CSC 325 Lab 4

C Pointers and Assembly

Table of Contents

This lab should be done with your Lab 4 partner.

1. Lab Overview and Goals

This lab consists of 2 parts. In the first part, you'll implement a C program that computes some statistics on a set of values read in from a file. The program will use C pointers and dynamic memory allocation to allocate enough space for the set of values it reads in. In the second part you'll write a `sum` function in ARM64 assembly that gets compiled into a program that you can then use test your function.

1.1. Lab Goals

- Gain experience using pointers and dynamic memory allocation (`malloc` and `free`) in C.
- Practice using `gdb` and `valgrind` to debug programs.

- Practice converting C code with conditionals and loops to equivalent assembly code.

2. Starting Point Code

2.1. Get the Starting Point Files

Both you and your partner should download the starting point files from Canvas.

2.2. Starting Point files

```
$ ls
Makefile    large.txt   prog.c    readfile.h  stats.c
readfile.c  small.txt   sum.s
```

1. `Makefile`: A Makefile simplifies the process of compiling your program. We'll look at these in more detail later in the course. You're welcome to look over this one, but you shouldn't need to edit it for this lab. `make` builds the `stats` program for Part 1 and `prog` for Part 2 of this lab assignment.

2. `readfile.h` and `readfile.c`: a library for file I/O. This is the same library used in Lab 1. Your program will make calls to functions in this library to read values from an input file. The instructions for using this library are explained below. **Do not modify any code in these two files**.

3. `stats.c`: starting point code for the C stats program. It contains some code to get the file name from the command line argument and the start of the `get_values` function that you will complete. **Your solution should be implemented in this file**.

4. `small.txt`, `large.txt`: example input files to use to test your stats program. You will want to create your own additional input files to more extensively test your program before you consider it finished.

5. `prog.c`: a main program for Part 2. **You do not need to modify this program**.

6. `sum.s`: starting point code for Part 2. **Your Part 2 solution should be implemented in this file.**

3. Part 1: Dynamic Memory Allocation

3.1. Compiling and Running

You will implement the program started in `stats.c` that takes a single command line argument, which is the name of a file of data values (floats, one per line), and computes and prints out a set of statistics about the data values.

Run `make` to compile the `stats` program. Make compiles your solution in the `stats.c` file, and also compiles and links in the `readfile.o` library, and links in the C math library (`-lm`). The math library has a `sqrt` function, which you will need for the standard deviation calculations.

```
$ make
```

Then run with some input file as a command line argument, for example:

```
$ ./stats small.txt

$ ./stats large.txt
```

Unlike Lab 1, the file format we'll be working with this time **does not** include at header on the first line. Thus, your program has no way of knowing in advance how many floats will appear in the file, so you can't allocate memory for all of them in advance.

This file format (lack of header) is an intentional part of the lab's design to necessitate potentially many calls to `malloc` and `free`.

3.2. Sample Output

The following shows an example of what a run of a complete `stats` program looks like for a particular input file specified at the command line:

```
$ ./stats small.txt
Results:
-------
num values:         16
       min:      0.500
       max:      3.000
      mean:      1.812
    median:      2.000
   std dev:      1.031
unused array capacity: 4
```

Note: you should test your implementation on multiple input files and create your own to test certain cases.

3.3. Lab Details

Experimental scientists often want to compute some simple statistical analyses over the set of data. A useful tool would be a program that could compute these statistical results for any an arbitrarily sized data set (i.e. it would work for 10 or for ten million data values without re-compilation); this is a good example of where using dynamic allocation makes your program more generic than a program using statically declared arrays.

When run, your program should do the following:

1. Make a call to `get_values`, passing in the filename of the file containing the data values, and passing in two pointers: the address of an `int` variable to store the size of the array (number of values read in) and the address of an `int` variable to store the total array capacity. This function is started for you. **See the Requirements Section for details on how to allocate the array that this function returns.**

2. The `get_values` function should return a `float *` that refers to an array of `float` values that stores the values read in from the file, or `NULL` on error (e.g., `malloc` fails or the file cannot be opened).

