Project 5 (50 points)
Assigned: Monday, February 13, 2017
Due: Tuesday, February 21, 2017, 6:00 PM

CS-2303, System Programming Concepts, C-term 2017

Programming Assignment #5 — Operator Overloading

Abstract

Design and implement a class for rational numbers, and write a test program to exercise that class. Overload the usual arithmetic operators to apply to objects of this class. Also overload the stream insertion and extraction operators (`<<` and `>>`) to read in and print out rational numbers.

Outcomes

After successfully completing this assignment, you should be able to:–

- Design a class of numerical objects and provide arithmetic on them.
- Implement operator overloading.
- Implement **`friend`** functions.
- Understand reference parameters and reference results.
- Understand and deal with **`const`** in classes and parameters.

Before Starting

Read Chapter 8 of *Absolute C++*, which introduces operator overloading, friends, and references. This assignment is adapted from Programming Project 2 of that chapter.

Notice and Warning

There are many, many designs and implementations of rational classes available in print and on the web. *You may not refer to these.* **You must design and develop your class *yourselves.*** You may consult your classmates, the teaching assistants, and the professor. You may speak to others inside or outside the Computer Science department *only to the extent that they have not referred to any previous definition of a rational class* in C++ or *Java*.

Optional Teams

You may, if you choose, work in a two-person team. To register your team for joint submission in *Canvas*, please send an e-mail to cs2303-staff@cs.wpi.edu. Teams should name their projects **`PA5_teamName`**, where **`teamName`** is the team name assigned by the instructor or course staff in *Canvas*. Existing teams from Programming Assignment #4 will automatically continue to this assignment. If you wish to change teams or break up a team, please let us know at the same address.

This Assignment

There are two parts to this assignment — the definition and implementation of the *Rational* class and the creation of a *test program* that tests and exhibits all of the features of that class.

The Rational class

A rational number is a number that can be represented as the quotient of two integers. For example, $\frac{1}{2}$, $\frac{3}{4}$, $\frac{64}{2}$, $\frac{32767}{65536}$, etc., are all rational numbers.[1] You can represent rational numbers as two values of type **`int`**, one for the numerator and one for the denominator. Your new class should be named **`Rational`**.

You need at least four constructors:–

- A constructor **`Rational(const int num, const int denom)`** to set the rational number to any legitimate value.
- A constructor **`Rational(const int wholeNumber)`** to set the numerator to the value of the argument and the denominator to *1*.
- A copy constructor **`Rational(const Rational &a)`**.
- A default constructor that sets the value of the **`Rational`** to zero (i.e., 0/1).

Overload the input and output operators **`<<`** and **`>>`** so that numbers can be input and/or output in the form **`1/2`**, **`15/32`**, **`300/401`**, etc. Note that the numerator and/or denominator may contain a minus sign, so input values of the form **`-1/2`**, **`15/-32`**, and **`-300/-401`** are legal.

Overload the following operators so that they correctly apply to the type **`Rational`**:–

`==`, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**, **`+`**, **`-`**, **`*`**, and **`/`**

In addition, define and implement a conversion function **`toDouble(const &Rational)`** that converts a **`Rational`** number to a **`double`**.[2]

Your class and all of its constructors and operators must *normalize* the rational number — that is, reduce it to its lowest terms. For this, you should use *Euclid's algorithm*, described here:–

http://en.wikipedia.org/wiki/Euclid_algorithm

This should be applied to any numerator-denominator pair that is not already known to be normalized. If the result is negative, the minus sign *should be in the numerator*. For example, 4/–8 would be normalized to –1/2.

Hints: Two rational numbers $\frac{a}{b}$ and $\frac{c}{d}$ are equal if $a*d$ equals $b*c$. If b and d are positive, a/b is less than c/d provided $a*d$ is less than $c*b$.

Simplifications: An input of a zero for a denominator is illegal. In normal circumstances, your class should "throw an error." If you are comfortable throwing errors, you may do so. However, since we have not yet covered error handling in this course, a simpler approach is to set both the numerator and denominator to zero and to print an error whenever such a number is used as an operand to any operator.

You may also ignore overflows. That is, multiplication and division may produce results so large that even when normalized, the numerator or denominator may not fit into variables of type **`int`**. However, your operators should hold intermediate values in variables of type **`long long int`** so that you do not gratuitously lose information before normalizing.

[1] By the quotients in this sentence, we mean everyday fractions, not the integer division result in *C* or *C++* that this expression would produce.

[2] It would be desirable to overload **`operator=`** to assign to **`double`**, but that is not possible under the "Rule on Overloading Operations" at the end of §8.2 of *Absolute C++*.

