

Carleton University  
Department of Systems and Computer Engineering  
SYSC 2006 - Foundations of Imperative Programming - Winter 2017

**Lab 3 - Functions that Process Arrays**

**Attendance/Demo**

To receive credit for this lab, you must demonstrate your solutions to the exercises. When you have finished all the exercises, call a TA, who will review your code, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

**Objective**

The objective of this lab is to write some C functions that process arrays.

**Prerequisite Reading**

*How to Think Like a Computer Scientist, C Version* (the URL for this book is in the course outline):

- Chapter 7

**A Brief Review of C Arrays**

The C variable declaration:

```
type name[capacity];
```

allocates an array with the specified *name*. The array's *capacity* is an integer expression, and specifies the number of elements in the array. Each element in the array stores a value of the specified *type*. So, the declaration

```
int numbers[10];
```

declares an array named `numbers` that has 10 elements, each one storing an integer.

An element in an array is accessed by specifying the array name and the element's position (index), which is given by an integer that ranges from `0` to *capacity*-1. For example, `numbers[0]` is the first element in array `numbers`, `numbers[1]` is the second element, and `numbers[9]` is the tenth element.

An array index does not have to be a literal integer; instead, we can use any expression that yields an integer. Often, the index is a variable of type `int`. As an example, here is a loop that initializes `numbers` with the first 10 even integers, starting with 0:

```
// initialize numbers to {0, 2, 4, 6, ..., 18}
int numbers[10];

for (int i = 0; i < 10; i += 1) {
    numbers[i] = 2 * i;
}1
```

There's an alternate way of declaring a C array that allows us to specify the initial values of the array elements by providing an *initializer list* as part of the declaration. For example, this statement:

```
int nums[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
```

declares and initializes array `nums`; the end result is the same as using a loop to initialize the array.<sup>2</sup> Notice that we didn't specify the array's capacity. The C compiler calculates the capacity, based on the number of values in the initializer list.

C arrays can be function arguments. Here's a function that returns the sum of the first  $n$  values in an array of integers:

```
int sum_array(int arr[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i += 1) {
        sum = sum + arr[i];
    }
    return sum;
}
```

Notice how parameter `arr` is declared - the parameter name is followed by square brackets, `[]`. This declares that the parameter is an array; however, we do not specify the capacity of the array.

---

<sup>1</sup> Here is an equivalent Python loop that creates and initializes a list:

```
# initialize numbers to [0, 2, 4, 6, ..., 18]
numbers = [0] * 10 # create a list of 10 0's
for i in range(10):
    numbers[i] = 2 * i
```

A more common approach in Python is to create an empty list, then append the ten integers, one at a time:

```
# initialize numbers to [0, 2, 4, 6, ..., 18]
numbers = []
for i in range(10):
    numbers.append(2 * i)
```

Python will expand the list's capacity, as required. We can't use this approach in C, because an array's capacity cannot be changed after it is allocated.

<sup>2</sup> This is equivalent to the Python statement:

```
nums = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

As a result, the function will process any array, regardless of its capacity, as long as each element in the array is of type `int`. (Of course, the sum of the array elements must not be greater than the largest `int` value.) It is the programmer's responsibility to ensure that the first  $n$  elements of the array have been initialized.

As an example of using an array as a function argument, here is how we call `sum_array` to sum all the integers in array `nums`:

```
int nums[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};

/* For an explanation of the next statement, see Section 7.5
 * in "How to Think Like a Computer Scientist - C Version".
 */
int capacity = sizeof(nums) / sizeof(nums[0]);

int total;
total = sum_array(nums, capacity);
```

Notice that the first argument is the name of the array, `nums`, and not `nums[]`.

We can call the same function to sum just the first five elements of array `numbers`; i.e., calculate `numbers[0] + numbers[1] + numbers[2] + numbers[3] + numbers[4]`:

```
int partial_sum;
partial_sum = sum_array(numbers, 5);
```

Functions can modify their array arguments. Here's a function that initializes the first  $n$  elements of an array to a specified integer value:

```
void initialize_array(int arr[], int n, int initial)
{
    for (int i = 0; i < n; i += 1) {
        arr[i] = initial;
    }
}
```

To initialize all 10 elements in `numbers` to 0, we call the function this way:

