**Lab 1.5 – Exploring Memory Management**

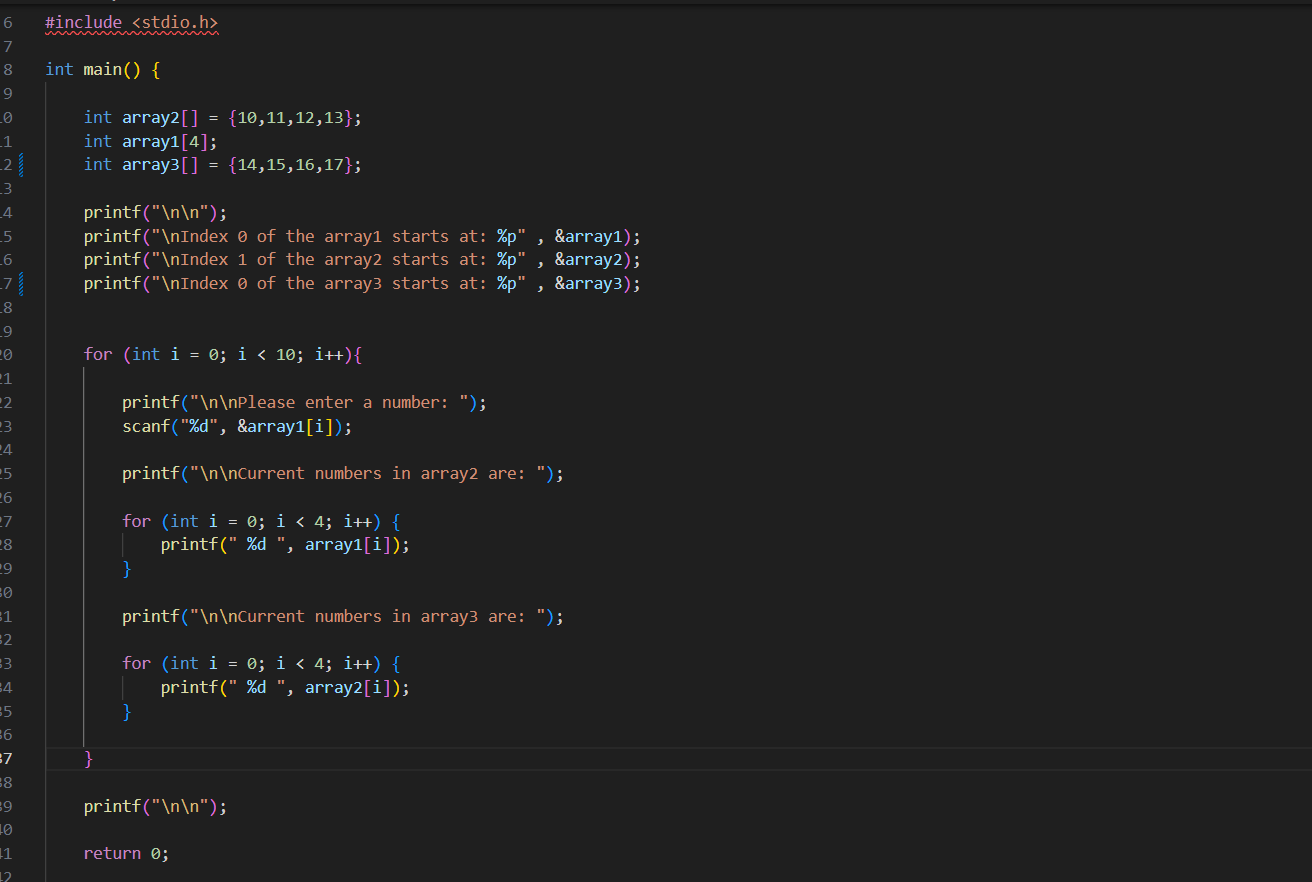
Github repo link:

