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.

```
#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;
}
```

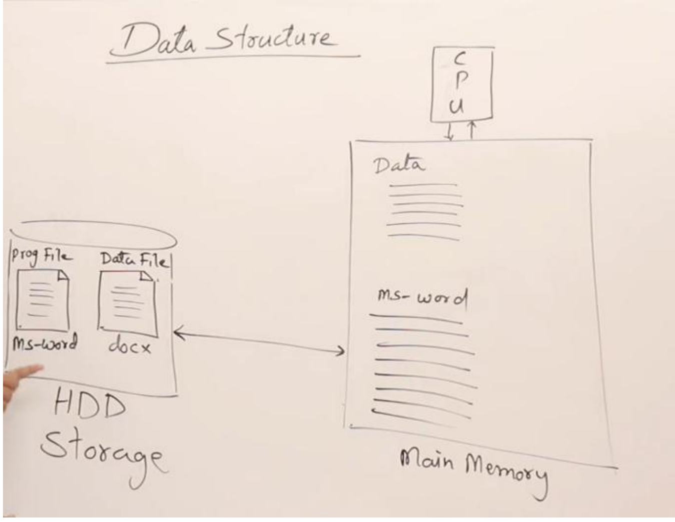
Template Class

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

Physical vs Logical Data Structures

ADT

Time and Space Complexity

Time and Space Complexity from Code

.