Micro-assignment 1: Reviewing, Critiquing, Testing, and Debugging C++ Code

I. Learner Objectives:

At the conclusion of this programming assignment, participants should be able to:

Develop C++ programs
Review, critique, test, and debug someone else's C++ code
Identify and understand syntax errors reported by g++
Critically analyze C++ code
Write test cases
Implement test code
Debug using a debugging tool

II. Prerequisites:

Before starting this programming assignment, participants should be able to:

- Analyze a basic set of requirements and apply top-down design principles for a problem
- Design, implement, and test medium programs in an object-oriented language
- Edit, build, and run programs through a Linux/WSL/MacOS environment

III. Overview & Requirements:

In this micro-assignment, you will be reviewing, critiquing, testing and debugging the provided code, in your environment. Please start with the main.cpp found on Canvas.

Overview

What is software testing?

Software testing is an activity to check whether the *actual* results match the *expected* results for the *unit* (we will consider a function or class) under test. If the actual results do not match the expected results, then we have found a *bug*! Without distinguishing too much between the various categories that a software or system problem could be categorized (i.e. failure, fault, error, etc.), we will group all software issues in the category of bug. A software bug is the result of one or more coding errors. These errors lead to a poorly working program, produce incorrect results, or cause a software crash.

Undoubtedly, you have practiced writing some test code in your prior CS courses. However, we want to solidify this concept further, and understand the great importance of software testing!

Why is software testing important?

Testing software is important not only because bugs could be expensive, but also because they could be dangerous or harmful. Testing should not be an afterthought, but really should be at the forefront of how we approach developing our software.

Note: you could even consider writing tests before you write production code! Test-driven development (TDD) is a software development process that prioritizes small evolutionary cycles to foster test-first development. The concept is that a test should be written before enough production code is implemented. It encourages clean and modular code that is *testable*. Testable code makes automated testing smoother, and in general the code more robust.

Software testing also provides confidence that limited bugs are left in the product. The level of confidence depends on the software testing *coverage* established and attained.

What is a test case?

A test *case* is a specific set of *inputs*, execution *conditions*, and expected results developed for a particular objective or requirement (def. Binder). A test case generally contains a unique test ID, test description, test steps, test data, preconditions, postconditions, and expected results to verify the objective or requirement. More information could be added to a test case, but we will leave it as at this point in the course.

Below is an example of a test case, structured within a comment block, that is required for this assignment. The style of the test case structure will vary in practice!

```
/*      Test ID: Empty queue check - EQC
        Unit: queue::isEmpty ()
       Description: test to determine if queue::isEmpty () returns 1 if a
       queue object is empty
        Test steps:
               1. Construct an empty queue object
               2. Invoke queue::isEmpty ()
               3. Conditionally evaluate the valued returned by
                  queue::isEmpty ()
        Test data: size = 0
        Precondition: queue object is empty
        Postcondition: queue object is still empty
       Expected result: queue is empty; 1 is returned
        Actual result: queue is empty; 1 is returned
        Status: passed
*/
```

What is white-box testing?

White-box testing, also known as *structural*, *glass* box, or *clear* box testing, is a method for providing tests based on knowledge and access to the internal implementation or structure of the code being tested. As a comparison, *black-box* testing, also known
as *behavioral* or *functional* testing, uses only the external interface and functional specification to develop test cases. In this assignment, we will focus solely on white-box testing single units or functions. Each unit may require several test cases.

What is unit testing?

Unit testing is a software testing method by which individual units (which can be functions, grouping of functions, or classes) are exercised with test cases. Each unit is generally tested to satisfy a level of code *coverage*. To design our tests, we need to understand the level of confidence or code coverage we aim to satisfy.

What is software testing code coverage?

Code *coverage* is a measure of the degree to which a unit under test has been tested. The higher the coverage, the more confident we are that our code does not have bugs. Code coverage also allows for us to quantitatively access the thoroughness of our test cases. To understand the thoroughness of our tests, we need to establish coverage *criteria*.

What is software testing coverage criteria?

Coverage *criteria* establish measure how extensively our test cases *exercise* the unit under test. There are several criteria including: *statement*, *branch, condition, path,* and more coverages. Branch coverage is more thorough than statement coverage, and condition coverage is more thorough than branch coverage, etc.

We will focus on branch coverage. We will also try to achieve 100% coverage. Branch coverage's goal is to ensure that all branches (false and true) from a given decision point are executed. Remember we know where these decision points exist because we have access to the implementation details of the unit. To achieve 100% coverage, we need to test all branches in each unit.

