EEE: 103

Computer Programming

L10: Dynamic Memory Allocation

🧠

Prepared by: Sk Tahmed Salim Rafid

Why Dynamic Memory Allocation?

The Problem with Fixed-Size Arrays

If you want to store details of 5 students, you create:

```
int students[5];
```

❓ But what if later you need to store 6 students?

- ✗ You cannot increase the size of this array
- ✗ You must create a new array
- ✗ Then copy each element one by one
- ✗ This is inefficient and inconvenient

Dynamic Memory Allocation - The Solution

⚡ **Allocated at runtime**

Memory size determined during program execution

📏 **Resized when needed**

Increase or decrease memory as required

♻️ **Freed when no longer required**

Efficient memory management

Stack vs Heap Memory

Understanding where different types of memory are stored

Heap Memory

malloc() allocations

Dynamic arrays

Resizable memory

Characteristics:

- Dynamic size
- Manual management
- Slower than stack
- Much larger space

Stack Memory

Local Variables

Function Parameters

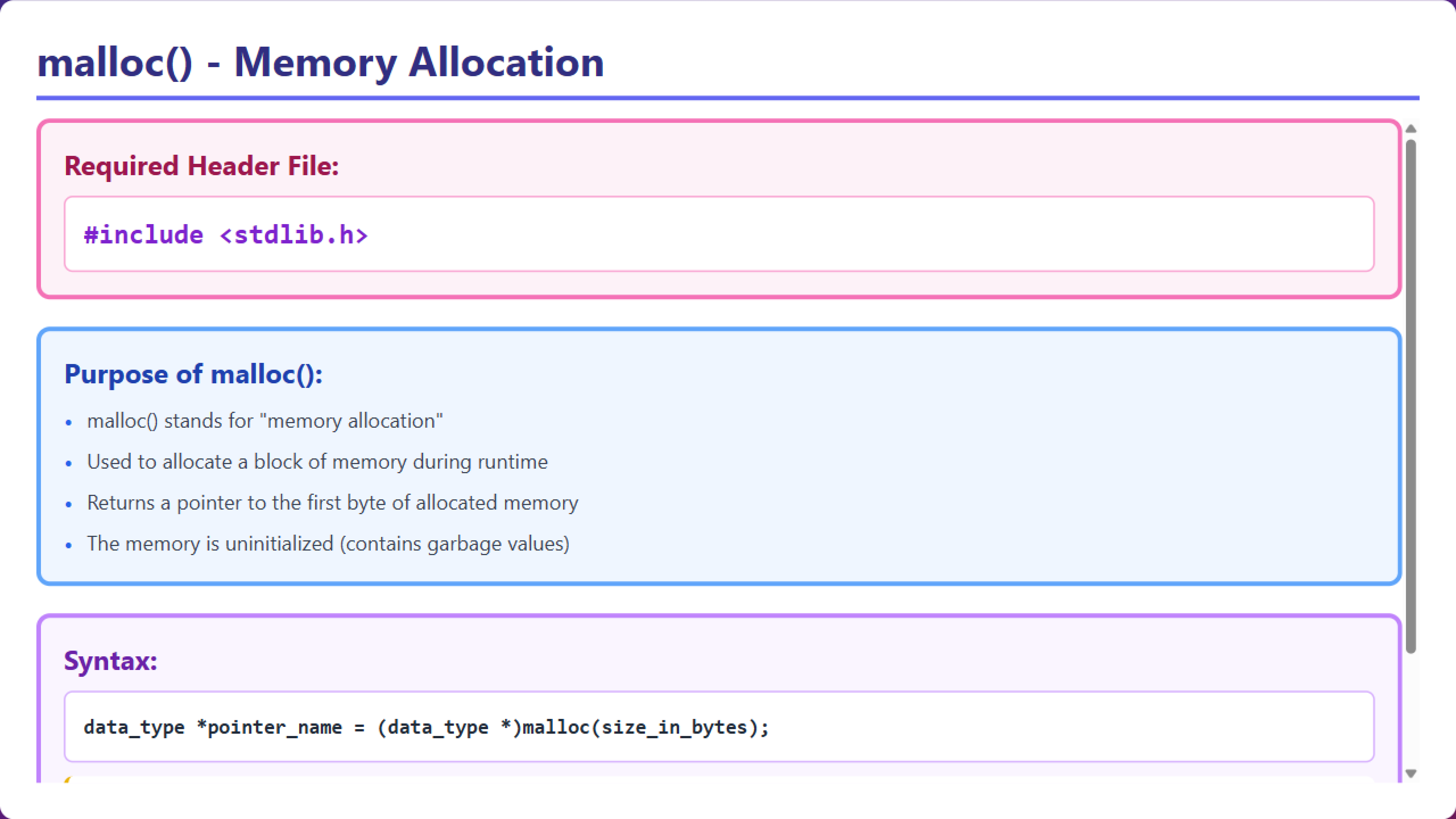
Return Addresses

Characteristics:

