Data Structures usng C and C++

Section 1: Before we Start

Introduction

Data structures are defined as the way data is organized in main memory so that a program can use it efficiently during execution.

Programs consist of **code and data**; data structures focus on how data is arranged for efficient operations.

Types of Data Structures

- **Physical data structures**: Arrays, Matrices, Linked Lists (define memory arrangement).
- **Logical data structures**: Stacks, Queues, Trees, Graphs, etc. (define how data is used).
- Arrays and matrices are built into most languages; linked lists must be implemented manually in C.

Why Study Data Structures

- They are a **core subject** in computer science academics.
- Essential for **software development**; applications cannot be built without them.

Levels of Learning Data Structures

1. Basic understanding of what they are and where to use them.
2. In-depth knowledge of operations and **time/space complexity analysis**.
3. Ability to **implement data structures from scratch**.

- The course targets **Level 3**, including implementation and analysis.

Programming Languages

- Any language can be used conceptually.
- Modern languages provide built-in data structures (STL in C++, Collections in Java/C#, containers in Python, etc.).
- Understanding usage requires only basic knowledge, but implementation requires deeper study.

Why C Language Is Used

- C has **no built-in data structures**, making it ideal for learning implementations from scratch.
- Helps clearly understand internal operations.
- Concepts can be easily transferred to **C++**, and adapted to Java or C#.

Course Organization

- Starts with a **brush-up of essential C and C++ concepts** (functions, structures, classes, templates, parameter passing).
- Covers **sorting techniques** (bubble, selection, insertion, etc.) with implementation and analysis.
- Begins with a detailed section on **recursion**, explaining its importance in problem-solving despite efficiency concerns.

Recursion and Problem Solving

- Recursion is fundamental to **mathematical problem-solving**.
- Many problems are first solved recursively, then converted to loops for efficiency.
- Understanding recursion is crucial for strong problem-solving skills.

Algorithms Clarification

- The course focuses on **data structures and algorithms applied to them**, not large-scale industry algorithms (e.g., Google or Facebook algorithms).
- General algorithms are treated as a **separate subject** and covered elsewhere.

Section 2: Essential C and C++ Concepts

Array as a Pointer

An **array** in C++ is a **contiguous block of memory** that stores multiple elements of the **same data type** under a single identifier.

It provides **indexed access** to elements using zero-based indexing.

```
/*
    Array as a pointer example
*/
#include <iostream>

void printArray(const int *A, int size)
{
    // A[1] = 10;
    for (int i = 0; i < size; i++)
    {
        // std::cout << A[i];
        std::cout << *(A + i);
    }
    const int *B = A;
    for (int i = 0; i < size; i++)
    {
        // std::cout << A[i];
        std::cout << *(B + i);
    }
}

int *func(int n)
{
    // let's create memory to store 5 variables in heap memory
    int *p;
    p = (int *)malloc(n * sizeof(int));
    return (p);
}

int main()
{
    int *a;
    a = func(3);

    int A[3] = {1, 2, 3};
    printArray(A, 3);
    return 0;
}
```

Structure as a Pointer

A **structure** in C++ is a **user-defined data type** that groups variables of **different data types** under a single name.

It is used to model a real-world entity by bundling related data together.

```cpp
#include <iostream>

/*
    Passing structure as a Paramter
*/

struct Rectangle
{
    int length;
    int breadth;
};

struct Rect
{
    int A[3] = {1, 2, 3};
};

void print(struct Rect r)
{
    for (int i = 0; i < 3; i++)
    {
        // std::cout << r.A[i] << std::endl;
        std::cout << *(r.A + i) << std::endl;
    }
}

// Pass by reference
void area(struct Rectangle &r)
{
    r.length = 20;
    std::cout << r.length * r.breadth << std::endl;
}

// Pass by value
// void area(struct Rectangle r)
// {
//     std::cout << r.length * r.breadth << std::endl;
// }

int main()
{
    struct Rectangle r = {10, 20};
    area(r);
    std::cout << r.length << std::endl;
    Rect r1;
    print(r1);
    return 0;
}
```

