Assignment 02: Data Types, Variables, Relational and Logical Operators, Selection Control Structures

COSC 1437: Programming Fundamentals II

Objectives

- Write some C/C++ code in a real program, using VSCode IDE
- Learn how to declare variables of different types and use them in programs
- Practice using arithmetic operators to perform calculations in our programs
- Practice declaring variables and using arithmetic operators together in programs
- Use some basic relational and logical operators to make some decisions.
- Create a basic if / else statement control structure using relational operators
- Create a switch multi-select statement.

Description

In this lab/assignment, you goal is to get more familiar with how assignments are done and graded for this class. You will be working on using basic C/C++ operators. You will write several expressions to do things like calculate the volume of objects, so that we can have you define some variables and use arithmetic operators to perform calculations.

You will also perform some tasks that will require you to use relational and logical operators. We will create some expressions in functions that will perform tasks conditionally, based on input values given.

And finally you will use a multi-selection switch statement to conditionally execute some task(s) based on input to a function.

Overview and Setup

For all assignments, it will be assumed that you have a working VSCode IDE with remote Dev Containers extension installed, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this assignment you have been given the following files (among some others):

File Name	Description
`src/assg02-tests.cpp`	Unit tests for the three tasks that you need to successfully pass
`src/assg02-library.cpp`	File that contains the code, all your work will be done here
`include/assg02-library.hpp`	Header include file of function prototypes

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub by accepting the assignment using the provided assignment invitation link for ‘Assignment 02’ for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure that you open the cloned folder and restart inside of the correct Dev Container.

3. Confirm that the project builds and runs, though no tests may be defined or run initially for some assignments. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Assignment Tasks

Lab Task 1 / In-Lab Work : Calculate the Volume of a Football

For the first task, you will be finishing the implementation of some code to calculate the volume of a football (technically known as an oblate-spheroid). The volume of a football can be calculated using the following formula What is the volume of a football:

$$v = \frac{4}{3}\pi a^2 b \tag{1}$$

Here a represents half of the length of the short axis (in other words the radius of the short axis). Likewise b represents the radius of the long axis.

In the file name `assg02-tests.cpp` you will find all of the tests you need to pass for this task. Start by changing the `#undef` for `task1` to be a `#define`. This will enable the test cases for the tests where the area of a football is calculated. If you enable these, your code should still compile and run. However, if you look at the result from running the tests, you should find that not all of these tests are passing.

Just to help you know what you are looking at here. The first test in this file looks like this:

```
REQUIRE_THAT(volumeOfFootball(0.0, 0.0), Catch::WithinAbs(0.0, 0.00001));
```

What this does is call a function named `volumeOfFootball()`. We have not yet learned about C/C++ functions, you will learn how to write your own functions later in the class. But what this is saying is that, I want the function named `volumeOfFootball()` to calculate the area of a football where both the short axis radius and long axis radius are 0.0. The short axis is passed in as the first value, and the long axis as the second value between the `()` after the function name.

The volume of any sphere or oblate-spheroid where an axis has a radius of 0 is of course 0. So the `WithinAbs()` is just testing that the result that is returned from this calculation is 0.0 (within an absolute error 0f 0.00001 here).

Let's implement the code to calculate this volume. Open the file named `assg02-library.cpp` and find the function there named `volumeOfFootball()`. The function defines a constant named `PI` that you will use. Notice how you define a named constant in C++ here. The function given to you only has the following statement other than defining `PI`:

```
return PI;
```

This function basically always returns an answer of 3.1415926... no matter what the `shortAxis` and `longAxis` are that are passed into the function. Try changing this to be

```
return 0.0;
```

What do you predict should happen when you save, recompile and run the tests now? You should find that now some of the tests pass. 3 of the tests pass in 0.0 for one or both of the axis, so the result should be 0.0.

But we want this function to pass all of the tests, and to correctly calculate the volume of a football given a short and long axis radius. Do the following to successfully complete this Lab/Task and pass all of the tests for the first task.

1. Define a variable of type `double` named `volume`.
2. Use the formula given above to calculate the volume of the football, given whatever is passed in as the `shortAxis` and the `longAxis`
   - Be careful about integer division (read our tutorial materials about this). Try using `4.0 / 3.0` instead of `4 / 3`.
   - You can use `shortAxis * shortAxis` to square (raise to second power) the `shortAxis`. We will learn of a better way to do this later when we talk about the C library functions.
   - Assign the result of this calculation to your variable named `volume`.

