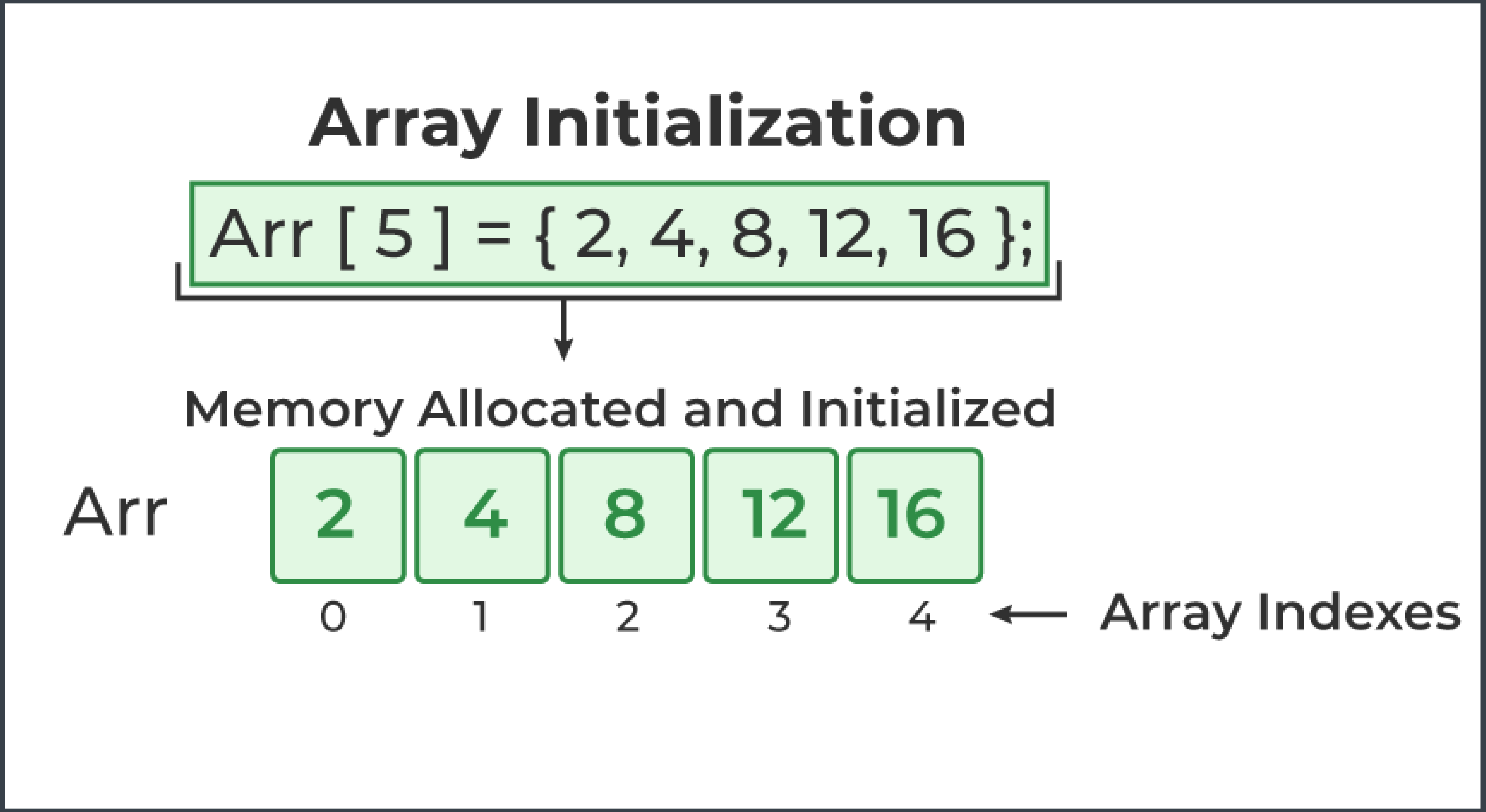
Array

Defnition

- An array is a data structure that can hold a fixed number of elements of the same data type.
- Elements within an array are stored in contiguous memory locations, making it efficient to access and manipulate them.

Defnition


Array Initialization
Arr [5] = { 2, 4, 8, 12, 16 };
Memory Allocated and Initialized
Arr
2
4
8
12
16
0
1
2
3
4
Array Indexes

Declaring an Array

- To declare an array in C, you specify the data type of the elements and the size of the array.

```
int numbers[5];
```

Initializing an Array

- You can initialize an array when declaring it by providing a list of values enclosed in braces

```
int numbers[] = {1, 2, 3, 4, 5};
```

Initializing an Array

- Alternatively, you can initialize an array element by element using indexing

```
int numbers[5];
numbers[0] = 1;
numbers[1] = 2;
// ...
```

Accessing Array Elements

- You can access individual elements of an array using square brackets and an index, starting from 0

```
int value = numbers[2];
```

Modifying Array Elements

- You can modify array elements by assigning new values using the assignment operator

