^{++}. In general, lab exercises are not as heavily checked for correctness as assignments, but we still check for good coding style, and that all parts have been completed. You have to take lab exercises very seriously because they teach you important things that may reoccur in assignments, quizzes, or the real life.
- During lab sessions you are allowed to ask help from TAs or instructors.
- You are allowed to discuss lab exercises and possible solutions with classmates and friends, but you have to put in serious effort by yourself when making the exercises. Copying without own effort is not allowed, and may still result in deduction of points, or filing an academic misconduct report.
- We strongly recommend that you use Linux (Ubuntu) for all exercises and assignments. You will have much more control over the processes, the files, and the compiler. Other operating systems do not have good memory checkers, and are too tolerant against programming errors.
- While working on the exercise, save your work frequently, so that you don't accidentily lose your work.

Introduction

Aim of this assignment is to show you:

1. a class that has a non-trivial invariant,
2. the use of member functions in C^{++},
3. how to define operators in C^{++},
4. the use of fields with non-trivial invariants. (It's all automatic.)

For this aim we will implement *exact rational numbers*. In order to allow unbounded precision, we will use an implementation of unbounded precision integers (called `bigint`) instead of machine `int`. We will define the standard operators `+,-,*,/`, so that one can write for example:

```
rational r = rational(1,2);
r = r * 2;
std::cout << r << "\n";    // prints 1.
r = r - rational(1,3);
std::cout << r << "\n";    // prints 2/3
```

The definition of class `rational` is as follows:

```
class rational
{
   bigint num;
   bigint denom;
      // Maintain the following class invariant:
      // num and denom's only common divisors are 1 and -1,
      // and denom > 0
};
```

The implementation of `bigint` is given, and you don't need to worry about how it is implemented. It supports the usual arithmetic operators `+,-,*,/,%` as well as the standard boolean operators `==,!=,<,>,<=,>=`.

In order to make sure that `num` and `denom` have no non-unit common divisors, the constructor computes a greatest common divisor (gcd) of `num` and `denom` and divides both by the gcd found. If `denom < 0`, then both `num` and `denom` can be multiplied by -1.

For two integers n_1 and n_2, a *greatest common divisor* is an integer n, such that both n_1, n_2 are divisible by n, and there exists no n' such that n_1, n_2 are also divisible by n' and $\text{abs}(n') > \text{abs}(n)$. A greated common divisor always exists, except when $n_1 = n_2 = 0$.

If one would not divide out common divisors, rationals would grow arbitrarily, even on simple computations, like $(3/3)^{1000}$, and a user would not recognize simple rationals like 33/231 when they are printed. (It is 1/7)

As usual in C^{++}, the specifications of class `rational` are in file **rational.h** and the implementations are in file **rational.cpp**.

Tasks:

1. First complete the function `bigint gcd( bigint n1, bigint n2 )` in file `rational.cpp`. This function **must use the Euclidean algorithm!**.

   In our experience, many problems originate from not carefully tested gcd functions. Test your function very carefully! The function must return a meaningful result when one or both of the arguments are negative. Your function must also return a meaningful result when one of the arguments is zero. You may assume that `gcd` is never called when $n_1 = n_2 = 0$. If you feel that something needs to be done in this case you can throw an `std::runtime_error( "GCD: both arguments are zero" )`.

   In case one or both of the arguments are negative, `gcd` may return a negative value. This is no problem. Don't worry about it.

2. Complete the method `normalize( )` in file **rational.cpp**. This method must establish the class invariant of `rational`. The constructor `rational( const bigint& num, const bigint& denom )` calls `normalize( )`.

   You may assume that `denom != 0`. Again if you don't like the idea, you can `throw std::runtime_error( "rational: denom == 0" );`

   `normalize( )` must be implemented in such a way that its correctness does not rely on the sign of the number that is returned by `gcd`.

3. Complete method `void print( std::ostream& ) const;` If `denom` is not 1, it must print `'num / denom'`, otherwise it must print only `num`.

4. Complete unary minus in **rational.cpp**

   ```
   rational operator - ( ) const;
   ```

   Implement it in such a way that `normalize` is not called. (This implies that you cannot call the constructor.) In order to make this possible, we defined `operator - ( )` as a member function. If the user writes `-r`, the compiler calls `r.operator - ( )`.

5. Complete the following binary arithmetic operators in file **rational.cpp**:

   ```
   rational operator + ( const rational& r2 ) const;
   rational operator - ( const rational& r2 ) const;
   rational operator * ( const rational& r2 ) const;
   rational operator / ( const rational& r2 ) const;
   ```

   These functions must use the constructor `rational( const bigint& num, const bigint& denom )`, in order to make sure that the result is normalized.

   The operators are defined as a member function: If the user writes `r1 + r2`, the compiler calls `r1. operator + ( r2 )`. This means that the fields of `r1` are just `num,denom`, while the fields of the right argument are `r2.num, r2.denom`.

6. Write the boolean comparison operators in file **rational.cpp**

   ```
   bool operator == ( const rational& r2 ) const;
   bool operator != ( const rational& r2 ) const;
   ```

   You can (and should) use the fact that rationals are normalized. This means that you don't have to do any multiplications, when comparing rationals.

When testing your functions, test that the invariant mentioned at the beginning is always true by printing the result.

Test the correctness of the operators. In order to do this, you can use the `approximation( )` method. You can also test standard arithmetic rules for different values, like for example

$$\begin{array}{lcl} (xy)z & = & x(yz) \\ x+(y-z) & = & -z + y + x \\ x(y+z) & = & yx + xz \end{array}$$