CPTS 223 Advanced Data Structure C/C++ Fall 2024

MA1: Reviewing, Critiquing, Testing and Debugging C++ Code

1 Learning 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.

2 Prerequisite

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.

3 Overview and 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" on Canvas.

3.1 Overview

3.1.1 What is software testing?

Software testing is an activity to check whether the actual results match the expected results for the unit (i.e., the smallest testable part of a software application and 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!

3.1.2 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.

3.1.3 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 value 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
*/
```

3.1.4 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.

3.1.5 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.

3.1.6 What is 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.

3.1.7 What are coverage criteria?

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

We will focus on branch coverage, i.e., the percentage of branches (decision points in the branches of `if`, `else`, `switch`, etc.) in the code that have been executed by a set of test cases and 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 is required to achieve 100% statement coverage (i.e., the percentage of executable statements in the code that have been executed/tested), that is `newData > max`. The tests that are necessary to achieve 100% branch coverage include:

1. The case when newData > 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.

3.1.8 Can 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 them.

3.2 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 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.

   Please 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 should construct a comment block with the information described in the background section 3.1.3, and implement a test function for the corresponding test case. Each test function should have a function declaration that is placed in a file called "testQueue.h" or "testQueue.hpp" and all test function definitions and comment blocks should 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.

   You will need one test case for each of the following functions:

   - queue::size()
   - queue::isEmpty()
   - queue::isFull()

   You will need two test cases for the following functions:

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

   Call your test functions from main(). At this point do NOT fix the bugs discovered, because it 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, VS Code, etc.). 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. They 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.

4 How to Submit: Github (and Share with TA)

1. In your Git repository for this class's coding assignments, **create a new branch called "MA1"** (Refer to this YouTube video about how to create a branch from Github web interface or the terminal). In the current working directory, also create a new directory called "MA1" and place all MA1 files in the "MA1" directory. All files for MA1 should be added, committed and pushed to the remote origin which is your private GitHub repository created when during PA1 (NO NEED TO CREATE A NEW REPO).

2. You should submit at least the following files:

   - two header file ("testQueue.h" or "testQueue.hpp" file, and "queue.h"),
   - two C++ source files ("main.cpp", which has your bug fixes, and "testQueue.cpp"),
   - one .pdf file with your screen shots or pictures (e.g., paste all figures to a MS Word and save as .pdf),
   - a "CMakeLists.txt" file containing your CMake building commands on Linux/WSL/MacOS which can compile your code.

3. You can refer the example GitHub template project for how to use CMake at https://github.com/DataOceanLab/CPTS-223-Examples.
4. Please invite the GitHub accounts of TAs (see Syllabus page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. You should submit a URL link to the branch of your private GitHub repository on Canvas. Otherwise, we will not be able to see your repository and grade your submission.
5. Please push all commits of this branch before the due date for submitting your Github link to Canvas portal. Otherwise it might be considered as late submission.

Here is a checklist for submitting your code:

- Invite all TAs and instructor as collaborators of your created repository "CPTS223_assignments". Their github username can be found in Syllabus on Canvas.
- Commit and push your local code to your repository.
- Make sure that your repository reflects and contains all your latest files.
- Copy the link to your repository in the Canvas submission portal.

5 Grading Guidelines

This assignment is worth 100 points. We will grade according to the following criteria: See Section 3.2 for individual points.