**Khaalid Abdirahman**

**1-usestaticarray.c**

A screenshot of a computer screen

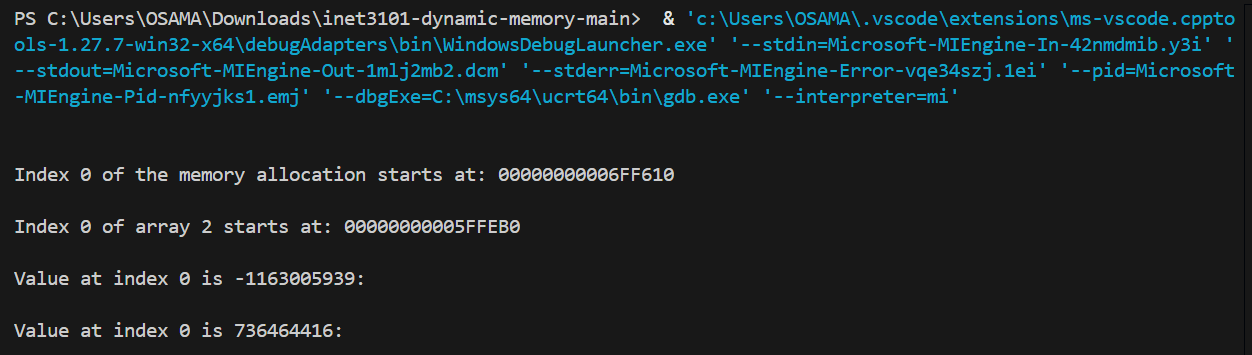
AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

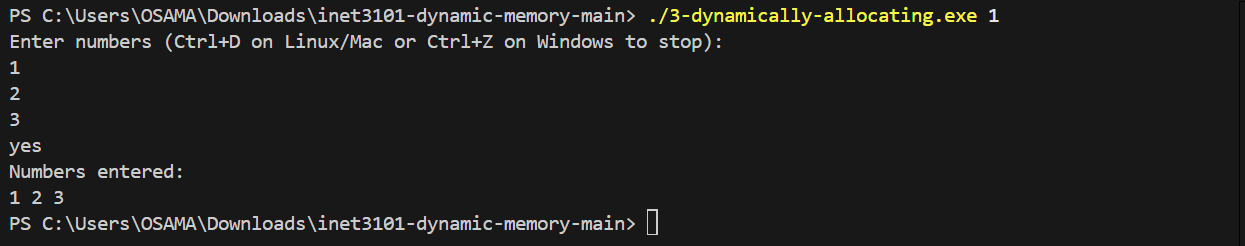
This program demonstrates why C arrays are considered *static structures*. The array array1 was only allocated 4 integers, but the program allows us to write more than 4 values into it. Since there are no runtime bounds checks in C, extra values overwrite neighboring memory locations — in this case, array2. This is a dangerous practice because it causes **buffer overflows**, which can corrupt data, crash programs, or even open security vulnerabilities. The key lesson is that arrays in C do not automatically prevent out-of-bounds access, so the programmer is responsible for ensuring safe indexing.

**2-usesdynamicarray.c**



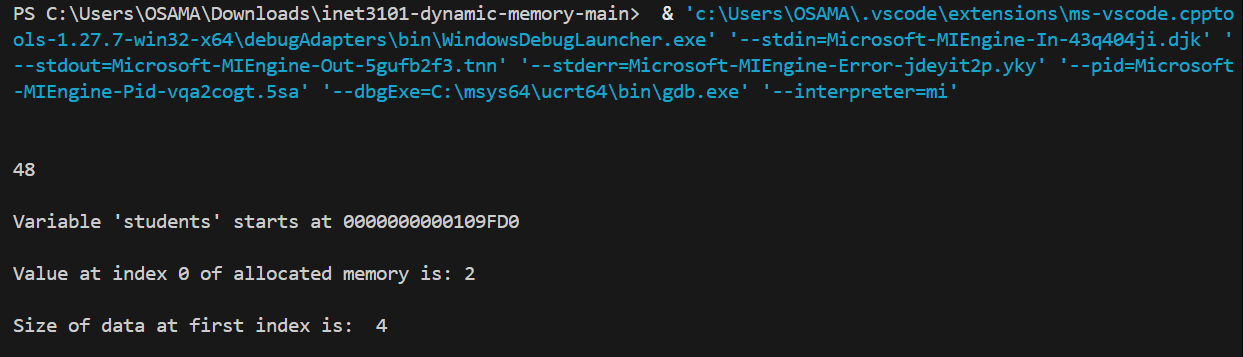
This example demonstrates the difference between allocating memory with malloc and declaring a static array. Both approaches reserve a block of memory that we can access with indexes, but the lifetime of that memory differs. A static array (array2) is automatically cleaned up by the operating system once the program ends, while dynamically allocated memory (array1) persists until explicitly freed with free(). If the programmer forgets to call free(), this leads to a **memory leak**, where memory remains reserved even though the program is no longer using it. This highlights why dynamic memory management in C requires careful handling by the programmer.