Part 01

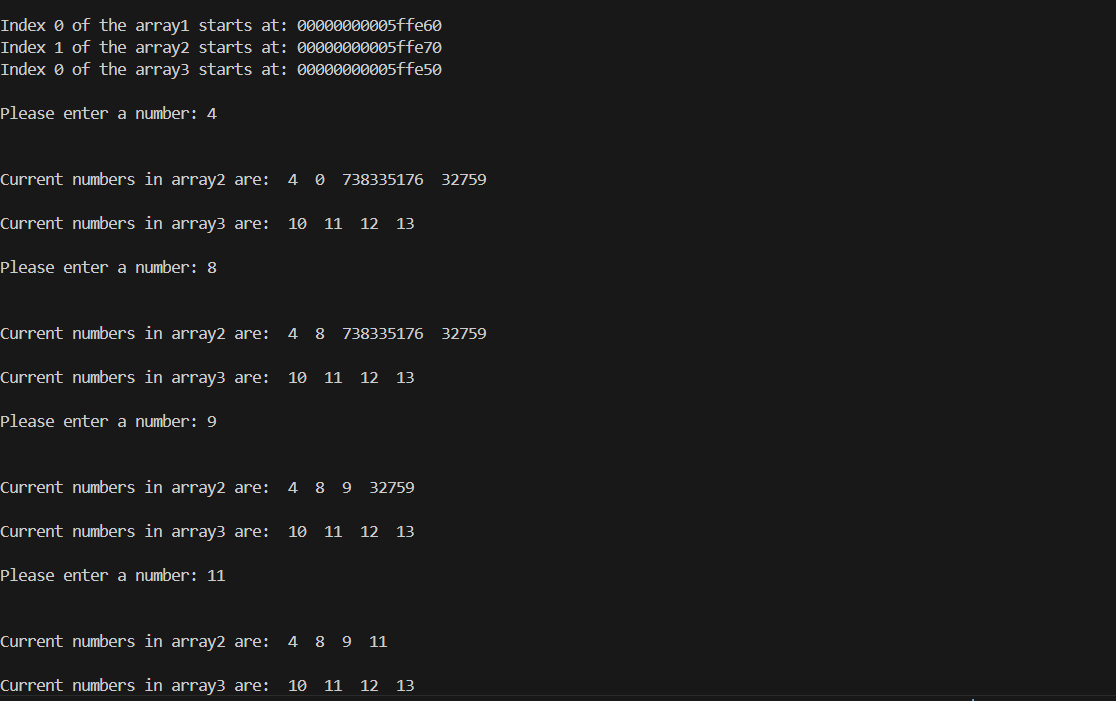
**1-usestaticarray.c :**

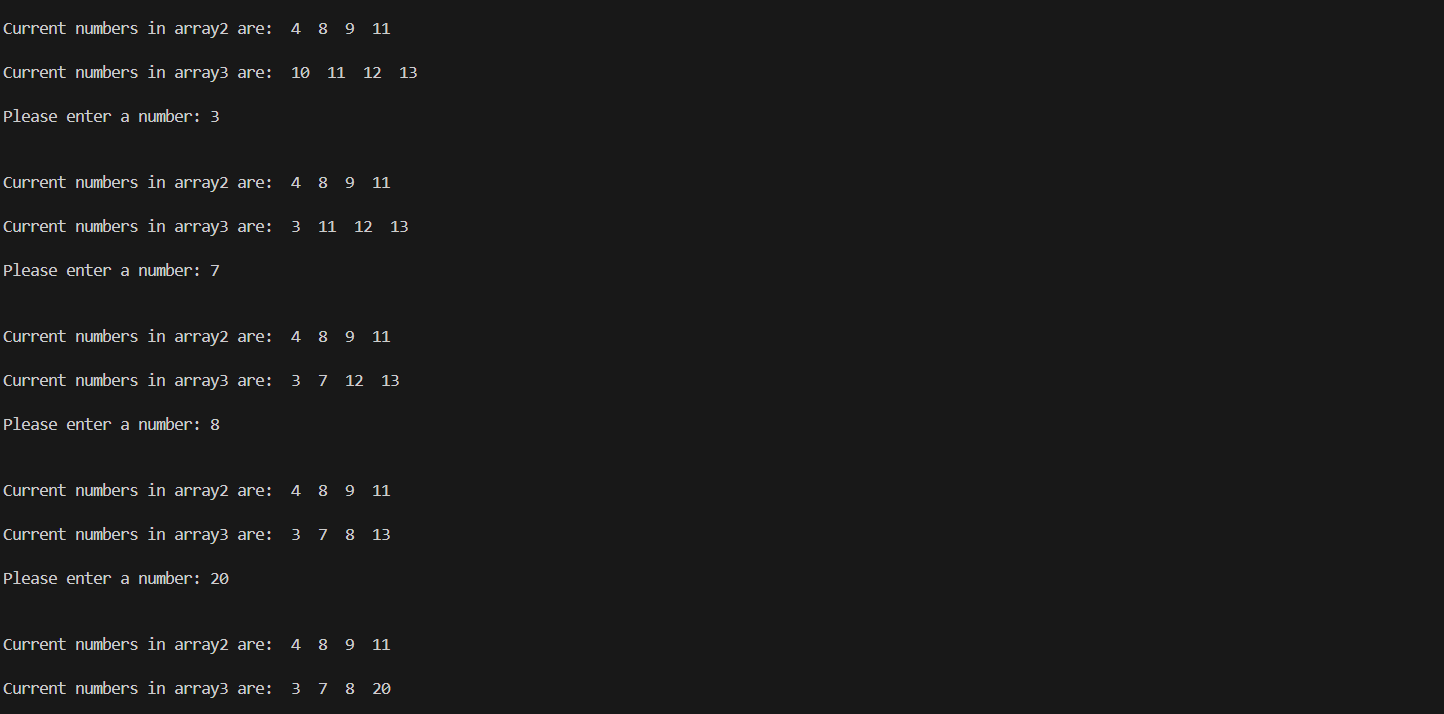
In C, arrays are statically allocated structures, meaning their size is fixed at compile time. However, C does not enforce strict array bounds checking, allowing unintended memory access beyond allocated limits. This behavior can cause unexpected results, such as modifying adjacent variables or structures in memory. In the given code, entering more values than array1 can hold may overwrite adjacent arrays (array2 or array3), demonstrating how memory layout influences data integrity. Additionally, different operating systems allocate memory differently, leading to variations in behavior across platforms.

Code



Output



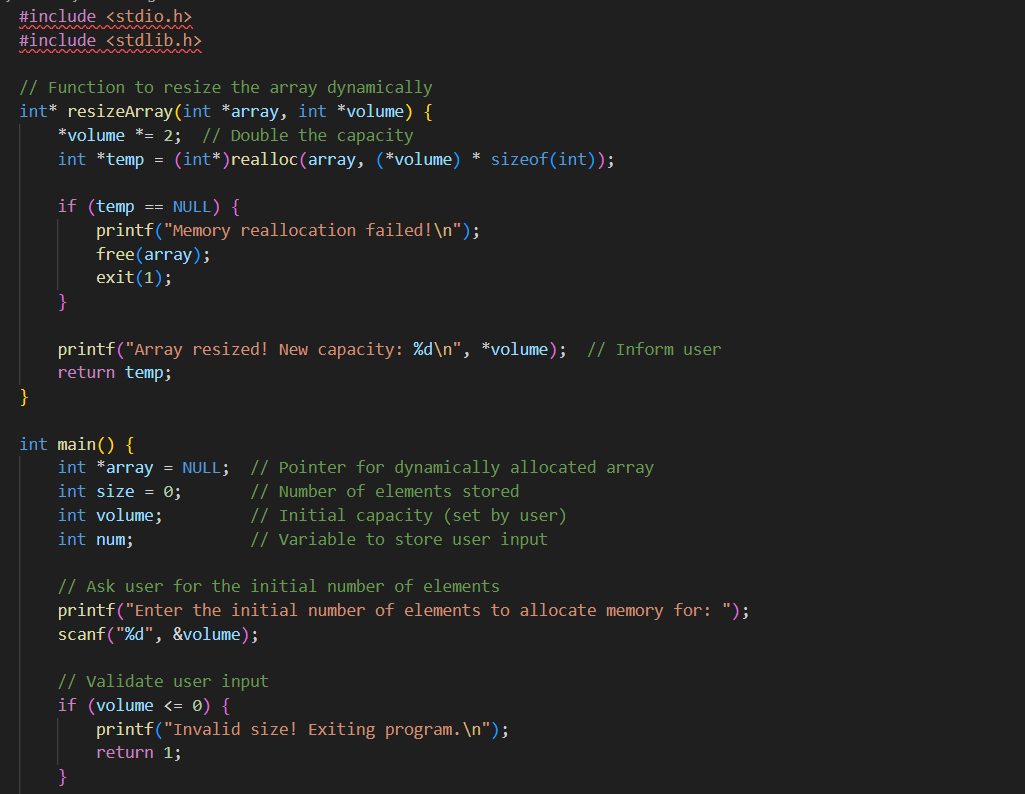


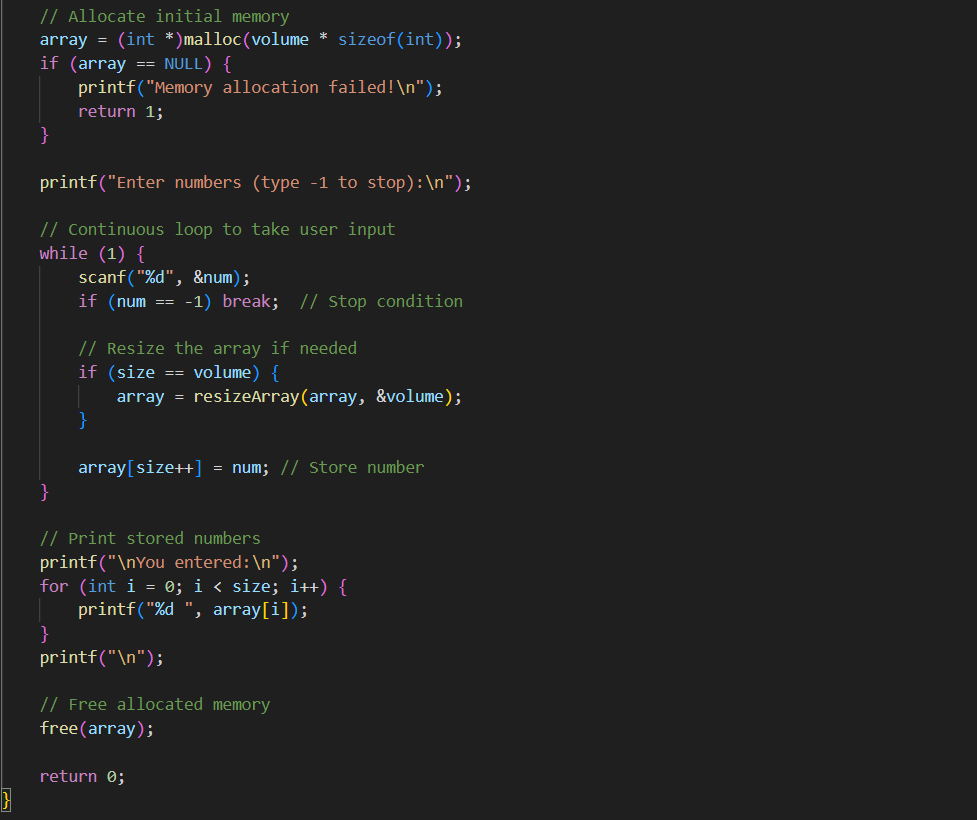
**2-mallocprinter.c :**

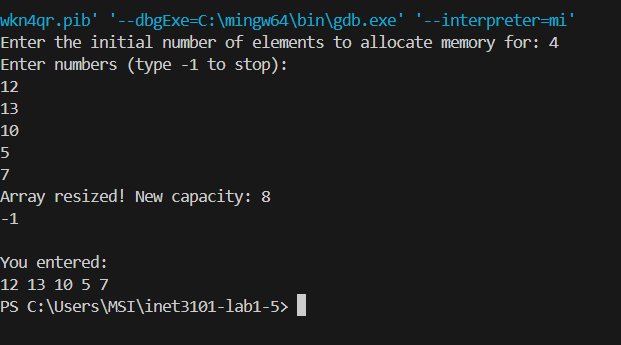


**3-dynamically-allocating.c :**

The code works by dynamically allocating memory for an array based on the user’s initial input. Instead of a fixed starting size, the program first asks the user how many elements they want to begin with. A while (1) loop is then used to continuously accept numbers until the user enters -1. If the array reaches its current capacity, the resizeArray function is called to **double** the array’s size using realloc(), ensuring there is enough space for additional numbers. By managing memory dynamically, we allow the program to scale based on user input while preventing unnecessary reallocation when not needed. The program also prints a message each time the array is resized, helping the user understand how memory is being managed. Finally, all allocated memory is freed at the end to prevent memory leaks.





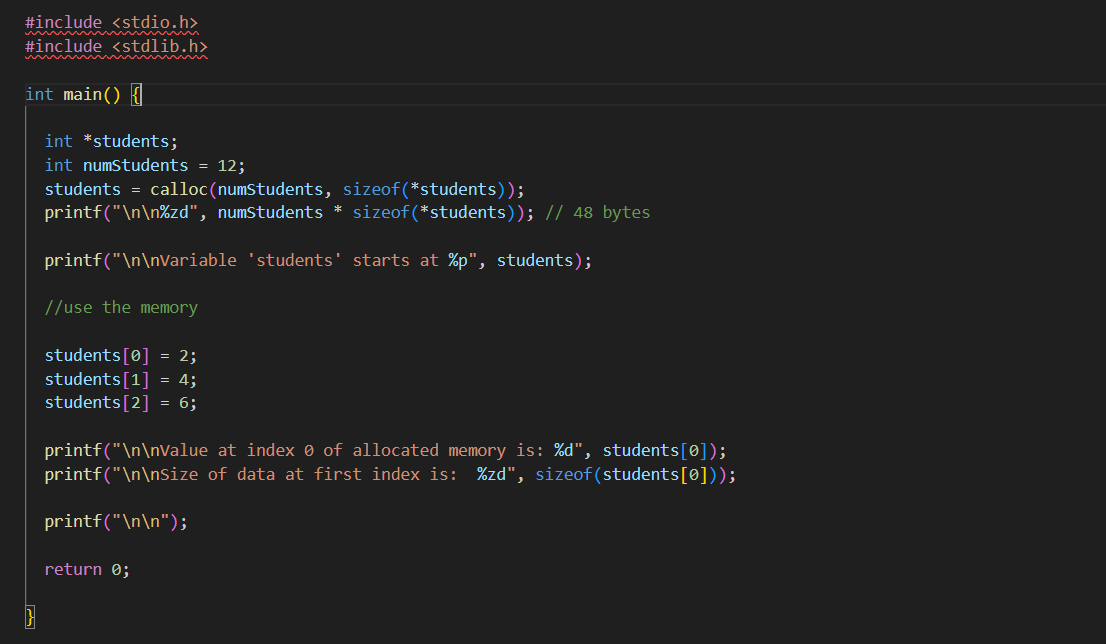


Code workflow

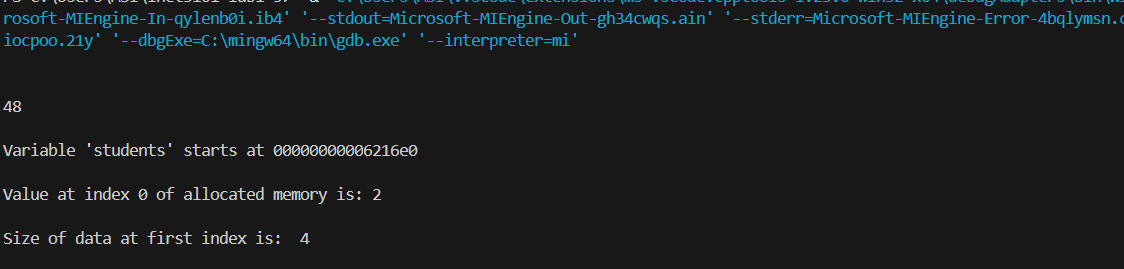
1. Starts by asking the user how many elements they want initially.
2. Uses a while(1) loop to continuously accept numbers until -1 is entered.
3. Checks if the array is full before adding a new element—if it is, realloc() is used to double the capacity dynamically.
4. Prints a message whenever the array is resized, so the user knows when memory is expanding.
5. Prevents memory leaks by calling free(array) at the end.

**4-simple-dynamic-example.c :**

**Code**

****

**Output**



**5-python-lists-are-objects-not-arrays.py:**

1. What is an Object in Object-Oriented Programming (OOP)?

In Object-Oriented Programming (OOP), an object is an instance of a class that contains both attributes and methods/functions. Objects provide a way to model real-world entities by collecting related properties and actions together. Code becomes more modular, reusable, and manageable as a result of its support for fundamental OOP concepts like encapsulation, inheritance, and polymorphism.

