[CS302-Data Structures] Homework 2: Stacks

Instructor: Kostas Alexis
Teaching Assistants: Shehryar Khattak, Mustafa Solmaz, Bishal Sainju
Fall 2018 Semester

Section 1. Stack ADT – Overview wrt Provided Code

Data Items

The data items in a stack are of generic DataType.

Structure

The stack data items are linearly ordered from the most recently added (the top) to the least recently added (the bottom). This is a LIFO scheme. Data items are inserted onto (pushed) and removed from (popped) the top of the stack.

Operations

```
Stack (int maxNumber = MAX_STACK_SIZE)
```

Requirements

None

Results

Constructor. Creates an empty stack. Allocates enough memory for a stack containing maxNumber data items (if necessary).

```
Stack (const Stack& other)
```

Requirements

None

Results

Copy constructor. Initializes the stack to be equivalent to the other Stack object parameter.

```
Stack& operator= (const Stack& other)
```

Requirements

None

Results

Overloaded assignment operator. Sets the stack to be equivalent to the other Stack object parameter and returns a reference to the modified stack.

```
~Stack()
```

Requirements

None

Results

Destructor. Deallocates the memory used to store the stack.

```
void push (const DataType& newDataItem) throw ( logic_error )
```

Requirements

Stack is not full.

Results

Inserts newDataItem onto the top of the stack.

```
DataType pop() throw ( logic_error )
```

Requirements

Stack is not empty

Results

Removes the most recently added (top) data item from the stack and returns the value of the deleted item.

```
void Clear()
```

Requirements

None

Results

Removes all the data items in the stack.

```
bool isEmpty() const
```

Requirements

None

Results

Returns true if the stack is empty. Otherwise, returns false.

```
bool isFull () const
```

Requirements

None

Results

Returns true if the stack is full. Otherwise, returns false.

```
void showStructure() const
```

Requirements

None

Results

Outputs the data items in a stack. If the stack is empty, it outputs "Empty stack". Note that this operation is intended for testing/debugging purposes only. It only supports stack data items that are one of c++ predefined data types (int, char, etc) or other data structures with overridden ostream operator<<.

Section 2. Implementation Approaches

As in the class, we consider both

- Array-based implementations
- Linked-List implementations

For studying we refer you to the course slides.

Section 3. Compilation Directions wrt Provided Code

Edit config.h and change the value of LAB6_TEST1 to 1 to test the link-based implementation. If set to 0, then the array-based implementation is tested. Recompile test6.cpp.

Section 4. Testing

Test your implementation of the linked list Stack ADT using the program in the file test6.cpp.

Section 5. Programming Exercise 1

We commonly write arithmetic expressions in the so-called infix form. That is, with each operator placed between its operand, as below

(3+4)*(5/2)

Although we are comfortable writing expressions in this form, infix form has the disadvantage that parentheses must be used to indicate the order in which operators are to be evaluated. These parentheses, in turn, complicate the evaluation process.

Evaluation is much easier if we can simply evaluate operators from left to right. Unfortunately, this evaluation will not work with the infix form of arithmetic expressions. However, it will work if the expression is in postfix form. In the postfix form of an arithmetic expression, each operator is placed immediately after its operands. The expression above is written in postfix form as

34+52/*

Note that both forms place the numbers in the same order (reading from left to right). The order of the operators is different, however, because the operators in the postfix form are positioned in the order that they are evaluated. The resulting postfix expression is hard to read at first, but it is easy to evaluate programmatically. We will do so with stacks.

Suppose you have an arithmetic expression in postfix form that consists of a sequence of single digit, nonnegative integers and the four basic arithmetic operators (+,-,*,/). This expression can be evaluated using the following algorithm in conjuction with a stack of floating-point numbers.

Read the expression character-by-character. As each character is read in:

- If the character corresponds to a single digit number (characters '0' to '9'), then push the corresponding floating-point number onto the stack.
- If the character corresponds to one of the arithmetic operators (characters '+', '-', '*', '/'), then
    - Pop a number off of the stack. Call it operand1.
    - Pop a number off of the stack. Call it operand 2.
    - Combine these operands using the arithmetic operartor, as follows
        - Result = operand2 operator operand1
    - Puhs result onto the stack.

When the end of the expression is reached, pop the remaining number off the stack. This number is the value of the expression. Applying this algorithm to the arithmetic expression

34+52/*

Results 17.5 as expected.

Exercise 1. Assignment 1. Create a program that reads the postfix form of an arithmetic expression, evaluates it, and outputs the result. Assume that the expression consists of single-digit, nonnegative integers ('0' to '9') and the four basic arithmetic operators ('+','-','*','/'). Further assume that the arithmetic expression is input from the keyboard with all the characters separated by white space on one line. Save your program in a file called postfix.cpp

Exercise 1. Assignment 2. Create a test plan involving the execution of 5 expressions for which you must provide the infix and postfix notations, alongside their result in your report document.

Section 6. Programming Exercise 2

One of the tasks that compilers and interpreters must frequently perform is deciding whether some pair of expression delimeters are properly paired, even if they are embedded multiple pairs deep. Consider the following C++ expression.

a=(f(b)-(c+d))/2;

The compiler has to be able to determine which pairs of opening and closing parentheses go together and whether the whole expression is correctly parenthesized. A number of possible errors can occur because of incomplete pairs of parentheses – more of one than the other – or because of improperly placed parentheses. For instance, the expression below lacks a closing parenthesis.

a=(f(b)-(c+2)/2;

A stack is extremely helpful in implementing solutions to this type of problem because of its LIFO behavior. A closing parenthesis needs to be matched with the most recently encountered opening parenthesis. This is handled by pushing opening parentheses onto a stack as they are encountered. When a closing parenthesis is encountered, it should be possible to pop the matching opening parenthesis off the stack. If it is determined that every closing parenthesis had a matching opening parenthesis, then the expression is valid.

```
bool delimetersOk ( const string& expression )
```

Requirements

None

Results

Returns true if all the parentheses and braces in the string are legally paired. Otherwise, returns false.

Exercise 2. Assignment 1. Save a copy of the delimiters.cs as delimeters.cpp. Implement the delimetersOk operation inside the delimeters.cpp program.

Exercise 2. Assignment 2. Add test 5 cases that check whether your implementation of the delimtersOk operation correctly detects improperly paired delimeters in input expressions.