UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

How To Test

1 Overview

This document serves as an example of how to write testing code, and should be read in conjunction with Practical 1. In Practical 1, you were given a `main.cpp` which contains a few basic examples of testing code. This should be used as an example/template, such that you can write your own testing code for future practicals and assignments, as you will receive marks for this.

2 Test structure

The testing framework that is discussed in Section 3 uses the hierarchy shown in Figure 1 to test your code:

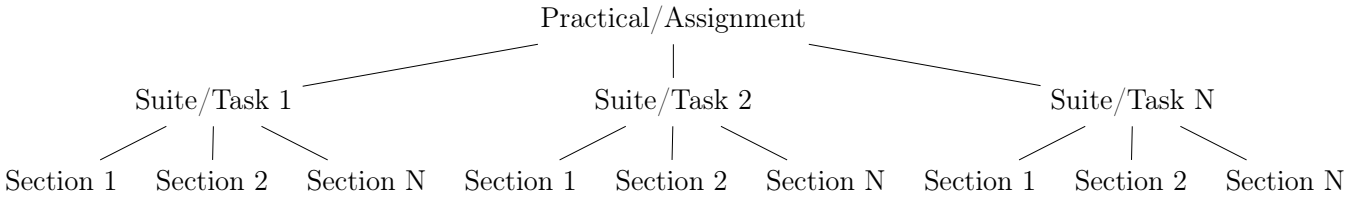

Figure 1: Abstracted view of testing framework layout

Figure 1 shows that for every practical or assignment that you need to test, you create a bunch of suites or tasks. A suite represents a set of sections which is usually related to a specific function or even a specific class. Each section is then a specific use case or edge case that you are testing. Within each section, you utilise the assert functions described in Section 3 to verify that what you think your code should do is what your code actually does.

Note, the more granular your suites and sections become, the easier it is to identify missed test cases and find errors in your code. Create as much suites and sections as you need to ensure you are confident in your code.

3 The testing framework code

You are provided with a `testingFramework.cpp` and `testingFramework.h` that contains variables and functions which automates some aspects of testing. A rundown of the functions are given below:

- changePrintMode
  - If you pass in 0 to this function, it will be in normal printing mode. This means that the received output is just printed normally. This can be used to directly see the program output.
  - If you pass in 1 to this function, then it will be in the testing mode. For each assert function, it will print out Test Passed/Test Failed.
- assert
  - There are currently two assert functions, one for ints, and one for strings.
  - If you need more functions for different data types, please add your own ones. (Later in the semester you will learn about templates and then we will show you how this can be done elegantly).
  - The assert function takes in two parameters, firstly an *expected value*, and then a *received value*.
  - The *expected value* should be seen as the memo answer, which is usually hardcoded when calling this function.
  - The *received value* is the output generated by your program.
  - If the *printMode* is 0, then it will just print out the received input.
  - If the *printMode* is 1, then it will print:
    - "Test #num passed" if the received and expected are the same
    - "Test #num failed" as well as the values of expected and received, if they were not the same.
- assertArray
  - This works the same as the assert functions, but it also needs a size parameter to indicate how many elements are in the array.
- assertNULL
  - This function can be used to check if a value is NULL or not NULL.
  - If the *isNULL* variable is true (or left out) then the expected value is NULL. If *isNULL* is false, then the expected value is not NULL.
  - You will need to copy this function for every pointer type you want to check if it is NULL.
  - *Hint: A similar function can be set up for testing if an expected value should be true, and further more a similar function can also be set up for false.*

- startSection
  - This starts a new section.
  - Sections are a way of organizing a group of tests.
  - A section can be given a name which makes it easier to read the output.
  - Sections can contain multiple assert statements, where each assert statement is one test.
- startSuite
  - A suite is a group of sections.
  - When calling *startSuite* you can give it a name which makes the output easier to read.
- endSuite
  - This ends the currently running Suite, by reseting the test number counters.
  - If the number of tests ran is equal to the number of tests passed, it will say "ALL Tests passed", otherwise it will say how many tests failed.

4 Practical 1 Example

- The first step is inside the main function to set the *printMode* to 1, as this makes it easier to see what tests ran and which tests failed.
- It is recommend to test each function on its own, but sometimes you need to group some of them together.
- My first set of tests are for the `createArray()` and `destroyArray()` functions. Thus in the main I made a function `createAndDestroyTest`.
  - Inside this function I start by calling *startSuite*, to show that a new Suite has started. At the end of this function, I then call *endSuite*.
  - The first section I test, is when a blank string is used as input for the `createArray` function. I thus call *startSection*, and give it a name.
  - Inside this section, I call the functions I want to test, which is `createArray()`, and `destroyArray()`.
  - After calling the functions, I use assert statements to check that the answer my program is getting, is the correct one.
  - Firstly, I know that the *numRows* variable should be 1, so I start by checking that. Secondly, I check that the arrays are also correct.
  - After calling `destroyArray()`, I know the pointers should be NULL, and the *numRows* should be 0. Thus, I use assert calls to check all of these.
  - I then start the next section, which tests an easy example of a small dynamic 2D array.
- To keep my code cleaner, I then made another function to test the print functions. They follow the same general example as was explained in the previous bullet.

5 Tips for testing

- NEVER delete old test cases. Keep all of them active, even if they have passed. This helps you as if you fix your code, but accidentally break an older test case, you will immediately know that you broke that older test case. (This is known as regression testing).
- Make sure you test every function thoroughly. Use code coverage tools like GCOV (there is a guide on ClickUp on how to use this) to know when each line has been run. Note that 100% coverage does not mean you tested everything, but it is a good starting place.
- Write your own testing code, as it is vital skill for all programmers to have, and you can get striked for plagiarism if you use the same testing code as your friend.
- It is suggested that you use the "*Given, When, Then*" structure to structure your code. This allows you to determine what the scenario or context was that caused the error to fail. You can read up more about this at the following link:
  `https://medium.com/geekculture/given-when-then-326d86a3c165`