- Fixed size
- Automatic management
- Fast access
- Limited space

Role of Pointers in Dynamic Memory

- When memory is dynamically allocated, it is created in the heap
- The allocation function returns the address of the allocated memory
- Pointers are required to store and access dynamically allocated memory
- Without pointers, we cannot work with dynamic memory
- The pointer holds the starting address of the allocated block



malloc() - Memory Allocation

Required Header File:

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

Purpose of malloc():

- malloc() stands for "memory allocation"
- Used to allocate a block of memory during runtime
- Returns a pointer to the first byte of allocated memory
- The memory is uninitialized (contains garbage values)

Syntax:

```c
data_type *pointer_name = (data_type *)malloc(size_in_bytes);
```

malloc() Example

```c
int *ptr = (int *)malloc(sizeof(int) * 5);
```

This allocates memory for 5 integers

Almost Identical Usage:

Dynamic:

```c
int *ptr = (int *)malloc(sizeof(int) * 5);
```

Static:

```c
int arr[5];
```

These are almost identical in usage!

Usage:

- Elements can be accessed using ptr[index]
- Array indexing works the same way
- ptr[0], ptr[1], ptr[2], ptr[3], ptr[4]

Checking for Allocation Failure

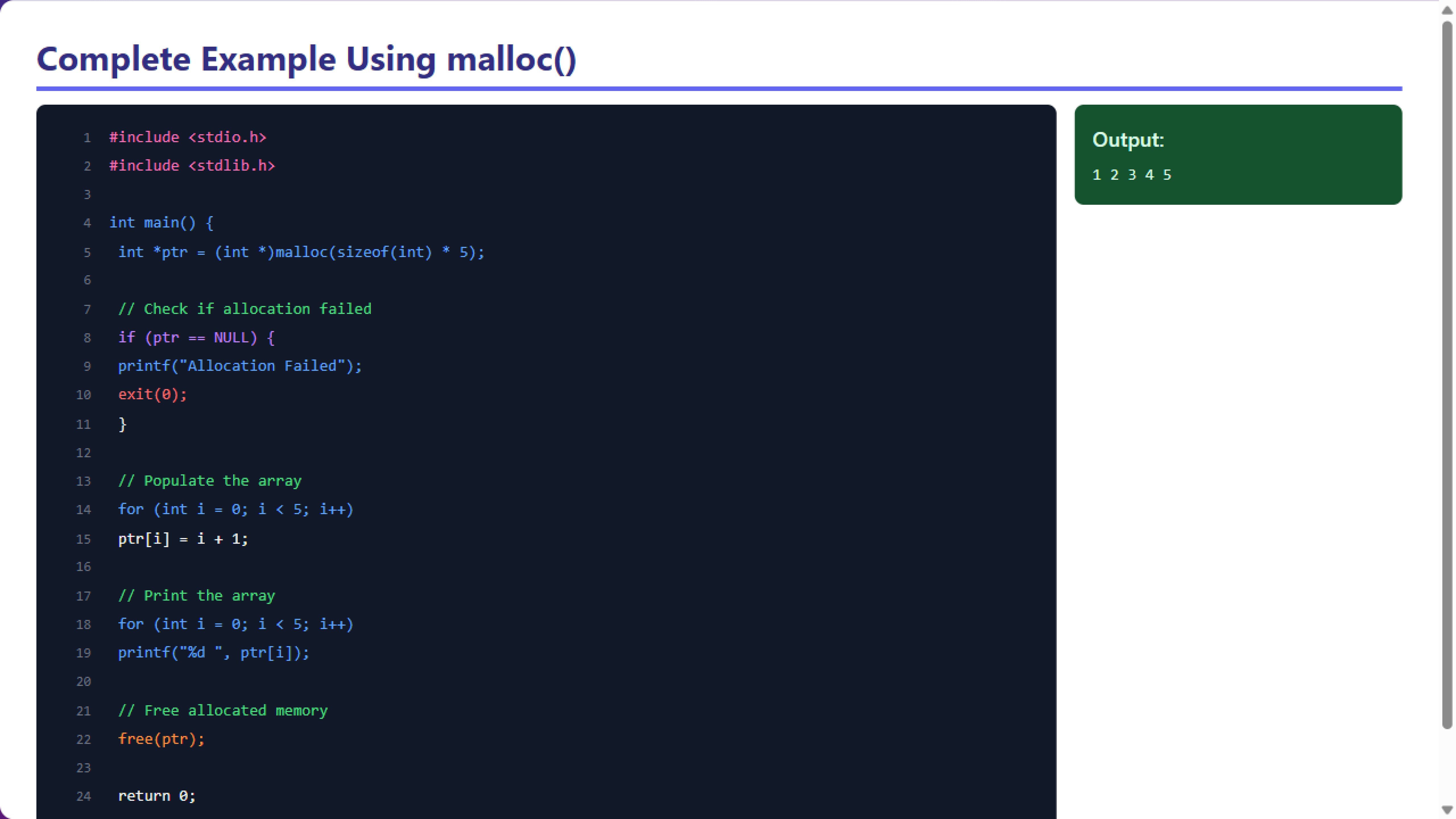
⚠ **If memory allocation fails, malloc() returns NULL**