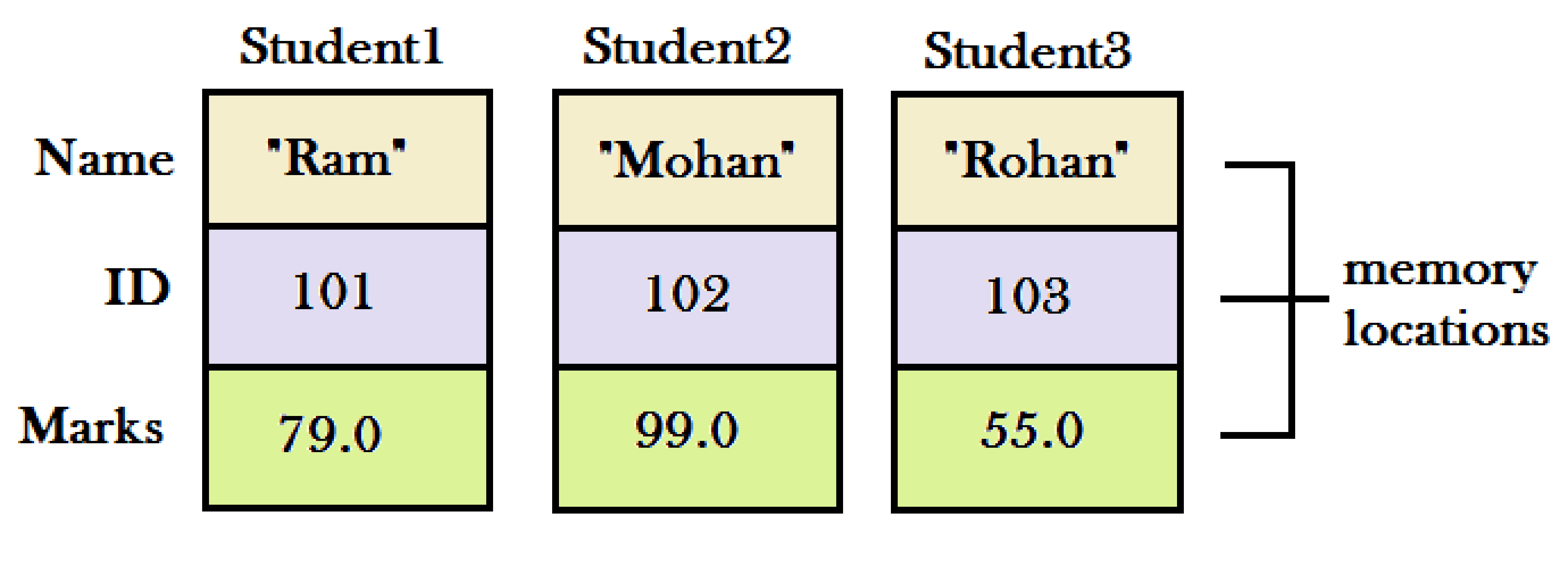
```
numbers[3] = 42;
```

Structure

Defnition

- A structure, in C, is a composite data type that allows you to group variables of different data types into a single unit.
- Each variable within a structure is referred to as a member or field, and you can access these members using the structure's name.

Defnition


Student1
Student2
Student3
Name
ID
Marks
'Ram'
101
79.0
'Mohan'
102
99.0
'Rohan'
103
55.0
memory
locations

Declaring a Structure

To declare a structure, you define its structure tag (a user-defined name for the structure type) and list its members.

```c
struct Person {
    char name[50];
    int age;
    float height;
};
```

Defining Structure Variables

After declaring a structure, you can define structure variables by specifying the structure name and assigning values to its members.

```c
struct Person person1;
person1.age = 30;
strcpy(person1.name, "John");
person1.height = 1.75;
```

Accessing Structure Members

You can access structure members using the dot (.) operator followed by the member name.

```
printf("Name: %s\n", person1.name);
printf("Age: %d\n", person1.age);
printf("Height: %.2f meters\n", person1.height);
```

Modifying Structure Members

You can modify the values of structure members by assigning new values using the assignment operator (=).

```
person1.age = 35;
```