The test program

Your test program should be called **`PA5`**. It must accept an indeterminate number of arguments on the command line, each of which specifies an input file containing a sequence of lines. Each line represents a rational expression to be evaluated as described below. Your program should be invoked from a command line by typing

```
./PA5 inputFile1 inputFile2 ...
```

Under this command, the program would open and read each input file in turn. The file may contain multiple lines. Each line would be an expression of rational numbers and operations in *postfix* form. That is, both operands precede their operator. You must scan the line, convert numeric input into rational numbers, apply each operation to the two operands preceding it, and output the (normalized) result in both rational and double format. For example, the input line

```
1/3 1/6 +
```

Means $(\frac{1}{3} + \frac{1}{6})$ and would result in the output line

```
1/3 1/6 +    : 1/2 (double 0.5)
```

That is, the output line should repeat the input line, followed by a tab and a colon, and followed by the answer in rational and double format.

Likewise, an input line of the form

```
1/4 1/8 + 2 *
```

should output

```
1/4 1/8 + 2 *     : 3/4 (double 0.75)
```

Boolean operators should output **`true`** or **`false`** (with no double equivalent) — e.g.,

```
3/4 6/8 ==   : true
```

If you encounter an error in an input line, or if the line is unintelligible, print an error message and proceed to the next input line. When you open a new input file, print a line identifying the file name before scanning and interpreting any input.

Design sufficient test expressions to exercise the entire class and all of its operators. *You may share and exchange test input files with your classmates.* Report the results of your testing in your **`README`** file.

Deliverables

You should carry out this project in *Eclipse CDT*. When you are ready to submit, clean the project and then export it to a *zip* file. The zip file should contain the following:–

- All of the *C++* files of your project, including your **`.cpp`** and **`.h`** files of the **`Rational`** class and one or more **`.cpp`** and **`.h`** files for your test program.
- A **`makefile`**. The target name should be **`PA5`**.
- One or more test files containing lines of the form described above. Each test file name should end in **`.txt`** and should begin with your user ID (or, if a team, the team name as assigned in *Canvas*). If you have more than one test file, suffix the names with sequence numbers or letters or some other identifying information

You must also submit a document called **README.txt**, **README.pdf**, or **README.doc** summarizing your program, how to run it, and detailing any problems that you had. If you borrowed any part of the algorithm or any test case for this assignment, be sure to cite your sources. Your **README** file does *not* need to be part of the *zip* file.

Before submitting your assignment, execute *make clean* to get rid of extraneous files. Submit it to *Canvas* under the assignment *Programming Assignment #5* (PA5). Programs submitted after the due date and time specified in *Canvas* will be tagged as late and will be subject to the revised late assignment policy for this course.

Grading

This assignment is worth fifty (50) points. Your program must be accompanied by a **makefile**, and the **makefile** must *compile your program without errors in order to receive* any *credit.*

- The Rational class itself – 25 points, allocated as follows:–
    - Correct definition of class and methods, and overloaded operators – 4 points
    - Correct implementation of four different constructors, of one destructor, and an internal normalization method – 6 points
    - Correct implementation of 13 operators (i.e., **<<**, **>>**, **!=**, **==**, **<**, **<=**, **>**, **>=**, **+**, **-**, *****, **/**, and conversion to **double**) – 13 points
    - Compile without warnings using **-Wall** – 2 points
- The Test program – 10 points
    - Correct processing of command lines, opening and closing of input files – 3 points
    - Correct scanning and parsing of input lines – 3 points
    - Correct evaluation of parsed input lines and output of results – 3 points
    - Compile without warnings using **-Wall** – 1 points
- Test cases – 10 points
    - Enough lines in test files to exercise each comparison operation at least 3 times with a range of input values – 3 points
    - Enough lines in test files to exercise each arithmetic operator (**+**, **-**, *****, **/**) at least three times with non-trivial denominators, including denominators that are relatively prime and those that are not relatively prime – 4 points
    - Enough test cases to show that normalization works in all situations – 3 points
- Satisfactory **README** file explaining your class and your testing and showing the output of all test cases – 5 points
- Penalty of ten points if your project is not clean before creating the zip file or for submitting your programs in something other than a zip file.

Note: This program must be developed in *Eclipse*. The graders will grade it on machines equivalent to the course virtual machine.

Extra Credit

For ten points of extra credit, define and implement the assignment operators **=**, **+=**, **-=**, ***=**, and **/=**. Also, construct test cases that show that they work. Then extend the input lines of the test cases to include simple variables that can be assigned to on one line and used on later lines of the same test case. (Variables do not carry across from one input test file to another.)

Make sure that the test cases for extra credit are clearly identified and separate from the regular test cases of this assignment.

For five additional points of extra credit, implement a **`friend`** function to overload the assignment operator to assign a rational number to a **`double`**. Extend the test program to demonstrate that this works. In your **`README`** document, explain how your tested it and how the graders can replicate you tests.