For example:

Given the following code, how many tests are required?

```
if (newData > max)
{
        max = newData;
}
```

Two tests are required to achieve 100% branch coverage, though only one test would be required to achieve 100% statement coverage (newData > max). The tests that are necessary include:

1. The case when newData is > max – tests the branch for when the decision is true
2. The case when newData is <= max – tests the branch for when the decision is false

Could we automate software testing?

There are many great testing environments and frameworks available to automate a testing process including: GoogleTest, Boost.Test, CppUnit, and many others. These environments allow for consistent setup for tests, test execution, and test evaluation. However, at this point in the course we will not use one of those.

What is required?

For this assignment complete the following requirements in order:

1. **(5 pts)** You should take a screen shot or picture of the syntax errors listed in the initial code after you build the code (with CMake, Make, g++, etc.) in the Linux/WSL/MacOS terminal. This should demonstrate that you are using a Linux/WSL/MacOS environment. The picture should end up in a .pdf file in your MA1 folder.

If you are using CMake, make sure your CMakeLists.txt contains the following "set" commands:

```
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_BUILD_TYPE Debug)
```

2. **(15 pts)** You are required to fix all syntax and build errors. If the error is directly related to an environment-specific condition (i.e. only works on Linux, but not Windows), then comment out the statement causing the problem. You do NOT need to find an equivalent fix for the other environment.

3. **(45 pts - 5 pts/test case)** Once you have fixed all syntax and build errors, please place the class queue and all its function definitions to a separate header file called `queue.h`. You are required to write *unit* tests for all functions or units in `queue.h` excluding the constructor, and the destructor, in the project. Your objective is to achieve 100% branch coverage for each unit. For each unit test you are required to construct a comment block with the information described in the background section "*What is a test case?*", and implement a test function for the corresponding test case. Each test function must have a function *declaration* that is placed in a file called `testQueue.h` or `testQueue.hpp` and all test

function *definitions* and comment blocks must be placed in a file called `testQueue.cpp`. A test function should always be declared as void testFunctionName (void). This means we want these tests to be self-contained. They do not accept any arguments and do not return any results. All setup and evaluation for the test is done inside the function.

Hint: you will need 1 test case for each of the following functions:

    queue::size ()
    queue::isEmpty ()
    queue::isFull (),

you will need 2 test cases for the following:

    queue::dequeue ()
    queue::enqueue ()
    queue::peek ()

Call your test functions from main (). At this point do NOT fix the bugs discovered! This will be done as part of the next step! Place the results of each test case, i.e. pass or fail in the same comment block as the test case comment block.

4. **(20 pts)** Fix all bugs revealed by your test cases. Show a screen shot or picture of one break point that you have added to a unit to debug, where the bug was identified by one of your test cases. To this end, you should use a debugging tool (gbd, CLion, or VS Code). The picture should end up in a .pdf file.

5. **(15 pts – 3 pts/attribute)** Using your understanding of design choices, software principles, and coding standards, which we will group under the general label "attributes" – list and describe 5 attributes demonstrated by the code that you would consider poor. These should NOT be related to the syntax errors. Examples of poor attributes could be related to comments, file structure, data structure selection, algorithm efficiency, etc. Place your list in a comment block at the top of the `main.cpp` file.

IV. Submission: Git (sharing with TA by creating a Github repository)

1. On your local file system, and inside of your Git repo for the class, create a new *branch* called MA1. In the current working directory, also create a new directory called MA1. Place all MA1 files in the directory, if you are not

directly working in your repo. All files for MA 1 must be added, committed, and pushed to the remote origin which is your **private** GitHub repo created in PA1 (DO NOT CREATE NEW REPO).

2. You should submit at least two header file (`testQueue.h` or `testQueue.hpp` file, and `queue.h`), two C++ source files (called `main.cpp`, which has your bug fixes, and `testQueue.cpp`), one .pdf file with your screen shots or pictures (paste all figures to a MS Word and save as .pdf), and a .txt file containing your building commands on Linux/WSL/MacOS (e.g., a CMakeLists.txt) which can compile your code.
3. You can start from the GitHub template project: https://github.com/DataOceanLab/CPTS-223-Examples
4. Please add the GitHub accounts of TAs (see Syllabus/Schedule page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. Otherwise, we will not be able to see your repository and grade your submission. DO NOT CREATE NEW REPO.
5. Do not push new commits the branch after you submit your link to Canvas otherwise it might be considered as late submission.

V. Grading Guidelines:

This assignment is worth 100 points. We will grade according to the following criteria:

See above in the section **What is required**? for individual points.