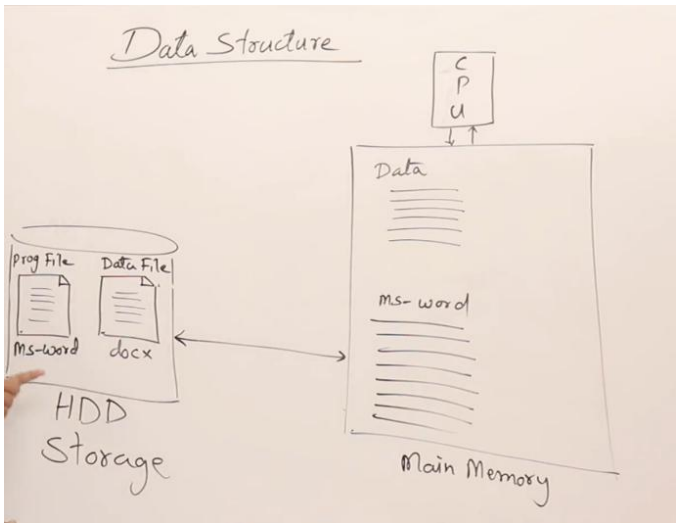
Template Class

A **template class** in C++ is a *blueprint for generating classes* where the data type is specified later, at compile time.
It enables **generic programming**—writing type-independent, reusable, and type-safe code.

```cpp
#include <iostream>

template <class T>
class Arithmetic
{
public:
    T _a;
    T _b;

    Arithmetic(T a, T b);

    T area();
};

template <class T>
Arithmetic<T>::Arithmetic(T a, T b) : _a{a}, _b{b}
{
}

template <class T>
T Arithmetic<T>::area()
{
    return _a * _b;
}

int main()
{
    Arithmetic<int> a(10, 20);
    std::cout << a.area() << std::endl;

    Arithmetic<float> f(10.2f, 11.3f);
    std::cout << f.area() << std::endl;

    return 0;
}
```

Section 4: Introduction

Introduction

- **Data is fundamental to any program.**
  A program is a set of instructions that performs operations on data to produce results. Without data, instructions have no purpose.
- **Data Structure – Definition:**
  A data structure is the arrangement and organization of data in **main memory (RAM)** during program execution so that operations on the data can be performed efficiently.
- **Where Data Structures Exist:**
  - Programs and data are stored permanently on **secondary storage** (hard disk, mobile storage).
  - When a program runs, both the program instructions and required data are loaded into **main memory (RAM)**.
  - The way data is organized in RAM for efficient processing is called a **data structure**.
  - Examples: Arrays, Linked Lists, Trees, Hash Tables.
- 

- **Execution Flow Example (MS Word):**
  - MS Word program is stored on disk.
  - When launched, it is loaded into RAM.
  - When a document is opened, the document data is also loaded into RAM.
  - The program processes the data in RAM using appropriate data structures.
- **Key Distinction:**
  - Data cannot be processed directly from disk.
  - It must be loaded into main memory.
  - Efficient organization in memory is critical for performance.

Database

- A **Database** is the organized arrangement of data on permanent storage (disk).
- Typically structured in tables (e.g., relational model).

- Used mainly for commercial and business applications.
- When applications use database data, it must be loaded into RAM and organized using data structures.

Difference:

- Data Structure → Organization in RAM (temporary, during execution)
- Database → Organization on disk (permanent storage)

Data Warehouse

- Large-scale storage of historical or legacy business data.
- Contains massive volumes of old, inactive data.
- Stored across arrays of disks.
- Used for analysis, decision-making, policy planning.
- Data mining algorithms are applied to analyze this data.

Big Data

- Refers to extremely large datasets, especially from the Internet.
- Includes data about people, places, transactions, behavior, etc.
- Focuses on storing, managing, and analyzing very large-scale data.
- Used in business intelligence, governance, and management decisions.

Final Comparison

Term	Location	Purpose
Data Structure	Main Memory (RAM)	Efficient data processing during execution
Database	Disk (Permanent Storage)	Organized storage of operational data
Data Warehouse	Array of Disks	Storage of historical/legacy business data
Big Data	Large-scale distributed systems	Analysis of massive internet-scale data

In essence, **data structures enable efficient processing inside memory**, while databases, data warehouses, and big data deal with organized storage and large-scale data management outside memory.

Stack vs Heap Memory

1. Introduction

The video explains:

- How main memory is organized
- How a program uses main memory
- Static memory allocation
- Dynamic memory allocation (introduction begins later)

2. Understanding Main Memory

- Memory is divided into **bytes**.
- Each byte has a **unique linear address** (not 2D coordinates).
- Addresses start from **0** and increase sequentially.

Example assumption used for explanation:

- Memory size = **64 KB**
- 1 KB = 1024 bytes
- 64 KB = 64 × 1024 = **65536 bytes**
- Address range: **0 to 65535**

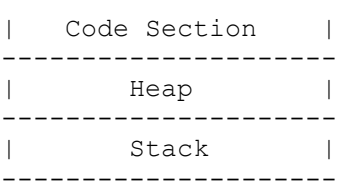
Although modern systems use RAM in GBs (4GB, 8GB, etc.), memory is often conceptually divided into manageable blocks (segments). For explanation, the video assumes one 64KB segment.

