CSCE 155 - C

Lab 7.0 - Arrays & Dynamic Memory

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 7 and 20 of the Computer Science I textbook
3. Watch Videos 7.1 thru 7.7 of the Computer Science I video series

Peer Programming Pair-Up

For students in the online section: you may complete the lab on your own if you wish or you may team up with a partner of your choosing, or, you may consult with a lab instructor to get teamed up online (via Zoom).

For students in the face-to-face section: your lab instructor will team you up with a partner.

To encourage collaboration and a team environment, labs are be structured in a *peer 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 Observing a Memory Leak

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

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. You can configure and run it as follows:

   ```
   ./memLeak 1000000 10
   ```

   which will run the program and create 10 random arrays each of size 1 million integer elements. It then sorts the array and reports the median element. The program will execute rather quickly so it will be difficult to observe the memory leak.
4. To observe the consequences of the leak, run the program with the following configuration:

   ```
   ./memLeak 10000000 20 > /dev/null &
   ```

   This will run the program with more arrays that are much larger, slowing it down. It will also suppress the output and run the program in the background so you can continue operating on the command line.
5. Run the command `top -u login` where `login` is replaced with your login (If you are on the cse server this is your cse login. If you are on a different server you can use `whoami` to determine your username). This will open up a list of all of the processes currently being executed by you.
6. Observe the column labeled "RES". This column will tell you the current amount of memory being used by a specific process. It should take about 2 minutes to execute. Type `q` to quit out of the `top` program.

3.2 Diagnosing & Fixing a Memory Leak

In this activity, you'll use a dynamic analysis tool called Valgrind (http://valgrind.org/) to diagnose the memory leak in this program. Valgrind is a *dynamic analysis* tool that can monitor a program and analyze the resources it is using and call attention to errors that may occur at runtime.

Instructions

1. Run the Valgrind tool using the following command:

   `valgrind --leak-check=full --show-leak-kinds=all ./memLeak 1000000 10`
   This starts Valgrind and runs your program within it. With the above parameters, it shouldn't take as long as before.

2. Once it is done, it should produce a report that includes a total number of bytes that was lost due to the memory leak as well as where the memory was originally allocated (though not where it was lost).

3. Using this report as a guide, fix the memory leak and answer the questions on your worksheet

3.3 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.

3.4 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 `generateRandomArray` function provided for you. Demonstrate it to a lab instructor and have them sign your lab worksheet.

4 Handin/Grader Instructions

1. Hand in your completed files:
   - `stats.c`
   - `stats.h`
   - `statsMain.c`
   - `worksheet.md`

   through the webhandin (https://cse-apps.unl.edu/handin) using your cse login and password.
2. Even if you worked with a partner, you *both* should turn in all files.
3. Verify your program by grading yourself through the webgrader (https://cse.unl.edu/~cse155e/grade/) using the same credentials.

4. Recall that both expected output and your program's output will be displayed. The formatting may differ slightly which is fine. As long as your program successfully compiles, runs and outputs the *same values*, it is considered correct.

5 Advanced Activity (Optional)

Read more about the capabilities of Valgrind (http://valgrind.org/, http://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)Debugging.html#Valgrind and how to use it. Think of other situations that this tool may have come in handy and try using it for the next bug that you encounter.