

**HW1: Programming Assignment** v1 1.28.21.2:00PM**WORKING WITH MEMORY ALLOCATIONS AND DEALLOCATIONS**

The objective of this assignment is to write and test a program with dynamic memory allocation. The program allocates several random sized arrays populated using randomly chosen characters for each iteration and computes the ratio vowels/consonants. You will also demonstrate how to use the Valgrind tool to test for memory leaks.

Due Date: Thursday, Feb 1, 2021, 11:00 pm

Extended Due Date with 20% penalty: Friday, Feb 2, 2021, 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: `Starter.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.

**Starter:** 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 vowels/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 `Starter` and printing the iteration number with maximum vowels in the array.

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

1. `int get_iteration_no(int rand)`: This function takes in a random value generated by `rand()` and returns a value between 50 (inclusive) and 100 (exclusive), which should be used as the 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 150 (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 65 (inclusive) and 90 (inclusive), which is then explicitly converted into a character and returned to the calling function. This assignment uses only upper case characters.
4. `float ratio(char *arr, int size, int *maxCountPointer)`: This function takes the populated array, its 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));
}

```

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. *Starter* 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 *Starter* 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 *Starter* 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 the seed using `srand()` and it should be printed.

```

srand(seed);
printf("With seed: %d\n", seed);

```

The *Starter* program should invoke the *Executor*.

```

double running_ratio = get_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 100 (exclusive, i.e. not including 100). The auxiliary method called `get_iteration_no(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 3 through 8 (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. Print the total number of iterations for this run.

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 150 (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 65 (inclusive) and 90 (inclusive). 65 is ASCII value for 'A' and 90 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 `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 `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 `ratio()` modified the `maxCount`, so the current iteration is 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 `Starter`. In `Starter` 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. [Starter]: With seed: 7  
 [Executor]: Number of iterations is 77  
 [Executor]: Iteration with maximum vowel count is 43  
 [Starter]: Running ratio: 0.241263
2. [Starter]: With seed: 91  
 [Executor]: Number of iterations is 85  
 [Executor]: Iteration with maximum vowel count is 59  
 [Starter]: Running ratio: 0.246685

#### Sample Valgrind output:

1. No leaks (with seed 7)

```
==2430878==
==2430878== HEAP SUMMARY:
==2430878==      in use at exit: 0 bytes in 0 blocks
==2430878==    total heap usage: 78 allocs, 78 frees, 10,366 bytes allocated
==2430878==
==2430878== All heap blocks were freed -- no leaks are possible
==2430878==
==2430878== For lists of detected and suppressed errors, rerun with: -s
==2430878== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

2. With leaks (with seed 7)

```
==2431104== HEAP SUMMARY:
==2431104==      in use at exit: 9,342 bytes in 77 blocks
==2431104==    total heap usage: 78 allocs, 1 frees, 10,366 bytes allocated
==2431104==
==2431104== LEAK SUMMARY:
==2431104==    definitely lost: 9,342 bytes in 77 blocks
==2431104==    indirectly lost: 0 bytes in 0 blocks
==2431104==    possibly lost: 0 bytes in 0 blocks
==2431104==    still reachable: 0 bytes in 0 blocks
==2431104==           suppressed: 0 bytes in 0 blocks
==2431104== Rerun with --leak-check=full to see details of leaked memory
==2431104==
==2431104== For lists of detected and suppressed errors, rerun with: -s
==2431104== ERROR SUMMARY: 0 errors from 0 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,

- Starter.c
- Executor.c
- Executor.h – This header file declares the methods exposed from Executor.c, so that they can be invoked from the Starter 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 must 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.

Jan 26: Example output has been corrected.

Ja 28: Valgrind output corrected.

