CSCI 161 Assignment 5: An indexed sequence ADT

Objectives

Upon completion of this assignment, you will to be able to:

- Implement the Stack ADT as both a vector and a singly-linked list as its underlying data structure.
- Understand the use of information hiding.
- Gain experience implementing a container object that can hold *generic* data types as the elements.
- Use parts of the implementations of the `Vector` and the `List` objects.
- Be familiar with a simple user-defined indexed sequence ADT, (abstract data type).
- Build on the programs created in previous assignments, getting more practice with some of the concepts.
- Use unit-testing for the implementation of different versions of the same ADT.

The expected completion time for this assignment should be between 2 to 4 hours. Please take advantage of the lab room drop in and Help Centre schedule to get help and support. Note that you have time over Study Week as well, but it is highly recommended you get most of this assignment done in the scheduled lab time.

Introduction

In this assignment, we will implement a basic stack object, based on the definition of a `Stack` *Abstract Data Type* (ADT) definition discussed in lecture. The basic functions and descriptions are in the `Stack.hpp` file. We will be creating two versions: `StackV.cpp` and `StackL.cpp`. They both provide the exact same functionality, but their implementation is different.

We *could* make the underlying attributes either the `Vector` or the `List` objects, which if you have perfected these in previous assignments, but for this assignment, we instead, implement a subset of these structures.

The `Stack` ADT is described as follows:

- It is a container of elements of unknown size that allows the following access:
  - ⋆ The `stack` knows if it is empty or not.
  - ⋆ Elements can be inserted one at a time (via the `push` operation).
  - ⋆ Only the last element that was inserted is viewable while inside the `stack` (via the `top` operation).
  - ⋆ The only element that can be removed in a single step is the ‘top’ element, the one that is viewable (via the `pop` operation).

  The `stack` is often defined as a `Last-In-First-Out` (LIFO) ADT.

Preparation

(1) Download this pdf file and store it on your computer.
(2) Create a directory / folder on your computer specific to this assignment, for example, `CSCI161/assn5` is a good name.
(3) Download and store the following files:
   `Stack.hpp`: DO NOT ALTER. This header file covers the ADT operations, as well as the private variables required for both implementations. Note that complete user definitions are contained in the comments of this file. Normally, the contents would be used as input to generate the `Assn5_UserGuide.pdf` file, but because there are not many member functions, it should be easy to access the information directly from this header file.
   `StackTemplate.cpp`: DO NOT ALTER. This is just a template to help you set up the implementation code so that all the necessary templates work. Use this as the *base* file for each of the versions of the stack. Copy this file into the files `StackV.cpp` and `StackL.cpp` files. Then fill in the functions.
   `makefile`: The optional simple IDE used on unix and linux machines. If you use it, it will help you sort out the compilation requirements.
(4) While doing the implementation for the individual versions, use the `main` function at the bottom as a unit tester. Once the individual file has been tested for each of the versions, then comment out the unit testers, and create a separate `main.cpp` file. This will test both files.

Tips and Implementation Details

the `Stack` class definition

The `class` is defined in the header file and except for the few private variables, gives no indication of the implementation. The `StackV.cpp` file will use the private variables that will make the `Stack` use an underlying expandable array as the base, while the `StackL.cpp` file will use the private varaibles that make the `Stack` use a singly-linked list as the base.

Note that because the `Stack` ADT has so few operations, the bases only require the functionality that will get the job done for the `Stack` class. It is up to you to decide how you will use the base in such a way that is most efficient in terms of the number of operations required (in terms of the number of elements). Note that constant time for each operation is most desirable.

compiling and testing

The `makefile`, even if you don't use it, should provide some clues as how to compile the files. Once the unit testing in a single implementation is working, then comment out the main function. Create a separate `main.cpp` file with a single main function that `#includes "Stack.hpp"` and write the generic tester. This `main.cpp` can test both implementations, but only one at a time.

To compile, run and test the `StackV` implementation, using the `makefile` type:

```
make goV
// once compilation done
./goV
```

To compilel, run and test the `StackL` implementation, using the `makefile` type:

```
make goL
// once compilation done
./goL
```

If you do not use the makefile, you can open it in a text editor to see that compiling both `main.cpp` with either `StackV.cpp` or `StackL.cpp` will produce the executable. For obvious reasons, both versions cannot be compiled and linked together.

Submission

Submit the following completed file to the Assignment folder on VIULearn before :

Monday, February 26, 23:59hrs

- `StackV.cpp`
- `StackL.cpp`
- `main.cpp`

A note about academic integrity

It is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

- All code must be clearly commented internally, describing the reasons for code choices.
- Any outside source of information must be cited.
- The code may not be generated by an AI program.

Grading

Marks are allocated for the following statements being true:

- All of the functions in `Stack.cpp` must perform as declared in the `Stack.hpp` file.
- The code submitted must compile on the lab machines,
- Internal commenting is present and include any of the following where appropriate:
    - ⋆ Reasons for code structure choices made during implementation.
    - ⋆ Explanations of constructs not discussed in lecture and source cited.
    - ⋆ Questions directed to the instructor.
    - ⋆ Ideas that extend any lecture discussions.
    - ⋆ Reasons for algorithmic choices.
- The underlying data structure for the `Stack` classes must be as specified. No outside or library classes may be substituted for the unlimited array or the singly linked list.
- No C++ container libraries may be included.
- Thorough testing that works for either implementations must be present in the `main.cpp` ffile.