Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Fall 2013

Lab 4 - Arrays and Functions

Objective

This is the third (and final) lab that reviews concepts that were taught in the prerequisite courses (i.e., C/C++ arrays for students who took ECOR 1606, and Python lists for students who took SYSC 1005). The objective of this lab is to write some C functions that process arrays.

Attendance/Demo

To receive credit for this lab, you must make a reasonable effort to complete the exercises and demonstrate the code you complete.

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. Finish any exercises that you don't complete by the end of the lab on your own time. Also, you must submit your lab work to cuLearn by the end of the lab period.

A Brief Review of C Arrays

The C variable declaration:

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

allocates an array with the the specified *`name`*. The array's *`capacity`* must be a constant 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 samples[10];
```

declares an array that can store 10 integer values.

Each element in an array is accessed by specifying the array name and the element's position (index), which is given by an integer expression. For example, `samples[0]` is the first element in array `samples`, `samples[1]` is the second element, and `samples[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. Most often, the index is specified by a variable of type `int`. Here is a loop that initializes the 10 integer elements in array `samples`:

```
// initialize samples to {0, 2, 4, 6, ..., 18}
int samples[10];
for (int i = 0; i < 10; i += 1) {
    samples[i] = 2 * i;
}
```

Here is an equivalent loop, written in Python, that creates and initializes a list:

```
# initialize samples to [0, 2, 4, 6, ..., 18]
samples = []
for i in range(9):
    samples.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 samples[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
```

does the same thing as the earlier code fragment containing a `for` loop. 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 a[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i += 1) {
        sum = sum + a[i];
    }
    return sum;
}
```

Notice how parameter `a` is declared. The `[]` indicates that the parameter is an array; however, we do not specify the capacity of the array. This means that the function can work with any array, regardless of its capacity, as long as each element in the array is of type `int`. 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 samples, we call the function this way:

```
int s;
sum_array(samples, 10);
```

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

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

```
int s;
s = sum_arrray(samples, 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 a[], int n, int initial)
{
    for (int i = 0; i < n; i += 1) {
        a[i] = initial;
    }
}
```

We can call this function to zero all 10 elements in `samples`:

```
initialize_array(samples, 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:

- We never know the capacity of a Python list. Python lists automatically grow as objects are appended or inserted in a list. In contrast, the capacity of a C array must be specified when it is declared. The array's capacity is fixed; there is no way to increase its capacity after values have been stored in every element.
- 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 items have been stored in an array. It is the programmer's responsibility to do this (usually by using an auxiliary variable to count the number of items in the array).
- 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 `samples[10]` will compile without error. At run-time, this expression accesses memory outside the array. Similarly, while `samples[-1]` is a perfectly valid Python expression, when used in a C program, this expression accesses memory outside the array.

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; i.e., contain `scanf` statements. Except for the `print_array` function you write for Exercise 1, none of your functions should produce console output; i.e., contain `printf` statements.

You have been provided with file main.c. This file contains incomplete implementations of six functions you have to design and code. It also contains a *test harness* consisting of 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.**

Instructions

1. Create a folder named Lab 4.
2. Launch Pelles C, and create a new project named lab4 inside the Lab 4 folder. The project type must be Win32 Console program (EXE). You should now have a folder named lab4 inside a folder named Lab 4 (check this).
3. Download file main.c from cuLearn. Move this file into your lab4 folder. **You must also add main.c to your project:** from the menu bar, select Project > Add files to project... In the dialogue box, select main.c, then click Open. An icon labelled main.c will appear in the Pelles C project window.
4. Open main.c.
5. Build the project. It should build without any compilation or linking errors.
6. Execute the project. The test harness will run. Read the console output carefully. The harness will report several errors while it tests the functions you'll complete for Exercises 1 through 5, which is what we'd expect, because you haven't started working on these functions.

Exercise 1

Write a function that prints the values stored in an array of doubles containing *n* elements. The function prototype is:

```
void print_array(double x[], int n);
```

The required format for the output is [$elem_0$ $elem_1$ $elem_2$ ... $elem_{n-1}$]; that is, a list of values enclosed in square brackets, with one space between each pair of values. There must be no spaces between the '[' and the first value, or between the last value and ']'. Here is an example of what the output should look like::

```
[1.0 5.2 -3.7 4.9]
```

Note: your function should assume that `n` is positive; i.e., it should not print an error message if `n` is zero or negative.

Exercise 2

Write a function that returns the maximum value in an array of doubles containing *n* elements. The function prototype is:

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

Note: your function should assume that `n` is positive; i.e., it should not print an error message if `n` is zero or negative. 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.

Exercise 3

Write a function that returns the minimum value in an array of doubles containing *n* elements. The function prototype is:

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

Note: your function should assume that `n` is positive; i.e., it should not print an error message if `n` is negative. 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.

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:

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

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

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

Write a function that returns the average magnitude of an array of doubles containing n elements. The function prototype is:

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

Your function should assume that `n` is positive.

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);
```

This prototype is found in header file `math.h`, so you'll have to remember to put

```
#include <math.h>
```

at the start of your program.

Write a function that returns the average power of an array of doubles containing n elements. The function prototype is:

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

Your function should assume that `n` is positive.

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.

Write a function named `normalize` that is passed an array containing *n* real numbers. This function will normalize the array using the technique described above. Your function should assume that the array will contain at least two different numbers.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.
2. The next thing you'll do is package the project in a ZIP file (compressed folder). From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. Click Save. Pelles C will create a compressed (zipped) folder named lab4.zip, 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 lab4).
3. Log in to cuLearn and click the Submit Lab 4 link. Submit lab4.zip. Instructions for submitting file-based assignments in cuLearn can be found here:

   www5.carleton.ca/culearnsupport/students/

   Click the link Evaluation Tools, then click the link Assignments.

   When you upload a file, its submission status is "Draft (not submitted)". After you upload the file, remember to click the Submit assignment button. This will change the submission status to "Submitted for grading".

Posted: September 29, 2013.