Data Organization

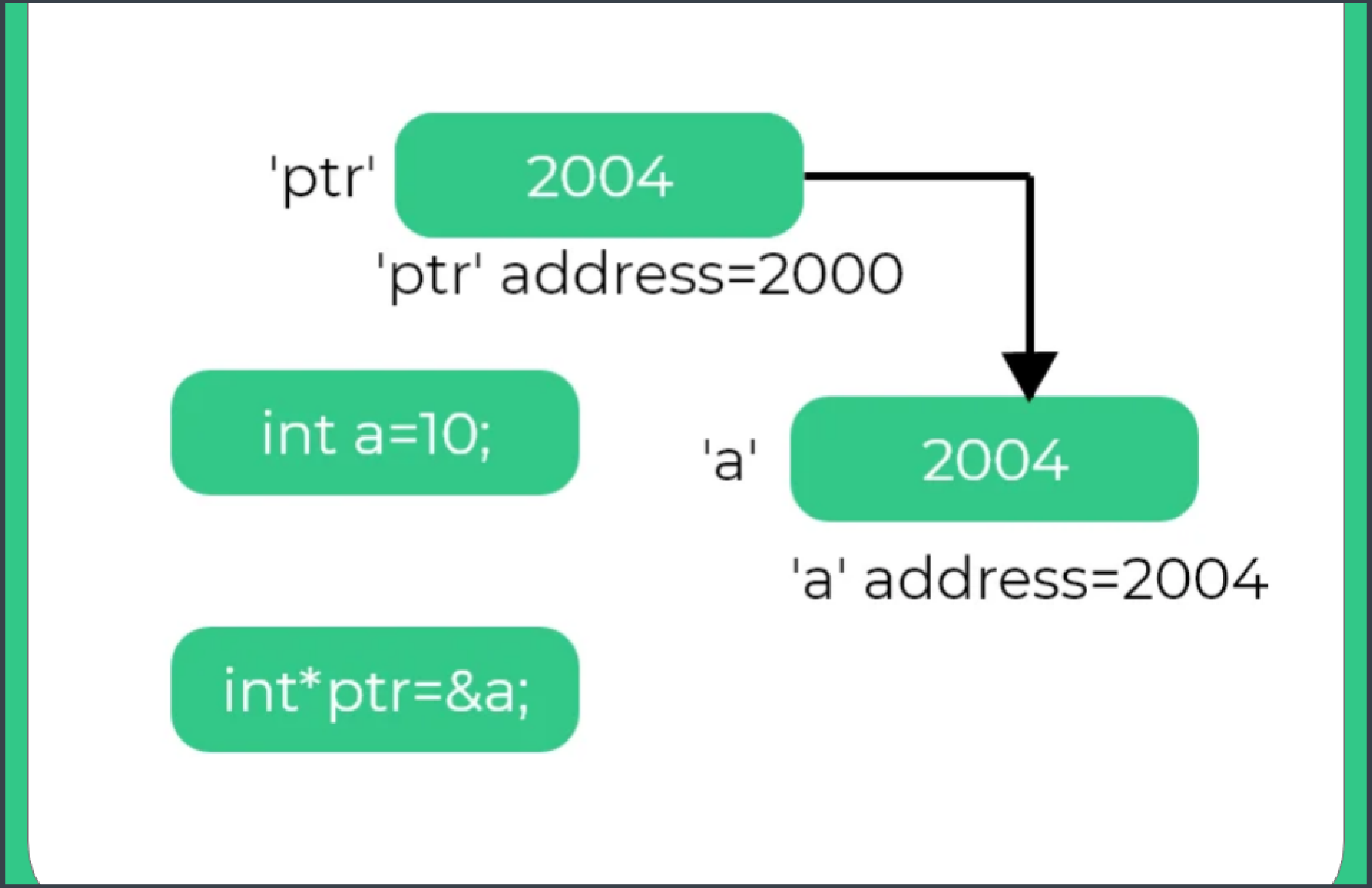
- Structures are commonly used to organize and manage data that naturally forms a group.
- They are essential for representing entities, records, or objects with multiple attributes.
- For example, you can use structures to represent employees, students, book records, and more.

Pointers

Defnition

A pointer is a variable that stores the memory address of another variable or object. It "points" to a location in memory.

Defnition



'ptr'
2004
'ptr' address=2000
int a=10;
'a'
2004
'a' address=2004
int*ptr=&a;

Declaring Pointers

Pointers are declared with a specific data type, indicating the type of data they point to.

```c
int *ptr;    // Declares an integer pointer
float *fp;   // Declares a float pointer
char *str;   // Declares a character pointer
```

Assigning a Pointer

Pointers can be assigned the memory address of an existing variable using the address-of operator &.

```c
int num = 42;
int *ptr = &num; // Assigns the memory address of 'num' to 'ptr'
```

Dereferencing a Pointer

To access the value stored at the memory location pointed to by a pointer, you use the dereference operator *.

```
int value = *ptr;
```

