Static Memory (Stack) In C:

int arr[5]; // fixed 5 items

- Allocated at compile-time on stack.

Dynamic Memory (Heap)

- **In C**
  - **`malloc` →**
  - **`calloc` →**
  - **`realloc` →**
  - **free**

Memory Layout Diagram

```
|------------------|
|  Code Segment    |  <- program instructions
|------------------|
|  Global/Static   |  <- global variables
|------------------|
|       Heap       |  <- grows upward (malloc, calloc, realloc)
|                  |
|                  |
|       Free Space |
|                  |
|------------------|
|       Stack      |  <- grows downward (local variables, function calls)
|------------------|
```

Stack = Short-term (automatic, fixed)
Heap = Long-term (manual, flexible)

Why Dynamic Memory Allocation?

In short: **Dynamic memory = "on-demand" memory, not fixed upfront.**

Technical (Programming) Reasons

1.Variable size arrays

```
int arr[100]; // fixed size at compile time
```

What if user needs 5 numbers? (95 wasted) Or 1000 numbers? (overflow!)
With dynamic memory:

```
int *arr = malloc(n * sizeof(int)); // n decided at runtime
```

2.Linked data structures

- Arrays are continuous and fixed size.
- If you want **linked list, trees, graphs**, you need **nodes created dynamically**.

Example:

```
struct Node {
   int data;
   struct Node *next;
};
struct Node *head = malloc(sizeof(struct Node));
```

3. **Reusable memory**
    - You can allocate when needed, free when done.
    - Prevents memory waste.
    - Important in **embedded systems** (like automotive ECUs, IoT devices) where memory is limited.

Ex: Fixed (static allocation)

```
#include <stdio.h>
int main() {
    int arr[100];   // fixed at compile-time
    int n;
    scanf("%d", &n);
    // if user enters n=5, 95 wasted; if n=200, overflow
}
```

1. What is malloc()?

- **malloc** = **Memory Allocation**
- It is used to **allocate memory at runtime** (from the heap).
- The memory size is given in **bytes**.
- It returns a **pointer** to the first byte of the allocated block.

2. Syntax

```
ptr = (cast-type*) malloc(size_in_bytes);
```

- `ptr` → pointer that will store the address of allocated memory.
- `(cast-type*)` → type of data (`int*`, `float*`, etc).
- `size_in_bytes` → how many bytes to allocate.

3. How to understand this

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

Breakdown:

1. `5 * sizeof(int)` → asks for memory of 5 integers.
   - If `sizeof(int)` = 4 bytes, then → 5 × 4 = 20 bytes.
2. `malloc(20)` → OS gives 20 bytes from heap.
3. `(int*)` → cast the returned address into `int*` type.
4. p now points to the first element of this memory block.

```
Heap memory (20 bytes) → [    ][    ][    ][    ][    ]
                          ↑
          p
```

Why do we need it?

In normal arrays:

```
int arr[100];
```

- Size is fixed at compile time.

With malloc:

```
int *p = (int*) malloc(100 * sizeof(int));
```

- Size is decided at runtime (depends on user/program need).

Mini Example (Simple Program)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 5;
    int *p = (int*) malloc(n * sizeof(int));

    if (p == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        p[i] = i + 1;   // store values
    }
```

```
    printf("Values: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", p[i]);
    }

    free(p); // free allocated memory
    return 0;
}
```

5. Tracing Example

- `malloc(5 * sizeof(int)) → allocate 20 bytes.`
- Let’s say the heap gives the address 1000.
- Then p = 1000.

First for loop:

Iteration (`i`)	**Expression**	**Value Stored**	**Memory Content (p[0] … p[4])**
0	p[0] = 0 + 1	1	1 _ _ _ _
1	p[1] = 1 + 1	2	1 2 _ _ _
2	p[2] = 2 + 1	3	1 2 3 _ _
3	p[3] = 3 + 1	4	1 2 3 4 _
4	p[4] = 4 + 1	5	1 2 3 4 5

Second for loop (printing values)

