COMP 3522 Lab #3: OOP and testing

Christopher Thompson, Jeffrey Yim
jyim3@bcit.ca

Due Friday 11:59pm

Welcome!

Welcome back! For your third lab, you will implement a familiar and well-loved data structure in C++, and then learn how to use CLion and a common unit testing framework to ensure your code is correct.

1 Set up your lab

Start by creating a new project:

1. Clone your repo using github classroom: https://classroom.github.com/a/FaIBQRTB
2. Fill out your **name** and **student number** at the top of **main.cpp**
3. Ensure you commit and push your work frequently. You will not earn full marks if you don't

2 Implement a stack

Let's implement a stack! Remember that a stack is a LIFO (last in first out) data structure. We add elements to the top of the stack, and when we want to take an element from the stack, we also take from the top:

1. Ensure you commit and push your work frequently. You will not earn full marks if you don't.
2. You must include a header file called myStack.hpp and a source file called myStack.cpp.
3. Name your stack class "MyStack" (hint: C++ has a built-in stack and we want to avoid naming conflicts)
4. The stack only stores ints.
5. The stack should be implemented using an **array**
6. It has a maximum size of 10 (define a constexpr).
7. There must be two private data members:
   (a) A C-style array of int of size 10 (don't use magic numbers!)
   (b) An integer which stores the index of the current "top" of the stack.

8. Implement a default constructor which initializes the index of the "top" to -1 (a new stack of int is empty so the index should be something that can never be a real index)
9. Implement a member function called push which accepts an int and adds it to the top of the stack, returning true if it was added, else false. This member function would return false if the stack was already full, for example.
10. Implement a member function called pop which accepts no parameters and returns void. This method should decrement the instance variable which stores the index of the "top" element. We are removing the int at the top of the stack by ignoring it now. You don't need to zero it out. Why not?
11. Implement a constant member function called top which accepts no parameters and returns (without removing) the int on the "top" of the stack.
12. Implement a constant member function called empty which accepts no parameters and returns true if the stack is empty, else false.
13. Implement a constant member function called full which accepts no parameters and returns true if the stack is full, else false.
14. Implement a constant member function called print which should print the contents of the stack (bottom of the stack to the top) to standard output in an easy-to-read format. Have the print function return something (string, stringstream etc) that you can use to test your output.

3 Test your code

You've probably noticed that I haven't asked you to put anything in the main function yet. We need a main function so our code can compile and execute, but it just has a print statement in it right now. So far this term, we've been testing our function in the main function. It's time to start unit testing:

1. Start by reviewing the **"Testing review.pdf"** slide deck included with this lab. These slides will be familiar to students who took COMP 2910, and will be helpful for students who were in co-op instead.
2. CLion integrates very nicely with three C++ unit testing frameworks. We will use the Catch unit testing framework first. It is easiest to use and has the shortest learning curve. Catch requires a header file, and that's IT!
3. Visit the Catch site at `https://github.com/catchorg/Catch2`.
4. I've already included "catch.hpp" in your cloned project, however, instructions for adding Catch to your project are here (Hint: you just have to drag a header file into your project!): `https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md#top`
5. Bonus reading: the developer of Catch works at JetStreams now (hooray for us!) and wrote an article about CLion and Catch which I would like you read: `https://blog.jetbrains.com/clion/2017/03/to-catch-a-clion/`
6. In order to use Catch, comment out the main method in main.cpp for now.
7. Create a new source file called unit_tests.cpp in your project.
8. Modify your project configuration. Go to Run | Edit Configurations and click the + in the upper left to add a run configuration. Choose "Catch" and call your configuration Unit Tests.
9. There are two lines of code you must add to the top of your unit_tests.cpp file in order for Catch to work:

```
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main(), put this in one cpp file
```

```
#include "catch.hpp"
```

This will automatically create an invisible main function for us, so go ahead and comment your main function out.

Here's the code for your first unit test. A freshly created stack should be empty. So I've created a test case for this. Inside the test case, I use the Catch **REQUIRE** statement to assert what should be true:

```
TEST_CASE("A new stack is empty", "testTag1")
{
    MyStack tester;

    REQUIRE(tester.empty() == true);
    REQUIRE(tester.full() == false);
}
```

10. To execute the test, select Run | Run unit tests. Mic drop.
11. Your final task is to test your code thoroughly. Each function must be tested. While testing your function, you should identify preconditions, postconditions, and invariants for your class, and comment each module appropriately.
12. Note: Running unit tests with catch is **very slow**. My suggestion: use the regular main function to thoroughly test edge cases for the stack first. Once you feel confident in your stack implementation, then write unit tests
13. You will find this page helpful. It describes the kinds of assertions you can make with Catch. Push it to its limits!
    https://github.com/catchorg/Catch2/blob/master/docs/Readme.md#top
14. Do NOT submit this Lab via D2L/The Learning Hub, submit it via GitHub

4 Grading

This lab will be marked out of 10. For full marks this week, you must:

1. (2 points) Commit and push to GitHub after each non-trivial change to your code
2. (3 points) Successfully implement the data structure exactly as described in this document
3. (3 points) Successfully test the data structure exactly as described in this document
4. (2 points) Write code that is consistently commented and formatted correctly using good variable names, efficient design choices, atomic functions, constants instead of magic numbers, etc.

Frequently asked questions/Troubleshooting:

1. **What is this "Too many sections"/"File too big error"?**
   If you're getting an error mentioning similar to the following:
   /usr/lib/gcc/x86_64-pc-cygwin/7.4.0/../../../../x86_64-pc-cygwin/bin/as:
   CMakeFiles/lab3template.dir/unit_tests.cpp.o: **too many sections** (32801)
   /tmp/cciusp7u.s: Assembler messages:
   /tmp/cciusp7u.s: Fatal error: can't write 171 bytes to section .text of
   CMakeFiles/lab3template.dir/unit_tests.cpp.o because: '**File too big**'

   Replace this line in CMakeLists.txt:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -pedantic")
```

to the following:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -pedantic
-Wa,-mbig-obj")
```

2. **How do I write multiple test cases?**

Write code in each TEST_CASE block as if it was its own separate main method

```
TEST_CASE("my_test_1_description","optional_test_tag")

{

  MyStack s;

  //test checking the top value on an empty stack

}



TEST_CASE("my_test_2_description","optional_test_tag")

{

  MyStack s;

  //test pushing to a full stack

}
```

Each test_case is run independently, so imagine there's multiple main methods, all run separately. Don't make ONE GIANT TEST_CASE function that tests everything

3. **How many test cases should I make?**

Be thorough in checking edge cases**.** Try testing at least the cases below. Remember**,** every test should be its own test case.

- Pop an empty stack
- Pop empty stack then add values
- Push more than maximum stack amount, check value of top of stack
- Check top value on empty stack
- Push X values pop Y values, check top for expected value
- Push exactly 10 values, check if stack is full and top value is expected value

4. **What value do I return when checking the top of an empty stack?**

Return -1 when checking the top from an empty stack. Save -1 in a global const so you can check for it

5. **What should the output be when printing an empty stack?**

   Print a helpful message stating the stack is empty

6. **Are magic numbers allowed for the unit tests?**

   Yes

7. **Do I need to add comments for the test cases?**

   No. Just make sure the description in the 1st parameter of the test case describes the test thoroughly.