```
initialize_array(numbers, 10, 0);
```

### **A Comparison of C Arrays and Python Lists (primarily for students who took SYSC 1005)**

An array can be thought of as a primitive Python list, but there are some important differences:

- When we create a Python list, we don't specify its capacity. Python lists automatically grow (increase their capacity) as objects are appended or inserted in a list. In contrast, the capacity of a C array is determined when it is declared. The array's capacity is fixed; there is no way to increase its capacity at run-time.
- We can determine the length of a Python list (that is, the number of objects stored in the list) by passing the list to Python's built-in `len` function. In contrast, C does not keep track of how many array elements have been initialized, and there is no function we can call to determine this. It is the programmer's responsibility to do this, usually by using an

auxiliary variable.

- Python generates a run-time error if you specify an invalid list index, but C does not check for out-of-bounds array indices. For example, a C expression such as `numbers[10]` will compile without error. At run-time, this expression accesses memory outside the array. Similarly, while `numbers[-1]` is a perfectly valid Python expression, when used in a C program, this expression accesses memory outside the array.
- Python provides functions, methods and operators that perform several common operations on lists; for example, append an object to the end of a list, insert an item in a list, delete an item from a specified position in a list, remove a specified object from a list, determine if a specified object is in a list, find the largest and smallest objects in a list, etc. In contrast, the only array operation C provides is the `[]` operator to retrieve or set the value at a specified index.

## General Requirements

You have been provided with four files:

- `exercises.c` contains incomplete definitions of five functions you have to design and code.
- `exercises.h` contains the declarations (function prototypes) for the functions you'll implement. **Do not modify `exercises.h`.**
- `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.**

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

Your functions should 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.

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). Pelles C makes it easy to do this. To select the formatting style:

- From the menu bar, select **Tools > Options...** An Options box will appear.
- Click the **Tabs** tab
- In the **C formatting style** box, click a radio button to select either **Style 1** (for K&R style) or **Style 2** (which appears to be close to BSD/Allman style).
- Click **OK**. The Options box will close.

To format the code in your editor window:

- Select Edit > Select all. Your code will be highlighted.
- Select Source > Convert to.
- From the submenu, select Formatted C code. Your highlighted code will be reformatted to conform to the selected style.

More information about C indentation styles can be found in Appendix A.3 of *How to Think Like a Computer Scientist, C Version*.

## Getting Started

### Step 1

Launch Pelles C and create a new project named `array_exercises`.

- If you're using the 64-bit edition of Pelles C, the project type should be Win 64 Console program (EXE). (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.)
- If you're using the 32-bit edition of Pelles C, the project type should be Win32 Console program (EXE).

When you finish this step, Pelles C will create a project folder named `array_exercises`.

### Step 2

Download files `main.c`, `exercises.c`, `exercises.h` and `sput.h` from cuLearn. Move these files into your `array_exercises` folder.

### Step 3

You must also add `main.c` and `exercises.c` to your project (moving the files to your project folder doesn't do this).

- Select Project > Add files to project... from the menu bar.
- In the dialogue box, select `main.c`, then click Open. An icon labelled `main.c` will appear in the Pelles C project window.
- Repeat this step for `exercises.c`.

You don't need to add `exercises.h` and `sput.h` to the project. Pelles C will do this after you've added `main.c`.

### Step 4

Build the project. It should build without any compilation or linking errors.

### Step 5

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

Execute the project. The test harness (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 the

harness tests.

The console output will be similar to this:

```
== Entering suite #1, "Exercise 1: avg_magnitude()" ==

[1:1] test_avg_magnitude:#1 "
avg_magnitude({5.7, 2.3, -1.9, 4.5, 6.2, -8.1, 9.7, 3.1}, 8) returns
5.19" FAIL
!      Type:      fail-unless
!      Condition: fabs(avg_magnitude(samples, 8) - 5.19) < 0.01
!      Line:      133

--> 1 check(s), 0 ok, 1 failed (100.00%)

== Entering suite #2, "Exercise 2: avg_power()" ==

...

==> 13 check(s) in 5 suite(s) finished after 0.00 second(s),
    0 succeeded, 13 failed (100.00%)

[FAILURE]
*** Process returned 1 ***
```

File `main.c` contains five *test suites*, one for each of the functions you'll write in Exercises 1-5.

In Exercise 1, you'll complete the implementation of a function named `avg_magnitude`. The first test suite is named "Exercise 1: `avg_magnitude()`". This test suite has one *test function*, named `test_avg_magnitude`. This function calls `avg_magnitude` once, and determines if `avg_magnitude` correctly returns the average magnitude of the array of doubles `{5.7, 2.3, -1.9, 4.5, 6.2, -8.1, 9.7, 3.1}`:

```
[1:1] test_avg_magnitude:#1 "
avg_magnitude({5.7, 2.3, -1.9, 4.5, 6.2, -8.1, 9.7, 3.1}, 8) returns
5.19" FAIL
!      Type:      fail-unless
!      Condition: fabs(avg_magnitude(samples, 8) - 5.19) < 0.01
```

The condition that determines if `avg_magnitude` returns 5.19,

$$\text{fabs}(\text{avg\_magnitude}(\text{samples}, 8) - 5.19) < 0.01$$

may appear a bit strange. Because of the way real numbers are represented in a computer, we should never use the `==` operator to compare two real numbers for equality. Instead, two real numbers are considered to be equal if they differ from each other 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 0.001), we consider the value returned by `avg_magnitude` to be equal to 5.19.

After the first suite has been executed, a summary is displayed, indicating that test performed by `test_avg_magnitude` fails:

```
--> 1 check(s), 0 ok, 1 failed (100.00%)
```

Next, suites #2 through #5 are executed, to test the functions you'll implement in Exercises 2 through 5. As expected, the tests in these suites also fail.

After all suites have been executed, a summary is displayed:

```
==> 13 check(s) in 5 suite(s) finished after 1.00 second(s),  
    0 succeeded, 13 failed (100.00%)
```

```
[FAILURE]
```

```
*** Process returned 1 ***
```

Your objective is to correctly implement all the functions, so that all the tests pass.

## Step 6

Open `exercises.c` in the editor. Design and code the functions described in Exercises 1 through 5. Don't make any changes to `main.c`, `exercises.h` or `sput.h`. All the code you'll write must be in `exercises.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  $x_0, x_1, x_2, \dots, x_{n-1}$ .

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 `main.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 doubles 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 calculate the absolute values of real numbers. The function prototype is:

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

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all the tests in the first test suite 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 `main.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 doubles 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**.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all the tests in the second test suite before you start Exercise 3.

## Exercise 3

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

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

This function returns the maximum value in an array of doubles 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, it **cannot** assume that all elements will be, for example, greater than 0 or greater than -999.0.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all the tests in the third test suite before you start Exercise 4.

## Exercise 4

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

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

This function returns the minimum value in an array of doubles 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, it **cannot** assume that all elements will be, for example, less

than 999.0.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all the tests in the fourth test suite 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  $k^{\text{th}}$  value in a list,  $x_k$ , is:

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

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 `main.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.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all the tests in the fifth test suite.

### Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.
2. Remember to back up your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.