HW1: Programming Assignment v.1.23.2022.16:00 PM
WORKING WITH DYNAMIC MEMORY ALLOCATION

For this assignment you will write and test a program which uses dynamic memory allocation. The program allocates several random sized arrays with integers. For each array it counts the integers that are prime and the ones that which are not. It then computes the ratio of the two numbers. The program keeps track of the iteration with the largest number of prime numbers and computes a running average of all ratios across all the iterations. You will also demonstrate that you can use the Valgrind tool to test for memory leaks.

Due Date: Monday, January 31, 2022, 11:00 pm
Late Due Date with 20% penalty: Tuesday, February 01, 2022, 11:00 pm

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

1. Task Description

For this assignment you will have with two C files: `Starter.c` and the `Executor.c`. The program will dynamically allocate and deallocate 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 `Executor.c`.

`Executor:` It is responsible for implementing the following:
1. Dynamically allocate and deallocate a random sized array for each iteration.
2. Populate elements in the array with random integers.
3. For each iteration, check all the elements in the array and determine whether each element is prime or not, and if so, count it.
4. Calculate the ratio of count for prime numbers to count for non-prime numbers.
   ---- count of prime numbers / count of non-prime numbers----
5. At the end, print the iteration number with maximum count of prime numbers.
6. Return the average value of the ratio prime/non_prime for all iterations to `Starter`.

All above six tasks are implemented in `get_running_ratio()` and `Starter` should call that function in the `Executor` file. The auxiliary functions that will be needed in the `Executor` are:
1. `int random_in_range(int lower_bound, int upper_bound):` This function takes a lower limit (inclusive) and an upper limit (exclusive) and returns a random integer value between them.
2. `int get_prime_count(int *array, int arraySize):` This function takes the reference to the array and the array size as an input. You need to iterate over the elements of the array and check if they have prime numbers. Return the counted number for prime.

Hints:
1. To generate a random number between an **inclusive** lower bound and an **exclusive** upper bound using a random number generator, you can use the following example:
```
int random_in_range(int lower_bound, int upper_bound)
{
      return ((rand() % (upper_bound - lower_bound)) + lower_bound);
}
```
2. To check each element of an array, the below link can be useful:
   - Checking Prime Number-Programiz

All print statements must mention 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("[Starter]: With seed: %d\n", seed);
   ```

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

   ```
   float running_ratio = get_running_ratio();
   ```

2. The `Executor` initializes **maxPrimeCount** and **maxCountIteration** in `get_running_ratio()` to 0. These are used to track the maximum count of prime numbers in an array and the iteration number that the array belongs to. The `Executor` 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 `random_in_range(int lower_bound, int upper_bound)` is called for the range specified above.

Steps 3 through 7 (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. For the loop, the iteration should start from 1, so the range of the loop can be described as [1, the random number between 50 and 100]. Note that "[" or "]" means that end of the range is inclusive. Print the total number of iterations.

```
printf("[Executor]: Number of iterations is: %d\n", totalIterations);
```

3. In `Executor` use the random number generator to compute the size of the array between 100 (**inclusive**) and 150 (**exclusive**). Use the auxiliary method `random_in_range(int lower_bound, int upper_bound`. 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 allocating the array, use the random number generator to populate it with elements between 50 (**inclusive**) and 199 (**inclusive**). Again, use the auxiliary method `random_in_range(int lower_bound, int upper_bound)`. This auxiliary method is called once for each element.

   *Note the bounds to obtain the random integer are both inclusive. Think about how to use the random_in_range() function to include 199 and **not** excluded it!*

5. The `Executor` calls `get_prime_count(int *array, int arraySize)`, by passing it the array and the size of the array. The auxiliary function returns the count of prime numbers.

6. Once the control returns to `get_running_ratio()`, calculate the ratio of the number of count for prime number to number of non-prime.

   Maintain a running sum of the ratios for the final average.

7. Further, the `Executor` checks if the returned count of prime numbers is greater than the previously stored maximum count (maxPrimeCount). If it is true, we update the values maxPrimeCount and maxCountIteration accordingly.

8. Once loop variable has reached its limit, you exit from the loop. In `Executor` print the iteration number with the maximum count of prime numbers.

9. Return the average value of the ratio prime/not_prime for all iterations to `Starter`

10. 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: 370
[Executor]: Number of iterations is: 72
[Executor]: Iteration with maximum prime count is 41
[Starter]: Running ratio: 0.263645
```

2.
```
[Starter]: With seed: 1031
[Executor]: Number of iterations is: 59
[Executor]: Iteration with maximum prime count is 56
[Starter]: Running ratio: 0.263244
```

Sample Valgrind output:

1. No leaks (with seed 370)

```
==3822050== HEAP SUMMARY:
==3822050==     in use at exit: 0 bytes in 0 blocks
==3822050==   total heap usage: 73 allocs, 73 frees, 37,220 bytes
allocated
==3822050==
==3822050== All heap blocks were freed -- no leaks are possible
==3822050==
==3822050== For lists of detected and suppressed errors, rerun with: -s
==3822050== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

2. With leaks (with seed 370)

```
==3822574== HEAP SUMMARY:
==3822574==     in use at exit: 36,196 bytes in 72 blocks
==3822574==   total heap usage: 73 allocs, 1 frees, 37,220 bytes allocated
==3822574==
==3822574== LEAK SUMMARY:
==3822574==    definitely lost: 36,196 bytes in 72 blocks
==3822574==    indirectly lost: 0 bytes in 0 blocks
==3822574==      possibly lost: 0 bytes in 0 blocks
==3822574==    still reachable: 0 bytes in 0 blocks
==3822574==         suppressed: 0 bytes in 0 blocks
==3822574== Rerun with --leak-check=full to see details of leaked memory
==3822574==
==3822574== For lists of detected and suppressed errors, rerun with: -s
==3822574== 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 files 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 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 1-10 (8 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. What is the main difference between Malloc() and Calloc()?
2. What is the main advantage of using dynamic memory allocation?
3. What happens if you attempt to free the memory already freed?
4. What is the purpose of the headerfile Executor.h, and why is Starter.h not necessary?
5. Describe the * and & operators in the context of pointers and references?

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/clarifications in the assignment will be noted below.

- Jan 21, 2022: Example output has been corrected.
- Jan 22, 2022: "Maintain a running sum of the ratios the final average." has changed to "Maintain a running sum of the ratios for the final average." at step 6.
- Jan 22, 2022: "*Steps 3 through 7 (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.* " has been elaborated as "*Steps 3 through 7 (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. For the loop, the iteration should start from 1, so the range of the loop can be described as [1, the random number between 50 and 100]. Note that "[" or "]" means that end of the range is inclusive. Print the total number of iterations.*" at the second paragraph of step 2.
- Jan 23,2022: Step 8 of Section 2 (Task Requirements) should be, "In Executor print the iteration number with the maximum count of prime numbers."