Lab 4 – C Arrays & Local and Global Variables

Lab 4 – Part I: C Arrays

Prerequisite Reading

C Arrays and Python Lists (provided in Brightspace).

General Requirements

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

Your functions must not be recursive. Repeated actions must be implemented using C's `while`, `for` or `do-while` loop structures.

None of the functions you write should perform console input; for example, contain `scanf` statements. None of your functions should produce console output; for example, contain `printf` statements.

Your functions must not declare local variables that are arrays; in other words, they must not have declarations similar to:

```
int temp[n];
```

Use the indexing (`[]`) operator to access array elements. Do not use pointers and pointer arithmetic. This means your functions should not contain statements of the form `*ptr = ...` or `*(ptr + i) = ...`, where `ptr` is a pointer to an element in an array.

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 (**lab4_part1_win.zip** or **lab4_part1_macOS.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Open the unzipped folder in VS Code

Step 3: lab4_part1_xxx folder contains the following files:

- arrays.h contains the declarations (function prototypes) for the functions you will implement. **Do not modify arrays.h.**
- arrays.c contains incomplete definitions of five functions you have to design and code.
- main.c and sput.h implement a *test harness* (functions that will test your code, and a `main` function that calls these test functions). **Do not modify `main` or any of the test functions.**

Step 5: Run the project. It should build without any compilation or linking errors, but you will see that all tests fail.

Step 6: Read this step carefully. To use the test harness, you need to understand the output it displays.

Double click the icons for main.c and arrays.c to open these files in editor windows.

File main.c contains five *test suites*, one for each of the functions you will write in Exercises 1-5. In Exercise 1, you will complete the implementation of a function named `avg_magnitude`. The test suite for this function is named "Exercise 1: avg_magnitude()". This test suite has one *test function*, named `test_avg_magnitude`. Here is the code for this function:

```
static void test_avg_magnitude(void)
{
    double samples[] = { 5.7, 2.3, -1.9, 4.5, 6.2, -8.1, 9.7, 3.1 };

    sput_fail_unless(fabs(avg_magnitude(samples, 8) - 5.19) < 0.01,
           "avg_magnitude({5.7, 2.3, -1.9, 4.5, 6.2, -8.1, 9.7, 3.1}, 8)");
    printf("Expected result: 5.19 (approximately), actual result:
           %.2f\n", avg_magnitude(samples, 8));
}
```

This function checks if `avg_magnitude` calculates the average magnitude of array `samples`, which contains eight `double`s. The check is performed by `sput_fail_unless`, which has two arguments. The first argument is the condition that must be `true` in order for the test to pass. The second argument is a descriptive string that is displayed when `sput_fail_unless` is executed.

The condition that determines if `avg_magnitude` returns the correct value (`5.19`, approximately) may appear a bit strange:

```
fabs(avg_magnitude(samples, 8) - 5.19) < 0.01
```

Because of the way floating-point numbers are represented in a computer, we should never use the `==` operator to compare them for equality. Two floating-point numbers are considered to be equal if they differ by a small amount. So, we subtract 5.19 (the expected result) from the value returned by `avg_magnitude`, and call `fabs` to obtain the absolute value of this difference. If this value is small (less than 0.01), we consider the value returned by `avg_magnitude` to be close enough to 5.19, and the test passes.

Execute the project.

The test harness will report errors as it runs, which is what we would expect, because you have not started working on the functions the harness tests. Once you start working on your lab and you correct the errors, all test harnesses should pass.

Step 7: Read C Arrays and Python Lists.pdf (Located on Brightspace).

Step 8: Do Exercises 1 through 5. Remember, you must not make any changes to main.c, arrays.h or sput.h. All the code you write must be in arrays.c.

Exercise 1

A sound (for example, a note played on a guitar or a spoken word) is recorded by using a microphone to convert the acoustical signal into an electrical signal. The electrical signal can be converted into a list of numbers that represent the amplitudes of *samples* of the electrical signal measured at equal time intervals. If we have *n* samples, we refer to the samples as x_0, x_1, x_2, … , x_{n-1}.

The *average magnitude*, or average absolute value, of a signal is given by the formula:

$$\text{average magnitude} = (|x_0| + |x_1| + |x_2| + \dots + |x_{n-1}|)/n = \sum |x_k|/n; \quad k = 0, 1, 2, \dots, n-1$$

An incomplete implementation of a function named `avg_magnitude` is provided in `arrays.c`. The function prototype is:

```
double avg_magnitude(double x[], int n);
```

This function returns the average magnitude of the signal represented by an array of `double`s containing *n* elements.

Finish the definition of this function. Your function should assume that `n` is positive; i.e., **it should not verify that `n` is > 0 before calculating the average magnitude of the first `n` array elements**.