**3-dynamic-allocation.c**

****

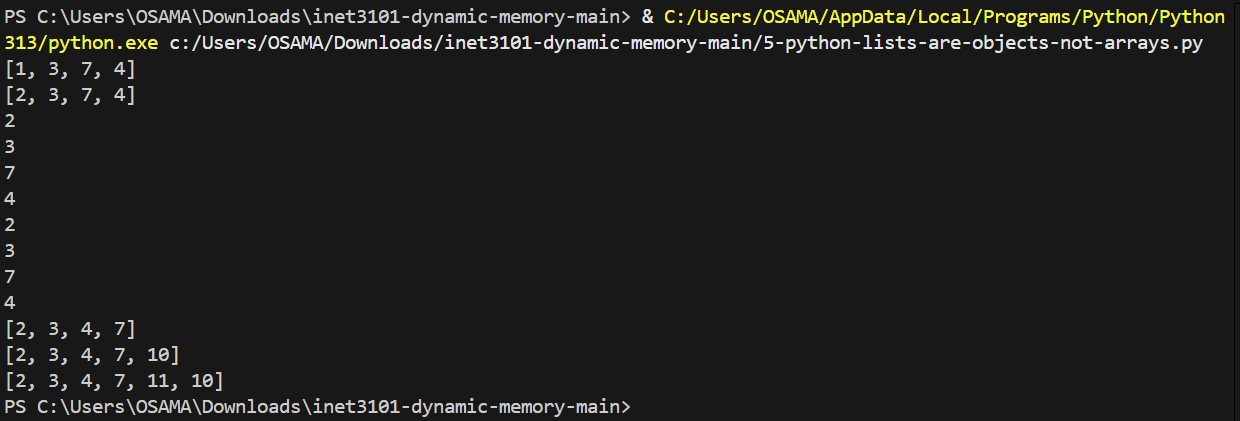
In this program, I modified the input logic from a for loop to an infinite while loop, allowing the user to keep entering numbers without a predefined limit. I used a **capacity** variable to track how much memory was currently allocated and a **size** variable to track how many numbers had been entered. When the number of inputs exceeded the capacity, I used realloc() to double the capacity, effectively resizing the array while preserving existing values. This approach solves the problem because it allows the program to grow dynamically as the user enters more data, instead of being limited to the original allocation size. The use of realloc() ensures we don’t lose previously entered values while expanding the array.

**4-usescalloc.c**



This program demonstrates the use of calloc() for dynamic memory allocation. Unlike malloc(), which allocates memory but leaves it uninitialized, calloc() allocates memory and also initializes all bytes to zero. In this example, memory for 12 integers (48 bytes) is allocated, and we can then use array-like indexing to assign and access values. The program shows the starting memory address of the allocation and verifies that the allocated elements behave like a normal array. The key difference is that calloc() ensures all values are set to zero initially, which can prevent unpredictable behavior from using uninitialized memory.

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



In Object-Oriented Programming (OOP), an **object** is an instance of a class that bundles both **data** (attributes) and **behavior** (methods) into a single entity. Objects allow us to model real-world concepts in code by encapsulating state and functionality together. A Python List is an example of an object. It not only stores data (like an array in C) but also provides built-in methods such as .sort(), .append(), and .insert() that manage memory resizing, shifting elements, and other operations internally. This demonstrates how Python abstracts away low-level memory management, making lists easier and safer to work with compared to static arrays in C.

**6-linkedlist.c**

**A screen shot of a computer

AI-generated content may be incorrect.**

A linked list solves some of the same problems as dynamic memory because it allows us to create data structures whose size can change during program execution. Like malloc, calloc, or realloc, each new node in a linked list is allocated dynamically at runtime, so we are not restricted to a fixed array size. Unlike arrays, linked lists don’t require shifting elements when inserting at the beginning or middle; instead, pointers are adjusted to link new nodes. This makes them more flexible for insertions and deletions. However, linked lists trade off efficiency in random access (no direct indexing) and use more memory due to storing pointers, which is why modern languages like Python often provide higher-level dynamic collections that hide these low-level details.