```
for (int i = 0; i < 5; i++) {
    printf("%d ", p[i]);
```

}

Iteration (`i`)	Expression	Printed Output	Final Output on Screen
0	printf("%d", p[0])	1	1
1	printf("%d", p[1])	2	1 2
2	printf("%d", p[2])	3	1 2 3
3	printf("%d", p[3])	4	1 2 3 4
4	printf("%d", p[4])	5	1 2 3 4 5

6. Key Points

malloc allocates **uninitialized memory** (garbage until you assign).

1. Always check if malloc returned **NULL** (allocation failed).
2. Always free() the memory after use.

Example: Just malloc and check

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p;

    p = (int*) malloc(5 * sizeof(int));  // allocate space for 5 integers

    if (p == NULL) {
        printf("Memory not allocated!\n");
        return 1;
    }

    printf("Memory successfully allocated at address: %p\n", p);

    free(p);  // release the memory
    return 0;
}
```

Step by Step (Dry Run)

1. int *p;
    - A pointer variable p is declared.
    - Currently, p has no valid address (garbage).
2. p = (int*) malloc(5 * sizeof(int));
    - sizeof(int) = 4 bytes (assume).
    - 5 * 4 = 20 bytes are requested from the heap.
    - OS gives 20 bytes, starting at (say) address 1000.
    - So p = 1000.

Heap (20 bytes allocated)
[1000][1004][1008][1012][1016]

3. Each slot is int sized (4 bytes).

4. if (p == NULL)
    - Check if allocation failed. If not, continue.
5. printf("%p\n", p);
    - Prints the starting address, e.g., 0x1000.
6. free(p);
    - `Frees the memory → returns 20 bytes back to heap.`
    - Now p becomes a **dangling pointer** (it still stores 1000, but memory is no longer yours).

1)malloc:

```
cdac@cdac-virtual-machine:~/Mahesh/pointers/dynamic_memory$ cat malloc_1.c
//Store N numbers and compute sum//


#include <stdio.h>
#include <stdlib.h>   // malloc

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    // Allocate memory for n integers
    //int *arr = (int*) malloc(n * sizeof(int));

    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
```

```c
    // Input elements
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i+1);
        scanf("%d", arr + i);   // same as &arr[i]
    }

    // Compute sum
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *(arr + i);
    }

    printf("Sum = %d\n", sum);

    free(arr);   // release memory
    return 0;
}
```

Note:

Q1: Why do we write (int*) in malloc?

int *arr = (int*) malloc(n * sizeof(int));

- malloc() **always returns** a void* (generic pointer).
- A void* just means: *"I have the address of something, but I don't know its type yet."*
- Since we want to store integers, we **typecast** it to int*.

Without (int*), we cannot directly assign it to an int* in **C++**, but in **C** it works even without casting.
So both are valid in **C**:

int *arr = malloc(n * sizeof(int)); // fine in C
int *arr = (int*) malloc(n * sizeof(int)); // more explicit

Tip: *"malloc gives us a general address, we put a label on it saying this is now an int address."*

Q2: Why do we check (arr == NULL)?

```
if (arr == NULL) {
   printf("Memory allocation failed!\n");
   return 1;
}
```

- If the system doesn't have enough memory (say you asked for 1 million integers), malloc **fails**.
- On failure, malloc **returns NULL** (special pointer meaning "points to nothing").
- If we don't check and still try to use arr, the program will crash (segmentation fault).

Analogy

- Imagine you ask the warehouse for 10 boxes.
- If the warehouse has no empty boxes, it gives you a **null ticket** (no boxes).
- If you don't check and try to put things in imaginary boxes, you're in trouble!

So in short:

- `(int*) → tells compiler: treat this address as an integer pointer.`
- `(arr == NULL) → safety check: did we actually get memory or not?`

1. What is calloc()?

- calloc = **Contiguous Allocation**
- Used to allocate memory **at runtime** (like malloc).
- Difference from malloc:
  - `malloc → allocates memory but keeps` **garbage values**.
  - `calloc → allocates memory and` **initializes all bytes to 0**.

