📘 Dynamic Memory Allocation in C

📘 Dynamic Memory Allocation in C

Functions: `malloc()`, `calloc()`, `realloc()` and their use in Linked Lists

🧠 1️⃣ Why We Need Dynamic Memory Allocation

In C, memory can be allocated in two ways:

Static (compile-time) and **Dynamic (run-time)**.

Type	Example	When Decided	Lifetime
Static / Compile-time	`int arr[5];`	At compile time	Fixed until program ends or function returns
Dynamic / Run-time	`int *arr = malloc(n * sizeof(int));`	At run time	Flexible until we `free()` it

🔷 Reasons to Use Dynamic Memory:

1. **When size of data is unknown at compile time.**

   (e.g. user enters `n` at runtime)

2. **When we need memory that persists beyond function scope.**
3. **When we need flexible data structures** like linked lists, trees, graphs, stacks, queues.
4. **When we need to resize allocated memory** at runtime.

Dynamic memory functions are declared in `<stdlib.h>`.

⚙️ 2️⃣ Functions of Dynamic Memory Allocation

🔷 `malloc()` — Memory Allocation

Syntax:

```
ptr = (int *)malloc(n * sizeof(int));
```

Explanation:

- Allocates `n * sizeof(int)` bytes from the **heap**.
- Does **not initialize** memory (contains garbage values).
- Returns a `void*` pointer (cast optional in C).
- Returns `NULL` if allocation fails.

Example:

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

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

    int *arr = malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory not allocated!\n");
        exit(0);
    }

    for (int i = 0; i < n; i++)
        arr[i] = i + 1;

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    free(arr);
```

```
    return 0;
}
```

🔷 `calloc()` — Contiguous Allocation

Syntax:

```
ptr = (int *)calloc(n, sizeof(int));
```

Explanation:

- Allocates memory for `n` elements, each of `sizeof(int)` bytes.
- Initializes all bytes to **zero**.
- Returns `NULL` if allocation fails.

Example:

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

int main() {
    int *arr = (int *)calloc(5, sizeof(int));
    if (arr == NULL) {
        printf("Memory not allocated!\n");
        return 1;
    }

    for (int i = 0; i < 5; i++)
        printf("%d ", arr[i]);  // prints: 0 0 0 0 0

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

✅ Difference between malloc and calloc:

Feature	malloc()	calloc()
Initialization	Garbage values	All zeros
Parameters	One (total bytes)	Two (blocks, size per block)
Speed	Slightly faster	Slightly slower (zeroing)

🔷 realloc() — Reallocation

Syntax:

```
ptr = (int *)realloc(ptr, new_size);
```

Explanation:

- Resizes previously allocated memory block.
- Keeps old data intact up to smaller of old/new sizes.
- Allocates new memory and copies data if necessary.
- Returns `NULL` if unable to allocate (old block remains valid).

Example:

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

int main() {
    int *arr = malloc(3 * sizeof(int));
    arr[0] = 1; arr[1] = 2; arr[2] = 3;

    // Increase array size to 5
    arr = realloc(arr, 5 * sizeof(int));
    arr[3] = 4; arr[4] = 5;

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

    free(arr);
```

```
    return 0;
}
```

✅ **Difference between realloc and malloc/calloc:**

Feature	`malloc()` / `calloc()`	`realloc()`
Use	First-time allocation	Resize existing block
Keeps previous data	❌ No	✅ Yes
Parameters	size only	old pointer + new size

💡 3️⃣ Why Dynamic Memory is Used in Linked Lists

Linked lists are **dynamic data structures** — number of nodes is not known in advance.

Each node is created or deleted **at runtime**.

Example Node:

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;
```

Creating a New Node Using `malloc()` :

```
Node *newNode = (Node*)malloc(sizeof(Node));
newNode->data = 10;
newNode->next = NULL;
```

Reasons for Using Dynamic Allocation:

1. The number of nodes changes at runtime.
2. Each node must **persist** even after the function returns (heap memory).
3. We can **insert/delete** nodes easily without worrying about fixed array size.

4. Static allocation (`Node arr[10]`) limits flexibility.

Function	Use in Linked List
`malloc()`	Create new nodes dynamically
`calloc()`	(Rare) Initialize nodes to zero
`realloc()`	Not used — nodes are not contiguous

🧾 4️⃣ Summary Table

Function	Meaning	Initialization	Common Use Case
`malloc()`	Memory Allocation	Garbage values	Dynamic arrays, linked lists
`calloc()`	Contiguous Allocation	Zeros memory	When zero-initialized memory needed
`realloc()`	Re-Allocation	Keeps old data	Resizing dynamic arrays
`free()`	Free Memory	—	Release allocated memory

🚀 5️⃣ In One Line

> We use malloc(), calloc(), and realloc() to allocate, initialize, and resize memory dynamically at runtime — enabling flexible and efficient data structures like linked lists, trees, and stacks.

✅ Example: Simple Linked List using malloc()

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

typedef struct Node {
    int data;
    struct Node *next;
} Node;

Node *createNode(int value) {
```

```c
    Node *newNode = malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

void printList(Node *head) {
    Node *temp = head;
    while (temp) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    Node *head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);

    printList(head);

    // Free memory
    Node *temp;
    while (head) {
        temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```
