# HW1: Programming Assignment v4 9.02.2019.7:00PM

## WORKING WITH MEMORY ALLOCATIONS AND DEALLOCATIONS

The objective of this assignment is to write and test a program with dynamic memory allocation.

Due Date: Thursday, September 12, 2019, 11:00 pm
Extended Due Date with 20% penalty: Friday, September 13, 2019, 11:00 pm

This document and the README.txt file are available at Canvas (Assignments > HW1)

## 1. Description of Task

For this assignment you will be working with two C files: `Invoker.c` and the `Executor.c`. It involves dynamically allocating and deallocating random sized arrays. You will use the Valgrind tool to ensure that there are no memory leaks.

**`Invoker:`** It is responsible for:

1. Setting the seed, whose value is passed as an argument, using srand().
2. Invoking functions in the `Executor`.

**`Executor:`** It is responsible for implementing the core functionality of this assignment, that is:

1. Dynamically allocating and de-allocating a random sized array for each iteration.
2. Populating elements in the array with random characters.
3. Counting the number of vowels and consonants in the array.
4. Computing the ratio vowel/consonants.
5. Track the iteration with maximum number of vowels in the array.
6. Obtaining the average value of vowels/consonants for all iterations and returning to `Invoker` and printing the iteration number with maximum vowels in the array.

All above tasks are implemented in `get_running_ratio()` and `Invoker` should call that function in the `Executor` file. The auxiliary methods that will be needed in the `Executor` are:

1. `int get_iteration_count(int rand)`: This function takes in a random value generated by rand() and returns a value between 50(inclusive) and 150(exclusive), which should be used as number of iterations.
2. `int get_arr_size(int rand)`: This function takes in a random value generated by rand() and returns a value between 100(inclusive) and 200(exclusive), which should be used as the array size for the array in one iteration. This function should be called once for each iteration.
3. `char get_arr_val(int rand)`: This function takes in a random value generated by rand() and generates a value between 97 (inclusive) and 123 (exclusive), which is then explicitly converted into a character and returned to the calling function. This assignment uses only lower case characters.
4. `float char_ratio(char *arr, int size, int *maxCountPointer)`: This function takes the populated array, it's size and the address to maxCount variable used in `get_running_ratio()`. It then counts the number of vowels and consonants, and returns the ratio of vowels/consonants. Also, it checks if the current value in address pointed by maxCountPointer is lesser than count of vowels in the current array (it means that the maximum number of vowels found in iterations so far is less than the current iteration). If yes, then it places the value of count in the address pointed by maxCountPointer.

**Hints**:

1. To generate a number between an inclusive lower bound and an exclusive upper bound using a random number generated by rand() you can use the following example.

```
int generator(int rand)
{
        return((rand % (upper_bound - lower_bound) + lower_bound) * 1);
}
```

2. To explicitly convert an integer to a character, you can use the following example.
```
int number = 97;
char character = (char)number;
```

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. `Invoker` or `Executor`. The example section below depicts these sample outputs.

Using Valgrind, ensure that there is no memory leak. Copy the Valgrind output indicating no leaks to README file. Then insert a memory leak by commenting out the code responsible for deallocation while ensuring that the program still functions as specified and copy the Valgrind output to README file. **Modify the program again so that it does not have a memory leak before submitting it by commenting the memory leak and placing a comment about it stating the below commented code is a memory leak.**

## 2. Task Requirements

1. The `Invoker` 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 actually pseudorandom. The generator is initialized with a "seed" value, then a mathematical formula generates a sequence of pseudorandom numbers. 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 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.
>
> We will be using different "seeds" to verify the correctness of your implementation.

In the `Invoker` file, the seed should be set for the random number generator based on the command line argument that is provided. The string/char* value received from the command line argument should be converted to integer using atoi() before being used to set seed using srand().

```
srand(seed);
```

The `Invoker` program should invoke the `Executor`.

```
double running_ratio = get_running_ratio();
printf("With seed: %d\t%d\n\n", seed, running ratio);
```

2. The `Executor` initializes maxCount and maxIteration in `get_running_ratio()` to 0 . This is used to track the maximum number of vowels in an array and the iteration number that the array belongs to . It then 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 50 (inclusive) and 150 (exclusive, i.e. not including 150). The auxiliary method called `get_iteration_count(int rand)` is to be used to map a given random integer into the above range. To generate a random number, invoke the `rand()`. Steps 5 through 9 (enumerated below)

are repeated in a loop and the number of times the loop is executed is dependent on the number of iterations that was returned.

3. The `Executor` uses the random number generator to compute the size of the array that must be allocated. The array size should be between 100 (inclusive) and 200 (exclusive). Again, auxiliary method called `get_arr_size(int rand)` is used to map a random number to this range. The `Executor` should allocate the memory in the heap; failure to do so will result in a 75-point deduction.

   **Allocating on the heap versus the stack**
   An array is created in 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

4. After the `Executor` has allocated the array, it uses the random number generator to populate each element of the array. The auxiliary method called `get_arr_val(int rand)` is used to map the random number to the range 97 (inclusive) and 123(exclusive). 97 is ASCII value for 'a' and 122 is the ASCII value for 'z'. This auxiliary method is called once for each element.

