

## HW1: Programming Assignment v.9/5/2024 9PM

### WORKING WITH DYNAMIC MEMORY ALLOCATION

For this assignment you will design, code and test a program which uses dynamic memory allocation. to allocate several random sized arrays with integers. For each array, it counts the number of prime numbers. It then computes the ratio of the count of primes to the count of non-primes (composite). The program keeps track of the iteration with the largest number of primes in an array and computes a running average of all ratios for all the iterations. You will also demonstrate the use of the Valgrind tool to test for memory leaks.

Due Date: September 11, 2024, 11:00 pm

Late penalty: 10% per day until September 13, 2024, 11:00 pm

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

#### 1. Task Description

You will have with two C files: `Primes.c` and the `Driver.c`. The program will dynamically allocate and deallocate random sized arrays. In addition to testing it for correct functionality, you will use the Valgrind tool to ensure that there are no memory leaks.

**Driver.c:** It is responsible for:

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

**Primes.c:** 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 primes to composites given by  
$$\text{Ratio} = \text{count of primes} / \text{count of composites}$$
5. At the end, print the iteration number with maximum count of primes.
6. Return the average value of the ratio for all iterations to `Driver`.

All above six tasks are implemented in `get_running_ratio()` and `Driver` should call that function in the `Primes` file. The auxiliary functions that will be needed in the `Primes` 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 how many primes they contain. Return the number of primes counted.

#### 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. There are several ways to test a number for primality. This link can be useful: [Prime Number program in C](#)

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. `Driver` or `Primes`. 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 `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 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 `Driver` 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. You can assume that the program will be passed the correct number of arguments and that the arguments will be in the correct format (matching those shown in the Example Outputs in Section 4).

```
srand(seed);  
printf("[Driver]: With seed: %d\n", seed);
```

The `Driver` program should invoke `Primes`.

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

2. `Primes` initializes **maxPrimeCount** and **maxCountIteration** in `get_running_ratio()` to 0. These are used to track the maximum count of primes in an array and the iteration number that the array belongs to. `Primes` 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("[Driver]: Number of iterations is: %d\n", totalIterations);  
_ratio();
```

3. In `Primes`, use the random number generator to compute the size of the array between 150 (**inclusive**) and 200 (**exclusive**). Use the auxiliary method `random_in_range(int lower_bound, int upper_bound)`. The `Worker` 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 `Primes` 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 in the array.
6. Once the control returns to `get_running_ratio()`, calculate the ratio.

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

7. Further, `Primes` must check 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 `Primes` print the iteration number with the maximum ratio of primes to composites.
9. Return the average value of the ratio primes/composites for all iterations to `Driver`
10. In `Driver` 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 also required to answer the questions in the README file.

### 4. Example Outputs:

```
a. Result of running $ ./Driver 1000
[Driver]: Seeding RNG with seed: 1000
[Primes]: Number of iterations is: 90
[Primes]: Iteration with MAX prime count: 72
[Driver]: AVG prime/composite ratio: 0.262219

b. Result of running $ ./Driver 906
[Driver]: Seeding RNG with seed: 906
[Primes]: Number of iterations is: 81
[Primes]: Iteration with MAX prime count: 67
[Driver]: AVG prime/composite ratio: 0.256258
```

#### Sample Valgrind output:

##### 1. No leaks (with seed 12345)

```
$ valgrind run 12345
==3276193== Memcheck, a memory error detector
==3276193== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et
al.
==3276193== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright
info
==3276193== Command: run 12345
==3276193==
[Driver]: Seeding RNG with seed: 12345
[Primes]: Number of iterations is: 99
[Primes]: Iteration with MAX prime count: 56
[Driver]: AVG prime/composite ratio: 0.259702
==3276193==
==3276193== HEAP SUMMARY:
==3276193==      in use at exit: 0 bytes in 0 blocks
==3276193==    total heap usage: 100 allocs, 100 frees, 71,012 bytes
allocated
==3276193==
==3276193== All heap blocks were freed -- no leaks are possible
==3276193==
==3276193== For lists of detected and suppressed errors, rerun with: -s
==3276193== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

##### 2. With leaks (with seed 12345)

```

$ valgrind run 12345
==3276441== Memcheck, a memory error detector
==3276441== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et
al.
==3276441== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright
info
==3276441== Command: run 12345
==3276441==
[Driver]: Seeding RNG with seed: 12345
[Primes]: Number of iterations is: 99
[Primes]: Iteration with MAX prime count: 56
[Driver]: AVG prime/composite ratio: 0.259702
==3276441==
==3276441== HEAP SUMMARY:
==3276441==       in use at exit: 69,988 bytes in 99 blocks
==3276441==   total heap usage: 100 allocs, 1 frees, 71,012 bytes
allocated
==3276441==
==3276441== LEAK SUMMARY:
==3276441==   definitely lost: 69,988 bytes in 99 blocks
==3276441==   indirectly lost: 0 bytes in 0 blocks
==3276441==   possibly lost: 0 bytes in 0 blocks
==3276441==   still reachable: 0 bytes in 0 blocks
==3276441==   suppressed: 0 bytes in 0 blocks
==3276441== Rerun with --leak-check=full to see details of leaked memory
==3276441==
==3276441== For lists of detected and suppressed errors, rerun with: -s
==3276441== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)

```

## 5. What to Submit

Use the CS370 *Canvas* to submit a single .zip file (not .tar) that contains:

- All .c and .h files listed below and descriptive comments within,
  - Driver.c
  - Primes.c
  - Primes.h – This header files declares the methods exposed from Primes.c, so that they can be invoked from the Driver program
- a Makefile that performs *make*, *make clean*, and *make tar*
- a README.txt file containing a description of each file and any information you feel the grader needs to grade your program, including the Valgrind outputs showing both no memory leaks and a memory leak

For this and all other assignments, ensure that you have submitted a valid .zip/.tar file. After submitting your file, you should download it and examine to make sure it is indeed a valid zip/tar file, by trying to extract it. If your program does not decompress properly, and we are unable to run it, you will receive a 0 on this assignment.

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

## 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 1-10 (Points for each task can be vary)	85 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
Providing a working Makefile	5 points

### 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.

9/4, 9/5 Updated example output to match assignment requirements.