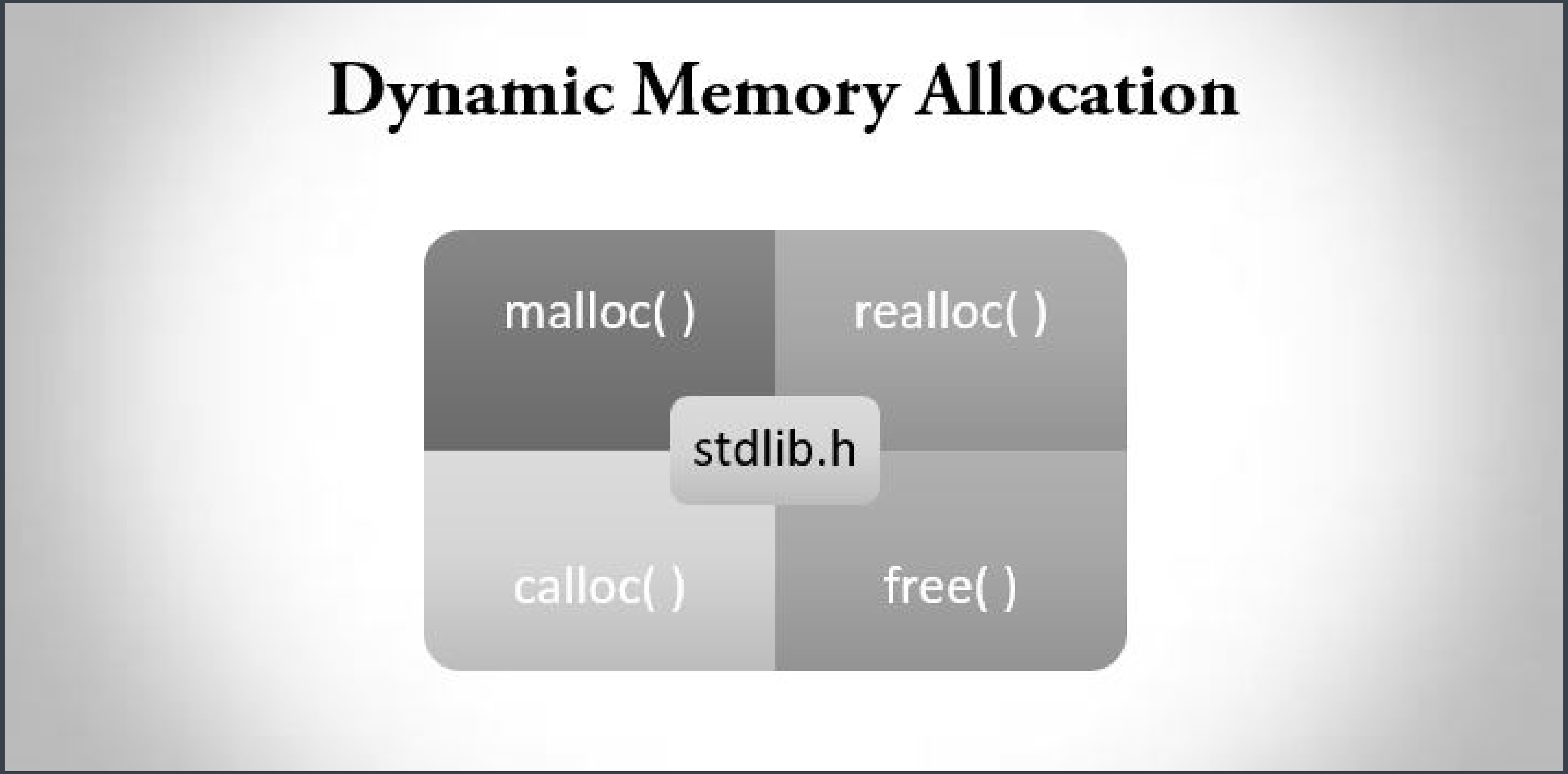
Retrieves the value stored at the memory location pointed to by 'ptr'

Dynamic Memory Allocation

Defnition

- Dynamic memory allocation is the process of allocating and deallocating memory for data structures during program execution.
- It allows you to allocate memory at runtime, enabling your program to adapt to varying data needs.

Defnition


Dynamic Memory Allocation
malloc( )
realloc( )
stdlib.h
calloc( )
free( )

Benefits

- Flexibility: Dynamic memory allocation allows you to adjust memory usage based on the actual needs of your program.
- Variable Data Structures: It is particularly useful for data structures like arrays and linked lists where the size can change during runtime.
- Efficient Memory Use: Memory is allocated only when required, minimizing memory wastage.

Allocating Memory with malloc

- malloc (Memory Allocation) is a library function in C used to allocate a block of memory of a specified size in bytes.
- It returns a pointer to the first byte of the allocated memory block.

```
data_type *pointer_variable = (data_type *)malloc(size_in_bytes);
```

Allocating Memory with calloc

- calloc (Contiguous Allocation) is another library function used for dynamic memory allocation.
- It allocates memory for an array of elements, initializes all bytes to zero, and returns a pointer to the first byte of the allocated memory block.

Deallocating Memory with free

free is used to release the dynamically allocated memory once it is no longer needed. Failing to do so can lead to memory leaks.

```
free(pointer_variable);
```

Error Handling

- It's important to check whether memory allocation was successful before using the allocated memory because both malloc and calloc can return a null pointer if there's insufficient memory.
- Proper error handling ensures that your program gracefully handles memory allocation failures.