   During development, at this step you might consider adding a debugging call to a `print_array` function to print out the values you read in to verify that they match what you expect. **Remove this output from your final submission if you do**.

3. Sort the array of values. (see the note in tips section about re-using your sorting function from Lab 1).

   During development, you might also consider printing the array at this step to verify that the values are correctly sorted. **Remove this output from your final submission if you do**.

4. Compute the min, max, mean (average), median, and standard deviation of the set of values and print them out. (See notes below about median)

5. Print out the statistical results, plus information about the number of values in the data set and the amount of unused capacity in the array storing the values.

The statistics your program should compute on the floating-point values are:

1. **num values**: total number of values in the data set.

2. **min**: the smallest value in the data set.

3. **max**: the largest value in the data set.

4. **mean**: the average of the set of values. For example, if the set is: 5, 6, 4, 2, 7, the mean is 4.8 (24.0 / 5).

5. **median**: the middle value in the set of values. For example, if the set of values read in is: 5, 6, 4, 2, 7, the median value is 5 (2 and 4 are smaller and 6 and 7 are larger). **If the total number of values is even, just use the value at index (total/2) as the median**. For example, for the set of 4 values (2, 5, 6, 7), the median value is 6 because 4/2 is 2 and the value in position 2 of the sorted ordering of these values is 6 (2 is in position 0, 5 in position 1, 6 in position 2, 7 in position 3).

6. **stddev**: is given by the following formula:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Where N is the number of data values, x_i is the ith data value, and $\overline{x}$ is the mean of the values.

7. **unused array capacity**: this is really a statistic about the program's behavior. This value refers to the number of spots that you allocated for your array without ever filling in. For example, if you allocate space for 80 floats, but you only fill in 72 of them, your *unused* capacity would be 8 (80 - 72).

3.4. File Format and File I/O

Your program will take as a command line argument the name of an input file that contains the data set. Each file contains some number of float values, one per line. The total number can vary from input file to input file, so it is up to your program to determine when it has read in all the values.

For example, a file with 4 values might look like this:

```
4.4
3.0
20.8
5.6
```

Included with the starting point code are a few sample input files (e.g., `small.txt`, `large.txt`) that you can use to run and test your program.

You will use the readfile library (that you also used in Lab 1) for doing file I/O. Documentation about library functions and how to use the library is in the library header file (`readfile.h`), which you can view in an editor. **You should not change any code in `readfile.c` or `readfile.h`.**

3.5. Lab Requirements

For full credit, your solution should meet the following requirements:

- Implement your solution in `stats.c`, which includes some starting point code.
- The `get_values` function: takes the name of a file containing input values, reads in values from the file into a dynamically allocated array, and returns the dynamically allocated array to the caller. The array's **size** and **capacity** are "returned" to the caller through pass-by-pointer parameters:

  ```
  float *get_values(int *size, int *capacity, char *filename);
  ```

  - The array of values **must be dynamically allocated** on the heap by calling `malloc`. **You must start out allocating an array of 20 float values**. As you read in values into the current array, if you run out of capacity:

    i. Call `malloc` to **allocate** space for a new array that is twice the size of the current full one.

ii. **Copy** values from the old full array to the new array (and make the new array the current one).

iii. **Free** the space allocated by the old array by calling `free`.

- There are other ways to do this type of allocation and re-allocation in C. However, **this is the way I want you to do it for this assignment**. Make sure you start out with a dynamically allocated array of 20 floats, then each time it fills up, allocate a new array of twice the current size, copy values from the old to new, and free the old.
- When all of the data values have been read in from the file, the function should return the filled, dynamically allocated, array to the caller (the function's return type is (`float *`). The array's size and total capacity are "returned" to the caller through the pass-by-pointer parameters `size` and `capacity`.