1. How Does a Python List Embody OOP Concepts?

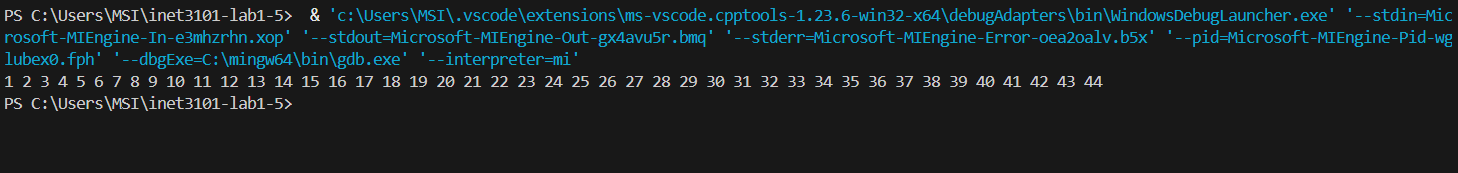
Since a Python list is an instance of the built-in list class, it is an object. It holds information (list elements) and offers methods (such as append(), sort(), and insert()) that let you work with that information. A Python list abstracts complicated actions like resizing, sorting, and insertion and manages memory dynamically, in contrast to a basic C-style array, which is merely a contiguous block of memory. This illustrates encapsulation, in which consumers communicate via clearly defined ways while the core operations of list management are concealed.

**6-linkedlist.c :**

1. How Does a Linked List Solve Dynamic Memory Problems?

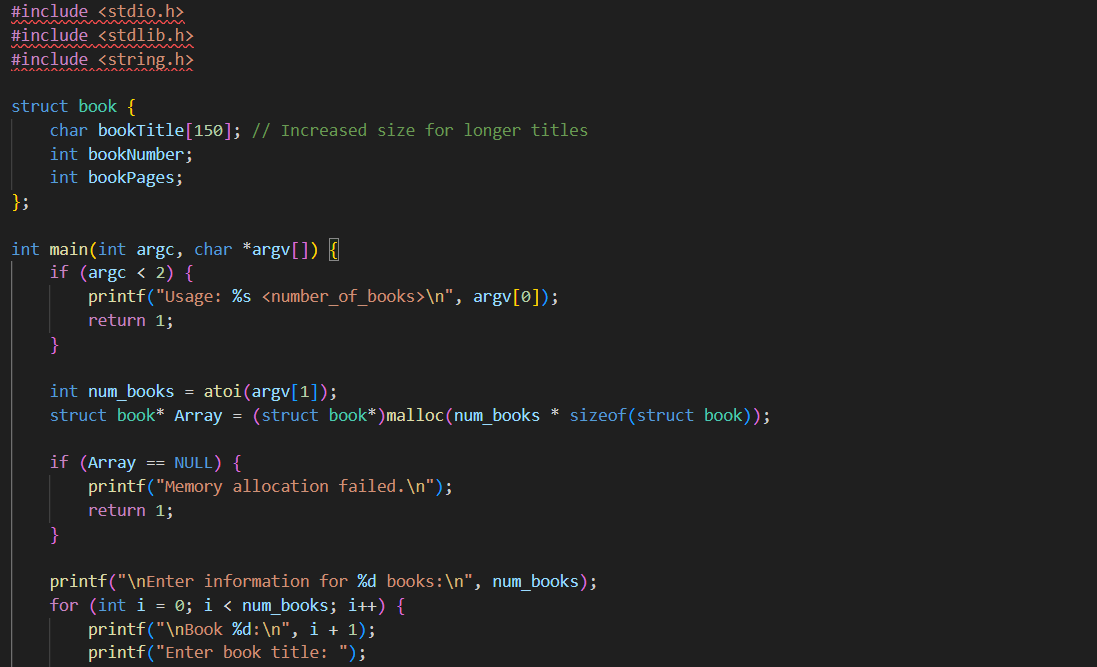
By enabling the effective insertion and deletion of elements without the need for contiguous memory blocks, a linked list aids in dynamic memory management. Linked lists reduce memory fragmentation problems by allocating memory for each node separately as needed, in contrast to arrays, which have a fixed size or need expensive resizing operations (malloc/realloc). Because nodes in this method are connected by pointers, moving items during insertion or deletion is avoided. Linked lists are helpful for frequent insertions and deletions but not for random access activities since they add additional memory overhead from pointer storage and may result in slower traversal than arrays.