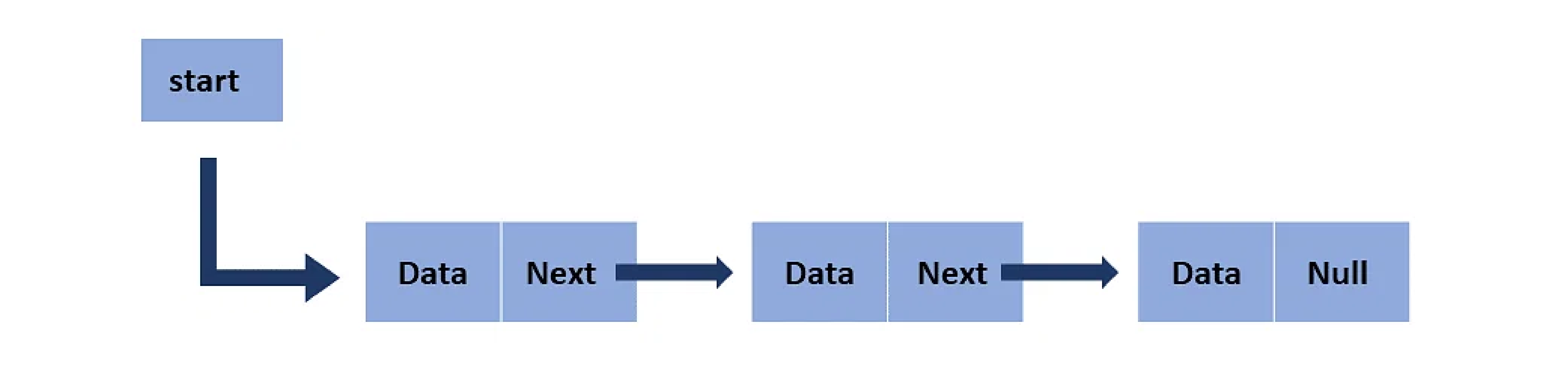
Error Handling

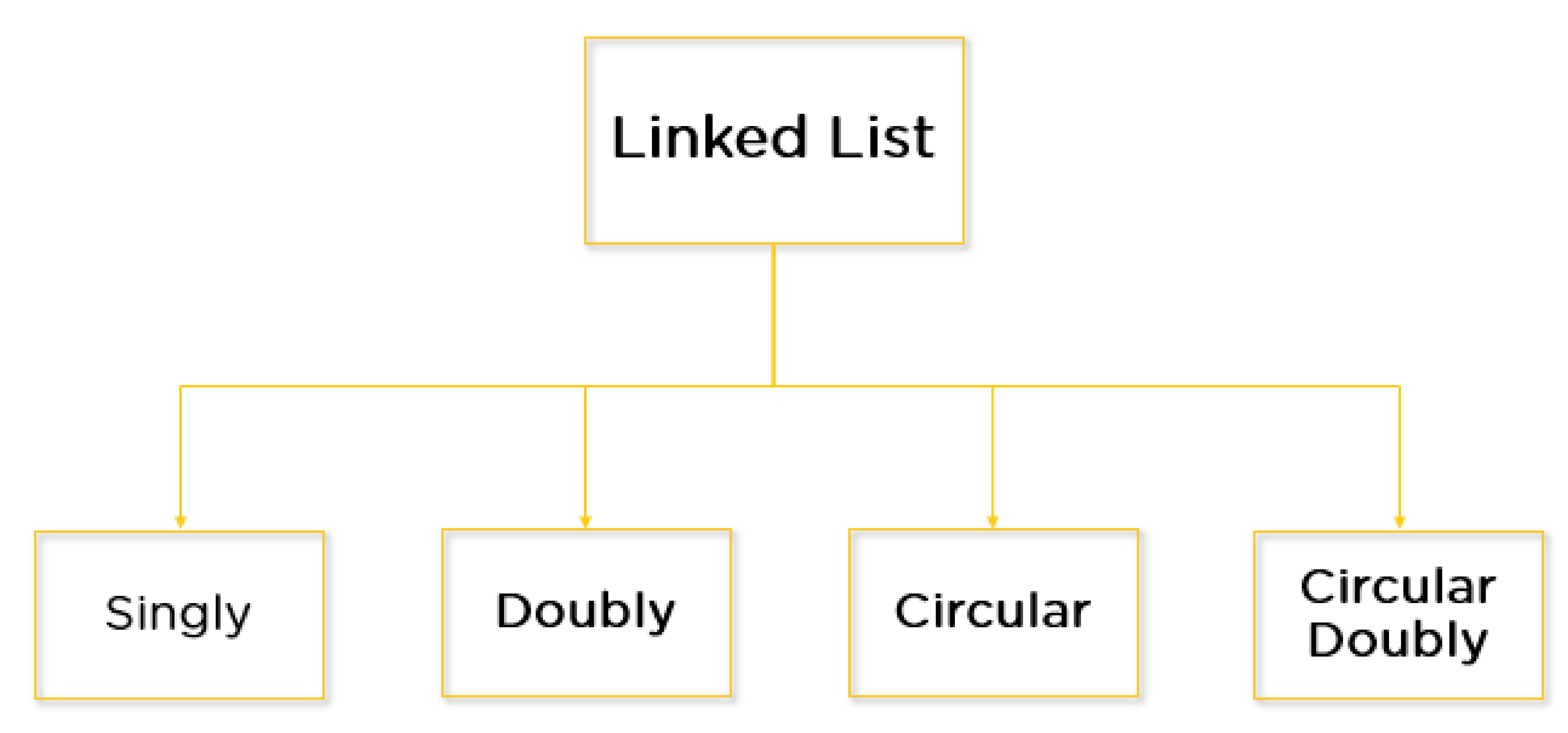
```
data_type *pointer_variable = (data_type *)malloc(size_in_bytes);
if (pointer_variable == NULL) {
    // Handle memory allocation failure
}
```

