Assignment 11: Pointers and Dynamic Memory Allocation

COSC 1437: Programming Fundamentals II

Objectives

- Practice declaring and using pointer variables
- Learn to use the dereference and address of operators
- Learn about dynamically allocating memory, and dynamically allocating arrays of memory
- Pass in pointers to functions and use then in functions for operations.
- More practice with processing arrays

Description

In this assignment you will write several functions as usual that should give you some practice using pointer variables, and dynamically allocating memory. We will also continue to get more practice in understanding and using arrays in this assignment. The last 2 tasks require you to dynamically allocate an array of integer values and return the resulting array from the functions, so you may want to review our class materials on dynamically allocating arrays in order to better understand those tasks.

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/assg11-tests.cpp`	Unit tests for the tasks that you need to successfully pass
`src/assg11-library.cpp`	File that contains the code implementations, all your work will be done here
`include/assg11-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 11' 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 : Use pointer variables

The in-lab task will give you a little bit of practice in using (dereferencing) pointer variables. In this task, you need to write a function named `toMetersAndFeetInches()`. This function needs 3 parameters as input. The first parameter should be a regular `int` type called `totalInches`. But the second and third parameters should be passed in as pointers to integers (`int*` data type), and named `feet` and `inches` respectively.

When we pass in a pointer to some variable as a parameter, the parameter will work basically the same as a reference parameter that you are familiar with from previous assignments. In fact the idea of a reference parameter was added to the C language later, so originally passing in a pointer was how reference parameters were done. There is one difference though. When you pass in a pointer as a parameter, it is a pointer data type. So to actually use the memory it points to, you have to dereference the pointer, as you read about in this weeks class materials.

The `toMetersAndFeetInches()` function should return a `double` result from it. This function actually performs two tasks. First of all, it should break down the `totalInches` into `feet` and `inches`, and return these using the pointer parameters to the function. For example, if we call the function with `totalInches = 28` then since there are 12 inches in a foot, you should return `feet = 2` and `inches = 4` from this function. HINT: use integer division to determine the whole number of feet from the total inches, and you can use the modulus operator `%` to determine the remainder number of inches.

After the whole number of `feet` are determined, this function also determines how many meters are equivalent to the whole number of `feet` that were passed in as the first parameter. You can use the formula

$$\text{meters} = \text{feet} * 0.3048$$

to convert the whole number of feet into meters. This `double` result should be returned as the result from this function. You might find it useful to read the tests for task 1 to get some examples of what this function is expected to do.

As a reminder, make sure that

1. You are passing in pointers to integers as the second and third parameters named `feet` and `inches` respectively.
2. That you are correctly using the pointer parameters, you need to use the dereference operator since these are pointer data types in order to access their values.
3. Assigning a value to the memory pointed to by the passed in pointer will work in a similar way as a reference parameter. We expect that upon return from this function the caller will see the expected calculated values for `feet` and `inches`.

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: Implement `growArray()` Function

As usual, create the function prototype and a stub, and uncomment the tests for task 2, to make sure your project still compiles and can now run the unit tests that you need to pass for this task 2.

In the second and third tasks, you will need to dynamically allocate arrays, and return the pointer to the base of these arrays as the result from the function you will implement.

In task 2 you need to implement a function called `growArray()`. This function takes the `numValues` and an array of integer `values` as input parameters, similar to previous assignments where we practiced passing arrays into functions. You will return an `int*` from this function, which will be a pointer to the base address of a new array of integers that is created dynamically by this function.

The purpose of this function is to dynamically create a new array that is twice as big as the input array, and copy and duplicate the values from the input array into the new array that is returned. For example if we have

```
int x[] = {1, 2, 3};
int* newValues = growArray(3, x);
```

Then the returned pointer to the `newValues` array should be an array of size 6 that has the values `{1, 1, 2, 2, 3, 3}` in indexes 0 through 5 of the array.

There is a bit of a tricky calculation/algorithm you need in order to successfully copy the values from the input array into the correct indexes. There are several ways to do this. The easiest solution might be to iterate through the indexes of the input array as normal. For example, given the array `x` above and `numValues` of 3, then a simple for loop that iterates `i=0, 1, 2` can be used to read out the values from the input array. But then you need to write those values into your new array correctly. Index 0 from the input array needs to go to indexes 0 and 1 for the new double sized array. Index 1 needs to be written to indexes 2 and 3, etc. There is an easy relationship between these indexes, e.g. the indexes in the new array will be at `2 * i` and `2 * i + 1` where `i` is the index of a value in the input array.

So the suggested algorithm for `growArray()` is as follows:

1. Dynamically allocate an array of integer values. This array should be twice as large as the `numValues`, the size of the input array. You are required to use the C++ `new` keyword for dynamic allocation in all assignments for this class (do not use old `malloc` / `calloc` C functions).
2. Iterate over the indexes from 0 to `numValues-1` of the input array.
   - Copy over the value from the input to the correct 2 locations of the new array that you dynamically allocated.
3. This function should return an `int*`, e.g. a pointer data type that points to an `int`. The base address of the array you allocate in step 1 is the `int*` that you need to return.

When you are satisfied your function works (make sure your project compiles and runs still), you should commit your changes and push them to the `Feedback` pull request in your classroom repository.

Task 3: Implement a `subArray()` Function

Your third task in this assignment is similar to the previous task. Write a function named `subArray()`. This function takes 3 input parameters. `start`, `length` should be the first 2 parameters, and they will indicate a start index and the number of values that will be copied. The third parameter will be an array of integer `values` as in the previous function.

This function will return the same result as the previous, an `int*`, which will be a pointer to a new dynamically array of values that is created by this function.

The purpose of this function is to dynamically create a new array and to copy a sub portion of the input array into the newly created array that will be returned. The `start` parameter indicates the starting index for the copy, and the `length` parameter indicates the number of values to be copied into the new sub array.

For example, in the following code:

```
int x[] = {0, 1, 2, 3};
int* newValues;

newValues = subArray(1, 2, x);
```

your function should dynamically create a new array of size 2 and return it.
The values at indexes 1 and 2 will be copied into this new array that is returned, so you should find that

```
newValues[0] == 1;
newValues[1] == 2;
```

So as before you do have to be careful in writing your loop to correctly copy the values from the input array into the correctly location of the new array you will dynamically allocate. So the suggested algorithm for `growArray()` is as follows:

1. Dynamically allocate an array of integer values. The new array should be `length` in size as it will hold that many values copied from the input array. You are required to use the C++ `new` keyword for dynamic allocation in all assignments for this class (do not use old `malloc` / `calloc` C functions).

2. This time is probably best to write a for loop that iterates over the indexes 0 to `length - 1`, which will be the indexes of the array you are going to return.
   - You need to copy from the input starting from the `start` index, so basically if you use `start + i` where `i` is the index into the array you are copying into, you will calculate the correct index to copy from the input array.
3. This function should return an `int*`, e.g. a pointer data type that points to an `int`. The base address of the array you allocate in step 1 is the `int*` that you need to return.

When you are satisfied your function is working, the project still compiles, and you can run and pass the tests, perform the usual to create and push a commit to the `Feedback` pull request.

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