3. How a Program Uses Main Memory

When a program runs:

1. The program (machine code) is loaded from disk into memory.
2. Main memory is divided into three sections:

Memory Layout



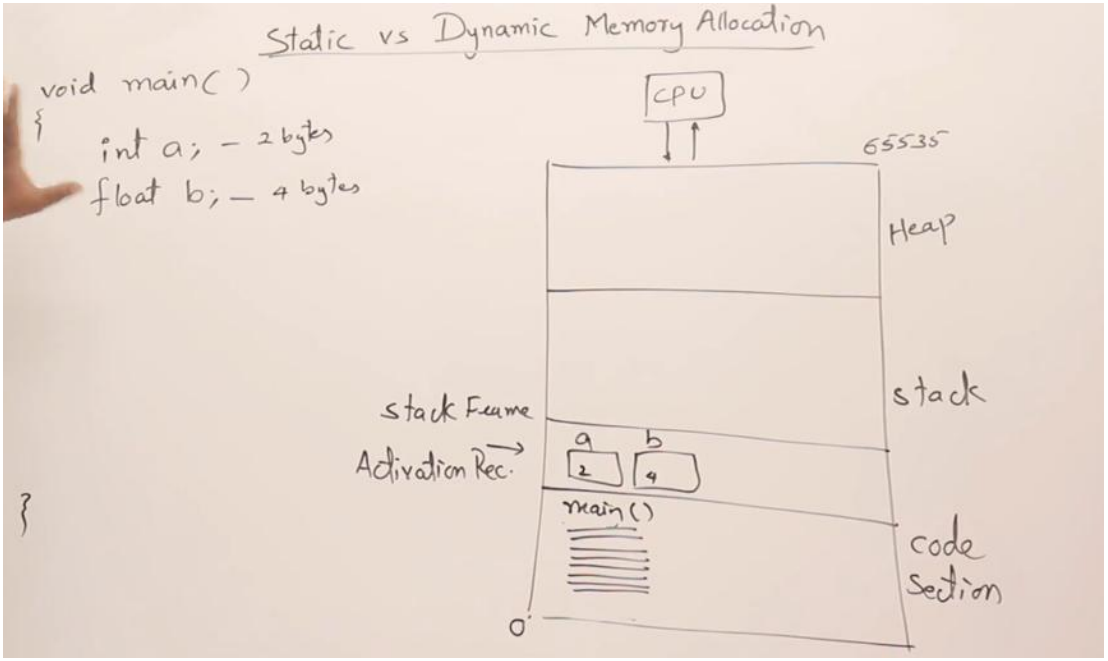
Sections Explained

1. **Code Section**
    - Contains the machine code of the program.

- Size depends on program size.

2. **Stack**
   - Stores local variables and function-related data.
   - Works automatically.
3. **Heap**
   - Used for dynamic memory allocation.
   - Managed manually by the programmer.

4. Stack Memory Example



Example program:

```
int main() {
    int x;
    float y;
}
```

Assumptions for explanation:

- `int` = 2 bytes
- `float` = 4 bytes
- Total memory required = 6 bytes

(Note: In real systems, `int` may be 4 bytes depending on compiler and architecture.)

What Happens?

- The compiler determines at **compile time** that `main()` needs 6 bytes.
- When the program runs, 6 bytes are allocated inside the **stack**.
- This memory block is called:

- **Stack Frame**
- **Activation Record**

For `main()`:

```
Stack Frame of main()
---------------------
| int x  (2 bytes)  |
| float y (4 bytes) |
---------------------
Total = 6 bytes
```

5. Static Memory Allocation

Definition

Static memory allocation means:

- The **size of memory required is determined at compile time**.
- Allocation happens automatically when the program runs.
- Memory is allocated in the **stack**.