3. Change the function so that instead of being hardcoded to return PI or 0.0 it returns your calculated value, which should be in the variable of type `double` named `volume` that you created.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 1 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Task 2: Roots of Quadratic Equation

In the second task you will continue to practice declaring variables and performing arithmetic calculations using arithmetic operators. We will also use relational operators and the `if / else if / else` control structure in this task.

The roots of a quadratic equation of the form $ax^2 + bx + c = 0, a \neq 0$ are given by the following formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

In this formula, the term $b^2 - 4ac$ is called the **discriminant**. If $b^2 - 4ac = 0$ then the equation has a single (repeated) root. if $b^2 - 4ac > 0$ then the equation has two real roots. And finally if $b^2 - 4ac < 0$, the equation has 2 complex roots.

Start by `#define` the task2 tests in `assg02-tests.cpp`. You might want to examine the tests again before continuing. We are testing code you will implement in a function called `determineQuadraticRoots()`, this function returns -1 if there are complex roots (based on the determinant), 1 if there is a single repeated root, and 2 if there are 2 distinct real roots.

In `assg02-library.cpp` the function named `determineQuadriticRoots()` always returns 0 when you begin this assignment. The values `a`, `b` and `c` of a quadratic equation of the form shown above are passed in again as `double` parameters to this function. However notice that this function returns an `int` type result.

Do the following.

1. Create a variable of type `double` named `discriminant` in this function.
2. Calculate the discriminant as shown using $b^2 - 4ac$ and save the result in the variable you declared.
   - You can again use `b * b` to square the value of `b`.
3. Create an `if / else if / else` control statement. You will need to use relational operator(s) here. Perform the tests exactly as described here,
   - First test if the discriminant is greater than 0. If it is then return 2 (to indicate 2 real roots)
   - else test if the discriminant is less than 0. If it is then return -1 to indicate complex roots.
   - else if the discriminant was not less than or greater than 0 it must be 0, so return 1 to indicate a single repeated root.

Once you successfully complete this function, save, compile and run the tests. You might want to consider testing incrementally, for example just test first that you pass the tests were it should return 2 to indicate 2 real roots. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Task 3: Switch Statements

For task 3 we will practice using the `switch` selection control structure in C/C++. Again start by `#define` the `task3` unit tests in the `assg02-tests.cpp` file. If you look at the tests, you should find that we are testing a function named `performOperation()` that takes 3 values as inputs. For example the first test in this file is:

```
CHECK_THAT(performOperation(3.3, 4.4, '+'), Catch::WithinAbs(7.7, 0.00001));
```

The `performOperation()` function takes two double values as its first two inputs. Then it takes a value of type `char` as the third parameter. Based on the operation passed in as the third parameter, this function is supposed to perform the indicated operation (addition or '+' in this case), and then return the result as a double. So this test expects a value of 7.7 to be returned as a result of performing the addition on the two inputs.

Find the `performOperation()` function in the `assg02-library.cpp` file. Initially the function given to you always just returns `-42.0`. Do the following for this task:

1. Define a variable of data type `double` named `result`. You will calculate the operation and put the result into this variable to be returned.
2. Create a `switch` statement that switches on the `operation` character that is passed in as the third parameter to this function.
3. Handle the cases for '+' to do addition, '-' to do subtraction, '*' to do multiplication and '/' to do division. In all of these cases, perform the indicated operation and save the calculated value in the `result` variable that you declared.
4. You also need to add in a `default` case for your `switch` statement. The default is to return a result of 0.0 if an invalid/unknown operation is asked for.
5. Don't forget that you need to return the `result` you calculate. So you need to modify the `return` statement of the function given to pass your tests.

Once you successfully complete this function, save, compile and run the tests. If you code the switch statement correctly, you should be able to pass all of the tests for the third task, including those asking for some operation other than '+', '-', '*' or '/'. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this assignment, up to 50 points will be given for having completed at least the initial Task 1 / In Lab task. Thereafter 25 additional points are given by the autograder for completing task 2 and task 3 respectively, for a total possible 100 points on the assignment.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull request comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop

index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).

4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax