**Lab 1.5**

1.

#include <stdio.h>

#define SIZE 4 // Define a fixed size for the array

int main() {

int array1[SIZE];

int array2[] = {10, 11, 12, 13};

printf("\n\n");

printf("\nIndex 0 of array1 starts at: %p", (void\*)&array1);

printf("\nIndex 0 of array2 starts at: %p", (void\*)&array2);

for (int i = 0; i < SIZE; i++) {

printf("\n\nPlease enter a number: ");

scanf("%d", &array1[i]);

}

printf("\n\nCurrent numbers in array1: ");

for (int i = 0; i < SIZE; i++) {

printf(" %d ", array1[i]);

}

printf("\n\nCurrent numbers in array2: ");

for (int i = 0; i < SIZE; i++) {

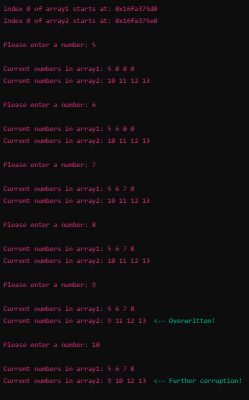
printf(" %d ", array2[i]);

}

printf("\n\n");

return 0;

}



3.

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

if (argc < 2) {

printf("Usage: %s <starting\_size>\n", argv[0]);

return 1;

}

int size = atoi(argv[1]); // Initial allocation size

int capacity = size; // Capacity keeps track of allocated memory

int \*array = (int \*)malloc(capacity \* sizeof(int));

if (array == NULL) {

printf("Memory allocation failed!\n");

return 1;

}

int count = 0;

int num;

printf("Start entering numbers (enter -1 to stop):\n");

while (1) {

printf("Enter a number: ");

scanf("%d", &num);

if (num == -1) // Stop condition

break;

if (count == capacity) {

capacity \*= 2; // Double the capacity

array = (int \*)realloc(array, capacity \* sizeof(int));

if (array == NULL) {

printf("Memory reallocation failed!\n");

return 1;

}

}

array[count++] = num;

}

printf("Numbers entered:\n");

for (int i = 0; i < count; i++) {

printf("%d ", array[i]);

}

printf("\n");

free(array); // Free dynamically allocated memory

return 0;

}

Approach:

Initial Allocation: The user provides the starting size via command-line arguments.

Infinite Input Loop: Uses a while(1) loop to continuously ask for input.

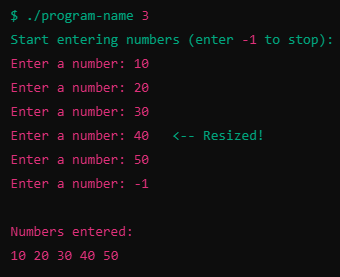
Resizing Strategy:

If the array reaches its capacity, realloc() is used to double the size.

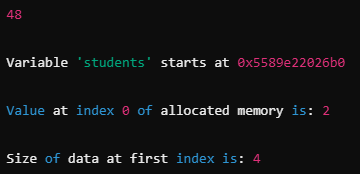
This prevents frequent resizing and improves performance.

Stopping Condition: The user can enter -1 to stop.

Memory Management: free() is called at the end to prevent memory leaks.



4.



5.

In Object-Oriented Programming (OOP), an object is a self-contained unit that bundles both data and functions that operate on that data. Think of it as a real-world object, like a car, that has both attributes (color, brand, speed) and behaviors (accelerate, brake, turn). In programming, an object is created from a class, which acts like a blueprint. This approach makes code more modular, reusable, and easier to manage. Instead of writing separate functions that manipulate raw data, objects keep everything organized within a single structure, making it easier to work with complex systems.

A Python list is not just a basic collection of values; it is an object with built-in functionality that makes working with lists easier. When you create a list, Python automatically provides a set of methods (like .append(), .sort(), and .insert()) that allow you to modify the list without worrying about low-level memory management. This is very different from how arrays work in C, where you need to manually allocate and resize memory. For example, when you use list.append(10), Python handles everything behind the scenes, allocating memory if needed, resizing the list, and ensuring everything runs smoothly. This is an example of encapsulation, where the complex details of how lists grow and shrink are hidden from the programmer. It also demonstrates abstraction, as Python provides an easy-to-use interface while taking care of the internal logic. This is why Python lists are more than just arrays, they are fully functional objects that make programming more efficient and intuitive.

6.

Using a linked list helps solve some of the same problems that dynamic memory allocation addresses, particularly when it comes to flexibility and efficient memory management. In a traditional array, memory is allocated in a fixed, contiguous block, meaning that resizing requires allocating a new block and copying data. This can be inefficient and lead to wasted memory.

A linked list, on the other hand, uses nodes that are dynamically allocated as needed, with each node pointing to the next one. This means that the list can grow or shrink without needing to reallocate or move memory, making it more memory-efficient in situations where the size of the data set is unknown or constantly changing. Additionally, inserting or deleting elements in a linked list is more efficient than in an array, as it does not require shifting elements. However, linked lists come with their own overhead, such as additional memory usage for pointers and slower access times compared to arrays due to lack of direct indexing.