3.26 Programming Assignment 2: Stack, Infix-to-Postfix, RPN Evaluation (Summer B 21)

Due Tuesday August 10, 2021 at 11:59 PM

Collaboration Policy Reminder

You may not use code from any source (another student, a book, online, etc.) within your solution to this PROGRAM. In fact, you may not even look at another student's solution or partial solution to this PROGRAM to use as a guide or otherwise. You also may not allow another student to look at any part of your solution to this assignment. You should get help on this assignment by coming to the instructors' office hours or by posting questions on Slack. Do not post your code on Slack.

Summary

Students will implement a Stack class template with exceptions.Then use it to implement two stack-based algorithms: *Postfix-to-Infix Conversion* and *RPN (Reverse Polish Notation) Evaluation*. In this assignment you will gain practical experience writing a generic stack class that can be of a different type depending on the purpose. The *Postfix-To-Infix* algorithm will require a stack of type `char`. The RPN evaluation algorithm should be a stack of type `double`. There will be floating point numbers in some of the test cases we use to grade.

Stack Class Template

The user-defined Stack class template will support `push(item), item& top(), pop(), int size(),` and `bool isEmpty()`. Included error-handling will make use of the `<stdexcept>` runtime error classes to `throw` either an *underflow exception* (when a `pop()`, or `item& top()` is attempted on an empty stack object) or an *overflow exception* (when a `push()` is attempted on full stack).

single file name Stack.h. To compile a test harness main.cpp with a Stack.h file use the following command: $ g++ -W -Wall -Werror -g --std=c++ *.cpp

Below is a *starter* Stack.h file. Most of the functionality still needs to be implemented. However, the ``top() function has been implemented to give you an example of how to throw an underflow error exception with correct syntax.

Stack.h

```
/*
 * CS010C Summer 2021
 * Programming Assignment 2
 * Stack.h
 * template, exceptions
 */

#ifndef STACK_H_
#define STACK_H_

#define CAPACITY 10
#include <stdexcept>

using namespace std;

typedef int T;

//template <class T>
class Stack {
public:
  int t; // type must be int, index into array
  T S[CAPACITY];
  Stack() { t = -1; }

  void push(T value) {
   //
  }

  void pop() {
    //;
  }

  unsigned int size() const {
   //
  }

};

#endif /* STACK_H_ */
```

Stack Class

First, implement a Stack class of type int and thoroughly test for correct functionality. Once you are confident the implementation is correct, convert the int Stack to a Stack class template. See class interface in next section.

- `void push(const int& x)` - Adds a new element x at the top of the stack.
- `int& top()` - Returns a reference to the next (top) element in the stack, if the stack is not already empty.
- `void pop()` - Removes item on top of the stack, reducing the stack's size by 1 if the stack is not already empty.
- `bool is_empty() const` - Returns true if stack is empty, false otherwise. Tests whether the stack size is zero. Primarily used as a helper function to ensure that a `pop()` or `item& top()` operation is not performed on an empty instance of a Stack.

Stack Class Template

The member functions of the stack class template will be nearly identical to the single type stack class above. The are semantically identical to the Stack class of type `int`. However, as a template class, it generalizes the class to be able to operate and store stack objects of various types. Examples: `int, double, string`.

harness should exhaustively and automatically test both data types in a single execution of a main function.

- Test both typical operational input/output behavior and edge cases.
- Input files for testing can be generated from **Random.org**:
    - Random.org integer sequences
    - Random.org string sequences

RPN Evaluation

Postfix notation represents binary arithmetic expressions as: `lhs\_operand rhs\_operand operator`, e.g. **64 31 +**. A left hand operand (number or variable) is written first, followed by the the right hand operand, followed by an arithmetic operator (e.g. addition (**+**), subtraction (**-**), multiplication (*****), or division (**/**). An example postfix expression is: 3.54 5.5 2.5 3.5 + 8 * + 5 + *.

RPN

```
Algorithm RPN(postfix_expression P)
  // input: postfix expression
  // output: result of evaluation of postfix expression
  create an empty stack S to holds chars
  size = P.size( )
  i <- 0
  for each p[i] in P // where i = {0, 1, ..., P.size( )-1}
    if p[i] is a number
        S.push( p[i] )
    else // p[i] is an operator
```

format, similar content, and scope.

postfix1.txt

```
12 5 6 + * 8 5 3 - * - 4 2 - 2 * / 8 12 + -
```

postfix2.txt

```
12 5 6 + * 8 5 3 - * - 4 2 - 2 * / 8 3 - - 2 / 10 - 2 * 4 * 8 *
```

postfix7.txt

```
3.54  5.5  2.5  3.5 +  8 *  + 5 + *
```

infix1.txt

```
(((12 * (5 + 6)) - (8 * (5 - 3))) / ((4 - 2) * 2)) - (8 + 12)
```

infix2.txt

```
(((12 * (5 + 6) - 8 * (5 - 3)) / ((4 - 2) * 2) - (8 - 3)) / 2 -
  for each p[i] in P // where i = {0, 1, ..., P.size( )-1}
    if p[i] is a operand
      output p[i]
    if p[i] is an operator (*, /, +, -)
      output S.top() and S.pop() from S until left paren at
S->top OR
      S.top() holds operator with lower precedence than p[i]
      S.push( p[i]  )
    if p[i] == '('
      S.push( p[i] )
    if p[i] == ')'
      // right paren causes stack to empty up to and including
left paren
      S.top() and S.pop() to output from S until '(' found
      S.top() and S.pop() '(' to output
   while ( !S.empty )
```

```
 * main.cpp
 * try-catch, exceptions
 */

#include <iostream>
#include <vector>
#include "Stack.h"
#include <stdexcept>

using namespace std;
```

```
} catch(underflow_error& e) {
  cerr << "Underflow Exception: "<< e.what() << endl;
} catch(...) {
  cerr << "Default Exception.\n";
}
```