Assignment 4: Simple Int Stack

Description: To introduce a simple ADT (abstract data type) and to understand the function and purpose of ADTs. To accomplish this, you will use generative AI to assist your coding. This is a "toy problem" to introduce stacks, as well as to advance and train you in proper architecture, file structures, loosely coupled architectures, and testing. When this assignment is complete, you should be able to build and test any ADT properly and also use generative AI to assist you.

Background

- This assignment assumes you have completed Assignment 3 successfully and have all the necessary tools installed and understand all the processes needed to complete assignments.
- This assignment deals with the stack ADT. The notes from your text for stacks are chapters 6 and 7. Those notes are **not** needed, but study them if you find them useful.

Methodology: At this point in the semester, you are familiar with best practices, proper development procedure, file structure, naming conventions, etc. So, you will not be reminded of every little detail. You will be given general instructions, and you are expected to architect, code, document, submit, etc., correctly more or less on your own. You will be given specifications and parameters, and then it is up to you to apply all the correct processes, procedures, formatting, and architecture. For example, this project begins by only giving you a README, and it is empty. Now it's up to you to know what to put in it, what files to add, make sure you don't miss anything, etc. Use all the best practices documentation along with the checklist to make sure you do not miss anything.

Instructions:

- Follow the assignment-specific instructions using this GitHub assignment invite: https://classroom.github.com/a/83xxskmj
- The invite will give you one file, **README.md**
- Make the following 4 code files with the following content and your comment headers (**do not make any other .cpp/.h files**):
    - **main.h** and **main.cpp**
    - **stack.h** and **stack.cpp**
- Do anything else you need to set up your project correctly. Don't code yet. Think carefully here! There is a lot you can and should do at this point to be ready to code.

Ready to code...

- **Stack**
    - Attributes:
        - A static integer array for the stack that uses a default value (see below).
        - int top; to track of the top of the stack.
        - No other attributes are allowed.
    - Methods:
        - **int pop();** or **bool pop(int*);** // remove and return the top value
        - **int peek();** or **bool peek(int*);** // return the top value

        - **bool push(int);** // insert to the top of the stack
        - **bool isEmpty();** // test for empty (i.e., top < 0)
        - **DO NOT MAKE ANY OTHER PUBLIC METHODS**

    - Other Specifications:
        - Default stack size to 10. Remember, do not use literals. Use a #define or a constant in stack.h. **Hint**: This value should be available outside the stack so you can use it in main for dynamic testing.
        - You may make private methods, but think them through first and follow best practices. Before making any, ask, do you really **need** them? Are they efficient? Is there a better way other than a new private method?

    - Design Notes: The **pop( )** and **peek( )** functions will require you to show error somehow, and **you cannot return an int to show error** (explained in the lectures). One way to do this is exception handling (explained in the lectures). The other way to do this (if you know how) is to use pass-by-reference. The alternate prototypes for pop and peek given above reflect these two ways. You may do it either way, but be consistent (i.e., pop and peek use the same technique).

    - Go through an iterative process (as shown in lectures). Ask generative AI to build you an integer stack, and then go through **at least <u>five</u> rounds** of re-specifying the code until you get something close to complete that follows best practices. **Keep a record** of the process you went through and the code generated; you will need it at the end of this assignment.

    - Copy the best code from the AI tool and put it into your stack.cpp and stack.h.

    - Clean and reconfigure the code to conform **perfectly** to all best practices for this course. Use the best practices pages [on the website]() as a checklist.

- **Testing**
    - Use main to **thoroughly** test your stack. Put all testing in main; main is simply your testbed.
    - Place all test code in main directly. Do not break it up into functions; just make one long main( ).
    - You must test every possible operation in every possible combination and explicitly show your stack is fully functional and can handle underflow, overflow, incorrect input, multiple random operations in every combination.
    - During testing, change the default value of your stack and test those different values. Does your stack still work? Does your testing dynamically change to keep up with the stack size? Can you change stack size with no other changes, and everything else works logically and consistently? The answer to those questions should be yes in all cases. If it isn't, you're not done.
    - Make sure you have watched [the video on testing]() and have read the testing guide below.
    - 50% of your code grade is testing. If you do not test thoroughly and comprehensively, the highest code grade you can achieve is 50%.

- **Report:** Write a report detailing the following:
    - Explain how AI-generated code was at first and what you had to do to improve it. In other words, explain the iterative process you went through.
    - Include screenshots of code showing differences as your prompts changed and the code evolved. Show your prompts and code.

    - Explain your experience. What AI did you use? Did AI help or hinder your efforts? Could you have done better without it? Did you learn anything about prompting AI, and if so, what did you learn? Did you learn any new coding techniques, and if so, what? Detail anything else you think is important about the process and experience.
    - Use whatever word processor you like, but submit a single .pdf (preferred) or docx. No other formats will be accepted.

Grading: The grading process is detailed on the course website. Read that page carefully! **Pay particular attention to the "automatic zero" section.** Do not lose points or get a zero for failure to follow specifications or best practices. Use that section like a checklist, and go over it **carefully**. For this assignment, your code will be 75% of your grade, and the report will be 25%. Each will be graded 0-100, and then the ratio applied.

Use this checklist to double-check your code. 👈 IMPORTANT!

Submission

- When you are ready for grading, use the Write Submission feature in Blackboard and submit your repo's link and attach your report (.pdf or .docx).
- Remember, if you need to fix/change something **after** you submit but **before** it's graded, just fix/change it and push again. **Do not resubmit the assignment before getting a grade**. Only re-submit after you get a grade and want a re-grading.

Guide to proper testing:

Developer testing is one of the most important aspects of writing code, and typically it is where you spend most of your time and do most of your coding. Proper testing means you must execute every operation, in every possible combination, and in every possible state for the ADT in question. In this case, you will need hundreds or even thousands of tests. Testing must also be dynamic based on the size of your stack. You must imagine, what if the stack size was changed to 10,000 or 100,000? Would your tests automatically keep up and be appropriate? You must show what happens when your stack is pushed beyond its limits in both directions for all operations, and you must include hundreds or even thousands of random operations to do proper testing.

To complete comprehensive testing, the best way to proceed is to first list every possible state your ADT can attain. For example, in the case of an int stack, there are three: 1) underflow, 2) overflow, and 3) neither underflow nor overflow. Then list every possible operation: 1) pop, 2) peek, 3) push, and 4) isEmpty. Once you know every possible state and operation, testing proceeds in two stages: 1) explicit testing, which tests all possible combinations of all states and operations, and 2) random testing, which randomly executes every possible operation a number of times proportional to your ADT's size (for example, 1000 random operations for size 10, 10,000 for size 100, and so on).