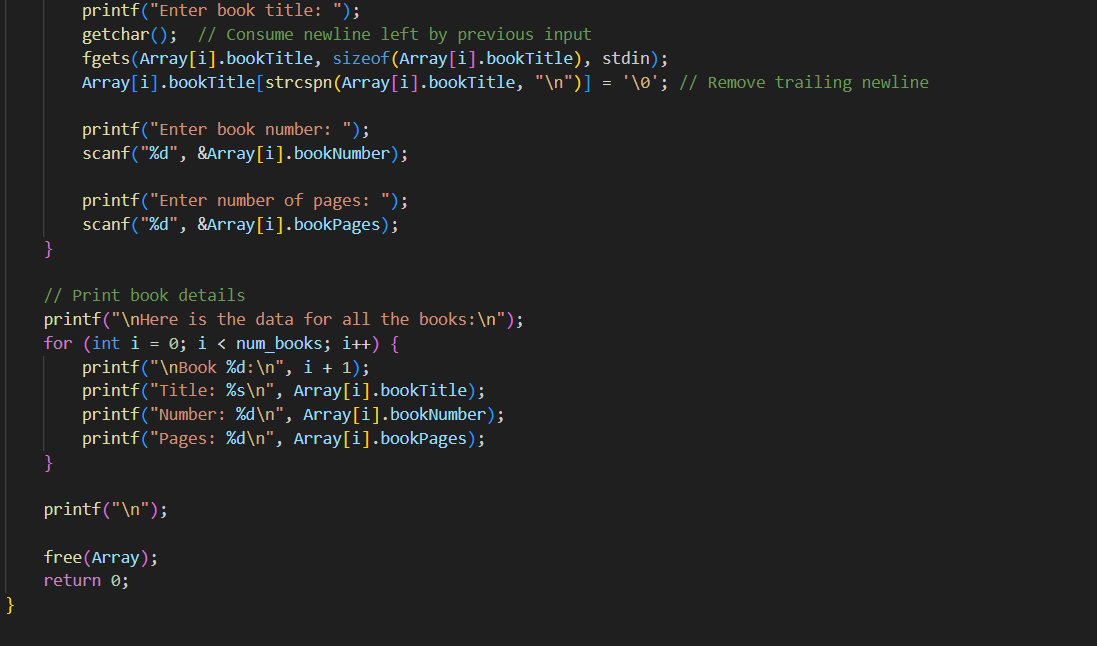
**7-managing-allocation-with-struct.c:**



**Mallocofstructs.c :**

Code





Output



Workflow of the Script

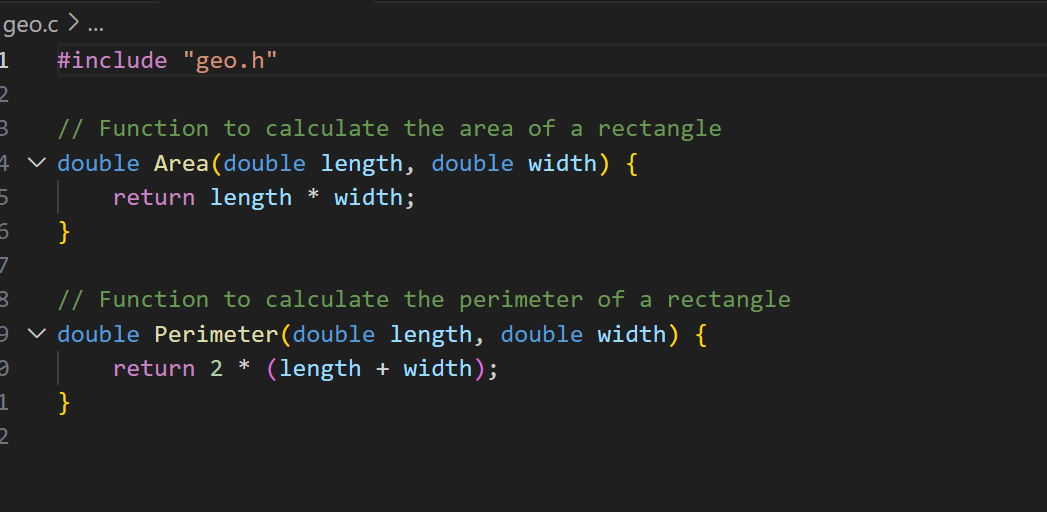
1. Take Input for Number of Books
2. Allocate Memory for Book Records
3. Collect Book Details
   * A loop runs num\_books times, prompting the user to enter:
     + Book Title (using fgets() to allow spaces)
     + Book Number
     + Number of Pages
4. Display Entered Data
5. Free Allocated Memory

