Homework 1

WORKING WITH MEMORY ALLOCATIONS AND DEALLOCATIONS

The objective of this assignment is a simple refresher on memory allocations and deallocations using C. Please note that the assignment is crafted such that using tools such as Valgrind will not help you.

Due: Wednesday, September 6th @ 8:00 pm MT

1 Description of Task

For this assignment you will be working with two programs: the `Driver` and the `MemoryManager`.

`Driver` The driver module is responsible for:

1. Invoking functions in the `MemoryManager`.
2. Setting upper bounds for memory consumption. This will be fixed, please do not change it.

A complete Driver program (`Driver.c`) is provided to you with the skeleton.

`MemoryManager` The memory management module is responsible for:

Implementing the core functionality of this assignment. It is responsible for:

1. Allocating and de-allocating data structures
2. Populating elements in the data structure
3. Computing the median element within the data structure
4. Checking to see if the median number is divisible by 13
5. Maintaining a running count of median elements that were divisible by 13.

Your functionality will be go inside the function `get_running_count()` in the `MemoryManager.c` file. Many of the auxiliary methods are already implemented for you in this file as discussed in the next section. You are strongly encouraged to use these functions instead of implementing your own. This will allow you to focus on the core segments of the assignment, avoid spending time on auxiliary paths, and also to avoid issues during grading due to bugs and misinterpretation of requirements of these utility functions.

All print statements must indicate the program that is responsible for generating them. To do this, please prefix your print statements with the program name i.e. `Driver` or `MemoryManager`. Section 4 below depicts these sample outputs.

2 Task Requirements

1. The Driver accepts one command line argument. This is the **seed** for the random number generator.

> **"Random" number generators and seeds**
> The random number generators used in software are usually not truly random. The generator is initialized with a "seed" value, then a mathematical formula generates a sequence of apparently random numbers. But, if you re-use the same "seed", you get that same sequence of numbers again.
>
> **Other uses of seeding the random number generator**
> Seeding the random number generator is useful for debugging in discrete event simulations (that are used to model complex phenomena) particularly stochastic ones. When a beta tester observes a problem in the program, you can re-create exactly the same simulation they were running. It can also be used to create a repeatable "random" run for timing purposes.
>
> Nota Bene: We will be using different "seeds" to verify the correctness of your implementation.

   In the `Driver.c` file, the seed is already set for the random number generator based on the command line argument that were provided.

```
srand(seed);
```

2. The driver will restrict the memory bounds that can be consumed by the process. These bounds have been set for you in the skeletal code that has been provided. Please do not change this.

3. The driver program will invoke the memory-manager. This has been already implemented for you in the `Driver.c` file.

```
int running_count = get_running_count();
printf("With seed: %d\t%d\n\n", seed, running_count);
```

4. The memory-manager uses the random number generator to compute the number of times that it must allocate and de-allocate arrays. The number of iterations should be between 100,000 (inclusive) and 120,000 (exclusive). A utility method called `get_iteration_count` is provided for you inside the `MemoryManager.c` skeleton file which will map a given random integer into the above range.

   Steps 5 through 9 (enumerated below) are repeated in a loop and the number of times the loop is executed is dependent on the random number that was returned.

   To generate a random number, invoke the `rand()` function available in `stdlib` library. This library is already included in the `MemoryManager.c` skeleton file.

5. The memory-manager uses the random number generator to compute the size of the array that must be allocated. The array size should be between 1000(inclusive) and 1500(exclusive). Again, another utility method named get_arr_size is provided in the skeleton file to port a random number to this range. The array should be allocated in the heap; failure to do so will result in a 5 point deduction.

   Allocating on the heap versus the stack
   An array is created on the heap by explicitly allocating memory using malloc or similar functions. On the other hand, allocating an array in the stack can be done as follows: int arr[num_of_elem];

   If memory is allocated on the heap, it should be released explicitly (e.g. using 'free') whereas memory is automatically released for stack variables when they go out of scope – hence the penalty.

6. After the memory-manager has allocated the array, it uses the random number generator to populate each element of the array.

7. This array is then sorted so that the numbers are in ascending order.

8. The median element of the array is retrieved. For simplicity, use the $\lfloor array\_size/2 \rfloor$ formula to calculate the index of the median in the sorted array irrespective of the number of elements in the array.

   Steps 7 and 8 are implemented in a utility method named `return_median` function in MemoryManager.c file for your convenience. You should pass the populated array and the number of elements in the array as input arguments when invoking this function.

9. Next, you must check if this median element is divisible by 13. If it is, you must increment the running count of such values by 1.

10. Once loop variable initialized in Step 4 has reached its limit, you exit from the loop and report on the total number of median elements that were divisible by 13.

3 Skeleton Code

Skeleton code is provided you with the following files. You do not need to use any other library while implementing the required tasks. All the required libraries are included in the skeleton files. The skeleton code can be downloaded from the course web site.

1. `Driver.c` – You are not required to modify this program.
2. `MemoryManager.c` – You need to implement the body of the function get_running_count in this file as per the instructions given above.
3. `MemoryManager.h` – This header files declares the methods exposed from MemoryManager.c, so that they can be invoked from the Driver program. You are not required to modify this file.

Please refer to the README.txt file inside the skeleton package on how to compile and run the program.

4 Example Outputs:

```
[Driver] With seed: 1234
[MemoryManager] Number of Iterations: 102414
[Driver] Running Count: 7850
```

```
[Driver] With seed: 7979
[MemoryManager] Number of Iterations: 100505
[Driver] Running Count: 7832
```

5 What to Submit

Use the CS370 *Canvas* to submit a single .zip file that contains:

- All .c and .h files related to the assignment (please document your code),
- a Makefile that performs both a *make clean* as well as a *make all,*
- a README.txt file containing a description of each file and any information you feel the grader needs to grade your program.

Filename Convention: You should keep the .c and .h filenames as they are in the skeleton code; any additional files can have the names you want. The archive file should be named as <FirstName>-<LastName>-HW1.zip . E.g. if you are Cameron Doe and submitting for assignment 1, then the zip file should be named Cameron-Doe-HW1.zip.

6 Grading

The assignments much compile and function correctly on machines in the CSB-120 Lab. Assignments that work on your laptop on your particular flavor of Linux, but not on the Lab machines are considered unacceptable.

This assignment will contribute a maximum of 5 points towards your final grade. The grading will also be done on a 5 point scale. The points are broken up as follows:

1 point each for correctly performing Task 1, 4, 5 and 9 (**4 points**)

1 point for getting the rest of the things right!

Deductions:
There is a 5-point deduction (i.e. you will have a zero on the assignment) if you:

(1) Modifying bounds specified in the `Driver` file.
(2) Allocate the array on the stack instead of the heap.
(3) Have an out of memory error or a segmentation fault: this is indicative of a memory leak, the very problem you are supposed to avoid.

You are required to **work alone** on this assignment.

7 Late Policy

Click here for the class policy on submitting late assignments.