2. Syntax

ptr = (cast-type*) calloc(num_elements, size_of_each_element);

- `num_elements → how many blocks`
- `size_of_each_element → size of each block (in bytes)`
- Total memory = num_elements × size_of_each_element

3. How to Understand

Example:

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

- 5 = number of integers
- sizeof(int) = 4 bytes
- Total = 5 × 4 = 20 bytes allocated
- All 20 bytes are set to **0**

So heap looks like this after allocation:

[1000] = 0
[1004] = 0
[1008] = 0
[1012] = 0
[1016] = 0

4. Example

- **malloc vs calloc (chairs analogy)**

Imagine a classroom:

- With malloc, you get 5 chairs, but **they may be dirty** (garbage values). You must clean them before using.
- With calloc, you get 5 chairs, and **all are cleaned and set to zero** `already → ready to use.`

```
cdac@cdac-virtual-machine:~/Mahesh/pointers/dynamic_memory$ cat calloc1.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n;
    printf("Enter number of students: ");
    scanf("%d", &n);

    // Allocate memory for n integers, initialized to 0
    int *marks = (int*) calloc(n, sizeof(int));

    if (marks == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
```

```c
    printf("Initial values (all should be 0):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", *(marks + i));
    }

    // Update values
    printf("\nEnter marks:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", marks + i);
    }

    printf("Final marks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", *(marks + i));
    }

    free(marks);
    return 0;
}
```

Assume:
int *marks = (int*) calloc(3, sizeof(int));

Suppose calloc gives address 1000.
So:

marks = 1000

Case 1: *(marks + i) in printf

Take loop for (int i=0; i<3; i++)

i = 0

- marks + 0 = 1000
- *(marks + 0) = value at address 1000
- Initially = 0 (because calloc clears memory)

i = 1

- marks + 1 = 1000 + (1 * sizeof(int))
- = 1000 + 4 = 1004
- `*(marks + 1) = value at address 1004 → 0`

i = 2

- marks + 2 = 1000 + (2 * 4) = 1008
- `*(marks + 2) = value at address 1008 → 0`

So printf prints:

0 0 0

Case 2: scanf("%d", marks + i)

Important: In scanf we don't use *.
We give the **address where scanf should put the number**.

i = 0

- marks + 0 = 1000
- `scanf will store user input at address 1000 → updates 1st element.`

i = 1

- marks + 1 = 1004
- `scanf will store user input at address 1004 → updates 2nd element.`

i = 2

- marks + 2 = 1008
- `scanf will store user input at address 1008 → updates 3rd element.`

Case 3: Printing final values

Now again loop with *(marks + i) will fetch values from 1000, 1004, 1008.

If user entered 50, 60, 70, then memory looks like:

```
1000 → 50
1004 → 60
1008 → 70
```

So printf prints:

50 60 70

Key Note:

- `marks + i → address of element` *i*
- `*(marks + i) → value at that address`

That's why in scanf we pass **address** (marks + i), but in printf we use * to get **value** (*(marks + i)).

7. Key Points\

- **malloc** → `uninitialized (garbage).`
- **calloc** → `initialized (all 0).`
- Both allocate from **heap**.
- Both return a pointer, must be checked against NULL.
- Always use free() when done.