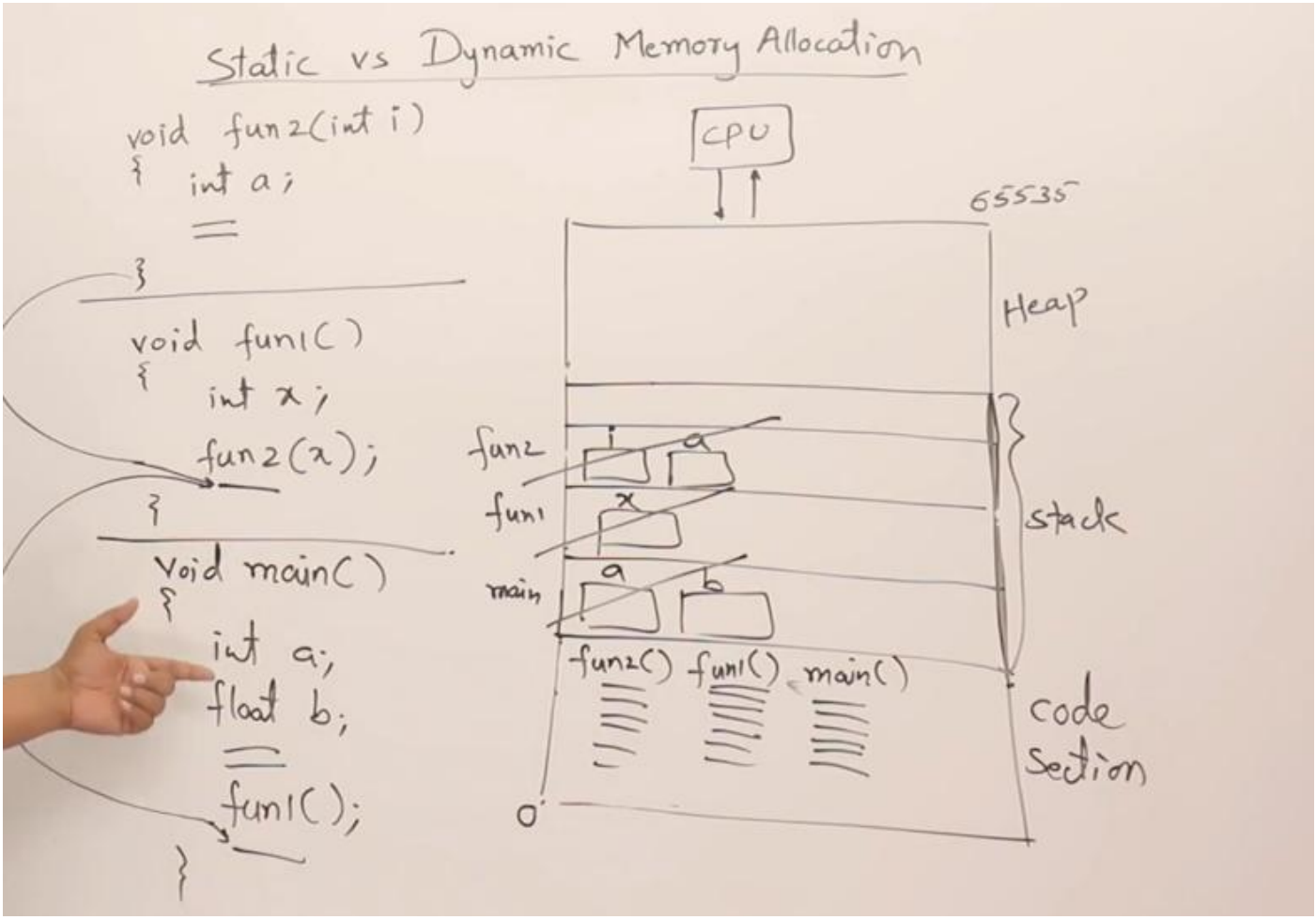
Why is it called "Static"?

Because:

- The size is fixed.
- The compiler already knows how much memory is needed.
- Everything is decided **before runtime**.

6. Key Takeaways

- Memory is divided into bytes with linear addresses.
- Programs divide memory into:
  - Code section
  - Stack
  - Heap
- Local variables are stored in the stack.
- Stack memory allocation is static because size is known at compile time.
- Each function gets a stack frame (activation record).



Stack Memory Allocation During Function Calls

- When a program starts, its **machine code** is loaded into the **code section** of memory.
- Execution begins with the main() function.

Sequence of Function Calls

Consider:

- main() has local variables a and b, and calls fun1().
- fun1() has a local variable x, and calls fun2(x).
- fun2() has a parameter i and a local variable a.

How Memory is Allocated in the Stack

1. **Execution enters main()**
    - Memory for a and b is allocated in the stack.
    - This forms the first **activation record** (stack frame).
2. **main() calls fun1()**
    - Control transfers to fun1().

- Memory for local variable x is allocated **above** main()'s activation record.
- fun1()'s activation record becomes the top of the stack.

3. **fun1() calls fun2(x)**
   - Control transfers to fun2().
   - Memory for parameter i and local variable a is allocated.
   - fun2()'s activation record is now at the top of the stack.

Stack Behavior (LIFO Principle)

- The **currently executing function** always uses the **topmost activation record**.
- When fun2() finishes:
  - Its activation record is removed (popped).
  - Control returns to fun1().
- When fun1() finishes:
  - Its activation record is removed.
  - Control returns to main().
- When main() finishes:
  - Its activation record is removed.
  - Program execution ends.

This follows **Last-In, First-Out (LIFO)** order:

