Assignment 5: Stacks

Description: In this assignment you will create a stack object that will work with a struct data type called **Data**. The struct data type contains an int called **id** and a string called **information**. The stack class will contain all the attributes and methods for a complete working, proper stack that will work with the struct Data type. You must implement a stack per the definition of a stack as given in class and the stack introduction assignment. Your stack will be capable of being created as any size from 2 to any number based on a parameter passed in from the command line. Your stack will be an array of pointers to **Data** structs (i.e. not the structs themselves).

Background

- This assignment assumes you have completed Assignment 4 (int stack) successfully.
- This assignment deals with the stack ADT. The notes from your text (optional) for stacks are chapters 06 and 07. Read those notes as a supplement if you find them useful.
- Watch the lectures and study the notes given on the assignment.

Algorithms:

- **push**: The process for pushing to the stack:
    - Pass data → pass *only* an int id and string pointer to the stack. Note that a string is an ADT itself so it must be passed by reference. The prototype for push should be one of the following, whichever notation and calls you prefer to use:
        - **bool push(int, string&);**
        - **bool push(int, string*);**
    - If there is room in the stack:
        - Test the validity of the data (positive int and non-empty string). Do not proceed to push if the data is invalid, return false.
        - Dynamically create a struct Data to hold the data.
        - Put the id and string in the struct Data.
        - Increment the stack counter.
        - Push the pointer for the struct onto the stack.
        - Return true.
    - If there is *not* room in the stack, return false.
    - Remember, only one actual return statement per function. Algorithms are not written like code, they are written to be logically consistent. The algorithm above looks like there are two or three returns, but in the actual code there should be only **one** return statement.
- **pop**: The process for popping from the stack.
    - Pass an 'empty' struct Data to the stack (by reference). Note that this means main will have a struct declared and it is passed by reference to the stack so the stack can fill it. This avoids a costly return by value. Make sure you understand this technique, it is common in programming.
    - If the stack is not empty…
        - Get the data from the top of the stack and put it in the struct Data passed from the caller.
        - Delete the allocated memory from the top of the stack.
        - Decrement the stack counter.
        - "Return" the data to the caller. Note that you will not actually "return" the data. The act of placing the data in the struct Data passed from the caller is the "return."
        - Return true (notice this pop is different from the last assignment, now it returns a bool).
    - If the stack is empty
        - Fill the passed struct with -1 and "" (empty string).
        - Return false (notice this pop is different from the last assignment, it returns a bool).

- **peek**: The process for peek() is the same as pop() but do not decrement the counter or deallocate any memory.
- **isEmpty**: The process for isEmpty() is to return true or false based on the top being -1 or not. This can be done in one very simple line of code. Think about it and see if you can do it in one line. You don't have to make it one line, but it's something to think about to improve your code.
- **Stack constructor**: Have one and only one constructor. The constructor accepts an int for the stack size and dynamically allocates the stack array in the constructor. Your stack must be able to be any size from 2 to n where n is any positive integer. If any other number is passed, handle that by setting the stack to a default value of 10.
- **Stack destructor**: You must deallocate anything left on the stack and the stack itself.
- In addition to those methods listed above, you will need a public **getSize()** method. This method can be called to check the size of the stack (i.e. it returns an int). You will need this because your stack is dynamically allocated and it's the only way to know from the outside how big the stack is.
- DO NOT MAKE ANY OTHER PUBLIC METHODS.

Remember ONE AND ONLY ONE return per function.
If you need more than one, your code is not structured properly.

Requirements:

- Follow the Assignment Specific Instructions using this GitHub assignment invite https://classroom.github.com/a/D-iD3xGu
- You are given all the files you need for the assignment except .gitignore. Make a proper .gitignore first and commit it.
    - **data.h**: Do not modify this file but you should study it.
    - **functions.cpp** and **functions.h**: Modify these as needed. You shouldn't need to, but are free to use this for your own functions and/or to modify the function that is there.
    - **main.h** and **main.cpp**: Modify these as needed. Follow the comment instructions in main.cpp.
    - **stack.h**: Modify this file as needed *except* for the attributes, *do not modify or add to them*.
    - **stack.cpp**: This is essentially blank, it's up to you to fill this in with the correct stack code.
    - **README.md**: Modify this file to describe your work.
    - DO NOT MAKE ANY OTHER FILES
- Place your comment headers on all files where indicated.
- Write a complete and proper stack in stack.cpp/h conforming to the traditional stack definition given above in the algorithms section and as explained in class, following all best practices and loose coupling.
- When your program runs, the user must pass in the size of the stack from the command line. For example, **a.out 5** or **a.exe 10**. Your program (main) must check the program is called correctly with one and only one parameter which is an int. If the program is not called correctly, main() must exit gracefully by telling the user they must enter a single parameter at the command line which represents an int for the stack size (it can be any int). Remember when you structure main() you may have only one return statement and may not use any functions that abruptly end the program (for example **exit();**).
- Make sure you properly allocate and deallocate memory, including deallocating whatever is left in the stack when it exits (i.e. use your destructor to do final clean up).
- Write ***complete*** ***and*** ***exhaustive*** tests for your stack in main.cpp.
    - DO NOT use user interaction to test your code. Your main() should be an automated test program demonstrating your stack is robust and fully functional.
    - Do proper reporting back to the user demonstrating your stack is working and being tested. You must report back from main to the user **proving** your stack works.

    - Because your stack can be any size, you must write automated tests that account for this. For example, if the user passes 3 for the stack size, you should be doing on the order of dozens of tests. If the user passes 10, then hundreds of tests. if the user passes 100, then thousands of tests. This means your testing must be dynamic and account for the stack size.
- All submission and good practice guidelines apply.

Grading: Your grade will be graded primarily on exactness to detail and specifications, architecture, and coding logic. You will also be graded on your repo and testing with the following guidelines:

- Failure to commit often, small, and smart will result in an **automatic -10% penalty** regardless of your code quality.
- Any stray files in your repo will result in an **automatic -10% penalty** regardless of your code quality.
- If you do not have proper comment headers and/or do not use the Write Submission feature (not the comment section), and/or do not submit your link correctly, it's an **automatic -10%** penalty regardless of your code quality.
- Failure to use the correct branch in your repo (main) will result in an **automatic -5% penalty** regardless of your code quality.
- Failure to test your code **thoroughly and exhaustively** will result in a maximum grade of 70% **regardless** of code quality.

Submission: When you are ready for grading, use the write submission feature in Blackboard and submit your repo link (this is the URL in the browser, not your SSH or .git link). If you need to fix/change something after you submit but *before* I grade it, just fix/change it and push again. **Do not re-submit the assignment before getting a grade**. Only re-submit after you get a grade and want a *re*-grading.