* Session 0

* Session 1: Introduction to Data Structures

Objectives:

- Understand what data structures are
- · Learn why they are needed
- Real-world analogies

Content:

- Definition of Data Structure
- Example: Think of a school storing student records we need organized ways to manage data
- Classification of Data Structures:
 - o **Primitive:** int, float, char, bool
 - o Non-Primitive: Arrays, Structures, Linked Lists, Stacks, Queues, Trees

Activities:

- Draw a chart of classifications
- Quiz: Primitive vs Non-Primitive

* Session 2: Data Structure Operations

Objectives:

Understand common operations on data structures

Content:

• Create (allocating space)

- Insert
- Delete
- Update
- Traverse
- Search
- Use diagrams and animations to show each operation

Activities:

• Live demo using an array (inserting and deleting values)

* Session 3: Review of Arrays

Objectives:

- Revise arrays in C++
- How data is stored in memory

Content:

- Static arrays: Declaration, initialization
- Accessing elements with index
- Memory layout of arrays

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

• Traversal using loops

Activities:

- Practice writing loops
- Show memory blocks for array

* Session 4: Structures in C++

Objectives:

• Learn to define and use structures

Content:

- What is a structure? (user-defined data type)
- Structure syntax
- Declaring variables
- Accessing members

```
struct Student {
    int id;
    char name[50];
    float marks;
};
```

Activities:

- Create a structure for Book, Car, Employee
- Show how it differs from arrays

* Session 5: Self-Referential Structures

Objectives:

- Understand how structures can reference themselves
- Foundation for Linked Lists

Content:

- Concept of self-referencing
- Example:

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

Use diagram to explain "chaining"

Activities:

Draw nodes on board and link with arrows

* Session 6: Unions in C++

Objectives:

• Understand what a union is and how it works

Content:

- Syntax of unions
- Difference between structure and union (memory sharing)
- Example:

```
union Data {
```

```
int i;
float f;
char str[20];
};
```

Activities:

- Show memory usage with sizeof
- Compare with struct

* Session 7: Pointers in C++

Objectives:

• Learn what pointers are and how they point to memory

Content:

- Pointer declaration and initialization
- & (address of), * (dereferencing)
- Pointers with arrays and structures

```
int a = 10;
int* p = &a;
cout << *p;</pre>
```

Activities:

- Memory address tracing with examples
- Live code tracing

* Session 8: Dynamic Memory Allocation

Objectives:

• Learn how to use new, delete, and dynamic arrays

Content:

- new operator
- delete operator
- Allocate memory for arrays dynamically

```
int* arr = new int[5];
delete[] arr;
```

Activities:

Practice allocating memory for arrays and structures

* Session 9: Representation of Arrays in Memory

Objectives:

Understand how linear arrays are stored in memory

Content:

- Contiguous memory allocation
- Memory address calculation: Base + index * size
- Practical example of array in RAM

Activities:

• Whiteboard memory layout diagrams

Quiz on memory addresses

* Session 10: Algorithm Analysis – Time and Space Complexity

Objectives:

- Introduce Big-O notation
- Measure efficiency of code

Content:

- What is time complexity? (focus on loop growth)
- What is space complexity?
- Common complexities: O(1), O(n), O(n²)
- Examples with array traversal, nested loops

```
for (int i = 0; i < n; i++) // O(n)
```

Activities:

- Analyze time for sorting or searching code
- Chart of complexity vs performance

** Session 1: Introduction to Data Structures

* Session 1: Introduction to Data **Structures**

Objectives

By the end of this session, students will be able to:

- 1. Define what a data structure is.
- 2. Understand why data structures are essential in programming.
- 3. Distinguish between primitive and non-primitive data types.
- 4. Use real-world analogies to understand abstract concepts.
- 5. Classify data structures properly.
- 6. Recognize the need for organizing data efficiently.

1. What is a Data Structure?

Definition:

A data structure is a way of organizing, storing, and managing data in a computer so that it can be used efficiently.

You can think of it as a **blueprint** or **template** for how data is handled in memory.

2. Why Do We Need Data Structures?

Computers deal with large amounts of data. Efficient organization is necessary for:

Reason	Explanation
Fast Access	To find/search data quickly (e.g., searching for a student by roll number)
Efficient Processing	For sorting, filtering, and analyzing data (e.g., sorting scores)

➡Memory Optimization Reduce memory waste and manage large datasets

Real-world Simulations Represent real-world systems (banking, ticket booking, etc.)

Real-World Analogy:

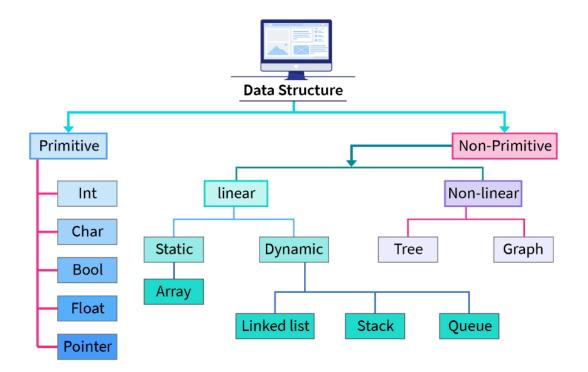
Imagine a school that has 500 students.

If the student records (name, age, ID, marks) are scattered across different files randomly:

- It's hard to find a student's info quickly.
- It's slow to update marks or search for top scorers.
- It's inefficient to add or remove students.

With data structures, we can organize all records in tables, databases, arrays, or linked lists, allowing easy and fast operations.

3. Classification of Data Structures



A. Primitive Data Structures

These are the basic building blocks provided by the language.

Туре	Description	Example
int	Integer numbers	int age = 20;
float	Decimal numbers	float pi = 3.14;
char	Single character	char grade = 'A';
bool	Boolean (true/false) values	bool pass = true;

These data types can store only one value at a time.

B. Non-Primitive Data Structures

These are more advanced and are used to store **multiple values** and complex data.

• 1. Linear Data Structures:

Elements are stored sequentially.

Туре	Description	
Array	Stores elements in continuous memory (fixed size)	
Structure	Groups different data types together	
Linked List	Nodes connected using pointers	
Stack	LIFO (Last In, First Out) data structure	
Queue	FIFO (First In, First Out) data structure	

• 2. Non-Linear Data Structures:

Type	Description
Tree	Hierarchical structure (parent-child)
Graph	Set of connected nodes (networks, maps, etc.)