Create: main → fun1 → fun2
Delete: fun2 → fun1 → main

Key Observations

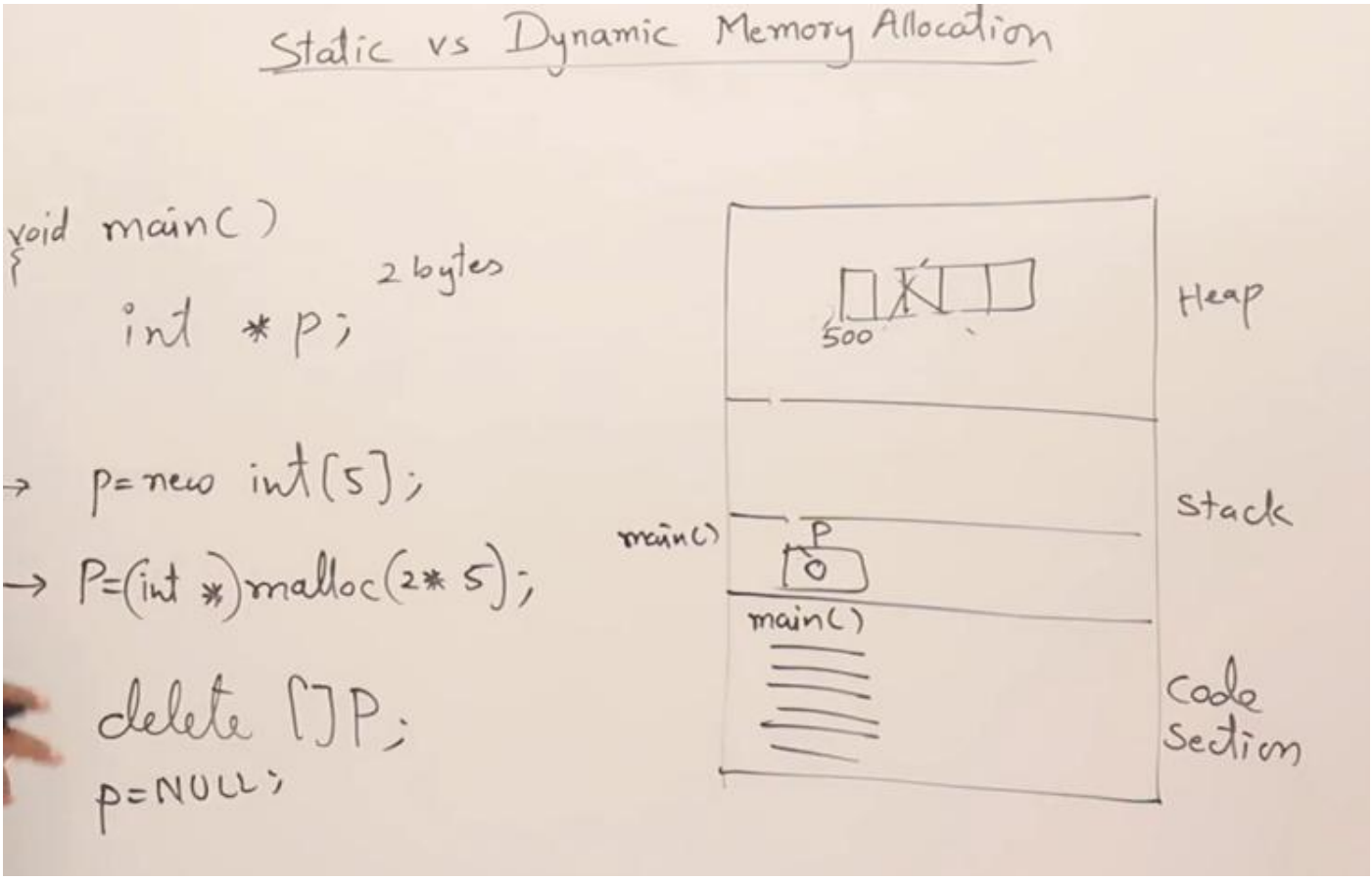
- Each function call creates a separate **activation record** in the stack.
- Activation records remain in memory until the function completes.
- Memory for:
  - Local variables
  - Function parameters
    is allocated inside the **stack**.
- Allocation and deallocation are:
  - **Automatic**
  - Managed by the compiler and runtime system
  - Not manually controlled by the programmer.

Final Conclusion

Stack memory is used for function calls because:

- Activation records are created in a top-down manner.
- They are deleted in reverse order.
- Memory is automatically allocated when a function starts.
- Memory is automatically freed when the function ends.

This structured push-pop behavior is why this memory region is called the **stack**.



Heap Memory and Dynamic Allocation

- **Heap memory** is used for **dynamic memory allocation**.
- The term *heap* refers to memory that is **not organized like the stack**.
    - Stack memory is structured (activation records are created and destroyed in order).
    - Heap memory is comparatively unstructured.

Key Points About Heap Memory

1. **Heap is treated as a resource**

- Like a printer resource in applications:
  - Request it when needed.
  - Release it when done.
- Memory must be explicitly allocated and deallocated.

2. **Programs cannot directly access heap memory**
   - Heap memory is accessed **through pointers**.
   - The pointer stores the address of the allocated heap memory.

How Memory is Allocated

- When a pointer is declared inside a function:
  - The pointer variable itself is stored in the **stack** (inside the activation record).
  - It holds the address of memory allocated in the **heap**.

In C++:

```
int *p = new int[5];
```

- Allocates memory for 5 integers in the heap.
- p stores the starting address.

