CS370 Fall 2025

HW1: Programming Assignment v.9/16/2025 8:57AM

DYNAMIC MEMORY ALLOCATION

You will design, code, and test a program that uses dynamic memory allocation to allocate several randomly sized arrays with integers. For each array, it counts the number of **perfect squares**. It then calculates the ratio of the number of perfect squares to the number of non-perfect squares. The program keeps track of the iteration with the largest number of perfect squares (squares) in an array and computes a running average of all ratios for all the iterations.

Due Date: Thursday, September 18, 2025, 11:00 pm
Late penalty: 10% per day until Saturday, September 20, 2025, **11:00 pm**

This document and the Makefile file are available at Canvas (Assignments > HW1)

1. Task Description

You will have with two C files: MemoryManager.c and the Driver.c. The program will dynamically allocate and deallocate random sized arrays with **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 MemoryManager.c

MemoryManager.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 square or not, and if so, count it.
4. Calculate the ratio of perfect squares to non-perfect squares given by
   Ratio = count of squares / count of non-perfect squares
5. At the end, print the iteration number with the maximum ratio of squares to non-squares.
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 MemoryManager file. The auxiliary functions that will be needed in the MemoryManager file 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 count_perfect_sqr(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 squares they contain. Return the number of squares 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. The C language has a math.h header file which may be of assistance when calculating square roots. This link may be helpful.

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 MemoryManager. The example section 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 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 samme simulation they were running. It can also be used to create a repeatable "random" run for timing purpose.

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 MemoryManager.

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

2. MemoryManager initializes **maxSquareCount** and **maxCountIteration** in *get_running_ratio()* to 0. These are used to track the maximum count of squares in an array and the iteration number to which the array belongs. MemoryManager then uses the random number generator to compute the number of times that it must allocate and deallocate arrays. The number of iterations should be between 50 (**inclusive**) and 200 (**exclusive**, i.e. not including 200). 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 0, so the range of the loop can be described as [0, (the random number between 50 and 200) - 1]. Note that "[" or "]" means that end of the range is inclusive. Print the total number of iterations.

3. In MemoryManager, use the random number generator to compute the size of the array between 50 (inclusive) and 200 (exclusive). Use the auxiliary method random_in_range(int lower_bound, int upper_bound). The MemoryManager should allocate the memory in the heap using int* tracked_malloc(int size).

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. The assignment must provide a robust and traceable memory management layer to prevent leaks and ensure allocation safety.

4.1 Safe Allocation Function

Description: The assignment shall provide a function to safely allocate memory for integer arrays. This function must perform a NULL check upon allocation and terminate the program gracefully with a descriptive error message if the system is out of memory, preventing undefined behavior.

int safe_malloc(int size);*
Parameters:
size: The number of integer elements to allocate.
Returns: A pointer to the allocated memory block.
Failure: The function does not return. It prints an error to stdout and terminates the program with EXIT_FAILURE.

4.2 Allocation Tracking

Description: The assignment shall maintain an internal counter to track the net number of active memory allocations in real-time for diagnostic purposes.This function will call safe_malloc to allocate memory and increment the internal memory_allocations counter.

int tracked_malloc(int size);*
Parameters:
size: The number of integer elements to allocate.
Returns: A pointer to the allocated memory block (inherits behavior from safe_malloc).

4.3 Tracked Deallocation

Description: The assignment shall provide a function to deallocate memory that correctly maintains the internal allocation counter, i.e., decrementing the internal memory_allocations counter.

*void tracked_free(void *ptr, int size);*
Parameters:
ptr: Pointer to the memory block to free. Handles NULL safely.
Returns: void

4.4 Leak Detection & Reporting

Description: The assignment shall provide a function to generate a final memory usage report, indicating if all allocated memory was successfully freed or if potential leaks are detected.

void print_memory_summary(); Output: Prints a summary to stdout showing the count of remaining allocations and a clear warning if that count is greater than zero.

In MemoryManager, use the random number generator to compute the size of the array between 50 (**inclusive**) and 200 (**exclusive**). Use the auxiliary method random_in_range(int lower_bound, int upper_bound). The code must allocate the memory in the heap.

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 that 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 MemoryManager calls count_perfect_sqr(int *array, int size), by passing it the array and the size of the array. The auxiliary function returns the count of square 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, MemoryManager must check if the returned count of square numbers is greater than the previously stored maximum count (maxSquareCount). If it is true, we update the values max_ratio and max_iteration accordingly.
8. Once loop variable has reached its limit, you exit from the loop. In MemoryManager print the iteration number with the maximum ratio of squares to non-squares.
9. Return the average value of the ratio squares/non-squares for all iterations to Driver
10. In Driver print the average ratio. Check your values using provided sample output.

3. Files Provided

Files provided for this assignment include the description file (this file) and a Makefile. This can be downloaded as a package from the course website. Please use the Makefile to compile and run the program.

Example Outputs:

a. If there is no memory leak the Result of running `$ ./Driver 1000` should be as follows:

```
[Driver] With seed: 1000
[MemoryManager] Number of iterations is: 90
[MemoryManager] Iteration with MAX perfect_sqr/non_perfect_sqr ratio: 9
[MemoryManager] Final summary - 0 memory blocks still allocated
[MemoryManager] All memory successfully freed
[Driver] AVG perfect_sqr/non_perfect_sqr ratio: 0.049688
```

b. If there is memory leak the Result of running `$ ./Driver 906` should be:

```
[Driver] With seed: 906
[MemoryManager] Number of iterations is: 131
[MemoryManager] Iteration with MAX perfect_sqr/non_perfect_sqr ratio: 100
[MemoryManager] WARNING: Potential memory leak detected!
[Driver] AVG perfect_sqr/non_perfect_sqr ratio: 0.050213
```

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
  - MemoryManager.c
  - MemoryManager.h – This header files declares the methods exposed from MemoryManager.c, so that they can be invoked from the Driver program
- a Makefile that performs *make*, *make clean,* and *make zip*
- a README.txt file containing a description of each file.

For this and all other assignments, ensure that you have submitted a valid .zip file. After submitting your file, you should download it and examine to make sure it is indeed a valid zip 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 file should be named John-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/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**
Successfully generating an error-free executable file	25 points
Creating seed properly	10 points
Successful Iterations	10 points
Check Max Iteration	15 points
Successful implementation of Avg Ratio	30 points
Check Memory Summary	10 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.
3. 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.

No	Changes	Revision
1	Example output corrected and updated	v.9/13/2025 12:37PM
2	Updated to uniform random number ranges	v.9/13/2025 12:37PM
3	Naming mismatch of method name (count_perfect_sqr and get_square_count) fixed	v.9/13/2025 12:37PM
4	`Worker` replaced by `code` to avoid confusion	v.9/13/2025 12:37PM
5	***"[Driver] AVG perfect_sqr/non_perfect_sr ratio"*** changed to ***"[Driver] AVG perfect_sqr/non_perfect_sqr ratio"***	v.9/13/2025 8:40PM
6	Fixed the start and end position of iteration. It should start at 0 and ends at (random number - 1) instead of (1, random number)	v.9/14/2025 2:24PM
7	Change the text in the 1Task Descript (MemoryManager) to *"with the maximum ratio of squares to non-squares."*	v.9/15/2025 12:05PM

8	Changed the documentation of `tracked_free` method's Returns to `void` instead of `A pointer to the allocated memory block (inherits behavior from safe_malloc).`	v.9/16/2025 8:57PM