4. Key Differences Between Primitive & Non-Primitive

Feature	Primitive	Non-Primitive
Nature	Basic data types	Composite data types
Size	Stores one value	Can store multiple values
Flexibility	Limited	More flexible and powerful
Examples	int, float, char, bool	array, structure, linked list

5. Practical Understanding: Real Code Examples

Primitive Example:

```
int age = 21;
float marks = 85.5;
char grade = 'A';
```

Stores a single value each.

Non-Primitive Example:

```
int marks[5] = {85, 90, 78, 92, 88}; // Array
```

Stores multiple values in a single variable.

```
struct Student {
    int rollNo;
    char name[50];
    float marks;
};
```

Groups different data types into one unit.

6. Key Takeaways

- Data structures are **critical** in solving complex problems efficiently.
- Think of them as **tools** or **containers** to manage data.
- Primitive types are basic; non-primitive are powerful and necessary for real-world applications.

7. Activities

Activity 1: Classification Chart

Ask students to draw the following chart:

```
Data Structures
  Primitive Non-Primitive
(int, float...) /
          Linear
                        Non-Linear
    (Array, Stack...) (Tree, Graph...)
```

Encourage them to add examples and illustrations.

Activity 2: Quick Quiz

Ask students to classify the following:

- 1. int, char
- 2. struct Student
- 3. int arr[10]

- 4. float
- 5. Linked List

Answers:

- 1. Primitive
- 2. Non-Primitive (Structure)
- 3. Non-Primitive (Array)
- 4. Primitive
- 5. Non-Primitive (Linear)

9. 8. Common Beginner Misconceptions (and how to handle them)

Misconception	Clarification
"Arrays and structures are not data types"	They are user-defined or derived data types
"All data types store only one value"	Arrays, structures store multiple values
"Data structure is only for advanced topics"	No – even an array is a data structure

📚 9. Suggested Homework

- 1. Define and explain 5 primitive data types with examples.
- 2. List 5 real-life examples where data structures are used.
- 3. Draw a classification diagram of data structures with 3 examples under each category.

Tab 4

Quiz: Introduction to Data Structures

Instructions:

• Time: 15-20 minutes

Total Marks: 20

• Attempt all questions.

Section A: Multiple Choice Questions (1 mark each)

(Choose the correct option)

- 1. Which of the following is a non-primitive data structure?
 - a) int
 - b) float
 - c) array
 - d) bool
- 2. A data structure is used to:
 - a) Perform calculations
 - b) Organize and store data efficiently
 - c) Print output
 - d) Execute loops faster
- 3. What is the correct way to declare an array of 5 integers in C++?

```
a) int arr(5);
b) int[5] arr;
c) array arr[5];
d) int arr[5];
```

- 4. Which data structure is **linear** in nature?
 - a) Tree
 - b) Graph
 - c) Linked List
 - d) Hash Table

- 5. What does a structure in C++ allow you to do?
 - a) Store only integers
 - b) Store a group of related values of different data types
 - c) Only store one value
 - d) Automatically sort values

Section B: True or False (1 mark each)

- 6. Structures can store values of different data types. $(_{---})$
- 7. Arrays can dynamically grow in size. (___)
- 8. int, char, and float are all primitive data types. (___)
- 9. A graph is an example of a non-linear data structure. (___)
- 10. Data structures are not useful for real-world applications. (___)

Section C: Short Answer Questions (2 marks each)

- 11. What is the difference between primitive and non-primitive data types?
- 12. Give two real-life examples where data structures are used.
- 13. What is the need for using data structures in a program?
- 14. Draw a small classification tree of data structures.
- 15. Write a C++ structure to represent a student with fields: name, rollNo, marks.

Section D: Match the Following (1 mark each)

A B

16. int a) Group of different types

17. Tree b) Non-linear structure

18. Structure c) Primitive data type

19. Linked List d) Linear data structure

20. char e) Primitive data type

Answer Key

Section A:

- 1. c
- 2. b
- 3. d
- 4. c
- 5. b

Section B:

- 6. True
- 7. False
- 8. True
- 9. True
- 10. False

Section C (Suggested Answers):

- 11. **Primitive types** are basic (like int, char), and store one value. **Non-primitive types** (like arrays, structures) store multiple or complex values.
- 12. Examples:
- Hospital patient records (structure/array)
- Ticket booking system (queue)
- 13. We use data structures to **organize data**, make searching faster, reduce memory waste, and solve complex problems efficiently.

14.

```
Data Structures
/
Primitive Non-Primitive
(int, float) / \
Linear Non-Linear
(Array, Stack) (Tree, Graph)

15.

struct Student {
   char name[50];
   int rollNo;
   float marks;
};
```

Section D:

16 - c

17 - b

18 – a

19 - d

20 - e

** Session 2: Data Structure Operations

* Session 2: Data Structure Operations

Objectives

By the end of this session, students will be able to:

- Understand the basic operations performed on data structures.
- Perform operations such as Create, Insert, Delete, Update, Traverse, and Search.
- Apply these operations on **arrays** (as the foundational data structure).
- Visualize how data changes inside memory during operations.

1. What Are Data Structure Operations?

Data structures allow you to organize data, but operations are how you work with that data.

" Operations are actions you can perform on a data structure to modify or access data."

2

2. Basic Data Structure Operations

Let's break down each operation one by one with examples.

A. Create (Initialization / Allocating Space)

This means **creating** or **allocating memory** for a data structure.

Example (Array):

```
int arr[5]; // Creates space for 5 integers
```

- The memory is reserved for 5 elements.
- The array now exists in memory.

B. Insert (Adding Data)

Used to add data into the data structure.

Parample (Array):

```
arr[0] = 10;
arr[1] = 20;
```

- You are **inserting** the values at specific positions.
- Arrays use index-based positions.

Diagram:

```
Index: 0 1 2 3 4 Value: 10 20 __ __ __
```

C. Delete (Removing Data)

Used to **remove** an element from the data structure.

In arrays, since size is fixed, we usually **shift** elements after deletion.

Example:

```
// Remove element at index 1
arr[1] = arr[2];
arr[2] = arr[3];
arr[3] = arr[4];
```

Then reduce logical size by 1.

iii Diagram (After deletion at index 1):

```
Index: 0 1 2 3 4 Value: 10 30 40 __ __
```

D. Update (Change Existing Data)

Modifying the value stored in the data structure.

Example:

```
arr[1] = 25; // Update index 1
```

Ⅲ Diagram:

Before:

```
Index: 0 1 2
Value: 10 20 30
```

After:

Index: 0 1 2 Value: 10 25 30

• E. Traverse (Visit All Elements)

Used to go through all elements of the data structure.

