# Assignment 01: Review of Functions and Arrays

COSC 2336: Data Structures and Algorithms

Fall 2021

## Objectives

- Practice using `<cmath>` predefined functions
- Review and practice writing user defined functions
- Review of arrays in C
- Practice passing arrays into functions

## Description

In this programming assignment you will write a program that will take an array of integers, and calculate the mean (average) and standard deviation of the numbers. If the numbers in the array are

```
int n = 5;
int x[] = {5, 3, 8, 2, 1};
```

Then to calculate the mean, we use the formula:

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

where $\bar{x}$ represents the calculated mean, and $n$ is the total number of values in the array $x$. In other words, the mean is simply the sum of the values divided by the total number of values ($n$), or

$$\bar{x} = \frac{x_0 + x_1 + x_2 + x_3 + x_4}{n} = \frac{5+3+8+2+1}{5} = 3.8$$

Likewise, the formula for calculating the standard deviation of a set of values is given by:

$$s = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(x_i - \bar{x})^2}$$

Here you should notice that $\bar{x}$ is the calculated mean of the values in the $x$ array we just showed previously. In English, the standard deviation is the sum of the square of the differences of each value from the mean. This sum of the squared differences is again divided by $n$ the total number of values, then we take the square root of this whole calculation to get the final standard deviation.

So for our example $x$ array, the standard deviation would be calculated as

$$\begin{aligned} s &= \sqrt{\frac{1}{n}\left[(x_0 - \bar{x})^2 + (x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + (x_3 - \bar{x})^2 + (x_4 - \bar{x})^2\right]} \\ &= \sqrt{\frac{1}{5}\left[(5 - 3.8)^2 + (3 - 3.8)^2 + (8 - 3.8)^2 + (2 - 3.8)^2 + (1 - 3.8)^2\right]} \\ &\approx 2.4819 \end{aligned}$$

## Setup

For this assignment you will be given the following files:

| File Name | Description |
|---|---|
| `assg01-tests.cpp` | Unit tests for the two functions you are to write. |
| `assg01-functions.hpp` | Header file for function prototypes you are to add. |
| `assg01-functions.cpp` | Implementation file for the functions you are to write for this assignment. |

Set up a multi-file project to compile the two `.cpp` source files together and run them as shown for the class. The `Makefile` you were given should be usable to create a build project using the Atom editor as required in this class. The general approach you should take for this assignment, and all assignment is:

1. Set up your project with the given starting templates. The files should compile and run, but either no tests will be run, or tests will run but be failing.
2. For this project, start by uncommenting the first `TEST_CASE` in the `assg01-tests.cpp` file. These are the unit tests to test the functionality of your function to calculate the mean of an array of integer values. When you uncomment this first unit test case, your program will no longer be able to compile, because you have not yet written the `calculateMean()` function.
3. Add the correct function prototype for your `calculateMean()` function to the `assg01-functions.hpp` header file. The prototype consists of the name of the function, its input parameters and their types, and the return value of the function.
4. Add a stub/empty implementation of `calculateMean()` to your own `assg01-functions.cpp` implementation file. The function should have the same signature as the prototype you gave in the header file. The function should initially just return a result of 0.0 since this is a value returning function that returns a double value as a result.
5. Your code should compile and run now. Make sure after adding the function prototype and stub your code compiles and runs. However, some but not all of the unit tests will now be passing, and most will be failing.
6. Incrementally implement the functionality of your `calculateMean()` function. You should try to add no more than 2 or 3 lines of code, and then make sure you program still compiles and runs. Start by adding code to get the first failing test to pass. Then once that test passes, move on to the next failing tests until you have all tests passing. If you write something that causes a previously passing test to fail, you should stop and figure out why, and either fix it so that the original test still passes, or remove what you did and try a new approach.
7. Once you have the `calculateMean()` function implemented and all unit tests passing, you should then move on to the `calculateStandardDeviation()` function. Your approach should be the same. Start by uncommenting the second `TEST_CASE` and making sure code still compiles after adding your function prototype to the header file and a stub/empty function to the implementation file. Then incrementally add code to this function, recompiling often and paying attention to which unit tests are passing and which are failing while implementing the function.

