Assignment 05: Predefined Functions and Defining Your Own Functions

COSC 1437: Programming Fundamentals II

Objectives

- Practice using predefined functions in the C and C++ libraries.
- Create simple functions that have one or more input parameters in them.
- Understand concept of passing in parameters by value.
- Learn how to return results using the return values from a function.
- Practice more mathematical operations and conditional control structures.

Description

In this assignment you will practice using some predefined functions from the C library, and you will write some simple functions of your own.

You have actually been using functions in our previous assignments. Each task has been defined as a function. Up until now you were given the signature of the function in the `assgXX-library.hpp` header file and its declaration in the `assgXX-library.cpp` file. You only had to write the code asked for inside of the curly braces for the function implementation so that it could be tested.

In this assignment you will have to write several functions. This time, for some of the tasks, you will not be given the signature and declaration in the header and implementation file. So you will have to figure out and write the appropriate function signature, including the function name, input parameters and return values, as well as add the declaration into the implementation file.

But first you will demonstrate reusing predefined functions from the C standard library.

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/assg05-tests.cpp`	Unit tests for the tasks that you need to successfully pass
`src/assg05-library.cpp`	File that contains the code implementations, all your work will be done here
`include/assg05-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 03' 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 : Volume of a Football revisited

For the first task, we will revisit and rewrite a function that you did in a previous assignment. In one of our first assignment, you had to calculate the volume of a football, given the short and long axis. In the `assg05-library.cpp` file you will see that we have a `volumeOfFootball()` function again. You should maybe now better understand the function declaration here and in the header file. This is a user defined function that takes two `double` parameters as input, the `shortAxis` and the `longAxis` parameters. These parameters are all passed by value.

For reference, here once again is the formula to calculate the volume of a football given the short axis `a` and long axis `b` radiuses:

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

In this version of the function, you are going to recreate the work to calculate the volume of the indicated football. But this time you will be required to reuse some functions and constants from the `cmath` C standard library. The `cmath` library defines functions for doing math, like sines, absolute value, etc. There is a function named `pow()` for raising some value to a power. In addition some constants are defined in the `cmath` library, including one named `M_PI` which contains the approximate value of π to 16 or so digits.

You should reimplement the expression to calculate the volume of a football with the given short and long axis values. But this time you are required to use the π constant from the `cmath` library. And instead of multiplying the short axis by itself to square it, you are required to use the `pow()` function from the library. In order to use these things from the `cmath` library, you need to add a `#include` statement at the top of the code in `assg05-library.cpp` so that this librarie's contents are included and available for you to use.

To complete the work for this task, perform the following steps:

1. First add in the needed `#include` statement at the top of the `assg05-library.cpp` file, so that you can reuse the functions and constants defined in that C standard library in your code.
2. As before, you should create a local variable named `volume` of type `double` used to hold the calculation of the volume and return it from this function.
3. Perform the calculation to calculate the volume.
    - You are required to use the `pow()` function from the `cmath` library to calculate the square of the `shortAxis` in this implementation.
    - You are required to use the `M_PI` constant declared in the `cmath` library for the value of π (instead of declaring your own constant for π).
4. After reading this weeks materials and tutorials, you should better understand the purpose of the `return` statement. This and all of the functions in this assignment are value returning functions. So you need to return the correctly calculated `volume` of the football as the last step in this function.

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. Though make sure you write the function as described, reusing the `pow()` and `M_PI` declarations from the `cmath` library. 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: Determine if a Character is a Digit or Not

A quick review, we haven't used the `char` data type much in this class yet. The `char` data type holds a single (ASCII encoded) character in C/C++. You can declare a variable of type character by doing something like:

```
char charToTest;
charToTest = '0';
```

Here we declared a variable of type `char` named `charToTest`. Below that we initialized the value of the character to be '0'. Notice that we didn't do

```
charToTest = 0;
```

This is incorrect. In the second initialization, the 0 is an integer constant value not a character. You need to use single quotes around a character for C/C++ to see it as a character constant. So in the first initialization we initialized `charToTest` to the character value of '0', a digit.

In task 2 and 3 you will write two versions of the following function. To begin with, you need to implement a function named `myIsdigit()`, which has already been declared for you again in the header and implementation file. This function takes a single `char` as an input parameter. The purpose of this function is it should return `true` if the character it is asked to test is a digit. That is to say if it is one of the set ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9') then the function should return `true`. But if it is not a digit character it should return false. You need to use selection control statement. You can use an `if` statement to test. You would need either 10 separate `if/ else if/ else` conditions, or you would need a complex boolean expression to get this to work. You could also use a `switch` and use the fact that case statements fall through if you don't do a `break`, to maybe implement this a bit more easily. You need to be careful when testing the character. For example

```
if (charToTest == 0)
```

does not do what you might think. Again this is because here we are testing integer 0 and not the character '0' in this selection statement.

However you decide to do it, your implementation needs to return `true` if it receives any of the 10 digit characters, and `false` if it receives any non digit character.

Do the following steps for this task:

1. Use some form of selection control statement to determine if the `charToTest` is a digit or not.
2. If it is one of the 10 digit characters, return a `true` boolean result.
3. If it is not a digit, return `false`.

When finished, save compile and rerun the tests. Once you are satisfied, make and push your commit of Task 2 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Task 3: Reuse `isdigit()` from the C standard library

This is really a continuation of task 2, but here you are going to reuse the `isdigit()` function from the C standard library to instead implement the algorithm by hand.

There is actually a function in the `cctype` library called `isdigit()` that performs exactly the task that you have just implemented. So for this task, lets practice some more code reuse. You need to put your code into the function named `clibIsdigit()` for this part of the task. This function again takes a character to be tested, and returns the same result of `true` if it is a digit character, and `false` if it is not.

