Dynamic Memory Allocation

By: **Annamalai A** with **GPT** assist

Dynamic Memory Allocation (DMA) in C refers to the process of requesting memory from the **heap** during program execution rather than deciding the required size at compile time. This approach allows programs to adapt to variable-sized inputs and avoid the rigid limitations of static memory allocation. Unlike fixed-size arrays declared at compile time, dynamic memory can grow or shrink as needed, offering much better flexibility and memory efficiency.

Motivation for Dynamic Memory Allocation

Programs that rely solely on static arrays often face two major issues:

1. **Overflow risk:**

   If the array is too small for incoming data, the program may attempt to write beyond its bounds.

   Since C does not perform automatic bounds checking, this can lead to buffer overflows, crashes, or security issues.

2. **Memory wastage:**

   If an overly large array is declared “just in case,” unused memory becomes wasted space, especially in memory-constrained environments such as embedded systems or low-RAM devices.

Dynamic memory allocation solves these problems by allowing memory to be allocated **only when needed** and released when no longer in use.

Static vs Dynamic Memory Allocation

In **static memory allocation**, array sizes must be known at compile time and cannot be modified during execution. This is suitable only for fixed-size problems.

In contrast, **dynamic memory allocation** delays the decision of required memory until runtime. Programs can request memory blocks of any size, use them as needed, and then return them to the system when finished.

This flexibility enables better performance, scalability, and memory usage, especially when working with unpredictable or large datasets.

Understanding the Stack and Heap

Memory in a running C program is broadly divided into two areas:

The Stack

The stack stores local variables, function parameters, return addresses, and function call states.

It operates in a **Last In, First Out (LIFO)** manner and grows downward towards lower memory addresses.

Memory on the stack is automatically freed when a function returns.

Although stack allocation is very fast, the stack is limited in size and unsuitable for large or dynamic data structures. Excessive usage may result in **stack overflow**.

The Heap

The heap is used for dynamic memory allocation and grows upward toward higher memory addresses.

Unlike the stack, the heap is manually managed by the programmer. Memory allocated on the heap persists until it is explicitly freed, regardless of which function allocated it.

The heap offers greater flexibility but requires careful management to avoid memory leaks, dangling pointers, or fragmentation.

`malloc()` — Memory Allocation

`malloc()` is used to allocate a block of memory on the heap. The allocated memory is **uninitialized** and may contain garbage values.

Example:

```
int *p = (int*) malloc(5 * sizeof(int));
```

`malloc()` returns the address of the first byte of the allocated block, or `NULL` if allocation fails. The programmer must explicitly assign values before using this memory.

Because it allocates only raw bytes, `malloc()` is suitable for dynamic arrays and custom data structures where initialization is handled manually.

`calloc()` — Contiguous Allocation

`calloc()` allocates multiple blocks of memory and initializes all the bytes to **zero**.

It takes two arguments: the number of elements and the size of each element.

Example:

```
int *p = (int*) calloc(5, sizeof(int));
```

This function is ideal when zero-initialized memory is required, such as arrays, buffers, or structures that must begin in a clean state.

Although slightly slower than `malloc()` due to the zero-initialization step, `calloc()` is safer for beginners and numerical applications.

`malloc()` vs `calloc()`

Feature	malloc()	calloc()
Initialization	No	Yes (zeros)
Arguments	One	Two (num, size)
Safety	Lower	Higher
Use case	General purpose	Arrays, clean memory

`realloc()` — Resizing Allocated Memory

`realloc()` is used to change the size of an already allocated memory block.

It can expand or shrink the memory region while preserving the existing data.

Example:

```
p = realloc(p, new_size);
```

If the memory block cannot be extended in-place, `realloc()` allocates a new block, copies old data, and frees the old block automatically.

If reallocation fails, it returns `NULL` but does **not** free the original memory, so best practice is to use a temporary pointer.

`realloc()` is essential for implementing dynamic arrays or buffers that must grow based on input size.

`free()` — Releasing Memory

`free()` is used to return previously allocated memory back to the heap.

Example:

```
free(p);
```

Failing to free memory leads to **memory leaks**, where unused memory remains allocated until the program terminates. This degrades performance and can exhaust heap space in long-running applications.

Every block allocated with `malloc()`, `calloc()`, or `realloc()` must eventually be released using `free()`.