Linked List

Defnition

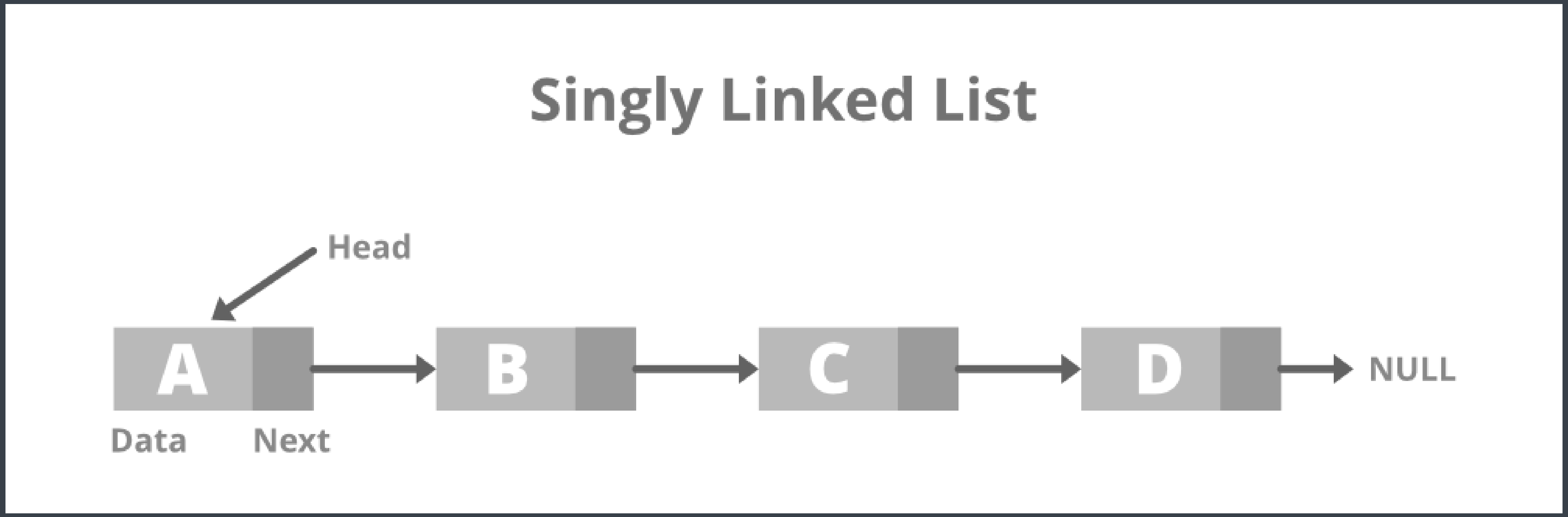
- A linked list is a linear data structure consisting of nodes. Each node has two components:
- Data: The actual data or value you want to store.
- Pointer (or link): A reference to the next node in the list.




Linked List
Singly
Doubly
Circular
Circular Doubly

Types

- Singly Linked List: Each node has a reference to the next node.

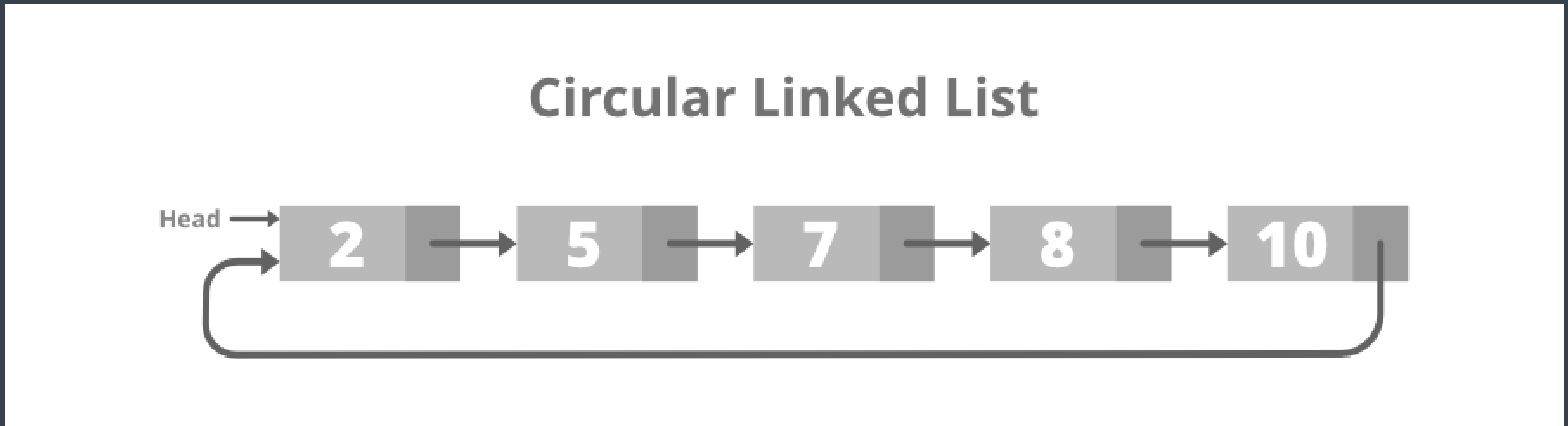


Types

- Doubly Linked List: Each node has references to both the next and previous nodes.