Example:

```
for (int i = 0; i < 5; i++) {
    cout << arr[i] << " ";
}</pre>
```

This loop visits each element and prints them.

Output:

```
10 20 30 40 50
```

F. Search (Find Specific Data)

Find the location of a value in the data structure.

§ Example (Linear Search):

```
int key = 30;
for (int i = 0; i < 5; i++) {
    if (arr[i] == key) {
        cout << "Found at index " << i;
    }
}</pre>
```

Summary Table of Operations

Operation	Meaning	Example Syntax
Create	Allocate space for data	<pre>int arr[5];</pre>
Insert	Put value at a specific position	arr[0] = 10;
Delete	Remove value (shift if needed)	arr[i] = arr[i+1];
Update	Change existing value	arr[1] = 25;
Traverse	Visit each element	for (int i)
Search	Find a specific value	if (arr[i] == key)

Live Demonstration Activity (Highly Recommended)

Use this code in your IDE (Code::Blocks / onlinegdb):

Demo: Insert, Delete, Traverse

```
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;

    cout << "Original Array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";

    // Insert 25 at index 2
    for (int i = n; i > 2; i--) {
        arr[i] = arr[i - 1];
    }
    arr[2] = 25;
    n++;
```

```
cout << "\nAfter Insertion at index 2: ";
for (int i = 0; i < n; i++) cout << arr[i] << " ";

// Delete element at index 3
for (int i = 3; i < n - 1; i++) {
    arr[i] = arr[i + 1];
}
n--;

cout << "\nAfter Deletion at index 3: ";
for (int i = 0; i < n; i++) cout << arr[i] << " ";

return 0;
}</pre>
```

** Session 2 Quiz: Data Structure Operations

* Session 2 Quiz: Data Structure Operations

✓ Part A: Multiple Choice Questions (MCQs)

Choose the correct option.

- 1. Which of the following is **not** a basic operation on a data structure?
 - a) Insert
 - b) Delete
 - c) Update
 - d) Compile
- 2. In C++, the operation to **visit each element** in a data structure is called:
 - a) Traversal
 - b) Debugging
 - c) Searching
 - d) Jumping
- 3. Which operation allows us to modify the value of an existing data element?
 - a) Create
 - b) Update
 - c) Delete
 - d) Allocate
- 4. The **Search** operation is used to:
 - a) Remove data
 - b) Input new data
 - c) Find a specific element
 - d) Change element's value
- 5. Which of the following happens first when working with data structures?
 - a) Traversal
 - b) Insertion
 - c) Searching
 - d) Creation



Write **T** for true and **F** for false.

6.	The Create operation is only used once in the lifetime of a data structure.
7.	You can perform Insert and Delete without traversing a data structure.
8.	Traversal is important when you want to display or process all data.
9.	You cannot perform an update operation on an array in C++.
10	Deletion always means removing the last element from the data structure.

Part C: Short Answer Questions

- 11. What is meant by the 'Create' operation in data structures?
- 12. Why is the 'Traversal' operation necessary in data structures?
- 13. Give an example in C++ that shows insertion into an integer array.
- 14. What is the difference between Update and Insert operations?
- 15. List all the 6 common operations on data structures.

Bonus (Application-based):

- 16. You're working with a list of student names. Describe what operation would you perform for each of the following tasks:
- a) Add a new student's name
- b) Correct a spelling mistake in an existing name
- c) Show all names on the screen
- d) Check if a student is in the list
- e) Remove a student who left school

Answer Key:

Part A:

- 1. d) Compile
- 2. a) Traversal
- 3. b) Update
- 4. c) Find a specific element
- 5. d) Creation

Part B:

- 6. T
- 7. F
- 8. T
- 9. F
- 10. F

Part C (Suggested Answers):

- 11. Allocating memory or defining space to store data.
- 12. To access or process each element in the data structure.
- 13. arr[2] = 99; (or using a loop and user input).
- 14. Update changes existing value; Insert adds new data.
- 15. Create, Insert, Delete, Update, Traverse, Search

Bonus:

- a) Insert
- b) Update
- c) Traverse
- d) Search
- e) Delete

* Session 3: Review of Arrays

* Session 3: Review of Arrays

© Objectives:

By the end of this session, students will be able to:

- Understand what arrays are and how they are declared and used in C++.
- Learn how data is stored in memory using arrays.
- Use index to access and modify array elements.
- Write loop-based code to traverse arrays.
- Visualize array storage using memory diagrams.

📚 1. What is an Array?

An array is a collection of fixed-size elements of the same data type, stored in contiguous memory locations.

Example:

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

- This creates an array of 5 integers.
- Elements are: arr[0] = 1, arr[1] = 2, ..., arr[4] = 5.

2. Why Use Arrays?

Without arrays:

```
int n1 = 10, n2 = 20, n3 = 30;
With arrays:
int arr[3] = {10, 20, 30};
```

- Easier to manage large amounts of similar data
- Can use loops for processing
- Efficient memory access

X 3. Declaration and Initialization

📌 Syntax:

```
int arr[5];  // Declaration (uninitialized)
int arr[5] = {1,2,3,4,5}; // Declaration + Initialization
```

If fewer elements are provided:

```
int arr[5] = \{1, 2\}; // Remaining will be 0 \rightarrow [1, 2, 0, 0, 0]
```

4. Accessing Elements Using Index

Each element in the array is accessed using an index that starts at 0.

Example:

```
cout << arr[2]; // Prints the 3rd element (3)
arr[4] = 99; // Changes last element to 99</pre>
```

Important:

- arr[0] is the first element
- arr[4] is the last element (if size = 5)
- Accessing arr[5] (outside size) = X Undefined behavior

★ 5. Memory Layout of Arrays

When we declare:

int $arr[5] = \{10, 20, 30, 40, 50\};$

Memory is allocated like:

Index	Element	Address (Assume start = 1000)
arr[0]	10	1000
arr[1]	20	1004
arr[2]	30	1008
arr[3]	40	1012
arr[4]	50	1016

© Explanation:

- Each int takes 4 bytes.
- Stored contiguously in memory.
- Address of arr[i] = Base address + (i * size of data type)

6. Traversal Using Loops

To access all elements, use a for loop:

• Example:

#include <iostream>

```
using namespace std;
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    for (int i = 0; i < 5; i++) {
        cout << "arr[" << i << "] = " << arr[i] << endl;
    }
}</pre>
```

Output:

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

Summary Table

Feature	Description
Fixed Size	Size is declared once and can't change
Same Data Type	All elements must be of same type
Indexing	Starts at 0
Contiguous Storage	Elements are stored next to each other

Activities for Class

Activity 1: Fill the Array

Ask students to:

- Declare an array of 5 integers.
- Take input using loop.
- Print the array using a loop.