Program: malloc vs calloc

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 5;

    // Using malloc
    int *p1 = (int*) malloc(n * sizeof(int));
    if (p1 == NULL) {
        printf("malloc failed!\n");
        return 1;
    }

    // Using calloc
    int *p2 = (int*) calloc(n, sizeof(int));
    if (p2 == NULL) {
        printf("calloc failed!\n");
        return 1;
    }

    printf("Initial values (malloc):\n");
    for (int i = 0; i < n; i++) {
        printf("p1[%d] = %d\n", i, p1[i]); // garbage
    }

    printf("\nInitial values (calloc):\n");
    for (int i = 0; i < n; i++) {
        printf("p2[%d] = %d\n", i, p2[i]); // zero
    }

    free(p1);
    free(p2);

    return 0;
}
```

Step by Step Tracing

1. malloc(n * sizeof(int))

    - Allocates 20 bytes (5 × 4).
    - Contents are **garbage** (whatever was in memory).
    - Example: [1000]=131, [1004]=-5, [1008]=4048, [1012]=0, [1016]=999 (random).

2. calloc(n, sizeof(int))
    - Allocates 20 bytes (5 × 4).
    - All initialized to **0**.
    - Example: [2000]=0, [2004]=0, [2008]=0, [2012]=0, [2016]=0.

Example Output (on real run, garbage may differ)

```
Initial values (malloc):
p1[0] = 131
p1[1] = -5
p1[2] = 4048
p1[3] = 0
p1[4] = 999

Initial values (calloc):
p2[0] = 0
p2[1] = 0
p2[2] = 0
p2[3] = 0
p2[4] = 0
```

1. Why realloc()?

- Sometimes, we don't know **in advance** how much memory we'll need.
- Example:
  - You allocate space for 5 students.
  - Later, 3 more students join.
  - Instead of creating a **new array manually** and copying old values, we use realloc().

- Stands for **Re-Allocation**.
- Used to **resize previously allocated memory** (from malloc/calloc).
- Keeps **old data safe** (up to min(old_size, new_size)) and adjusts memory.

2. Syntax

ptr = realloc(ptr, new_size_in_bytes);

- ptr → previously allocated memory (from malloc, calloc, or even realloc).
- new_size_in_bytes → new total size.
- If successful → returns pointer to **new block**.
- If fails → returns NULL (old memory still valid).

4. Technical Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 5;
    int *p = (int*) malloc(n * sizeof(int));

    if (p == NULL) {
        printf("malloc failed!\n");
        return 1;
    }

    // Assign initial values
    for (int i = 0; i < n; i++) {
```

```
        p[i] = (i + 1) * 10;
    }

    printf("Before realloc:\n");
    for (int i = 0; i < n; i++) {
        printf("p[%d] = %d\n", i, p[i]);
    }

    // Now expand to 8 integers
    n = 8;
    p = (int*) realloc(p, n * sizeof(int));

    if (p == NULL) {
        printf("realloc failed!\n");
        return 1;
    }

    // Assign values for new space
    for (int i = 5; i < n; i++) {
        p[i] = (i + 1) * 10;
    }

    printf("\nAfter realloc:\n");
    for (int i = 0; i < n; i++) {
        printf("p[%d] = %d\n", i, p[i]);
    }

    free(p);
    return 0;
}
```

---

5. Dry Run

1. **Before realloc (n=5)**

```
p[0] = 10
p[1] = 20
```

p[2] = 30
p[3] = 40
p[4] = 50

2. **After realloc (n=8)**

p[0] = 10
p[1] = 20
p[2] = 30
p[3] = 40
p[4] = 50
p[5] = 60
p[6] = 70
p[7] = 80

---

6. Key Points

- `malloc → allocate memory.`
- `calloc → allocate + initialize to 0.`
- `realloc → resize previously allocated memory.`
- Always check for NULL.
- Always use free() when done.

1. What is free()?

- When we allocate memory using malloc, calloc, or realloc, it comes from the **heap**.
- If we don’t release it, memory will be wasted → called a **memory leak**.
- free() is used to release that memory back to the system.

Syntax: free(ptr);

- When you use malloc, calloc, or realloc → memory is taken from the **heap** (a special area of RAM).
- That memory **does not get automatically released** when the function ends (unlike normal local variables stored in stack).
- If you don’t release it manually → it stays occupied until program ends.
- The function free(ptr) releases that block back to the heap so it can be reused.

Why is free() Important?