In C:

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

Important Practice: Deallocation

- If memory is allocated using new, it must be released using:

```
delete[] p;
```

- After deleting, set pointer to:

```
p = nullptr;
```

What Happens If You Don't Release Memory?

- The memory remains reserved.
- It cannot be reused.
- This causes a **memory leak**.
- Continuous leaks may eventually exhaust heap memory.

Final Conclusion

- **Static memory allocation** → Done in the **stack** (automatic).
- **Dynamic memory allocation** → Done in the **heap** using pointers (manual control).
- Heap memory must be **explicitly allocated and explicitly released** to avoid memory leaks.

Physical vs Logical Data Structures

1. Classification of Data Structures

Data structures are categorized into:

A. Physical Data Structures

- **Array**
- **Linked List**

These are called *physical* because they define **how memory is organized and allocated**.

Array

- Collection of **contiguous memory locations**.
- Fixed size (cannot grow or shrink after creation).
- Can be created in **stack or heap**.
- Suitable when the **maximum size is known in advance**.
- Directly supported in languages like C, C++, and Java.

Linked List

- Collection of **nodes** (each node contains data + link to next node).
- **Dynamic size** (can grow or shrink at runtime).
- Always created in the **heap**.
- Head pointer may be in stack.
- Used when the **size is not known beforehand**.

Key Difference

Array	Linked List
Fixed size	Variable size
Contiguous memory	Non-contiguous nodes
Can be stack or heap	Always heap
Best when size is known	Best when size is unknown

2. Logical Data Structures

Examples:

- Stack
- Queue
- Tree
- Graph
- Hash Table

These define **how data is used and accessed**, not how it is stored physically.

Types of Logical Data Structures

Linear:

- Stack → LIFO (Last In First Out)
- Queue → FIFO (First In First Out)

Non-Linear:

- Tree → Hierarchical structure
- Graph → Nodes with connections

Tabular:

- Hash Table → Key-value based structure

Important Concept

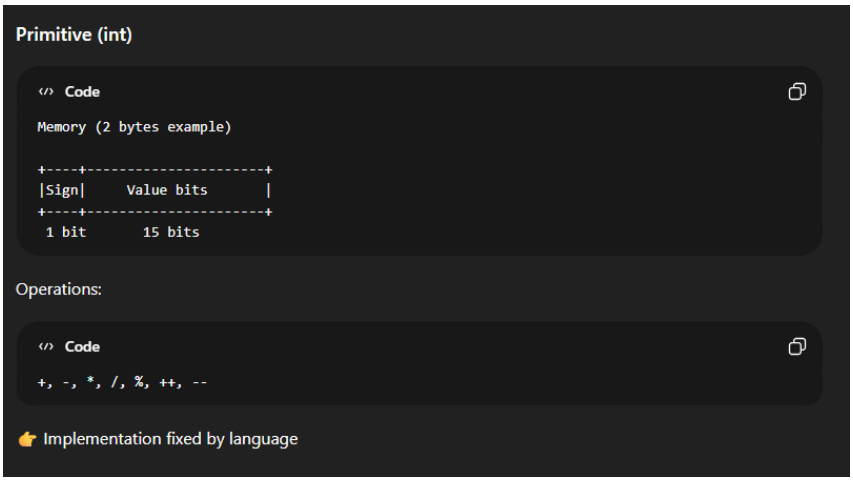
Logical data structures are **implemented using physical data structures**:

- Using **Array**
- Using **Linked List**
- Or a combination of both

ADT

An **Abstract Data Type (ADT)** is a conceptual model that defines a data structure by:

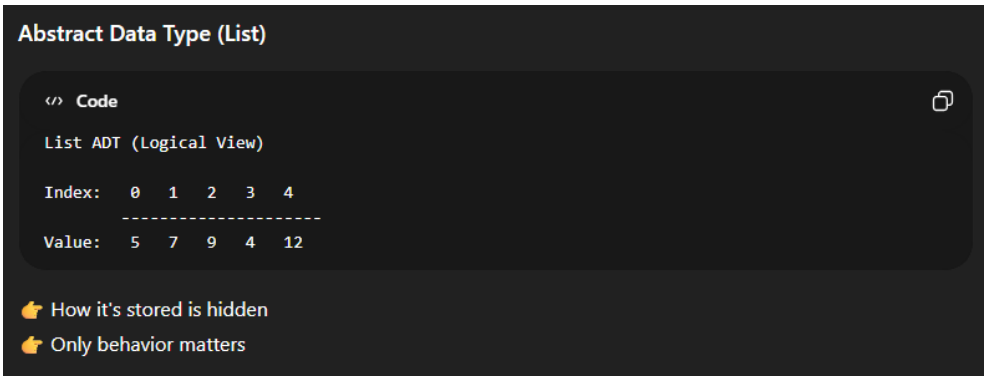
1. **How data is represented**
2. **What operations are allowed on the data**
3. **Without exposing how those operations are implemented**

