COMP220 — Project 1

Fall 2016

Abstract

For your first project you'll be writing a modified version of the Reverse Polish Notation (RPN) calculator discussed in Chapter 5 section 2 of the text. It combines elements of exercises 8, 9, and 11 from chapter 6. The end result of your efforts is a fairly robust Rational number class, a library for tokenizing and evaluating RPN expressions, and a basic command-line interface based RPN calculator program.

CLI-RPNRC

The program you'll be writing is a command-line interface (CLI) based reverse polish notation (RPN) calculator for rational numbers. The user will pass a RPN expression involving rational operands and the operators $+$, $*$, $/$, $-$, $<$, $>$, $<=$, $>=$, $==$, and $!=$. When comparisons are combined with arithmetic, then boolean *true* should be interpreted as 1/1 and *false* as 0/1. All comparison operators have the same precedence and should be evaluated after the arithmetic operators.

Rational Operands

Rational numbers are numerical values which can be written as fractions with integer valued numerators and denominators. You'll be extending the Rational class defined and discussed in chapter 6 section 3 of the text by completing exercise 8 from chapter 6. In addition to the operators listed in exercise 8, you should implement *operator¡¡* and *operator¿¿* to enable easy streaming I/O with a Rational datum. The streaming I/O operators also get used by gTests for better error reporting.

CLI Tokenizing

Your program will read the entire expression as a command-line arguments, *tokenize* the expression, then carry out the calculation specified by the expression or indicate there is something wrong with the expression. Tokenizing is the act of splitting the sequence of raw characters into a sequence of logical tokens. In this case tokens will be strings that represent operators or rational numbers. The text discusses a class-based solution to tokenizing. We'll focus on a simple procedural library, but the underlying logic is still the same so the class-based solution merits some study. This library should not only tokenize the expression, but provide a means to recognize properly formated tokens. For example, using this library we should be able to recognize if a string can be read as a rational or not. This allows our calculator to avoid errors that would occur when something like *hello world +* is encountered.

Rationals can be expressed as any two integers separated by a /. No spaces should be found on either side of the / character. This means 123/456 is a valid rational expression but 123 /456, 123 / 456, and 123/ 456 are not. Users are allow to express negative numerators and denominators and unsimplified rationals, but we will always store them in simplest form and associate negative values with the numerator only.

Operands and operators in full expressions must be separated by whitespace but the type and amount of whitespace doesn't matter. This means tokenizing a complete expression means breaking up the string by whitespace and discarding all the whitespace.

Error Handling

Once an expression has been tokenized, the process of evaluating the expression occurs. Several things can go wrong during this process:

- Improperly formatted operands and zero denominator operands
- Unknown operators
- Too many or too few operands
- Divide by zero

We'll catch all these problems dynamically, while we're evaluating the expression, rather than statically, prior to evaluation as we're tokenizing or parsing the expression. The evaluation procedure should detect the errors as they occur and throw exceptions specific to the errors. The program's main procedure, or some other caller to the evaluation procedure, can then catch the error, report the problem to the user, and end the program gracefully.

Logistics

Event	Date
Project Lab Assignment	10/20
Free Lab	10/27
Project Due	11/2

Grades

Grades are based on the completeness of the program and the design. Complete programs have a fully functioning calculator program that includes robust error handling. Complete designs have a complete set of documentation and tests. Programs that fail to compile on either the Debug/Release or Test build will not receive not better than a D.

Lab Assignment

Your first project lab assignment is straightforward, you must get the Rational Class started in the book, write gtests for all of the existing methods, then begin work on the extensions required for this project. You may want to implement *operator==* sooner rather than later in order to more easily test the arithmetic operators. At the end of lab, submit your source code as assignment *labp1*.