But this time, you are required to reuse the `isdigit()` function from the C `cctype` standard library, instead of writing your own version. You will need to use the appropriate `#include` statement again to reuse this function. Perform the following steps for this part of task 2:

1. Make sure you `#include` the library you need in order to reuse the `isdigit()` function from it.
2. Call the `isdigit()` library function on the character to be tested, passing it the parameter you received in your function.
3. Return the result that the `isdigit()` returns as the boolean result for your `clibIsdigit()` function. Technically the `isdigit()` function from the library returns an `int` value. But it returns 1 if the character was a digit, to indicate `true`, and 0 to indicate `false`. Originally C did not have an actual `bool` type. But 0 / 1 will get automatically converted (or type cast) into boolean `false` / `true`. So you can just return whatever the call to `isdigit()` returns and it will be converted to the correct `bool` type.

When finished, save, compile and rerun the tests. Once you are satisfied, make and push your commit of Task 3 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Task 4: User Defined Absolute Value Function

For this task (and probably for most tasks starting from this point in the class), you have not been giving the function signature in `assg05-library.hpp` nor a stub implementation of the function in `assg05-library.cpp`. So you will first have to add the signature and the function implementation before you can implement it in this task.

Start by declaring a function named `myabs()` in the `assg05-library.hpp` header file. This function should take an `int` as its input parameter, and it will return an `int` result. This will be the first time you have added a function signature to the header file. Make sure you end the signature in the header file with a `;`.

Then you should create a stub implementation in `assg05-library.cpp` that compiles and runs. The function implementation should go immediately below the function documentation provided for the `myabs()` function in `assg05-library.cpp`. It should have exactly the same signature as what you just put into the header file, but instead of ending in a `;`, you should end with `{` and `}` curly braces. The implementation code for this function goes in between the curly braces in the `assg05-library.cpp` implementation file.

Since this is a value returning function, you should go ahead and add a

```
return 0;
```

inside of the curly braces. Then define the `task4` tests in `assg05-tests.cpp` and try and compile and run the tests. By returning 0, you have hardcoded the function to just return a result of 0 whenever it is called. This should be enough to allow you to compile the code and run the tests. But since your function just returns 0, it will fail most all of the tests for task 3.

To complete this function do the following steps:

1. You first need to add in the correct signature described into the `assg05-library.hpp` header file. Don't forget the ending `;`
2. And you need to add in a declaration of the function in the `assg05-library.cpp` implementation file. You should start with a stub function that returns `0.0`, and ensure it compiles and that if you enable the `task4` tests, the tests run (but fail).
3. Hopefully the implementation is relatively obvious to you at this point in the course. You need a selection statement that tests if the input value parameter is negative (e.g. less than 0). If the input parameter is negative, you want to negate, or alternatively return the negation of the value.
4. If the value was positive (or 0) you don't have to do anything, just return the original value.
5. You will need 1 or possible 2 separate return statements to return the absolute value you calculate, depending on how you write your implementation.

There are several ways to accomplish this, but most likely the easiest is to just use an `if` or maybe `if / else` selection statement.

When finished, save compile and rerun the tests. Once you are satisfied, make and push your commit of Task 4 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Task 5: User Defined Fahrenheit to Celsius Function

For more practice, lets declare and implement one more function. Again the function signature and declaration has not been given to you for this function. You need to start by adding in a signature into the '.hpp' header file, and an empty function in the `.cpp` implementation file. As suggested before, it would be a good idea to first create an implementation that returns 0.0, and ensure that your code then compiles and runs the tests if you enable `task5` unit tests.

In this task you need to code a function that will take a `double` value as its input parameter. This parameter will hold a temperature in degrees Fahrenheit that we want to convert to degrees Celsius and then return that as the result of the function. So the function will be a value returning function that returns a `double` result. The name of the function must match what is being tested, so you must call this function `fahrenheitToCelsius()` in the header and implementation file.

Once your stub function compiles and runs, you need to perform a calculation that converts a temperature in Fahrenheit to the corresponding temperature in Celsius. The formula to perform this conversion is:

$$C = (F - 32.0)\frac{5}{9}$$

In other words, subtract 32 from the temperature in degrees Fahrenheit (your input parameter to this function) and then multiply by $\frac{5}{9}$. Be careful that you are not performing integer division here, you may want to use `5.0` and `9.0` in your expression to ensure that the C/C++ compiler treats these values as `double` values.

So to summarize, perform the following steps for this task:

1. You first need to add in the correct signature described into the `assg05-library.hpp` header file. Don't forget the ending `;`
2. And you need to add in a declaration of the function in the `assg05-library.cpp` implementation file. You should start with a stub function that returns `0.0`, and ensure it compiles and that if you enable the `task4` tests, the tests run (but fail).
3. Use a local variable named something like `degreesCelsius`. This should be of type `double`. You will store the result of the calculation in this variable and return it.
4. Perform the arithmetic operations to convert the input parameter in degrees Fahrenheit to degrees Celsius. Save the result in your local variable.
5. Add or modify your stub return statement to return the calculated temperature in degrees Celsius.

When finished, save compile and rerun the tests. Once you are satisfied, make and push your commit of Task 3 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Task 6: Extra Credit, Do the Inverse

For 5 points of extra credit, create a function that converts degrees in Celsius to degrees in Fahrenheit. The arithmetic calculation is similar to the one given previously, but will be the inverse (you can google and find an expression to do this if not sure how to invert the function).

There are tests for `task6`, so if you want the extra credit, make sure your function is named as expected, and that it passes these tests given in `task6`.

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 tasks 2 and 3 successfully.

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