CPSC 221 Programming Assignment #2 Specification

Due: 23:00 Friday, November 03, 2017

October 24, 2017

Introduction

For the primary purpose of causing you, the students of CPSC 221 much grief and suffering, Aylo invented a postfix-like mathematical language called ?Lang, and it is your task to implement both the interpreter and the container abstract data types to support it!

Secondarily, it will be a practical exercise in working with C++ templates, exceptions, and dynamic array structures. It is recommended that you work in pairs for this assignment.

Your code must be made with C++ using Clang++ as a compiler. Class header files and at least one skeleton implementation have been provided.

Terminal input and output will be done using standard IO, so `cin` and `cout` will be helpful. An automated grader will be used, so make sure your output and/or return values are formatted correctly.

Details about the output format are given in the Interpreter definition.

Specification

ArrayClass

This is to become your own implementation of a templated dynamic array container which mimics much of the functionality of the `std::vector` class. Why? It builds character and it will be used as the data storage of two other classes in this assignment.

As templated classes require a different method of separating definition from implementation, a stub implementation has been provided for you. Pay special attention to the format; you may follow a similar format for the other templated classes.

This class contains a major portion of the work for this assignment and is crucial to the correct operation of all of the other classes. Thus it is highly recommended to start working early on this and to test it very thoroughly before proceeding further in this assignment.

Stack

This template class implements the standard Stack ADT functions, using an ArrayClass object as its underlying data structure.

Queue

This template class implements the standard Queue ADT functions, also using an ArrayClass object as its underlying data structure.

While the ArrayClass allows random access to only those indices which have been "added" to the object, you must implement your Queue functions such that they will (usually) run in constant time.

Interpreter

The Interpreter processes a postfix-like language, where instructions are space-delimited strings consisting of operands and operators. Operands may be positive or negative integers, and operators may be binary mathematical operators, or special operators. Binary operators are processed in a standard postfix way using a stack for operand storage. However, a key difference is that the result of the operation is to be stored into the queue instead of returned to the stack.

Special operators perform some more specific interactions with the stack and queue. Please refer to `interpreter.h` for details about each operator's behaviour.

Instructions and output

A single instruction takes the form of a space-delimited string and can contain any number of operands and operators. For example:

```
12 3 2 ^ -6 RQ /
```

Stored Data

An *operand* is always an integer.
An *operator* is either a *binary operator* or a *special operator*. A list of each can be found below.
An *element* can be either an *operand* or an *operator*.

The Machine Structure

These instructions are interpreted on a machine with two data structures:

A queue (named the *tube*) containing *element*s.

A stack (named the *bin*) containing only *operand*s.

Execution

Program execution can be placed into three primary behaviours:

Next: The program will pop a single element from the *tube*, and execute the relevant operation.
If this element is an operand, the relevant operation is to then push the operand to the top of the *bin*.
If this element is instead an operator, then the relevant operation is defined below. Note that since the *bin* contains only *operands*, an *operator* will never be pushed to the *bin*.
Finally, if the element's operation cannot be executed (or if there is no element at all), an exception should be thrown.

Run: When this behaviour begins, the program begins a loop:
Until there are no more elements in the *tube*, the program will run the *Next* behaviour. However, if the *pause* operator (see below) is executed, then the program will halt execution, without altering the program state.

Run All: This behaviour is identical to that of *Run*, except *pause* operator executions are ignored.

List of Binary Operators

A *binary operator* removes and evaluates on exactly two elements from the *bin*, and stores its result in the *tube*. For order-specific operators, the second element removed (the one that was found deeper in the stack) is treated as the left operator, and the first element is treated as the right. The following is a list of binary operators valid in the language:

+: Adds two numbers together.

−: Subtracts one number from another.

∗: Multiplies two numbers together.

/: Divides one number by another.

ˆ: Exponentiates one number to the power of another. Note that this operator should run in the complexity of the logarithm of its exponent.

Evaluation of these *binary operator*s returns the empty string.

List of Special Operators

A *special operator* can perform different operations on the *bin* and *tube*. The following is a list of special operators valid in the language:

RQ: This is the *requeue* operator. When this operator is popped from the *tube*, the following process will be performed:

(1) Each *operand* will be popped from the *bin* one by one. A copy of the *operand* will be made, and the copy will be enqueued into the *tube*. Meanwhile, the original item will be placed into a *spare bin* (another stack).

(2) Each *operand* in the *spare bin* will be removed and placed into the *bin*. After the completion of this operation, the *spare bin* will be empty, and the *bin* will be in the state is started in prior to the execution of the *requeue* operation.

While the operands are being removed from the *bin*, they should also be appended into a string with a space in between each operand. There will be no space appended after the last operand. This string will be printed as the result of evaluating the *RQ* operator.