✓ Always check before using the pointer!

```c
int *ptr = (int *)malloc(sizeof(int) * 5);

if (ptr == NULL) {
 printf("Allocation Failed");
 exit(0);
}

// Safe to use ptr now
```



Complete Example Using malloc()

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

int main() {
 int *ptr = (int *)malloc(sizeof(int) * 5);

 // Check if allocation failed
 if (ptr == NULL) {
 printf("Allocation Failed");
 exit(0);
 }

 // Populate the array
 for (int i = 0; i < 5; i++)
 ptr[i] = i + 1;

 // Print the array
 for (int i = 0; i < 5; i++)
 printf("%d ", ptr[i]);

 // Free allocated memory
 free(ptr);

 return 0;
```

Output:

```
1 2 3 4 5
```

free() - Releasing Memory

After using dynamically allocated memory, it must be released

```
free(pointer_name);
```

🚫

Prevents memory leaks

Avoids accumulation of unused memory

🔓

Releases unused heap memory

Makes memory available for other programs

✅

Good programming practice

Professional and efficient code

⚠ **Warning: Never access memory after freeing it!**

realloc() - Resizing Memory

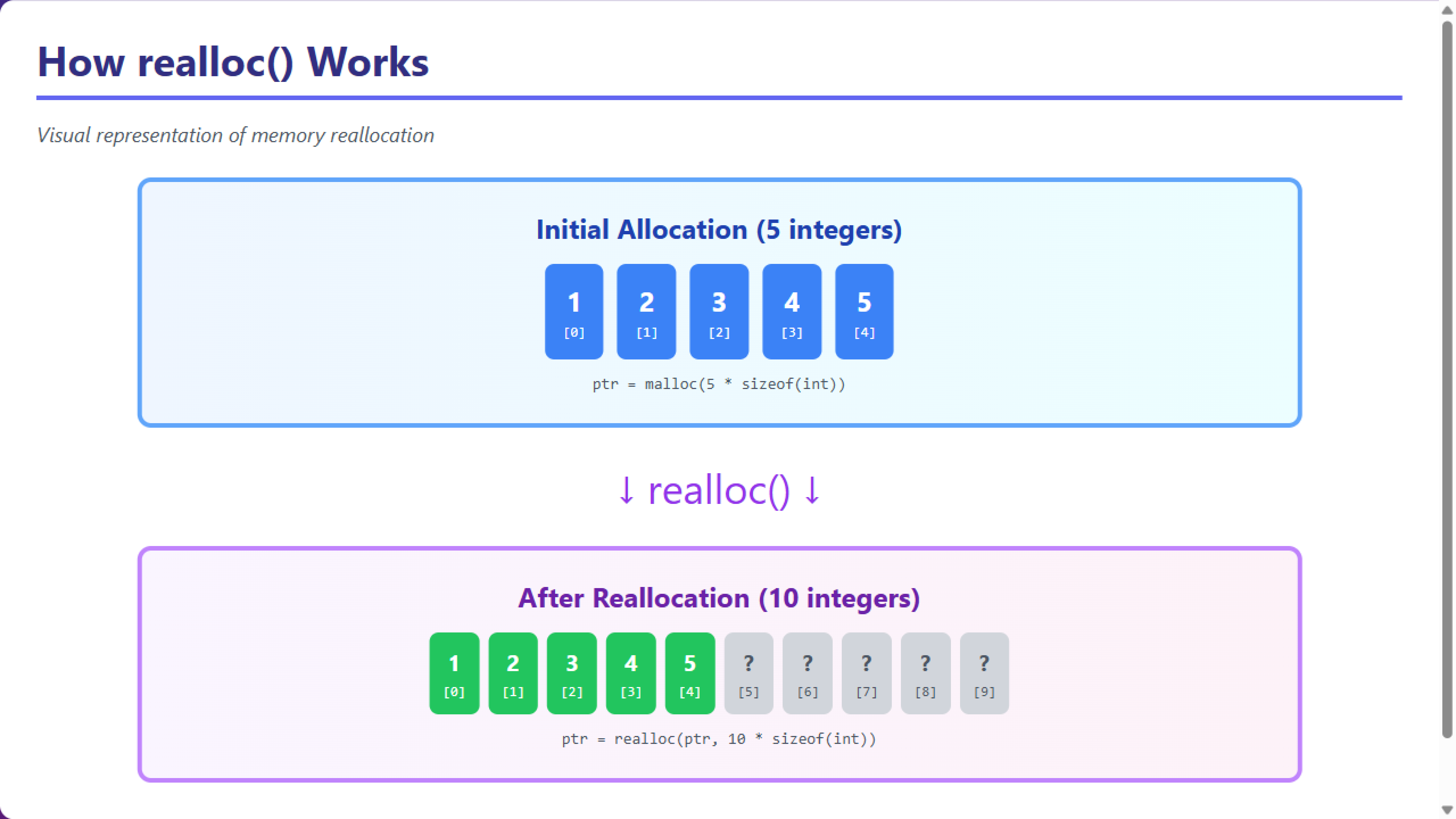
Purpose of realloc():

- realloc() is used to resize an existing memory block
- Can increase or decrease memory size
- Preserves existing data (up to the smaller size)
- May move memory to a new location

Syntax:

