Algorithms and Data Structures I - Fall 2018 - Programming Assignment 3

Directions: This assignment will be demoed in lab on Thursday, November 8th or Friday, November 9th.

Lab 3: Stacks Assignment Details: As an object-oriented language, C++ allows you to define your own types. This capability can be used to wrap data and functions that operate on this data into a self-contained unit which is called a class. C++ includes extensive support for classes. For an introduction, see: http://www.cplusplus.com/doc/tutorial/classes/

An example C++ `Elevator` class implementation is listed below. Code that uses this `Elevator` class has no knowledge of the `private` data members and, in fact, direct access to private data is not permitted (try uncommenting the annotated lines in main()). Instead, the class provides public methods that allow users to perform all necessary functions. In this way, the `Elevator` class *encapsulates* data, providing a simple interface to complex state detail, a key benefit of object-oriented design.

```
#include <iostream>

class Elevator {
private:    // private data members
  int currentFloor;
  bool doorsOpen;
  bool movingUp;
  bool movingDown;

public:     // public methods
  Elevator() { currentFloor = 0; }  // object constructor
  int getCurrentFloor() {
    return currentFloor;
  }
  void call(int toFloor) {
    if (toFloor == currentFloor) {
      doorsOpen = true;
    } else if (toFloor < currentFloor) {
      doorsOpen = false;
      movingUp = false;
      movingDown = true;
      while (toFloor < currentFloor) {
        currentFloor--;
      }
    } else if (toFloor > currentFloor) {
      doorsOpen = false;
      movingUp = true;
      movingDown = false;
      while (toFloor > currentFloor) {
        currentFloor++;
      }
    }
  }
};
```

```
int main() {
  Elevator A;

  // if uncommented, the two lines below will generate compile-time errors
  // you are not permitted to directly access private member variables
  // A.currentFloor = 1;
  // std::cout << A.currentFloor;

  std::cout << "Elevator on: " << A.getCurrentFloor() << std::endl;
  std::cout << "Calling elevator to floor 10" << std::endl;
  A.call(10);
  std::cout << "Elevator on: " << A.getCurrentFloor() << std::endl;
}
```

Your task in this lab is to implement a `Stack` class in C++ that can hold `char` values. Your class should define the following public methods:

1. texttt int size() - return an integer representing the number of characters currently on the stack
2. `void push(char c)` - push character `c` onto the stack
3. `char pop()` - pop the top character off the stack

It is up to you to choose appropriate data structures to manage internal state of your `Stack` class. For additional detail, see Chapter 4 of A Practical Introduction to Data Structures and Algorithm Analysis.

Once you have implemented your `Stack` class, use it to solve problem 4.19 from A Practical Introduction to Data Structures and Algorithm Analysis, copied here for convenience:

A common problem for compilers and text editors is to determine if the parentheses (or other brackets) in a string are balanced and properly nested. For example, the string "((())())()" contains properly nested pairs of parentheses, but the string)()(does not, and the string ()) does not contain properly matching parentheses.

Implement an algorithm that returns `true` if a string contains properly nested and balanced parentheses, and `false` otherwise. Use a stack to keep track of the number of left parentheses seen so far. *Hint:* At no time while scanning a legal string from left to right will you have encountered more right parentheses than left parentheses.

Deliverable:

C++ code that compiles to a single executable (for example: `a.out`). This executable should accept a string of parentheses characters as a command-line argument and should print "true" if the string contains balanced parentheses. Print "false" otherwise.

```
$ ./a.out "((())())()"
true

$ ./a.out "(((((((()"
false

$ ./a.out ""
true
```