C`s math library (`math.h`) contains a function that calculates the absolute values of real numbers. The function prototype is:

```
// Return the absolute value of x.
double fabs(double x);
```

Use the console output printed by the test harness to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #1 before you start Exercise 2.

Exercise 2

The *average power* of a signal is the average squared value, which is given by the formula:

$$\text{average power} = (x_0^2 + x_1^2 + x_2^2 + \dots + x_{n-1}^2)/n = \sum x_k^2/n; \quad k = 0, 1, 2, \dots, n - 1$$

An incomplete implementation of a function named `avg_power` is provided in `arrays.c`. The function prototype is:

```
double avg_power(double x[], int n);
```

This function returns the average power of the signal represented by an array of `double`s containing *n* elements.

Finish the definition of this function. Your function should assume that `n` is positive; i.e., **it should not verify that `n` is > 0 before calculating the average power of the first `n` array elements**.

Use the console output printed by the test harness to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #2 before you start Exercise 3.

Exercise 3

An incomplete implementation of a function named `maxi` is provided in arrays.c. The function prototype is:

```
double maxi(double arr[], int n);
```

This function returns the maximum value in an array of `double`s containing *n* elements.

Finish the definition of this function. Your function should assume that `n` is positive; i.e., **it should not verify that `n` is > 0 before calculating the maximum value in the first `n` array elements**. Your function **cannot** assume that all elements in the array will be greater than any particular value; in other words, each element could be any of the double-precision floating point numbers that can be represented in C.

Use the console output printed by the test harness to help you identify and correct any flaws. Verify that your function passes all the tests in test suite # 3 before you start Exercise 4.

Exercise 4

An incomplete implementation of a function named `mini` is provided in arrays.c. The function prototype is:

```
double mini(double arr[], int n);
```

This function returns the minimum value in an array of `double`s containing *n* elements.

Finish the definition of this function. Your function should assume that `n` is positive; i.e., **it should not verify that `n` is > 0 before calculating the minimum value in the first `n` array elements**. Your function **cannot** assume that all elements in the array will be smaller than any particular value; in other words, each element could be any of the double-precision floating point numbers that can be represented in C.

Use the console output printed by the test harness to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #4 before you start Exercise 5.

Exercise 5

There are several different ways to *normalize* a list of data. One common technique scales the values so that the minimum value in the list becomes 0, the maximum value in the list becomes 1, and the other values are scaled in proportion. For example, consider the values in this unnormalized list:

```
[-2.0, -1.0, 2.0, 0.0]
```

The normalization technique described above changes the list to:

```
[0.0, 0.25, 1.0, 0.5]
```

The formula for calculating the normalized value of the k^{th} value in a list, x_k, is:

$$\textit{normalized value of } x_k = (x_k - min_x) \,/\, (max_x - min_x)$$

where min_x and max_x represent the minimum and maximum values in the list, respectively. If you substitute min_x for x_k in this formula, the dividend becomes 0, so the normalized value of min_x is 0.0. If you substitute max_x for x_k in this formula, the dividend and divisor have the same value, so the normalized value of max_x is 1.0.

An incomplete implementation of a function named `normalize` is provided in arrays.c. This function is passed an array containing *n* real numbers, and normalizes the array using the technique described above.

Finish the definition of this function. Your function should assume that the array will contain at least two different numbers, so the expression $max_x - min_x$ will never be 0. Your function must call the `max` and `min` functions you wrote for Exercises 3 and 4.

Use the console output printed by the test harness to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #5.

Note: In C, arrays are passed by reference to functions.

Lab 4 – Part II: Local and Global Variables

Step 1: Depending on your operating system, download (**lab4_part2_win.zip** or **lab4_part2_macOS.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Open the unzipped folder in VS Code

Step 3: lab4_part2_xxx folder contains the following files:

- lab4test.h contains the declarations (function prototypes) for the functions defined in lab4test.c **Do not modify lab4test.h and lab4test.c.**
- main.c contains the main function. **Do not modify main.c**

Step 4: Open each of the files, get familiar with the variables and identify the scope of each of them.

Step 5: Answer the following questions and show your answers to the TAs:

- What is the scope of variable1 defined inside main.c?
- Are any of the variables variable1 defined inside lab4test.c the same as the variable defined inside main.c?
- Does the assignment statement inside lab4test.c (line 7) modify the content of variable1 inside main.c?
- How many distinct variables i there are in main.c? If more than one, what is the scope of each of them?
- How many distinct variables variable1 there are in main.c? If more than one, what is the scope of each of them?
- In lab4test.c, is the variable1 inside test2 the same as the variable inside test1?

Wrap-up

Submit arrays.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 arrays.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.