Code:

```
int arr[5];
for (int i = 0; i < 5; i++) {
    cout << "Enter number " << i + 1 << ": ";
    cin >> arr[i];
}
for (int i = 0; i < 5; i++) {
    cout << "You entered: " << arr[i] << endl;
}</pre>
```

Activity 2: Memory Mapping on Whiteboard

Draw 5 blocks on the board and write:

- Indexes: 0-4
- Values: 10, 20, 30, 40, 50
- Assign assumed addresses (e.g., 1000, 1004, 1008...)

Make students guess how the addresses are calculated.

Activity 3: Error Correction

Give this code to students and ask them to correct it:

```
int arr[3] = {10, 20, 30, 40}; // Too many elements
cout << arr[3]; // Out of bounds</pre>
```

Expected corrections:

- Either increase size or reduce elements.
- Avoid accessing arr[3] in size 3 array.

A Homework Suggestions

- 1. Declare an array of 10 integers and print all even numbers.
- 2. Write a program to find the **sum** of elements in an array.
- 3. Write a program to reverse an array using a loop.

Optional Challenge (For Curious Students)

Write a program to input 5 student marks and find:

- Highest mark
- Average mark
- Marks above 80

** Session 3 Quiz: Review of Arrays in C++

** Session 3 Quiz: Review of Arrays in C++

✓ Part A: Multiple Choice Questions (MCQs)

Choose the correct option.

```
1. Which of the following is the correct way to declare an integer array of size 5?
    a) int arr[5];
    b) arr[5] int;
    c) int[5] arr;
    d) array int[5];
2. What is the index of the first element in a C++ array?
    a) 1
    b) 0
    c) -1
    d) Undefined
3. Which loop is commonly used to traverse an array?
    a) if
    b) switch
    c) while
    d) for
4. How many elements are there in the array declared as int marks[6];?
    a) 5
    b) 6
    c) 7
    d) Depends on user input
5. What is the correct way to access the third element of array arr?
    a) arr(3)
    b) arr[3]
    c) arr[2]
    d) arr[1]
```

Write T for true and F for false.	
6 Arrays can only store values of the same data type.	
7 In C++, arrays can grow or shrink in size after declaration.	
8 The last element of an array with size 10 is at index 9.	
9 An array's elements are stored in non-contiguous memory locations.	
10 You must always initialize an array at the time of declaration.	
✓ Part C: Fill in the Blanks	
11. The size of the array int arr[5]; is	
12. Arrays in C++ are stored in memory.	
13. To access an array element, we use its	
14. Arrays store values in memory blocks.	
15. The loop commonly used to go through all elements of an array is called	
✓ Part D: Short Answer Questions	
16. Define an array in C++ and initialize it with 5 integers.	
17. Write a loop to print all elements of an array arr[5].	
18. Why do array indices start from 0 in C++?	
19. How is memory laid out when you declare int $a[3] = \{10, 20, 30\}$;?	
20. What is the difference between int $arr[5]$; and int $arr[] = \{1,2,3,4,5\}$;?	



Answer Key (for teacher reference):

Part A:

- a) int arr[5]; 2. b) 0
- 3. d) for
- 4. b) 6
- 5. c) arr[2]

Part B:

- 6. T
- 7. F
- 8. T
- 9. F
- 10. F

Part C:

- 11.5
- 12. contiguous
- 13. index
- 14. continuous / contiguous
- 15. for loop

Part D (Sample Answers):

```
16. int arr[5] = \{1, 2, 3, 4, 5\};
17.
for(int i = 0; i < 5; i++) {
    cout << arr[i] << endl;</pre>
}
```

- 18. Historical reasons from C; makes offset calculations easier (index 0 = base + 0 offset).
- 19. Memory blocks store 10, 20, and 30 in sequence (e.g., 1000, 1004, 1008 assuming 4 bytes).

es size only; s			

* Session 4: Structures in C++

* Session 4: Structures in C++

Objectives

By the end of this session, students will:

- Understand what structures are and why they are used.
- Learn to define and use structures in C++.
- Know how to declare structure variables and access their members.
- Compare structures with arrays.
- Practice by creating structures for real-world examples.

📚 1. What is a Structure?

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

Why use Structures?

Arrays can store many values, but **only of the same type**.

Structures allow storing **different types** of information **together**.

Real-life Analogy

Think of a Student Record:

- Roll Number → Integer
- Name → Character array (string)
- Marks → Float

Instead of using:

int rollNo;

```
char name[50];
float marks;
We group them into one:
struct Student {
    int rollNo;
    char name[50];
    float marks;
};
```

% 2. Syntax of Structure

```
struct StructureName {
    dataType member1;
    dataType member2;
};
```

Example:

```
struct Student {
    int id;
    char name[50];
    float marks;
};
```

This defines a new **type** called Student.

3. Declaring Structure Variables

After the structure:

```
// Declare a structure variable s1
Student s1;
Student s2, s3; // Multiple variables
```

You can also define variables inside the struct:

```
struct Student {
    int id;
    char name[50];
    float marks;
} s1, s2;
```

4. Accessing Structure Members

```
Use dot operator (.):
s1.id = 101;
strcpy(s1.name, "Ali");
s1.marks = 85.5;
To display:
cout << s1.id << endl;</pre>
cout << s1.name << endl;</pre>
cout << s1.marks << endl;</pre>
```

5. Full Example

```
#include <iostream>
#include <cstring>
using namespace std;
struct Student {
    int id;
    char name[50];
    float marks;
};
int main() {
    Student s1;
    s1.id = 101;
```

```
strcpy(s1.name, "Ali");
    s1.marks = 92.5;
    cout << "ID: " << s1.id << endl;</pre>
    cout << "Name: " << s1.name << endl;</pre>
    cout << "Marks: " << s1.marks << endl;</pre>
    return 0;
}
```



💡 6. Structure vs Array

Feature	Array	Structure		
Same data type	Yes	No (can have mixed types)		
Use case	List of similar values	Record with different types		
Syntax	int arr[5];	struct Book { };		
Memory layout	Contiguous elements	Each member stored individually		

7. Practice Structures

Mathematical Book Structure:

```
struct Book {
    char title[100];
    char author[50];
    int pages;
};
```

Car Structure:

```
struct Car {
    char model[50];
    int year;
   float price;
};
```

M Employee Structure:

```
struct Employee {
    int id;
    char name[50];
    float salary;
};
```

Let students fill in values using code:

```
Employee e1;
e1.id = 201;
strcpy(e1.name, "Sara");
e1.salary = 65000;
```

Memory Layout Visualization

If you define:

Total size: 58 bytes (may be more with padding)

Classroom Activities

Activity 1: Create Structure for Laptop

Ask students to define:

- Brand (string)
- Model (string)
- Price (float)

Activity 2: Difference Exercise

Give a problem and ask:

• Would you use array or structure?