Types

- Circular Linked List: The last node's reference points back to the first node, forming a closed loop.



Operations

- Insertion: Adding a new node to the list.
- Deletion: Removing a node from the list.
- Traversal: Iterating through the list to access its elements.
- Searching: Finding a specific element in the list.
- Modification: Changing the value of a node.

Implementing Linked Lists

To implement a linked list in C, you need to define a structure for the list's nodes

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

Insertion in a Linked List

- Create a new node.
- Adjust the links (pointers) to ensure the new node is correctly integrated into the list.

```
new_node->next = previous_node->next;
previous_node->next = new_node;
```

Deletion in a Linked List

- Finding the node to be deleted and locating its previous node (if it exists).
- Adjusting the links to bypass the node to be deleted.
- Freeing the memory occupied by the deleted node using free

```
temp = previous_node->next;
previous_node->next = temp->next;
free(temp);
```

Traversal of a Linked List

Traversal is the process of visiting each node in the list. You can use a loop to iterate through the list, following the next pointers.

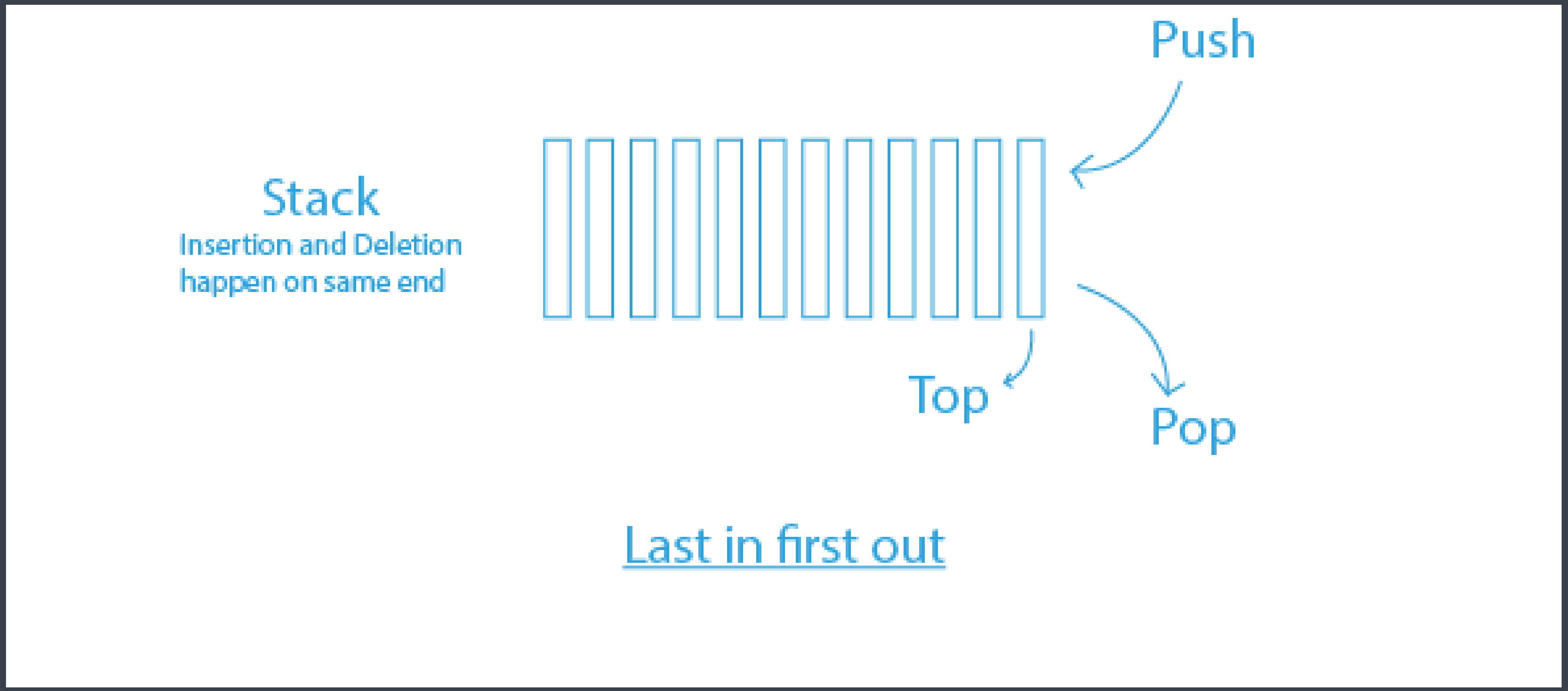
```
struct Node *current = head; // 'head' points to the first node
while (current != NULL) {
    // Process the current node (e.g., print its data)
    printf("%d ", current->data);
    current = current->next;
}
```

Stacks and Queues

Stack Defnition

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle.
- In a stack, the most recently added item is the first to be removed.
- Stacks are used in various scenarios, such as function call management (the call stack) and expression evaluation.