- After reading in the values from the file, your program should **sort the array** (and note that your program should be able to easily compute min, max and median on a sorted array of values). See the Tips section about how to re-use your sort function from the Lab 1.
- **For full credit, your program must be free of valgrind errors**. Do not forget to close the file you opened! If you don't, valgrind will likely report a memory leak.
  - You can install valgrind on your ELSA VM by the following commands:

```
sudo apt update
sudo apt full-upgrade
sudo apt install valgrind
```

- Your code should be commented, modular, robust, and use meaningful variable and function names. This includes having a top-level comment describing your program and listing your and your partner's names and the date. In addition, every function should include a brief description of its behavior. **You may not use any global variables for this assignment.**
- It should be evident that you applied top-down design when constructing your submission (e.g., there are multiple functions, each with a specific, documented role). **You should have at least 4 function-worthy functions.**
- You should not assume that I will test your code with the sample input files that have been provided.
- When run, your program's output should look like the output shown below from a run of a working program. To make the job of grading easier, **please make your output match the example as closely as possible**.

```
$ ./stats large.txt
Results:
-------
num values:         94
       min:      0.333
       max:      3.000
      mean:      1.161
    median:      1.000
   std dev:      0.788
unused array capacity: 66
```

Note: just like you can use `\n` in the printf format string to insert a new line, you can use `\t` to insert a tab character for pretty formatting. Also, you can specify formatting of each placeholder in the printf string. For example, use `%10.3f` to specific printing a float/double value in a field with of 10 with 3 places after the decimal point. Section 2.8 of the textbook has more information about formatting printf placeholders.

- For increased precision, **use the C `double` type to store and compute the mean and the square root**.
- Your code should be commented, modular, robust, and use meaningful variable and function names. Each function should include a brief description of its behavior.

3.6. Tips and Hints

- Before even starting to write code, use top-down design to break your program into manageable functionality.
- Try building **get_values** without the re-allocation and copying part first (for files with fewer than 20 values). Once that works, then go back and add functionality to handle larger input files that require mallocing up new heap space, copying the old values to the new larger space, and freeing up the old space.
- **Sort function**: You can use your sort function from Lab 1 in this program. To do so, just copy your sort function from the previous lab's .c file into this file, and then change the array parameter's type to be a pointer to float (`float *`). That's it — you can use the same sort function to sort the dynamically allocated array!

```
// change the prototype and function definition of your sort function
// so that its first parameter is float *values
void sort(float *values, int size);
```

- You can similarly copy over your `print_array` function from your Lab 1 solution and use it to help you debug (note: you likely don't want to do this with large files, but you can create some smaller input files on which to test with printing out array contents). **Be sure to remove or comment out calls to `print_array` in your submitted program if you do this (this is just debugging output).**

- The C math library has a function to compute square root: `double sqrt(double val)`. You can pass `sqrt` a float value and C will automatically convert the float value to a double.
- See the Lab 1 lab assignment for documentation about using the `readfile` library (and also look at the `readfile.h` comments for how to use its functions).
- Take a look at the textbook, weekly lab code and in-class exercises to remind yourself about malloc, free, pointer variables, dereferencing pointer variables, dynamically allocated arrays, and passing pointers to functions.
- Use **Ctrl-C** to kill a running program stuck in an infinite loop.

4. Part 2: ARM64 Programming

In this part of the assignment, you will write a `sum` function in ARM64 assembly that is compiled into a program (`prog`) that you can use test your `sum` function.

4.1. Compiling and Running

The files for this part are:

- The `sum.s` file contains the starting point ARM64 code for the sum function that you will complete for Part 2. **Your Part 2 solution should be implemented in this file.**
- `prog.c`: a main program for testing your sum implementation, **You do not need to modify this program**.