5. The `Executor` makes a temporary copy of maxCount and then calls `char_ratio(char *arr, int size, int *maxCountPointer)`, by passing it the array, size of the array and the address of maxCount. In this auxiliary function the number of vowels and consonants in the array is found, and so is the ratio vowels/consonants. It also compares the maximum number of vowels in all the iterations so far, which is the value stored in the address pointed by maxCountPointer. If the maximum number of vowels found so far is less than the count of the vowels in the current iteration, the value at the address pointed by maxCountPointer is changed to count.

6. Once the control returns from `char_ratio(char *arr, int size, int *maxCountPointer)` to `get_running_ratio()`, if the temporary copy of maxCount is not the same as maxCount, it means the modified the maxCount, so the current iteration as the highest count of vowels. Change the value of maxIteration to the current iteration number.

7. Once loop variable initialized in Step 4 has reached its limit, you exit from the loop and obtain the average value of the ratio.

8. In `Executor` print the iteration number with maximum vowels, and then return the average value of the ratio to `Invoker`. In `Invoker` print the average ratio. Check your values using provided sample output.

   **Testing for randomness**
   There exist a number of rigorous tests for randomness for sequences generated by pseudorandom generators. The test here is a rather simple one.

## 3. Files Provided

Files provided for this assignment include the description file (this file), a README file. This can be downloaded as a package from the course web site.

Please refer to the README.txt file inside the package on how to compile and run the program. You are needed to answer the questions in the README file.

## 4. Example Outputs:

```
1.   <system_name>:<folder_path> $ ./Invoker 3
[Invoker]: With seed: 3
[Executor]: Number of iterations is 96
[Executor]: Iteration with maximum vowel count is 27
[Invoker]: Running ratio: 0.244940

2.   <system_name>:<folder_path> $ ./Invoker 64
[Invoker]: With seed: 64
[Executor]: Number of iterations is 146
[Executor]: Iteration with maximum vowel count is 5
[Invoker]: Running ratio: 0.236935
```

**Sample Valgrind output:**

```
1. No leaks

==23191== HEAP SUMMARY:
==23191==     in use at exit: 0 bytes in 0 blocks
==23191==   total heap usage: 97 allocs, 97 frees, 15,013 bytes allocated
==23191==
==23191== All heap blocks were freed -- no leaks are possible
==23191==
==23191== For lists of detected and suppressed errors, rerun with: -s
==23191== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

2. With Leaks

==23221== HEAP SUMMARY:
==23221==     in use at exit: 13,989 bytes in 96 blocks
==23221==   total heap usage: 97 allocs, 1 frees, 15,013 bytes allocated
==23221==
.
.
.
==23221==
==23221== LEAK SUMMARY:
==23221==    definitely lost: 13,989 bytes in 96 blocks
==23221==    indirectly lost: 0 bytes in 0 blocks
==23221==      possibly lost: 0 bytes in 0 blocks
==23221==    still reachable: 0 bytes in 0 blocks
==23221==         suppressed: 0 bytes in 0 blocks
==23221==
==23221== For lists of detected and suppressed errors, rerun with: -s
==23221== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

## 5. What to Submit

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

- All .c and .h files listed below and descriptive comments within,
  - `Invoker.c`

  - `Executor.c`
  - `Executor.h` – This header files declares the methods exposed from `Executor.c`, so that they can be invoked from the `Invoker` program
- a Makefile that performs both a *make build* as well as a *make clean,*
- a README.txt file containing a description of each file and any information you feel the grader needs to grade your program, and
  - Valgrind outputs showing both no memory leaks and a memory leak
  - Answers for the 5 questions

For this and all other assignments, ensure that you have submitted a valid .zip/.tar file. After submitting your file, you can download it and examine to make sure it is indeed a valid zip/tar file, by trying to extract it.

**Filename Convention:** The archive file must be named as: <FirstName>-<LastName>-HW1.<tar/zip>. E.g. if you are John Doe and submitting for assignment 1, then the tar file should be named John-Doe-HW1.tar

## 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/Mac OS X, but not on the Lab machines are considered unacceptable.
The grading will also be done on a 100 point scale. The points are broken up as follows:

| Objective | Points |
|---|---|
| Correctly performing Tasks 1-8 (10 points each) | 80 points |
| Descriptive comments | 5 points |
| Correctly injecting and then fixing the memory leak, and providing copies of Valgrind outputs showing both no memory leaks and a memory leak was detected | 5 points |
| Questions in the README file | 5 points |
| Providing a working Makefile | 5 points |

**Questions:** (To be answered in README file. Each question worth 1 point)

1. Malloc allocates memory dynamically on heap? – True/False
2. When dynamically allocating an integer array, Malloc takes the number of elements as the input? – True/False
3. free() is defined inside which header file?
4. How many executable(s) are required to be generated by the Makefile for this assignment?
5. What command is used to call the default target in Makefile?

**Deductions:**
There is a 75-point deduction (i.e. you will have a 25 on the assignment) if you:

(1) Allocate the array on the stack instead of the heap.
(2) Have memory leak or a segmentation error which cannot be plugged by commenting the memory leak code provided, which is identified by placing a comment just above it.

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

## 7. Late Policy

Click here for the class policy on submitting late assignments.

**Revisions**: Any revisions in the assignment will be noted below.

9/2/19: The following revisions have been made in the assignment text.

Page 1 Hint 1 should be: "To generate a number between an inclusive lower bound and an exclusive **upper** bound using a random number generated by rand() you can use the following example"

Task requirement 1 should be "**double** running_ratio = get_running_ratio();"

Task requirement 2 should be "The number of iterations should be between 50 (inclusive) and 150 (exclusive, i.e. not including **150**)"