Dynamic Memory Allocation

What is Dynamic Memory Allocation?

Dynamic memory allocation allows programs to **request memory at runtime** rather than compile time.

Used when:

- The amount of data is not known beforehand.
- Data structures need to grow or shrink during execution.

Why Not Static Allocation?

Static Memory	Dynamic Memory
Size fixed at compile time	Size determined at runtime
Allocated on stack	Allocated on heap
Automatically freed	Must be freed manually
Limited size	Flexible and scalable

Functions in `<stdlib.h>`

Function	Description
`malloc(size)`	Allocates uninitialized memory
`calloc(n, size)`	Allocates and zero-initializes memory
`realloc(ptr, size)`	Changes the size of an existing allocation
`free(ptr)`	Frees allocated memory

🧩 Example 1: Using `malloc()`

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

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

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

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

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

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

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

🧩 Example 2: `calloc()` for Zero Initialization

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

int main() {
    int *arr = calloc(5, sizeof(int));
    for (int i = 0; i < 5; i++) printf("%d ", arr[i]);
    free(arr);
}
```

✅ `calloc()` ensures all elements are initialized to 0.

🧩 Example 3: `realloc()` for Resizing

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

int main() {
    int *arr = malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) arr[i] = i + 1;

    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);
}
```

✅ Expands or shrinks a previously allocated block.

Common Use Case 1: Dynamic Arrays

When user input size is unknown at compile time.

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

🔷 Common Use Case 3: Dynamic 2D Arrays

Allocate memory for a matrix when dimensions are unknown.

```
int **matrix = malloc(rows * sizeof(int *));
for (int i = 0; i < rows; i++)
    matrix[i] = malloc(cols * sizeof(int));
```

💡 Used in:

- Dynamic table storage
- Graph adjacency matrices
- Image data buffers

◆ Common Use Case 2: Linked Lists

Each node is created dynamically to store data.

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

Node *newNode(int val) {
    Node *temp = malloc(sizeof(Node));
    temp->data = val;
    temp->next = NULL;
    return temp;
}
```

🔷 Common Use Case 4: Binary Trees

Dynamic allocation enables recursive structures.

```c
typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

Node* newNode(int val) {
    Node *temp = malloc(sizeof(Node));
    temp->data = val;
    temp->left = temp->right = NULL;
    return temp;
}
```

💡 Used in:

- inary Search Trees
- Expression Trees

◆ Common Use Case 5: Graphs (Adjacency List)

```c
typedef struct Node {
    int vertex;
    struct Node *next;
} Node;

Node* createNode(int v) {
    Node *n = malloc(sizeof(Node));
    n->vertex = v;
    n->next = NULL;
    return n;
}
```

⚠️ Memory Management Best Practices

✅ Always check if `malloc()` returned `NULL`.

✅ Always `free()` memory after use.

✅ Avoid **memory leaks** and **dangling pointers**.

✅ Use tools like Valgrind to detect leaks.

🚫 Common Errors

Error	Cause
Segmentation fault	Using uninitialized or freed pointer
Memory leak	Forgetting to call `free()`
Double free	Freeing the same pointer twice
Buffer overflow	Writing beyond allocated memory

🧠 Summary

Function	Use	Key Point
`malloc()`	Allocate memory	Uninitialized memory
`calloc()`	Allocate + zero initialize	Slower, safer
`realloc()`	Resize memory	Retains old data
`free()`	Release memory	Must call manually

💡 Exercises

1. Implement a dynamically growing array that doubles in size.
2. Create a linked list of student records using dynamic memory.
3. Write a function to dynamically allocate a 2D array.
4. Simulate dynamic allocation of memory for a tree.

🏁 Takeaway

> Dynamic memory makes C powerful — but with great power comes great responsibility.

Always pair every `malloc()` with a matching `free()`!

References

- https://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture08.pdf
- https://web.eecs.utk.edu/~bvanderz/teaching/cs140fa08/labs/lab1/cs102/malloc.html
- Section 6.8 in the book linked