Before we building or running part 2, you need to log in to a machine that has an ARM64 CPU! You can use the `ssh` command to remotely log into a different computer. We have a small cluster of ARM machines set up for CS 31. You can log into one using:

```
ssh arm.cs.swarthmore.edu
```

The `Makefile` is set up to compile both the ARM64 `sum.s` and `prog.c` files into an executable named `prog` that you can use to run and test your ARM64 implementation of the `sum.s`.

```
$ make     # compiles prog from sum.s and prog.c
$ ./prog
```

When run, the `prog` reads in user input for the value `n`, which gets passed as a parameter to your `sum` function, and then prints out the result (you can see this main control flow in the `prog.c` file's main function, **which you do not need to modify**):

```
$ ./prog
This program computes the sum of 1-N
Enter an value for n: 10
The sum of 1 to 10 is 55
```

4.2. Sample Output

The following shows some example output of a couple runs of `prog` that makes calls to your ARM64 version of the sum function (in `sum.s`):

```
./prog
This program computes the sum of 1-N
Enter an value for n: 10
The sum of 1 to 10 is 55

./prog
This program computes the sum of 1-N
Enter an value for n: 50
The sum of 1 to 50 is 1275
```

4.3. Details

For this part, you'll implement a sum function in ARM64. You should implement your program in `sum.s`, which has a starting point of this function.

In `sum.s` is the starting point of an ARM64 `sum` function. The starting point handles the stack set-up and function return. As a result, you just need to implement the ARM64 translation of the function body. See the comments in `sum.s` about where on the stack is space for local variables, and where the parameter `n` is located.

The C function you will implement in ARM64 is:

```
/* computes the sum of the values 1-n
 * n: a long value (taken from user input)
 * returns: the sum of values from 1-n, or -1 if n is not positive
 *
 * NOTE: we use longs rather than ints so that we can take advantage of
 * the 64-bix x-prefixed registers (rather than the 32-bit w-registers).
 */

long sum(long n) {
    long i, res;

    if (n <= 0) {
        return -1;
    }

    res = 0;
    for (i = 1; i <= n; i++) {
        res = res + i;
    }

    return res;
}
```

The program `prog.c` makes a call to this `sum` function (you do not need to modify `prog.c`, but you can open it in an editor to see what it's doing).

4.4. Requirements

- Your ARM64 implementation of the `sum` function should be added to the `sum.s` starting point file.
- You do not need to edit `prog.c`, but looking at it might help you understand how your sum function is being called.
- Your solution must include a loop. Yes, we know that the sum of the first `n` positive integers is `n * (n+1) / 2`, but you **are not allowed** to use that in your answer. The purpose of this exercise is to give you experience using control flow instructions (branches).

4.5. Tips

- Write C-goto versions of sum function, the `if` and `for` loop parts in particular, to help you with the translations of those parts.
- Try adding a little bit of assembly code to `sum.s` at a time, then compile and test it out by running `prog`, and then add some more. Code that you test out may not make it in to your eventual solution, but testing small pieces can help you put together the individual steps you need to do things like like returning a value, and initializing a local variables, etc. For example, you might:
    a. first try to just return the value of n.
    b. next, try to initialize res to 0 and return the value of res.
    c. then, try to construct the `if` statement, and test passing positive and negative values to check that your function returns -1 when n is not positive.
    d. finally, try implementing the for loop to build the result.
- If you have errors, trace through the instructions on paper, drawing memory contents and register contents as you go.
- You can also try using `gdb` to debug your program at the assembly level. Set a breakpoint in `sum` and use `ni` to step through instruction execution. Use `print` (e.g., `print $x0`) to print out register values.
- Use **Ctrl-C** to kill a running program stuck in an infinite loop.

5. Submitting your Lab

It is good practice to run `make clean` before

6. Handy References

C programming Resource

- Dive into Systems, Chapter 2 on C programming
- Dive into Systems, Chapter 3 on gdb