Activity 3: Print Student Records

Ask students to:

- Create an array of 3 students
- Input and display their details

Homework

- 1. Create a structure for Movie with:
 - Title, Genre, Duration, Rating
 Write a program to input and display 3 movies.
- 2. Define a structure for MobilePhone with:
 - o Brand, Model, RAM, Price
- 3. Compare array vs structure with 2 examples.

Summary

- Structures are powerful for real-world data grouping.
- Use them when data types vary.
- Access members using the **dot operator**.
- Structures can be combined with **arrays**, **functions**, **and pointers** (coming next sessions!).

Tab 12

* Session 4 Quiz: Structures in C++

Part A: Multiple Choice Questions (MCQs)

Choose the correct option.

- 1. What is a structure in C++?
 - a) A predefined data type
 - b) A user-defined data type
 - c) A loop structure
 - d) A pointer
- 2. Which of the following correctly defines a structure?

```
a) define struct { int x; };
b) struct { int x; } myStruct;
c) structure myStruct { int x; };
d) struct myStruct ( int x );
```

3. What does the following code do?

```
struct Student {
    int id;
    float marks;
};
```

- a) Declares a function
- b) Defines a loop
- c) Defines a structure
- d) Compiles nothing
- 4. How do you access a member of a structure variable?

```
a) struct.member
```

- b) variable->member
- c) object.member
- d) variable.member
- 5. Which of the following is NOT true about structures?
 - a) They can store multiple variables of different types
 - b) Structures can include arrays as members
 - c) Structures cannot contain other structures
 - d) Structures are useful for grouping related data

Part B: True / False

Write **T** for true and **F** for false.

6.	Structures in C++ are user-defined data types.
7.	You must initialize all members of a structure at the time of declaration.
8.	You can have an array of structures.
9.	Structures can hold different types of data in one unit.
10	Structures and arrays are the same thing in C++.

Part C: Fill in the Blanks

11. The keyword used to define a structure is
12. To access structure members, we use the operator.
13. A structure groups types of variables together.
14. A structure can contain types like int, float, and arrays.
15. Structure variables are declared using the of the structure.

Part D: Short Answer Questions

- 16. What is the main difference between a structure and an array?
- 17. Write a structure definition for a Book containing:
 - title (char array)
 - author (char array)
 - price (float)

18. How do you declare a variable from the structure Student defined as:

```
struct Student {
    int rollNo;
    float marks;
};
```

- 19. How can you store 5 employees using a structure named Employee?
- 20. Why are structures important in real-world applications?

Bonus: Code Understanding

What will be the output of the following code?

```
#include <iostream>
using namespace std;

struct Car {
   int id;
   float price;
};

int main() {
   Car c1 = {101, 550000.5};
   cout << c1.id << " " << c1.price;
   return 0;
}

a) 101 550000
b) 101 550000.5
c) Error
d) 550000.5 101</pre>
```

Answer Key :

Part A:

- .. .
- 2. b
- 3. c
- 4. d
- 5. c

Part B:

- 6. T
- 7. F
- 8. T
- 9. T
- 10. F

Part C:

```
11. struct
```

- 12. . (dot)
- 13. different
- 14. multiple
- 15. name

Part D:

```
16. Arrays store only same-type data; structures store mixed types 17.
```

срр

CopyEdit

```
struct Book {
    char title[50];
    char author[50];
    float price;
};
```

- 18. Student s1;
- 19. Employee emp[5];

20. Useful for organizing related data like student info, employee records, etc.

Bonus Answer:

b) 101 550000.5

** Session 5: Self-Referential Structures in C++

* Session 5: Self-Referential Structures in C++

Objectives

By the end of this session, students will:

- Understand the concept of self-referential structures.
- Learn how structures can reference themselves.
- See how this concept forms the basis for data structures like linked lists.
- Practice visualizing and drawing "linked" nodes.

📚 1. What is a Self-Referential Structure?

A self-referential structure is a structure that includes a pointer to a variable of its own type.

This is the **foundation** for complex dynamic data structures like:

- Linked Lists
- Stacks (using linked lists)
- Trees
- Graphs

Definition

```
struct Node {
                 // data of the node
   int data;
                // pointer to the next node (same type)
   Node* next;
};
```

Here, Node is a structure that contains:

- data: An integer (or any data type).
- next: A pointer of type Node* it "points" to another structure of the **same type**.

📌 Real-Life Analogy

Imagine a train:

- Each **coach** has some **data** (passengers, seats).
- Each coach also has a connector to the next coach.

This is how **linked lists** work — each unit points to the next.

Why is it useful?

Arrays have fixed size.

But using self-referential structures, you can dynamically:

- Add more data at runtime.
- Build chains of data.
- Easily insert/delete in the middle.

X 2. Syntax Explanation



Let's create two linked nodes:

```
Node* first = new Node;
first->data = 10;
Node* second = new Node:
second->data = 20;
first->next = second; // link first to second
second->next = nullptr; // last node points to nothing
This creates a chain:
[10 | next] → [20 | null]
```

4. Diagram to Explain Chaining

You can draw the following on the board or in slides:

```
| 10 | * | --> | 20
+----+
            +----+
 first
             second
```

Each box has two parts:

- Left: data
- Right: next pointer (address of next node)

🗩 5. Visualize with Pointer Addresses (Advanced)

```
Node* A = new Node; // Suppose at address 0x100
A->data = 50;
```

```
Node* B = new Node; // Suppose at address 0x200
B->data = 70;
A \rightarrow next = B;
B->next = nullptr;
Memory view:
0x100: [50 \mid 0x200]
0x200: [70 | NULL ]
```

