Lab 5 – Functions with Array and Pointers

General Requirements

For those students who already know C or C++: do not use structs. They are not necessary for this lab.

For exercise 1, your functions must not be recursive. Repeated actions must be implemented using C's `while`, `for` or `do-while` loop structures. For exercises2 and 3, you **must** use recursion.

None of the functions you write should perform console input (i.e., `scanf` statements are forbidden). None of your functions should produce console output (i.e., `printf` statements are forbidden).

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). Instructions were given in the VS Code installation instructions.

Finish each exercise (i.e., write the function and verify that it passes all its tests) before you move on to the next one. Do not leave testing until after you have written all the functions.

Getting Started

Step 1: Depending on your operating system, download (**lab5_Windos.zip** or **lab5_macOS.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Double click on Lab5.code-workspace

Step 3: lab5_xxx folder contains the following files:

- array_pointers.h contains the declarations (function prototypes) for the functions you will implement. **Do not modify array_pointers.h.**
- array_pointers.c contains a few complete functions, and incomplete definitions of the functions you have to design and code. **Do not modify the complete functions.**
- main.c will call the functions in array_pointers. Note that we are not using sput for this lab, because we are interested in the code and syntax within the functions. Passing a set of test functions would not be sufficient for correct solutions to this lab. **Do not modify `main.c`.**

Step 4: Run the project. It should build without any compilation or linking errors.

Step 5: Execute the project. For the functions where the test harness is provided (the functions in main.c) will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions.

Step 6: Open array_pointers.c in the editor. Solve exercises 1, and 3. Remember, you must not make any changes to main.c or array_pointers.h. All the code you write must be in array_pointers.c.

Exercise 1

File array_pointers.c contains a complete implementation of find_max_v0 which finds the largest value in the first n elements of an array of integers, using the familiar index notation, i.e. ar[i].

Your first task is to write find_max_v1 which does exactly the same thing as find_max_v0 except that find_max_v1 uses pointers to access the array elements, i.e. *(arp+i).

Your second task is to write find_max_v2 which does exactly the same thing as find_max_v0 and find_max_v1 except that find_max_v2 uses a walking pointer to access the array elements, e.g. p moves through the array such that *p is always the contents of the current element.

Check that the output for exercise 1 is correct. In other words, check that the output for find_max_v1 and find_max_v2 is now the same as for find_max_v0 when you run the project. In addition, double-check that you have followed the instructions above and in the function descriptions.

Exercise 2

File array_pointers.c contains an incomplete definition of a function named `count_in_array`. The function prototype is:

```
int count_in_array(int a[], int n, int target);
```

This function counts the number of integers in the first `n` elements of array `a` that are equal to `target`, and returns that count. For example, if array `arr` contains the 11 integers 1, 2, 4, 4, 5, 6, 4, 7, 8, 9 and 12, then `count_in_array(arr, 11, 4)` returns 3 because 4 occurs three times in `arr`.

Implement `count_in_array` as a recursive function. Your `count_in_array` function cannot have any loops.

Hint: Review the recursive `sum_array` function that was presented during the lectures. Before writing any code. formulate a recursive definition of the solution to the problem, "What is the number of integers in the first `n` elements of array `a` that are equal to `target`?" Then, convert that definition to C code. We recommend that you use an iterative, incremental approach when implementing the function. For example, during the first iteration, write just enough code to handle the base case(s). Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code recursive case(s).

Read the definitions of `test_exercise_2` and `test_count_in_array` in main.c. Function `test_exercise_2` has six test cases for the `count_in_array` function. It calls `test_count_in_array` six times, once for each test case. Notice that `test_count_in_array` has four arguments: the three arguments that will be passed to `count_in_array`, and the value that a correct implementation of `count_in_array` will return.

Build and execute the project. Use the console output to help you identify and correct any flaws. Verify that `count_in_array` passes all of its tests before you start Exercise 3.

Exercise 3

File array_pointers.c contains an incomplete definition of a function named array_compare. The function prototype/signature is:

```
_Bool array_compare(int arr1[], int arr2[], int n);
```

The function array_compare, recursively compares two arrays to determine if the first n elements of array arr1 and arr2 are the same. The arrays are compared element by element, starting from the first element and progressing to the last element. The function should return true if the n first elements of the arrays are equal and false otherwise. For example if arr1 contains the 5 integers 1, 2, 3, 4, 5 and arr2 contains the 5 integers 1, 2, 3, 4, 5, then `array_compare(arr1, arr2, 5)` returns `true` since both arrays are identical. If arr1 contains the 5 integers 1, 2, 3, 4, 5 and arr2 contains the 5 integers 1, 2, 3, 4, 6, then `array_compare(arr1, arr2, 5)` returns false as the last element in arr1 is not the same as in arr2. If arr1 contains the 5 integers 1, 2, 3, 4, 5 and arr2 contains the 5 integers 1, 2, 3, 4, 6, then `array_compare(arr1, arr2, 4)` returns true as four fist elements in arr1 are the same as in arr2. You may assume that both `arr1` and `arr2` will have at least n elements.

Implement `array_compare` as a recursive function. Your `array_compare` function cannot have any loops.

Read the definitions of `test_exercise_3` and `test_array_compare` in main.c. Function `test_exercise_3` has four test cases for the `array_compare` function. It calls `test_array_compare` five times, once for each test case. Notice that `test_array_compare` has four arguments: the three arguments that will be passed to `test_array_compare`, and the value that a correct implementation of `array_compare` will return.

Build and execute the project. Use the console output to help you identify and correct any flaws.

Wrap-up

Submit array_pointers.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that array_pointers.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the deadline.