PRINT: Upon execution, the entire contents of the *tube* should be dequeued and appended into a string, in order, separated by spaces. Note that there should be no additional space at the end of the string. This string is returned as a result of evaluating the *PRINT* operator.

#: This is the *pause* operation. Upon execution, the program will pause its execution until otherwise specified.

Sample Programs

The text in the parentheses denotes the state of the *tube*, the text in the brackets denotes the state of the *bin* (where the left side is the bottom, and the right side is the top), and the text in the braces denotes the printed output of the program so far.

$(1, 2, +, PRINT, \#)$:
$[1], (2, +, PRINT, \#)$
$[1, 2], (+, PRINT, \#)$
$(PRINT, \#, 3)$
$\{\#\ 3\}$

$(2, 5, \hat{}, 1, RQ, PRINT, \#)$
$[2], (5, \hat{}, 1, RQ, PRINT, \#)$
$[2, 5](\hat{}, 1, RQ, PRINT, \#)$
$(1, RQ, PRINT, \#, 32)$
$[1](RQ, PRINT, \#, 32)$
$[1](PRINT, \#, 32, 1), \{1\}$
$[1], \{\#\ 32\ 1\}$

$(12, 3, 2, \hat{}, -6, RQ, /, \#, 4, 2, RQ, \#, *, PRINT, \#)$
$[12], (3, 2, \hat{}, -6, RQ, /, \#, 4, 2, RQ, \#, *, PRINT, \#)$
$[12, 3], (2, \hat{}, -6, RQ, /, \#, 4, 2, RQ, \#, *, PRINT, \#)$
$[12, 3, 2], (\hat{}, -6, RQ, /, \#, 4, 2, RQ, \#, *, PRINT, \#)$
$[12], (-6, RQ, /, \#, 4, 2, RQ, \#, *, PRINT, \#, 9)$
$[12, -6], (RQ, /, \#, 4, 2, RQ, \#, *, PRINT, \#, 9)$
$[12, -6], (/, \#, 4, 2, RQ, \#, *, PRINT, \#, 9, -6, 12), \{-6\ \ 12\}$
$(\#, 4, 2, RQ, \#, *, PRINT, \#, 9, -6, 12, -2)$
$(4, 2, RQ, \#, *, PRINT, \#, 9, -6, 12, -2)$
$[4], (2, RQ, \#, *, PRINT, \#, 9, -6, 12, -2)$
$[4, 2], (RQ, \#, *, PRINT, \#, 9, -6, 12, -2)$
$[4, 2], (\#, *, PRINT, \#, 9, -6, 12, -2, 2, 4), \{2\ \ 4\}$
$[4, 2], (*, PRINT, \#, 9, -6, 12, -2, 2, 4)$
$(PRINT, \#, 9, -6, 12, -2, 2, 4, 8)$
$\{\#\ \ 9\ \ -6\ \ 12\ \ -2\ \ 2\ \ 4\ \ 8\}$

Hints

Class declarations have been provided for you. You can use the given files to complete the project. Please read the comments for each function carefully for information about required inputs, outputs, and behaviours.

Deliverables and submission

This assignment is to be submitted via `handin` (https://my.cs.ubc.ca/docs/hand-in). Each member of your team should submit a ZIP archive containing the following files:

- partners.txt
- arrayclassprivate.h
- arrayclass.cpp
- stackprivate.h
- stack.cpp
- queueprivate.h
- queue.cpp
- interpreterprivate.h
- interpreter.cpp

Your ZIP file should be titled "pa2.zip".
For `handin`, our course is **cs221** and the assignment is **pa2**.

The `partners.txt` file should contain **your** CSID on the first line, and your partner's CSID on the second line. Each line should contain only a CSID and nothing else; this file should contain at most two lines.

Late submissions will be accepted up to 48 hours past the due date, at a 20% penalty per 24-hour period. No submissions will be accepted beyond 48 hours past due.

Appendix: A Tangible Understanding of the Language

This section is not necessary to complete the programming assignment, but may aid you in visualizing the system you are trying to implement.

The language can be thought of as the program executed by a robot for an assembly line for rabbits. The robot has a long (literal) tube that brings it either instructions on paper. or cages full of rabbits.

The *operands* are effectively cages of rabbits. while the *operators* are effectively intructions for the robot.

As an example, say the robot receives a cage with four rabbits, a cage with two rabbits, and then an instruction that says "multiplication" Then, the robot will place the cage with four rabbits into a bin, then the robot will place the cage with two rabbits into the bin, then finally the robot will follow the multiplication instruction and remove both original cages from the bin, and place a new cage with eight rabbits into the other end of the tube.

Similarly, the print operator can be seen as the robot emailing a picture (before knocking everything out of the tube in a glitchy robotic frenzy), and the pause operator can be viewed as a really long book of useless instructions that the robot must spend an indefinite amount of time trying to read.