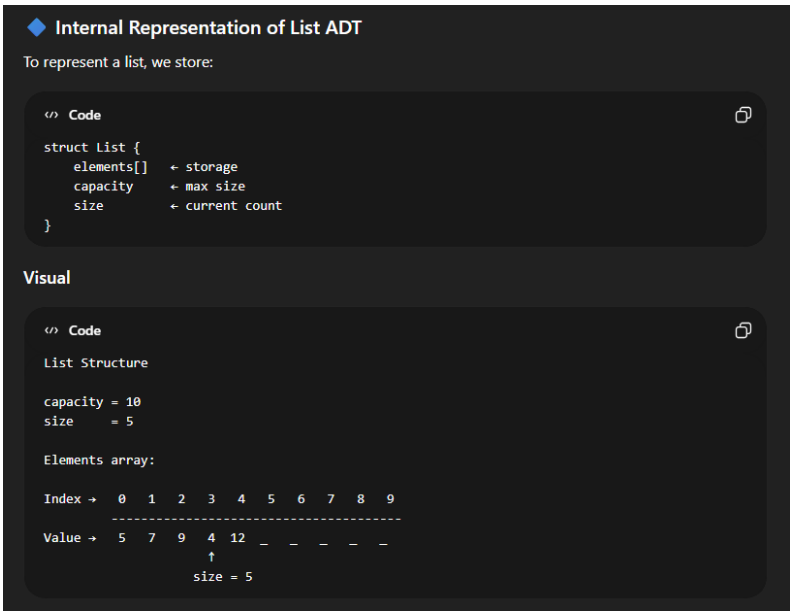


A normal data type (like `int`) specifies both storage and permitted operations (e.g., arithmetic, relational, increment/decrement). The term **abstract** means the internal implementation details are hidden from the user—only the interface matters.

ADT emerged with object-oriented programming, where developers can create their own data types using classes. Users of an ADT do not need to know whether it is implemented using arrays, linked lists, or any other structure.

List as an ADT





A **list ADT** represents a collection of elements indexed sequentially.

Representation requires:

- Storage space for elements
- Capacity (maximum size)
- Current size (number of elements present)

The list can be implemented internally using:

- Arrays
- Linked lists
  (implementation choice is hidden from the user)

Common Operations on List ADT

- **Add / Append** — add element at the end
- **Insert** — add element at a specific index (requires shifting)
- **Remove** — delete element at an index (shift remaining elements)
- **Set / Replace** — update element at an index
- **Get** — retrieve element at an index
- **Search / Contains** — find index of a key
- **Sort** — arrange elements in order
- Other possible operations: reverse, merge, split

Key Takeaway

An ADT combines **data representation + allowed operations** while **hiding implementation details**. In C++, any class that encapsulates data and its operations effectively defines an ADT. Data structures like arrays, linked lists, stacks, queues, trees, graphs, and hash tables are commonly expressed as ADTs.

Time and Space Complexity

Time Complexity measures how the running time of an algorithm grows with input size. Instead of exact time, we express it in terms of **n** (number of elements). The key idea: time depends on the **procedure or algorithm used**, not on the machine.

Core Patterns

- **Single pass through n elements → Order(n)**
  Example: summing or searching an array using one loop.
- **Nested loops over n elements → Order(n^2)**
  Example: comparing each element with every other element (sorting-style logic).
- **Triangular nested processing** (n-1, n-2, ...) → still **Order(n^2)**
  Because the highest-degree term dominates.

- **Repeated halving of data → Order(log n)**
  Example: binary-search-like behavior where the problem size is divided by 2 each step.

👉 Key rule: focus on the **degree of growth**, not exact counts.

How to Analyze

You can estimate time complexity:

- From the **algorithm's behavior** (preferred)
- From the **code structure** (loops, nesting, halving patterns)

Examples:

- for (i=0; i<n; i++) → O(n)
- Two nested loops → $O(n^2)$
- Loop where i = i/2 each iteration → O(log n)

Always read the loop carefully—don't assume every loop is O(n).

Data Structure Examples

- **Array / Linked list full traversal** → O(n)
- **Matrix (n × n) full traversal** → $O(n^2)$
- **Binary tree path traversal (height)** → O(log n)
- **Binary tree full traversal** → O(n)

Space Complexity

Space Complexity measures how much memory a program uses.

- Array of n elements → **O(n)**
- Linked list with n nodes → **O(n)**
- Matrix n × n → **$O(n^2)$**
- Tree with n nodes → **O(n)**

We focus on how space grows with **n**, not exact bytes.

Key Takeaway

- Time complexity depends on the **work performed by the algorithm**.
- Space complexity depends on **memory growth with input size**.
- In analysis, we keep the **dominant term** (highest power of n).
- Understanding loop behavior is the fastest way to estimate complexity.

