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

Lab 3 - Arrays

Online Submission

Submit file exercises.c through cuLearn before the deadline.

Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.

Prerequisite Reading

Programming in C zyBook, Chapter 8, *Arrays*

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.

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

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

Instead, your functions should modify their array arguments, as required.

Use the indexing (`[]`) operator to access array elements. Do not use pointers and pointer arithmetic (which have not yet been covered in lectures). 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 on how to do this were provided in Lab 1 and Lab 2.

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

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.

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

Execute the project. The test harness will report 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)"  FAIL
!    Type:      fail-unless
!    Condition: fabs(avg_magnitude(samples, 8) - 5.19) < 0.01
!    Line:      25
Expected result: 5.19 (approximately), actual result: -1.00

--> 1 check(s), 0 ok, 1 failed (100.00%)
Tests for remaining exercises won't be run until avg_magnitude passes
all tests.
```

```
==> 1 check(s) in 1 suite(s) finished after 0.00 second(s),
    0 succeeded, 1 failed (100.00%)

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

In Exercise 1, you'll complete the implementation of a function named `avg_magnitude`. The test suite for this function is is named "Exercise 1: avg_magnitude()". This test suite has one *test function*, named `test_avg_magnitude`:

```
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 doubles.

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 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 0.01), we consider the value returned by `avg_magnitude` to be close enough to 5.19, and the test passes.

The incomplete implementation of `avg_magnitude` in exercises.c always returns -1, so this condition is `false`, and the test fails. The harness then displays the expected and actual results.

After the first suite has been executed, a message is displayed, indicating that the tests performed by test suite #1 failed:

```
--> 1 check(s), 0 ok, 1 failed (100.00%)
Tests for remaining exercises won't be run until avg_magnitude passes
all tests.
```

A summary is displayed as the test harness finishes:

```
==> 1 check(s) in 1 suite(s) finished after 1.00 second(s),
    0 succeeded, 1 failed (100.00%)
```

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

After you have correctly implemented `avg_magnitude`, the output displayed by sput should be:

```
== 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)"  pass
Expected result: 5.19 (approximately), actual result: 5.19

--> 1 check(s), 1 ok, 0 failed (0.00%)
```

When you review the output, you can quickly determine that your `avg_magnitude` function passes the test performed by `test_avg_magnitude`.

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:

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

An incomplete implementation of a function named `avg_magnitude` is provided in exercises.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. Review the console output and 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:

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

An incomplete implementation of a function named `avg_power` is provided in `exercises.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. Review the console output and 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 `max` is provided in `exercises.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. Review the console output and 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 `min` is provided in `exercises.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. Review the console output and 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 `exercises.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.Review the console output and verify that your function passes all the tests in test suite #5.

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 grading/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.