EECE.3220: Data Structures

Spring 2019

Programming Assignment #3: Stacks and strings

Due **Monday, 4/8/19**, 11:59:59 PM

1. Introduction

This assignment provides an introduction to working with stack objects. You will write a linked stack class, as well as a basic main program to interact with the stack.

2. Deliverables

You should submit three files for this assignment:

- ***prog3_main.cpp:*** Source file containing your main function.
- ***Stack.h:*** Header file containing the `Stack` class definition.
- ***Stack.cpp:*** Source file containing definitions of `Stack` member functions.

Submit a single archive file (.zip only) containing all .h and .cpp files to the "Program 3" assignment on Blackboard. *If you complete either of the extra credit sections, you may have more than 3 files to submit. Please note in the "Comments" section of your Blackboard submission that you have submitted extra credit if applicable.*

3. Specifications

Class Design: Your `Stack` class should be implemented as a linked stack holding values of type `string`. The class should contain, at a minimum, the following functions:

- `Stack()`: Default constructor
- `~Stack()`: Destructor
- `bool empty()`: Returns true if stack is empty, false otherwise
- `string top()`: Return the top value on the stack
- `void push(const string &val)`: Push val on top of the stack
- `void pop()`: Remove the top element from the stack
- `void display(ostream &out)`: Function that prints the contents of the calling object to the output stream `out`, starting at the top of the stack and printing one element per line.

3. Specifications (continued)

Command Line Input and Corresponding Output: Your main program should accept the commands below:

- **push:** Prompt the user to enter a word that should then be pushed on the top of the stack. Immediately print the state of the stack after the push operation is complete.
- **pop:** Remove the topmost word from the stack. Immediately print the state of the stack after the pop operation is complete.
- **match:** Prompt the user to enter a word and test if that word matches the topmost item on the stack. Print an appropriate message in either case; the message should contain both the user input and the word at the top of the stack.
- **exit:** End the program.

Error checking: Your program should print error messages in the following cases:

- The user enters a command other than the four valid ones listed
- The user attempts to pop a value from an empty stack

EXTRA CREDIT: You may earn up to **7 points** of extra credit for each of the following upgrades (and may implement both for up to 15 points (yes, I know 7 + 7 = 14)). Please specify which upgrade(s) you completed (if any) in the "Comments" for your submission:

- Write an overloaded output operator << for your Stack class and demonstrate its use in your main function.
  - This operator prints an object directly—given `Stack S1`, `cout << S1;` would behave like `S1.display(cout)` if the << operator is correct.
- Write your own code to make all string input case-insensitive. In particular:
  - All commands should be case-insensitive (in other words, `push`, `PUSH`, and `pUsH` should all perform the same operation)
  - All words entered should be stored with lowercase letters only (`Geiger` would be pushed on the stack as `geiger`)
  - The match command should indicate two words match even if the cases of all letters do not match

  Note: Any comparisons formed in your solution should not simply pass the strings into a built-in function, such as (but not limited to) `stricmp()`, `strcasecmp()`, or `boost::iequals()`. You *may* use built-in functions to build a case-insensitive comparison function, but you won't get extra credit for simply calling one function that does the work for you.

I strongly suggest you get the base program working first, then copy your working files and attempt the extra credit in a new set of files. It's very easy to "break" a working program and find yourself unable to reverse the changes and fix the program again!

4. Hints

We discussed the following basics of a linked stack implementation in class Wednesday, 3/27. I strongly suggest you watch the lecture video if you could not attend class, as I did not present any slides that day:

- A linked `Stack` implementation strictly requires just one data member—a pointer to the top node in the stack that should be `NULL` when the stack is empty.
- In addition to defining a `Stack` class, you will have to define a node data type. I recommend creating the `Node` type within your `Stack` definition, ensuring this `Node` type can only be used inside `Stack` functions:

```
class Stack {
public:
   // List of public member functions
private:
   class Node {
   public:
      string word;   // Word in each node
      Node *next;    // Pointer to next node
   };

   Node *top;        // Pointer to top of stack
};
```

  Each `Node` contains a pointer to the next `Node` in the stack. The last `Node` (the "bottom" of the stack) should have its `next` pointer set to `NULL`.

- We also discussed basic algorithms for some of the member functions:
  - `~Stack()`: In a linked data structure, the destructor deletes <u>all</u> of the dynamically allocated nodes. The general algorithm, in pseudocode:

```
Start at top
while (more nodes exist) {
   Find next node
   Delete current node
}
```

  - `push()`: The `push()` function adds a new node at the top of the stack:

```
Dynamically allocate new node & store data in it
Make new node point to old top node
Make top pointer reference new node
```

  - `pop()`: The `pop()` function removes the top node from the stack:

```
Store address of current top node
Make top pointer reference 2nd node (since 1st node
   will be deleted)
Delete old top node (using ptr from first step)
```

5. Test Cases

Your output should closely match these test cases exactly for the given input values, at least in terms of format and general functionality. I will use these test cases in grading of your assignments, but will also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.

In the test cases below, user input is underlined. *The program may behave differently if you add the case-insensitive upgrade.*

```
Enter command: pop
ERROR: Stack is empty

Enter command: push
Enter word: program
Stack: program

Enter command: push
Enter word: four
Stack: four
       program

Enter command: push
Enter word: EECE
Stack: EECE
       four
       program

Enter command: match
Enter word: EECE
User input EECE matches top of stack

Enter command: match
Enter word: four
User input four doesn't match top of stack (EECE)

Enter command: pop
Stack: four
       program

Enter command: pop
Stack: program

Enter command: pop
Stack is empty

Enter command: quit
ERROR: Invalid command quit

Enter command: exit
```