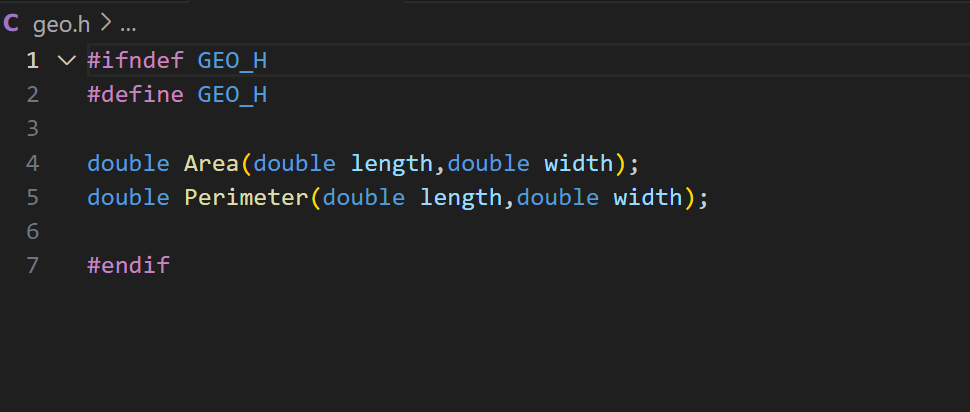
**header-examples :**

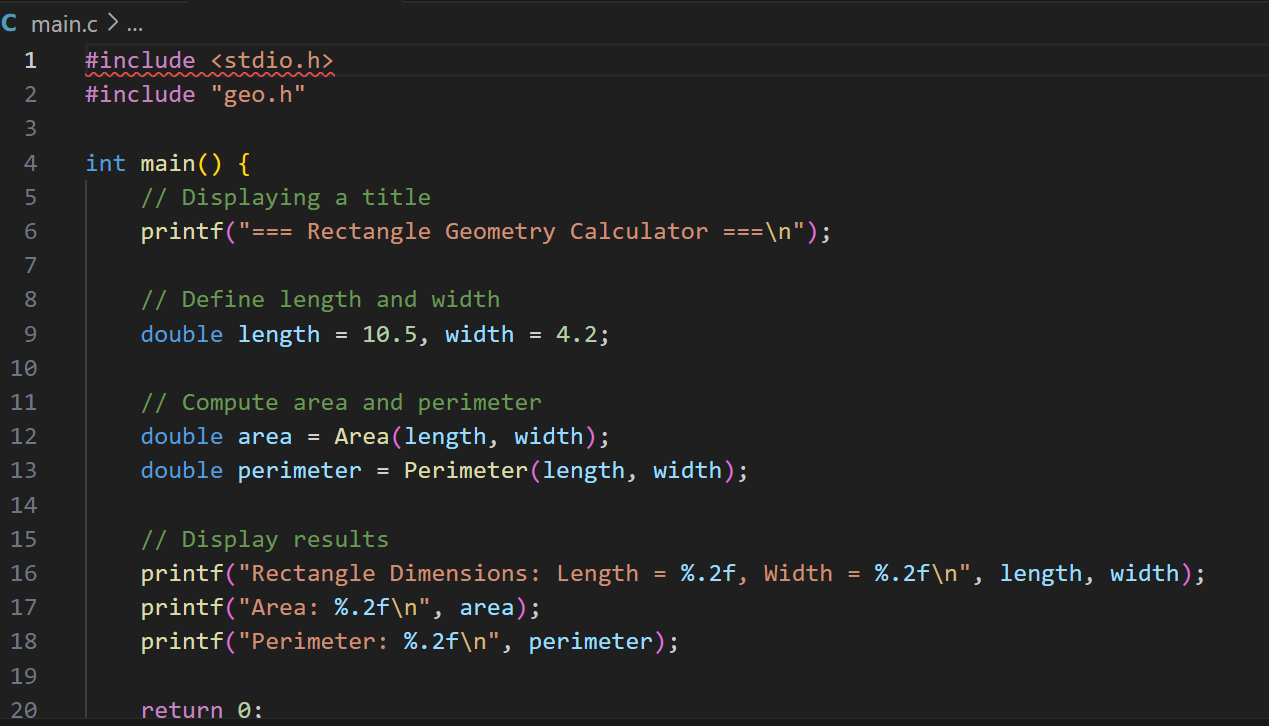
output for given code



Updated Code







Outptut

