Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Fall 2015

Lab 3 - Functions that Process Arrays

Attendance/Demo

To receive credit for this lab, you must make reasonable progress towards completing the exercises and demonstrate the code you complete. **Also, you must submit your lab work to cuLearn by the end of the lab period**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. 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 part of the lab is to write some C functions that process arrays.

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`*.

For example,

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

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

Each element in an array is accessed by specifying the array name and the element's position (index), which is given by an integer. 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 specified by a variable of type `int`. Here is a loop that initializes the 10 integer elements in array `numbers`:

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

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

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

Here is an equivalent Python loop that creates an empty list, then initializes it by appending the same ten integers:

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

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

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

C arrays can be used as 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. 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.

To sum all 10 integers in array `numbers`, we call the function this way:

```
int total;
total = sum_array(numbers, 10);
```

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

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

```
int partial_sum;
partial_sum = sum_arrray(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);
```

Aside (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

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

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 have been provided with file `main.c`. This file contains incomplete implementations of five functions you have to design and code. It also contains 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.

Getting Started

Step 1

Create a new project named array_functions (all letters are lowercase, with underscores separating the words). The project type can be either Win 64 Console program (EXE) or Win32 Console program (EXE). After creating the project, you should have a project folder named array_functions. Check this. If you do not have a folder named array_functions, close this project and repeat Step 1.

Step 2

Download files main.c and sput.h from cuLearn. Move these files into your array_functions folder.

Step 3

You must also add main.c to your project. To 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.

You don't need to add 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. You'll need to understand the output the test harness displays when tests fail (i.e., when your code has flaws) as well as when tests pass.

Execute the project. The test harness 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: max()" ==

[1:1]  test_max:#1  "max({1.0, 2.0, 3.0, 4.0}) ==> 4.0"  FAIL
!    Type:      fail-unless
!    Condition: fabs(max(data1, 4) - 4.0) < 0.001
!    Line:      93
[1:2]  test_max:#2  "max({1.0, 2.0, 4.0, 3.0}) ==> 4.0"  FAIL
!    Type:      fail-unless
!    Condition: fabs(max(data2, 4) - 4.0) < 0.001
!    Line:      95
[1:3]  test_max:#3  "max({4.0, 3.0, 2.0, 1.0}) ==> 4.0"  FAIL
!    Type:      fail-unless
!    Condition: fabs(max(data3, 4) - 4.0) < 0.001
!    Line:      97
[1:4]  test_max:#4  "max({5.0}) ==> 5.0"  FAIL
```

```
!     Type:      fail-unless
!     Condition: fabs(max(data4, 1) - 5.0) < 0.001
!     Line:      99
[1:5]  test_max:#5  "max({2.0, 2.0}) ==> 2.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data5, 2) - 2.0) < 0.001
!     Line:      101

--> 5 check(s), 0 ok, 5 failed (100.00%)

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

...

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

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

In Exercise 1, you'll complete the implementation of a function named `max`. The first test suite is named "Exercise 1: max()". This test suite has one *test function*, named `test_max`. This function calls `max` five times. Each time, the value returned by `max` is compared to the value we expect a correct implementation of the function to return.

For example, the first test performed by `test_max` checks if `max` correctly returns the largest value in the array of doubles `{1.0, 2.0, 3.0, 4.0}`:

```
[1:1]  test_max:#1  "max({1.0, 2.0, 3.0, 4.0}) ==> 4.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data1, 4) - 4.0) < 0.001
```

The condition 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 4.0 (the expected result) from the value returned by `max`, 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 `max` to be equal to 4.0.

Step 6

Open main.c in the editor. Design and code the functions described in Exercises 1 through 5.

Exercise 1

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 check whether `n` is passed a positive or negative value. 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 of the tests in the first test suite before you start Exercise 2.

Exercise 2

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 check whether `n` is passed a positive or negative value. 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 0 or 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 of the tests in the test suite before you start Exercise 3.

Exercise 3

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 `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. Your function must call the `max` and `min` functions you wrote for

Exercises 1 and 2.

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 of the tests in the test suite before you start Exercise 4.

Exercise 4

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 $x_0, x_1, x_2, \ldots, x_{n-1}$.

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

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

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 check whether `n` is passed a positive or negative value.

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 of the tests in the test suite before you start Exercise 5.

Exercise 5

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

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

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 check whether n is passed a positive or negative value.

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 of the tests in the 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. The next thing you'll do is package the project in a ZIP file (compressed folder) named array_functions.zip. To do this:

   2.1. From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. If you named your Pelles C project array_functions, the zip file will be named array_functions.zip by default; otherwise, you'll have to edit the File name: field and rename the file to array_functions before you save it. **Do not use any other name for your zip file** (e.g., lab3.zip, my_project.zip, etc.).

   2.2. Click Save. Pelles C will create a compressed (zipped) folder, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder array_functions).

3. Before you leave the lab, log in to cuLearn and submit array_functions.zip. To do this:

   3.1. Click the Submit Lab 3 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag array_functions.zip to the File submissions box. **Do not submit another type of file (e.g., a Pelles C .ppj file, a RAR file, a .txt file, etc.)**

   3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you aren't ready to do this; for example, you want to do some more work on the project and resubmit it later, you can leave the file with "draft" submission status. When you're ready to submit the final version, you can replace or delete your "draft" file submission by following the instructions in Step 3.3, then finish the submission process by following the instructions in Step 3.4.

   3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

      3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

      3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit the final version of your file. This will change the submission status to "Submitted for grading".