6. Full Example Program

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
int main() {
    Node* first = new Node;
    Node* second = new Node;
    first->data = 100;
    first->next = second;
    second->data = 200;
    second->next = nullptr;
    // Traversing the nodes
    Node* temp = first;
    while (temp != nullptr) {
        cout << temp->data << " ";</pre>
        temp = temp->next;
    }
```

```
return 0;
}
```

Output:

100 200

7. Summary

Term	Meaning
Self-referential	A structure that holds a pointer to its own type
Node	A unit with data and a pointer to the next node
Chain	Multiple nodes linked together
Head	Starting point of the chain
NULL	Marks the end of the chain

Activities

1. Board Exercise:

Draw 3 boxes and manually link them with arrows. Let students label data and next.

2. Code Practice:

Ask students to:

- o Create 2-3 nodes
- o Assign data
- o Link them
- o Print all values

3. Group Task:

Break students into pairs and ask them to build a linked structure manually with diagrams.

Homework

- 1. Define a structure EmployeeNode with:
 - o int id
 - o float salary
 - EmployeeNode* next
- 2. Write a program to:
 - o Create 3 employees
 - Link them
 - o Traverse and display their IDs and salaries
- 3. Compare arrays and self-referential structures in terms of:
 - Memory use
 - Flexibility
 - o Insertion/deletion

Tab 13

* Session 5 Quiz: Self-Referential Structures in C++

✓ Part A: Multiple Choice Questions (MCQs)

Choose the correct answer.

- 1. A self-referential structure is a structure that:
 - a) Refers to a function
 - b) Refers to another class
 - c) Contains a pointer to a structure of the same type
 - d) Contains a static variable only
- 2. What is the correct syntax for a self-referential structure?

```
a) struct Node { int data; struct Node next; };
b) struct Node { int data; Node *next; };
c) struct Node { int data; pointer next; };
d) struct Node { data int; next Node*; };
```

- 3. What is the purpose of the next pointer in the Node structure?
 - a) To delete the node
 - b) To hold data
 - c) To point to the next node
 - d) To point to an integer
- 4. What does NULL represent in a self-referential structure?
 - a) A function
 - b) The beginning of a structure
 - c) An uninitialized variable
 - d) The end of a linked structure chain
- 5. What will happen if you try to access node->next->data when node->next is NULL?
 - a) It prints 0
 - b) It throws a compile-time error
 - c) It causes a runtime error (segmentation fault)
 - d) It runs successfully

6 Self-referential structures are useful for building dynamic data structures.
7 Every node in a self-referential structure must have the same memory address.
8 The keyword struct is needed only once during definition.
9 A structure cannot have a pointer to itself.
10 Linked lists are built using self-referential structures.
✓ Part C: Fill in the Blanks
11. A structure that contains a pointer to a structure of the same type is called a
12. In the structure Node, the pointer to the next node is usually named
13. A self-referential structure is the foundation of data structures.
14. The last node in a linked list usually points to
15. In C++, the pointer to a structure is defined using the symbol
Part D: Short Answer Questions
16. Write the structure definition of a self-referential node containing int data and a pointer to the next node.
17. What is the purpose of the next pointer in a node structure?
18. Draw a simple diagram showing 3 nodes linked using the next pointer.

19. How is a self-referential structure different from a regular structure?

20. Write code to create two nodes and link them using next.

Bonus Code Output

What will be the output of this code?

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
int main() {
    Node* a = new Node;
    Node* b = new Node;
    a->data = 5;
    b->data = 10:
    a - next = b;
    b->next = nullptr;
    cout << a->data << " " << a->next->data;
}
a) 5 10
b) 10 5
c) Error
d) 50
```



Answer Key (For Teacher Use)

Part A:

- 1. c
- 2. b
- 3. c

```
4. d
```

```
5. c
```

Part B:

- 6. T
- 7. F
- 8. T
- 9. F
- 10. T

Part C:

- 11. self-referential structure
- 12. next
- 13. linked
- 14. NULL

16.

};

15. * (asterisk)

Part D: Sample Answers

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

17. To link the current node to the next node in a dynamic chain.

```
18. [10 \mid *] \rightarrow [20 \mid *] \rightarrow [30 \mid NULL]
```

19. A regular structure holds basic data types only, but a self-referential structure holds a pointer to its own type, enabling dynamic linking.

20.

```
Node* a = new Node;
Node* b = new Node;
a->data = 10;
b->data = 20;
a->next = b;
```

```
b->next = nullptr;
```

Bonus: a) 5 10

* Session 6: Unions in C++

* Session 6: Unions in C++

© Objectives:

- Understand what a union is in C++.
- Learn how memory is shared in unions.
- Differentiate between structures and unions.
- Practice through examples and memory demonstrations.

1. What is a Union?

A **union** is a user-defined data type, similar to a structure, but with a key difference in how memory is managed.

- A structure allocates separate memory for each member.
- A union allocates one shared memory block for all members.

Only one member can contain a value at any given time.

Use Case:

Unions are used when:

- You need to store **only one** value out of multiple possible options.
- Memory efficiency is important (like in embedded systems, low RAM environments).

🔠 2. Syntax of Union

```
union Data {
   int i;
   float f;
   char str[20];
};
```

This union contains:

- An integer i
- A float f
- A character array str



All members share the same memory location.

3. Declaring and Using a Union Variable

```
#include <iostream>
using namespace std;
union Data {
    int i;
    float f;
    char str[20];
};
int main() {
    Data d;
    d.i = 10;
    cout << "d.i = " << d.i << endl;</pre>
    d.f = 220.5;
    cout << "d.f = " << d.f << endl;
    d.i = 99;
    cout << "d.i = " << d.i << endl;
    return 0;
}
```

Output Understanding:

- When you assign a new value to any member, previous data is overwritten.
- Memory is shared among all members.

4. Memory Usage and sizeof()

```
#include <iostream>
using namespace std;

union Data {
   int i;
   float f;
   char str[20];
};

int main() {
   cout << "Size of union: " << sizeof(Data) << endl;
   return 0;
}</pre>
```

Explanation:

- Memory allocated = size of largest member.
- Here, char str[20] is 20 bytes \rightarrow So, sizeof(Data) is 20.

Compare with a structure:

```
struct Data {
   int i;
   float f;
   char str[20];
};
```

- Here, total size = size of int + float + char[20] (+ padding)
- Typically around **28–32 bytes** depending on system architecture.