```
data_type *new_ptr = (data_type *)realloc(old_pointer, new_size * sizeof(data_type));
```

📌 Always assign the result to a pointer and check for NULL



How realloc() Works

Visual representation of memory reallocation

Initial Allocation (5 integers)

1	2	3	4	5
[0]	[1]	[2]	[3]	[4]

```
ptr = malloc(5 * sizeof(int))
```

↓ realloc() ↓

After Reallocation (10 integers)

1	2	3	4	5	?	?	?	?	?
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

```
ptr = realloc(ptr, 10 * sizeof(int))
```

Example Using realloc()

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

int main() {
 // Initially allocate memory for 5 integers
 int *ptr = (int *)malloc(5 * sizeof(int));

 // Resize memory to hold 10 integers
 ptr = (int *)realloc(ptr, 10 * sizeof(int));

 // Check for failure
 if (ptr == NULL) {
 printf("Memory Reallocation Failed");
 exit(0);
 }

 // Use the resized memory
 // ...

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

Step 1 Step 2 Step 3 Step 4 Step 5 Step 6

Output:

⚠️ **Note:** After realloc(), the old pointer may become invalid if memory moved!

Dynamic Memory Functions Summary

Function	Purpose	Returns	Notes
`malloc()`	Allocate memory	Pointer to allocated memory	*Memory is uninitialized*
`realloc()`	Resize allocated memory	Pointer to resized memory	*May move data to new location*
`free()`	Release memory	void (nothing)	*Must be called to prevent leaks*
`Pointer`	Stores address of allocated memory	N/A	*Essential for dynamic memory*

Common Mistakes to Avoid

❌ Using memory without checking NULL

Consequence: Program crash / Segmentation fault

❌ Forgetting to call free()

Consequence: Memory leaks

❌ Accessing memory after freeing it

Consequence: Undefined behavior / Crash

❌ Using incorrect size in malloc() or realloc()

Consequence: Buffer overflow / Data corruption

Practice Problems

1

Basic

Dynamic Sum Calculator

Write a program to dynamically allocate memory for n integers, take input from the user, and print the sum.

2

Intermediate

Even Numbers Storage

Dynamically allocate memory for 10 integers and store only even numbers entered by the user.

3

Advanced

Dynamic Array with realloc()

Write a program that: (1) Initially allocates memory for 3 integers (2) Takes input from user (3) If user wants to enter more numbers, resize using realloc() (4) Display all entered numbers

Best Practices Checklist

- Always check if malloc() or realloc() returns NULL
- Always call free() for every malloc() or realloc()
- Set pointer to NULL after freeing: ptr = NULL
- Never free the same memory twice
- Calculate size using sizeof() for portability
- Use realloc() instead of manual copy when resizing
- Keep track of allocated memory size

Thank You

Allocate Wisely, Code Efficiently! 🚀