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

A computer screen with blue and white text

AI-generated content may be incorrect.

This example shows how a struct can be used to manage dynamic memory in C by keeping track of the data, its size, and the number of items currently in use. The addToList function demonstrates how memory can be resized automatically when more space is needed, similar to how Python or Java handle dynamic lists internally. Functions for deleting or inserting would follow the same idea: shifting items in memory while updating numItems, allowing the list to grow or shrink as needed without the programmer having to manually manage raw memory each time.

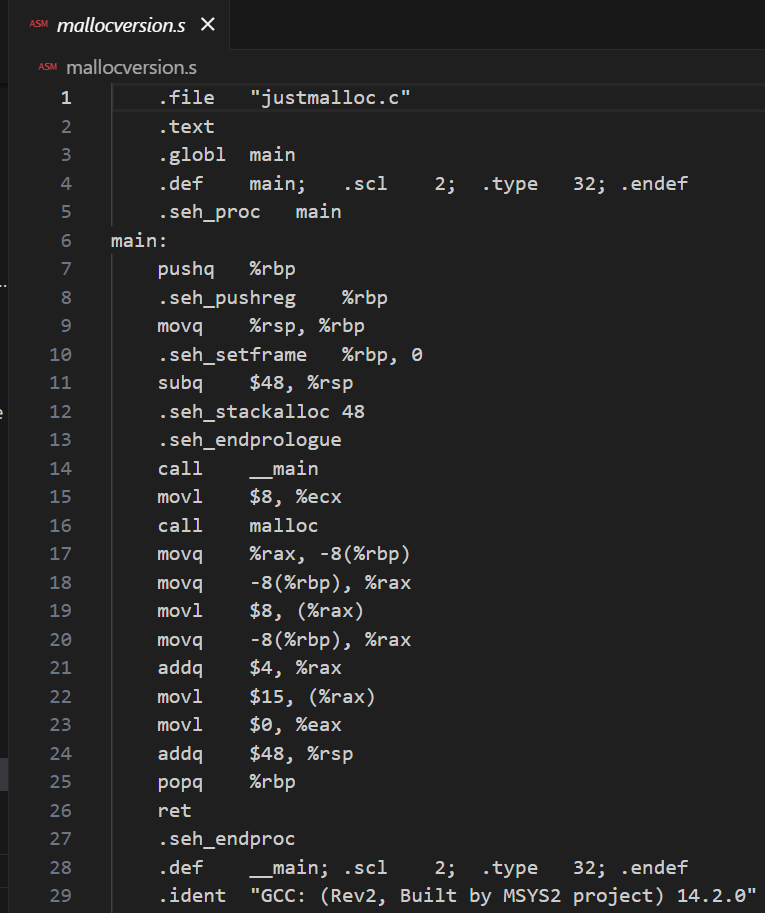
**Justarray.c**

**A screenshot of a computer program

AI-generated content may be incorrect.**

When this program is compiled to assembly, both the array (int array[] = {8, 15};) and the variable (int x = 15;) are just memory locations with values stored in them. From the CPU’s perspective, there is no difference between memory allocated as an array, with malloc, or as a simple variable. In all cases, the compiler translates the code into instructions that reserve space in memory and store integers, so the distinction only matters at the C language level, not at the hardware level.

**Justmalloc.c**

****

In this code, memory is allocated dynamically at runtime using malloc instead of declaring a static array. We request space for two integers, then assign values to them with array[0] = 8; and array[1] = 15;. If you assemble this code and compare it to the previous example with a static array, the assembly looks different at first because malloc is a function call. However, the end result is the same from the CPU’s perspective: values are stored in memory at specific addresses and accessed using pointers. Whether the memory is created at compile-time (array) or runtime (malloc), the CPU still just reads and writes integer values at allocated memory locations.

**Mallocofstructs.c**

**A computer screen with white text

AI-generated content may be incorrect.**

This modified program uses a **real-world structure** (struct book) to store information about books, such as title, inventory number, and page count. Unlike the earlier simple example with just a character and integer, this struct represents meaningful data and demonstrates how dynamic memory allocation can manage collections of complex objects. By letting the user input and store multiple books, it simulates how programs like library systems or inventory managers keep track of data, showing why structs combined with dynamic memory are powerful in C.