CSCE 155 - C

Lab 07 - Arrays & Dynamic Memory

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Review the following free textbook resources:
   - http://www.cs.cf.ac.uk/Dave/C/node7.html
   - http://en.wikibooks.org/wiki/C_Programming/Arrays
   - http://www.cs.cf.ac.uk/Dave/C/node11.html
   - http://en.wikibooks.org/wiki/C_Programming/Memory_management

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Understand what memory leaks are, how they're caused, and how to prevent them
- Declare and fill an array with values
- Access an array's values individually and iterate over the entire array
- Pass an array as a parameter to a function
- Understand the relationship between arrays and pointers

2 Background

Arrays hold collections of variable values of a certain type (`int` or `double` for example). Static arrays are arrays whose size is specified at compile time and specified by hardcoding the size when declared. For example:

```
int array[100];
```

This declares an integer array that holds 100 integers (in addition, static arrays are allocated on the program stack while dynamic arrays are allocated on the program heap, but we will not go into those details here). Once we have an array, we can access individual elements in the array by specifying an index:

```
1  array[0] = 5;
2  printf("the last element is %d\n", array[99]);
```

You should take care that you do not access elements beyond the range of the array's indices. Indices start at 0 (first element) and end one less than the size of the array. Attempts to access elements outside this range results in undefined behavior and may result in a segmentation fault or other such run time error.

Unfortunately, static arrays are quite limited and for most applications, we will not know how big an array needs to be at compile time. Instead, we typically need an array whose size is not known until runtime. Dynamic arrays provide such functionality.

Dynamic arrays are arrays whose size is not specified at compile time. Rather, we write code such that at run time, a chunk of memory is dynamically allocated to accommodate the space we need to hold a certain number of variable values. C provides a family of functions to dynamically allocate memory in the standard C library (`stdlib.h`): `malloc()`, `calloc()` and `realloc()`.

In this lab, we will focus on malloc (memory allocation), which takes exactly one argument: the number of bytes you wish to allocate in memory. To make our code portable, we do not hardcode the number of bytes each type of variable requires. Instead, we use the `sizeof` operator to determine how many bytes each type takes. Finally, `malloc` returns a generic void pointer (`void *`) because as far as `malloc` is concerned, it is just allocating memory, it doesn't care what you intend to use it for. Therefore, it is best practice to explicitly type case the void pointer into the type of pointer that we intend to use it as. Altogether, we have:

```
int *arr = (int *) malloc(sizeof(int) * 100);
```

This creates an array that is holds 100 integers. Once allocated, we can treat the array like any other static array with indices starting at 0 just as before.

Memory is a resource and when using dynamic memory, memory management becomes an issue. Memory management means that we responsibly "clean up" after ourselves. Once the dynamic memory that was allocated is no longer needed, we need to tell the operating system that we're done with it and that it should be made available to be reused by our program or allow other programs to use it. We release memory by using the `free()` function and passing it a pointer to our memory chunk:

```
free(arr);
```

Failure to release unneeded memory may result in a memory leak whereby a program continually allocates memory but never releases it, becoming a resource hog. Eventually, the program uses so much memory that it slows to a halt or is terminated as the operating system is unable or unwilling to allocate more resources to it.

3 Activities

Clone the repository from GitHub containing the code for this lab by using the following URL: https://github.com/cbourke/CSCE155-C-Lab07.

3.1 Fixing a Memory Leak

In this exercise, we'll observe a memory leak in action and fix it.

Instructions

1. Change to the `memoryLeak` directory
2. Build the executable using the `make` command. Make is a utility that reads a makefile–a specification for source file dependencies and the process by which they are built. The makefile for this project has been provided for you.
3. A program named `memLeak` will be created in your directory. The program will generate an array of 500,000 integers using `malloc()` several times (the exact number is specified by the user as a command line argument). It will then find a random ith Order Element and print the result to the screen.
4. Open a second putty session (right click the putty icon on the task bar), and run the command `top -u login` where `login` is replaced with your cse login. This will open up a list of all of the processes currently being executed by you.
5. In the first putty session, run the `memLeak` program and generate 40 different random lists by running `./memLeak 40` (this may take some time).
6. In the second putty session (the one running top), pay close attention to the `memLeak` program, specifically the column labeled "RES". This column will tell you the current amount of memory being used by a specific process.
7. Keep track of how much memory it uses towards the end of its execution on your worksheet.
8. Once the program is done, navigate to the `src` directory and open the file `memLeak.c`. Fix the memory leak by adding a call to `free()` in the appropriate place (hint: it will be inside the loop in the `main()` function).
9. Repeat steps 6–8. Close top by typing 'q'
10. Finish questions 1 and 2 on your worksheet

3.2 Static Arrays

Navigate back to the statistics directory. Several files have been provided for you. The `stats.c` and `stats.h` file define several utility functions to manipulate arrays and to compute statistics on those arrays. The `statsMain.c` file contains a main driver program. Recall that you will compile these files using:

```
gcc -c stats.c
gcc -o runStats stats.o statsMain.c
```

In this activity, you will implement and use several functions that use arrays as parameters. When passed as a parameter to a function, the function also needs to be told how

large the array is; typically an integer size variable is passed with any array parameter. You will also declare and populate a static array to test your functions.

Instructions

1. A function, `readInArray` has been provided for you that takes an array and its size and prompts the user to enter values to populate the array. However, there is an error in the function: examine the `scanf` call and determine what is wrong and fix it before proceeding.
2. Using the other functions as a reference, implement the `getMin`, `getMax`, and `getMean` functions. You will need to provide the correct function signature in both the header file and source file.
3. In the `statsMain.c` declare a static array that is "large enough" as indicated by the other clues in the code. Use your declared static array as an argument to `readInArray` and the other function calls.
4. Run your program and demonstrate it to a lab instructor.

3.3 Dynamic Arrays

In this activity you will modify the code in `statsMain.c` to use a dynamic array instead of a static array.

Instructions

1. Alter your static array declaration to be an integer pointer and add code that calls malloc to initialize the appropriate amount of memory.
2. Alter any other code as necessary to remove the "large enough" restriction that we had in the previous activity.
3. Run your program again and demonstrate it to a lab instructor

Follow Up

Demonstrate your program using a "large" array. Obviously prompting for and entering a lot of numbers is tedious. So instead, alter your code again to instead take advantage of the `createRandomArray` function provided for you. Demonstrate it to a lab instructor and have them sign your lab worksheet.

4 Advanced Activity (Optional)

So far we've only been working with single dimensional arrays, but often times it is important to represent something like a table or a matrix in code. One way to accomplish this is to use a multidimensional array. You can declare a statically allocated 2D array with the statement

```
int matrix[100][50];
```

You can think of 100 as the number of rows in the table and 50 as the number of columns. Begin by writing a program that will initialize a matrix with some random numbers and find the average of each row (you'll probably want to use nested for loops). Once you've gotten the hang of accessing the elements in a 2D matrix, try writing a program that will find the product of two (square) matrices. To assist you, write a function that will automatically create a randomly populated two dimensional array and return a reference to it (a pointer to a pointer array!).