Stack Defnition



Stack Operations

- Push: To add an item to the stack, you push it onto the top of the stack.
- Pop: To remove the top item from the stack, you pop it.
- Peek (or Top): To view the top item without removing it, you peek.

Push Operation

- Pushing an item onto the stack involves adding an item to the top of the stack.
- It can be implemented using arrays or linked lists.

```
void push(int stack[], int *top, int data) {
    if (*top < MAX_SIZE - 1) {
        stack[++(*top)] = data;
    } else {
        // Stack overflow
    }
}
```

Pop Operation

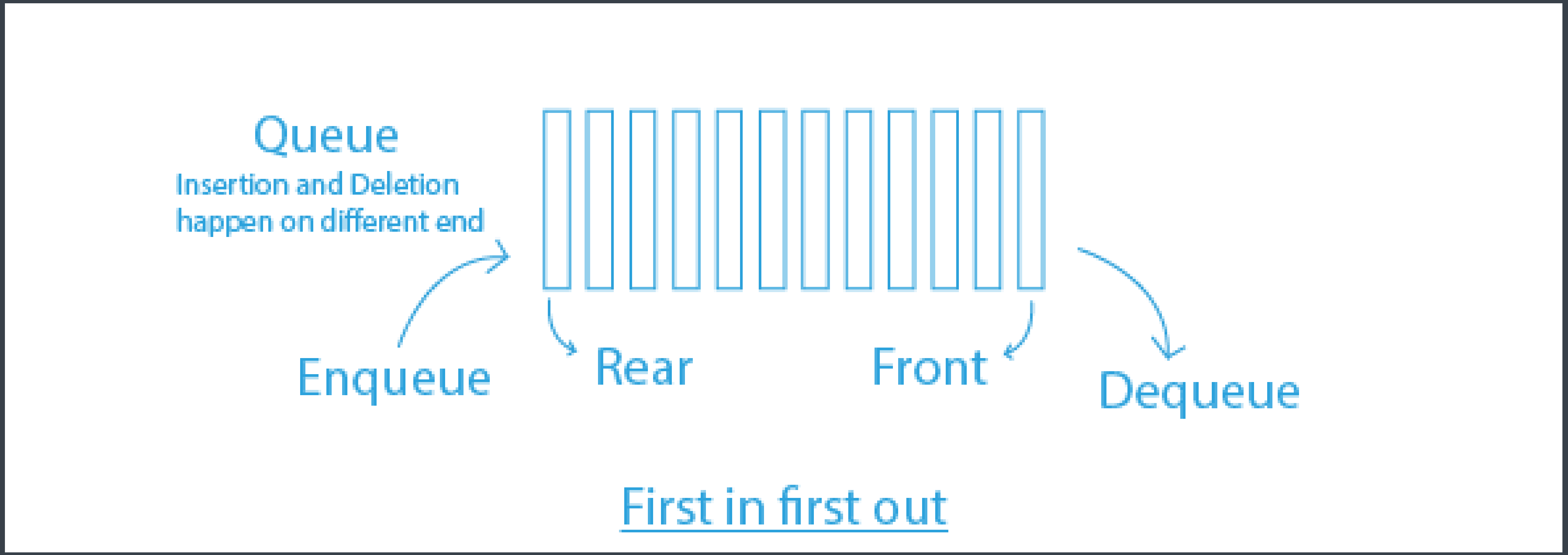
- Popping an item from the stack involves removing the top item.

```
int pop(int stack[], int *top) {
    if (*top >= 0) {
        return stack[(*top)--];
    } else {
        // Stack underflow
        return -1; // An error value
    }
}
```

Queue Defnition

- A queue is another linear data structure, but it follows the First-In-First-Out (FIFO) principle.
- In a queue, the oldest item is the first to be removed.
- Queues are used in scenarios like task scheduling, print job management, and more.

Queue Defnition



Queue Operations

- Enqueue: To add an item to the queue, you enqueue it at the rear.
- Dequeue: To remove the front item from the queue, you dequeue it.
- Front: To view the front item without removing it, you check the front

Enqueue Operation

Enqueuing an item onto the queue involves adding an item to the rear.

```
void enqueue(int queue[], int *front, int *rear, int data) {
    if (*rear < MAX_SIZE - 1) {
        queue[++(*rear)] = data;
    } else {
        // Queue is full
    }
}
```

Dequeue Operation

Dequeuing an item from the queue involves removing the front item.

```
int dequeue(int queue[], int *front, int *rear) {
    if (*front <= *rear) {
        return queue[(*front)++];
    } else {
        // Queue is empty
        return -1; // An error value
    }
}
```