Technical Reasons

1. **Prevent memory leaks**
   - If you forget to free(), heap keeps shrinking, program consumes more RAM, and system slows down/crashes.
   - Example: a server running 24/7 without free → will eventually run out of memory.
2. **Efficient use of limited memory**
   - Embedded systems (IoT, automotive ECUs, medical devices) often have only a few KB of RAM.
   - Every byte matters, so freeing unused memory is critical.
3. **Good programming practice**

- Helps OS/RTOS manage heap properly.
- Prevents fragmentation issues in long-running applications.

2. Key Points

- Only memory allocated dynamically (malloc, calloc, realloc) should be freed.
- After free(ptr), the pointer becomes **dangling** → `it still holds the old address, but that memory is no longer valid.`
- Safe practice: set **ptr = NULL**; after freeing.

3. Technical Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*) malloc(3 * sizeof(int));

    if (p == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    p[0] = 10;
    p[1] = 20;
    p[2] = 30;

    printf("Before free:\n");
    for (int i = 0; i < 3; i++) {
        printf("p[%d] = %d (address = %p)\n", i, p[i], &p[i]);
    }

    free(p);   // release memory
    p = NULL;  // avoid dangling pointer

    printf("\nAfter free:\n");
```

```
    if (p == NULL) {
        printf("Pointer is NULL, memory released.\n");
    }

    return 0;
}
```

---

4. Step by Step (Tracing)

1. `malloc(3 * sizeof(int)) → allocate 12 bytes. Suppose addresses:`

```
[1000] = 10
[1004] = 20
[1008] = 30
```

2. free(p);
    - Memory [1000–1008] is returned to heap.
    - `But p still contains 1000 →` **dangling pointer**.
3. p = NULL;
    - Now p is safe.
    - `Trying to access memory via NULL → segmentation fault (better than using freed memory by mistake).`

Note:

Rule of scanf

`scanf("%d", X); → expects the` **address of an int** (so that it can place the user's input there).

Normally, with variables:

int x;
scanf("%d", &x);   // &x = address of x

What happens with arrays and pointers?

When we say:

int *arr = malloc(n * sizeof(int));

- arr itself is already a **pointer**.
- It stores the **address of the first element**.

So:

- arr + i = the **address of element i**.
- That's exactly what scanf needs!

Therefore, scanf("%d", arr+i) is the same as scanf("%d", &arr[i]).

We don't need another & because arr+i is **already an address**.

Tracing

Suppose malloc returned base address 1000.

- arr = 1000
- `i=0 → arr+0 = 1000 → scanf stores value at address 1000`
- `i=1 → arr+1 = 1004 → scanf stores value at address 1004`
- `i=2 → arr+2 = 1008 → scanf stores value at address 1008`

If user enters: 10 20 30, memory becomes:

```
1000 → 10
1004 → 20
1008 → 30
```

Tip:

```
&arr[i]   ==   arr+i
```

Then show with an example:

```
scanf("%d", &arr[i]);  // works
scanf("%d", arr+i);    // works (same thing!)
```

1) malloc (Uninitialized Memory Allocation)

- Technical Uses
    - When size is known, but we don't need values initialized.
    - Example: Storing marks of N students where N is entered at runtime.
    - Example: Allocating buffer for network packet, image, or sensor data.
    - Example: Linked list / tree nodes (each node created with malloc).

---

2) calloc (Initialized Memory Allocation)

- Technical Uses
    - When size is known **and** you want all elements **zeroed** at start.
    - Example: Initializing an array of counters (all 0).
    - Example: Bitmaps or frequency tables (start from 0).
    - Example: Matrices where all values should start at 0.

---

3) realloc (Resizing Memory)

- Technical Uses
    - Dynamic arrays when size grows/shrinks at runtime.
        - Example: Expanding array when more students join a class.
        - Example: Text editor buffer (user types more characters → buffer grows).
        - Example: Growing stack/queue in data structures.
    - Memory optimization: if you initially over-allocated, shrink using realloc.