☼ 5. Structure vs Union

Feature	Structure	Union
Memory	Separate memory for each	Shared memory for all
Access	All members at once	One member at a time
Size	Sum of all member sizes	Size of largest member
Use case	Multiple values together	Only one value at a time
Example use	Student data, Employee	Network packets, Storage

6. Activities for Students

Activity 1: Memory Comparison

- 1. Create a struct Data and union Data.
- 2. Use sizeof() to compare memory usage.
- 3. Print outputs after assigning values.

Activity 2: Value Overwrite

- 1. Assign value to union member i
- 2. Assign new value to f
- 3. Print i again \rightarrow observe how it's **overwritten**

Activity 3: Visual Diagram

Draw memory blocks:

- For **structure**: show separate blocks per member.
- For **union**: show one shared block.

Key Takeaways

- Unions are used to **optimize memory usage**.
- Only **one member** can hold a meaningful value at a time.
- Structures store all members independently, but unions overlay them in memory.
- Accessing multiple members in a union simultaneously leads to **unpredictable** results.

Tab 15

* Session 6 Quiz: Unions in C++

Part A: Multiple Choice Questions (MCQs)

Choose the correct option.

- 1. What is the key feature of a union in C++?
 - a) Each member gets its own memory space
 - b) Members can be functions only
 - c) Only one member uses memory at a time
 - d) All members store strings only
- 2. Which of the following correctly defines a union in C++?

```
a) union Data { int a; float b; };
b) structure Data { int a; float b; };
c) define union { int a; float b; };
d) data union { int a; float b; };
```

- 3. In a union, if you assign values to multiple members one after another, what happens?
 - a) All values are stored together
 - b) Only the last assigned value remains valid
 - c) The union will crash
 - d) None of the values are stored
- 4. What does sizeof(union Data) return when Data contains int a; char str[10]; float b;?
 - a) 4
 - b) 10
 - c) Size of the largest member
 - d) Sum of all members
- 5. Which is a valid use case for a union?
 - a) Store a student's details (name, marks, roll no)
 - b) Store different types of a value at different times
 - c) Implement a 2D array
 - d) Save multiple records at once

6 A union can store multiple values in all members simultaneously.
7 The memory size of a union is equal to its largest member.
8 Unions are more memory-efficient than structures when only one value is used at a time.
9 Unions are defined using the keyword union.
10 You can access multiple union members at the same time reliably.
✓ Part C: Fill in the Blanks
11. The keyword used to define a union in C++ is
12. In a union, all members share the same location.
13. Unions are useful for saving when only one field is used at a time.
14. Only the assigned member holds valid data in a union.
15. Using sizeof() on a union returns the size of its member.
✓ Part D: Short Answer Questions
16. Define a union named Data that contains int i, float f, and char str[10].
17. Why does a union occupy less memory compared to a structure?
18. What happens when you assign a value to Data.i and then to Data.f?
19. List two real-world scenarios where using a union is beneficial.
20. Write a small code snippet that shows how to declare and access a union variable.

Bonus: Code Output

What will be the output of the following code?

```
#include <iostream>
using namespace std;
union Data {
    int i;
    float f;
};
int main() {
    Data d;
    d.i = 10;
    d.f = 5.5;
    cout << "i = " << d.i << endl;</pre>
    cout << "f = " << d.f << endl;
    return 0;
}
a) i = 10, f = 5.5
b) i = 0, f = 5.5
c) i = garbage value, f = 5.5
d) Compilation error
```

Answer Key

Part A:

- 1. c
- 2. a
- 3. b
- 4. c

```
5. b
```

Example e;

```
Part B:
6. F
7. T
8. T
9. T
10. F
Part C:
11. union
12. memory
13. space/memory
14. last
15. largest
Part D: Sample Answers
   16.
union Data {
     int i;
     float f;
     char str[10];
};
   17. Because all members share the same memory space instead of using separate
      blocks.
   18. The value of i becomes invalid (overwritten) after assigning a value to f.
   19. Example answers:
   • Union for storing sensor input (int, float, or byte data).
   • Network protocol packet interpretation.
   20.
union Example {
     int x;
     float y;
};
```

```
e.x = 5;
cout << e.x;
```

Bonus Answer:

c) i = garbage value, f = 5.5 (because assigning to f overwrites i)

* Session 7: Pointers in C++

* Session 7: Pointers in C++

Objectives:

By the end of this session, students should:

- Understand what pointers are in C++
- Know how to declare, initialize, and use pointers
- Grasp the concepts of address-of (&) and dereferencing (*)
- Use pointers with variables, arrays, and structures
- Visualize how pointers map to memory

1. What is a Pointer?

A **pointer** is a variable that stores the **memory address** of another variable.

A pointer "points to" the location in memory where a value is stored.

Example:

```
int a = 10;
int* p = &a; // pointer p holds the address of variable a
```

- a stores the value 10
- p stores the address of a
- *p accesses (dereferences) the value stored at that address → 10

2. Syntax and Terminology

➤ Declaration:

```
int* ptr;  // pointer to int
float* fptr;  // pointer to float
char* cptr;  // pointer to char
```

➤ Initialization:

```
int a = 5;
int* ptr = &a; // ptr now points to a
```

Q 3. Important Operators

Operator	Meaning	Example
&	Address-of (returns address)	&a
*	Dereference (value at address)	*ptr

Visualization Example:

```
int a = 5;
int* p = &a;
```

Memory layout:

Variable	Value
а	5
p	address of a (e.g. 0x61ff08)

Access:

- $p \rightarrow address$
- *p → value at address = 5

📚 4. Pointers and Arrays

Arrays and pointers are closely related.

```
int arr[3] = {10, 20, 30};
int* p = arr; // points to arr[0]
```

Accessing elements using pointers:

```
cout << *(p + 0); // 10
cout << *(p + 1); // 20
```

```
arr[i] is the same as *(arr + i)
```

5. Pointers with Structures

➤ Defining a structure:

```
struct Student {
    int id;
    float marks;
};

Student s1 = {101, 89.5};
Student* ptr = &s1;
```

➤ Accessing members using pointers:

- Using dot: s1.id
- Using pointer and arrow: ptr->id, ptr->marks

```
cout << ptr->id;  // prints 101
cout << (*ptr).marks; // also valid, prints 89.5</pre>
```

6. Pointer Activities for Students

Activity 1: Memory Address Tracing

```
int x = 42;
int* p = &x;

cout << "x = " << x << end1;
cout << "&x = " << &x << end1;
cout << "p = " << p << end1;
cout << "*p = " << *p << end1;</pre>
```