## Tasks

You should set up your project/code as described in the previous section. In this section we give some more details on implementing the two functions you are to write for this assignment. You should perform the following tasks for this assignment, once you have your project set up and building:

1. Implement the function named `calculateMean()`. You should already have a stub of this function that compiles and returns a result of 0.0 before starting to implement the function. This function takes two parameters, an integer indicating the size of the array of values ($n$), and an array of integers. This function will return a value of type `double`. The signature and name of the function should match how it is tested in the unit tests file. The function should calculate the mean for any array of 1 or more integer values passed in to it, as described above. The function should calculate the mean using `double` valued precision, because even though the values

are integers, the calculated mean can often be a non integer result. **HINT**: be careful about integer division. If you sum up the values in the array as an integer sum, and you divide by an integer, the result by default in C/C++ will be an integer. You need to cast one or both values to a double value when performing the division to get a double result.

2. Implement a second function named `calculateStandardDeviation()`. You should not start on this function until your `calculateMean()` function is completely working, as you are required to reuse your first function in the implementation of this function. You should start like before, by first making sure your program compiles and runs when you uncomment the unit tests after adding the function prototype and a stub that returns a value of 0.0. This function takes the same two parameters, the size of the array of values ($n$) and an array of integers. This function will also return a double result. The function will calculate and return the standard deviation of any array of integers of any size passed to it. Some further requirements of this function. You must demonstrate the use of predefined functions from `<cmath>` such as the `pow()` and `sqrt()` functions. Also, you must call the `calculateMean()` function from this function, e.g. do not repeat the calculation of the mean value, but reuse your previous function to do this work to obtain $\bar{x}$.

## Example Output

Here is the correct output you get from your program if you correctly implement the two functions and successfully pass all of the unit tests given for this assignment. If you invoke your function with no command line arguments, only failing tests are usually shown by default. In the second example, we use the `-s` command line option to have the unit test framework show both successful and failing tests, and thus we get reports of all of the successfully passing tests as well on the output.

```
$ ./test
===============================================================================
All tests passed (24 assertions in 2 test cases)



# using the -s option to show successful check/assertions
$ ./test -s

-------------------------------------------------------------------------------
assg01 is a Catch v2.7.2 host application.
Run with -? for options

-------------------------------------------------------------------------------
<calculateMean()> function tests
  test array of 1 item
-------------------------------------------------------------------------------
assg01-tests.cpp:35
.......................................................................

assg01-tests.cpp:38: PASSED:
  CHECK( calculateMean(1, x) == Approx(3.0) )
with expansion:
  3.0 == Approx( 3.0 )

... output snipped ...

===============================================================================
All tests passed (24 assertions in 2 test cases)
```

# Assignment Submission

A MyLeoOnline submission folder has been created for this assignment. There is a target named `submit` that will create a tared and gziped file named `assg01.tar.gz`. You should do a `make submit` when finished and upload your resulting gzip file to the MyLeoOnline Submission folder for this assignment.

```
$ make submit
tar cvfz assg01.tar.gz assg01-tests.cpp assg01-main.cpp assg01-functions.hpp assg01-functions.cpp
assg01-tests.cpp
assg01-main.cpp
assg01-functions.hpp
assg01-functions.cpp
```

# Requirements and Grading Rubrics

## Program Execution, Output and Functional Requirements

1. Your program must compile, run and produce some sort of output to be graded. 0 if not satisfied.
2. 40 pts. for correctly implementing the `calculateMean()` function.
3. 50 pts. for correctly implementing the `calculateStandardDeviation()` function.
4. 5 pts. for correctly demonstrating functions as required.
5. 5 pts. for showing reuse of function to calculate mean.

## Program Style

Your programs must conform to the style and formatting guidelines given for this class. The following is a list of the guidelines that are required for the assignment to be submitted this week.

1. Most importantly, make sure you figure out how to set your indentation settings correctly. All programs must use 2 spaces for all indentation levels, and all indentation levels must be correctly indented. Also all tabs must be removed from files, and only 2 spaces used for indentation.
2. A function header must be present for all functions you define. You must give a short description of the function, and document all of the input parameters to the function, as well as the return value and data type of the function if it returns a value.
3. Do not include any statements (such as `system("pause")` or inputting a key from the user to continue) that are meant to keep the terminal from going away. Do not include any code that is specific to a single operating system, such as the `system("pause")` which is Microsoft Windows specific.