Time and Space Complexity from Code

🎯 Core Assumption

- Every **simple statement** (assignment, arithmetic, condition) takes **1 unit time**.
- Build a **time function f(n)** by counting executions.
- The **highest-degree term** determines the time complexity.

✅ Example 1: Swap Function

Pseudo:

```
temp = x
x = y
y = temp
```

Analysis

- 3 simple assignments → each takes 1 unit
- Total time = **3**

Result

- $f(n) = 3$
- Constant time → **O(1)**

Key idea: No dependence on input size.

✅ Example 2: Single Loop (Array Sum)

Structure:

```
for i = 0 to n-1:
    sum = sum + A[i]
```

Execution Count

- Loop condition ≈ **n+1**
- Body executes **n times**
- Final return → **1**

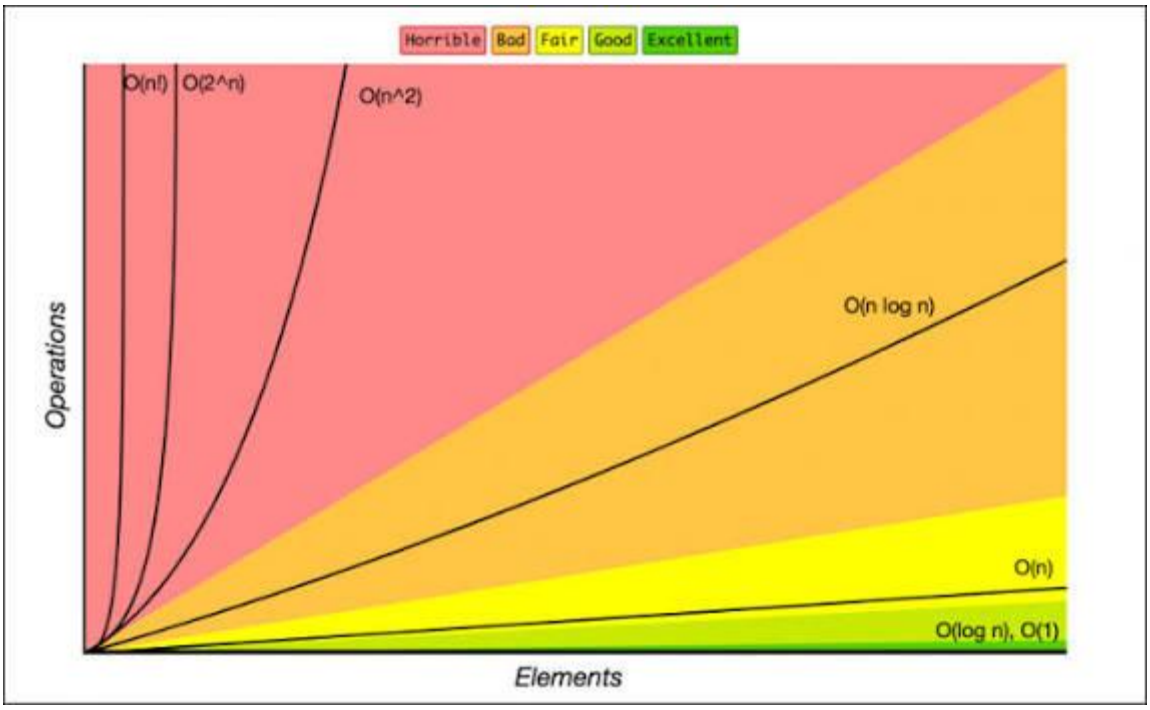
Time Function

$$f(n) = 2n + 3$$

Result

- Highest degree = 1
- Time complexity = **O(n)**

📊 Diagram: Single Loop Growth



Interpretation: Time grows proportionally with input size.

✅ Example 3: Nested Loops

Structure:

```
for i = 0 to n-1:
    for j = 0 to n-1:
        statement
```

Analysis

- Outer loop → n
- Inner loop → n for each outer iteration
- Total executions → $n \times n = n^2$

Time Function

$$f(n) = 2n^2 + 2n + 1$$

Result

- Highest degree = 2
- Time complexity = **O(n²)**

✅ Important Concept: Function Calls

If a function calls another function:

`fun1() → calls fun2()`

and

- `fun2()` takes **O(n)**

then

- `fun1()` is also **O(n)** (not O(1))

Rule: Include the cost of called functions

🔑 Key Rules to Remember

- Simple statement → **O(1)**
- Single loop (1…n) → **O(n)**
- Nested loops → **O(n²)** (or higher)
- Function calls → include callee complexity
- Ignore constants and lower-order terms
- Focus on **highest power of n**

🧠 Practical Insight

Most time complexity comes from **loops**:

- One loop → usually **n**
- Two nested loops → usually **n²**
- Three nested loops → usually **n³**

But if loop increments differently (e.g., `i *= 2`), complexity may be **O(log n)** — you must read carefully.

.