- Ask students to draw the memory diagram showing:
 - x with its value
 - p pointing to x

Activity 2: Pointer with Array

```
int nums[3] = {5, 10, 15};
int* ptr = nums;

for (int i = 0; i < 3; i++) {
    cout << "Address: " << (ptr + i) << " Value: " << *(ptr + i) << endl;
}</pre>
```

Students will observe:

- How pointer moves through array
- How values are accessed via pointer arithmetic

Activity 3: Pointer with Structure

```
struct Car {
    string model;
    int year;
};
```

```
Car c1 = {"Honda", 2018};
Car* cp = &c1;
cout << cp->model << ", " << cp->year << endl;</pre>
```

Ask students to trace:

- Memory layout of c1
- Pointer cp accessing structure members

🔁 7. Live Code Tracing Ideas

Use whiteboard or code visualization tools (like cpp.sh or onlinegdb):

- Create variables, assign pointers
- Show values and addresses side by side
- Update values using *p = new_value; and show changes reflected in original variable

🔚 8. Summary of Concepts

Concept	Explanation
Pointer	A variable that stores an address
& operator	Returns address of a variable
* operator	Accesses value at a memory address
Pointers with arrays	Support pointer arithmetic
Pointers with structs	Use -> to access members

Tab 17

* Session 7 Quiz: Pointers in C++

Part A: Multiple Choice Questions (MCQs)

Choose the correct option.

- 1. What does a pointer store?
 - a) A function name
 - b) The value of a variable
 - c) The address of a variable
 - d) A constant
- 2. What is the meaning of the & operator in C++?
 - a) Bitwise AND
 - b) Value of variable
 - c) Address of variable
 - d) Logical AND
- 3. What does the * operator do when used with pointers?
 - a) Multiplies two values
 - b) Gives the memory size
 - c) Returns the address
 - d) Dereferences the pointer (gets the value)
- 4. What will this code print?

```
int a = 5;
int* p = &a;
cout << *p;</pre>
```

- a) Address of a
- b) 0
- c) 5
- d) Error
- 5. Which of the following is the correct way to declare a pointer to an integer?
 - a) int p;
 - b) int *p;
 - c) pointer int p;
 - d) int &p;

Part B: True or False

- 6. ___ A pointer must be initialized before it can be used.
- 7. ___ The * symbol is used to declare and dereference a pointer.
- 8. ___ A pointer can store the address of any data type.
- 9. ___ Pointer variables do not need memory themselves.
- 10. ____ You can access array elements using pointers.

Part C: Fill in the Blanks

- 11. The symbol & is used to get the _____ of a variable.
- 12. The symbol * is used to _____ a pointer.
- 13. A pointer that is not initialized is called a _____ pointer.
- 14. Pointers can be used with _____ to traverse their elements.
- 15. The arrow operator (->) is used to access members of a _____ using a pointer.

Part D: Output-Based Questions

16. What is the output of the following code?

```
int x = 20;
int* p = &x;
*p = *p + 5;
cout << x;

a) 5
b) 25
c) 20
d) Error</pre>
```

17. What will this code output?

```
int a = 7;
int* ptr = &a;
cout << ptr;</pre>
a) 7
b) 0
c) Address of a
d) Compile-time error
```

Part E: Short Answer Questions

- 18. Write a program to declare an integer variable, assign a value, then create a pointer and print the value using dereferencing.
- 19. What is the difference between &x and *p in the context of pointers?
- 20. Why are pointers important in C++? Give any two real-world uses.

Answer Key (For Teachers)

Part A:

- 1. c
- 2. c
- 3. d
- 4. c
- 5. b

Part B:

- 6. True
- 7. True
- 8. False
- 9. False
- 10. True

Part C:

- 11. address
- 12. dereference
- 13. dangling
- 14. arrays
- 15. structure

Part D:

```
16. b (25)
```

17. c (Address of a)

Part E (Sample):

18.

```
int a = 10;
int* p = &a;
cout << *p;</pre>
```

19. &x gives the address of variable x, while *p gives the value stored at the address p.

20.

- Used in dynamic memory allocation (e.g., new, delete)
- Useful for creating data structures like linked lists, trees, etc.

** Session 8: Dynamic Memory Allocation in C++

* Session 8: Dynamic Memory Allocation in C++

Objectives:

- Understand what **dynamic memory** is.
- Learn how to use **new** and **delete** keywords.
- Allocate memory for variables and arrays during program execution.
- Learn how to avoid memory waste and memory leaks.

Why Dynamic Memory?

Real Life Analogy:

Imagine you're organizing a party and don't know how many guests will come.

- If you prepare 100 chairs, and only 10 guests arrive waste of space. (This is like static memory)
- If you bring chairs only when guests arrive, you save space. (This is dynamic memory)

real numbers of the programming, when you don't know in advance how many items you'll need, use dynamic memory.

1. Static vs Dynamic Memory

Feature	Static Memory	Dynamic Memory	
Fixed size	Yes	No (decided at runtime)	



2. What is the new keyword?

• new is used to **create (allocate)** memory **dynamically** on the **heap** (a memory area managed by the system).

Syntax:

```
int* ptr = new int;
*ptr = 25;
```

Explanation:

- ptr is a pointer to an int.
- new int asks the system to give memory for one integer.
- *ptr = 25; stores value 25 in that memory.

X 3. What is the delete keyword?

delete is used to free memory created by new.
 If you don't use it, memory stays reserved → causes memory leak.

```
delete ptr;
```

4. Dynamic Memory for Arrays

Suppose the user wants to enter 5 numbers — we don't need to fix size.

Code:

```
int* arr = new int[5]; // creates an array of 5 integers in heap
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10; // Store 0, 10, 20, 30, 40
}
for (int i = 0; i < 5; i++) {
    cout << arr[i] << " ";
}
delete[] arr; // Free the memory</pre>
```

Important:

- Use delete[] for arrays
- new[] → delete[]

🧱 5. Dynamic Memory for Structures

Let's say we have a structure for a student.

```
struct Student {
    int id;
    float marks;
};
```

Now we create it dynamically:

```
Student* s = new Student;
```

```
s->id = 101;
s->marks = 88.5;
cout << s->id << " " << s->marks;
delete s; // Free structure memory
```

✓ Use -> to access members with pointers.

6. Full Real-World Example

Problem:

Ask the user how many numbers they want to enter, take input, and print them.

Code:

```
#include <iostream>
using namespace std;

int main() {
   int n;
   cout << "Enter number of elements: ";
   cin >> n;

   int* arr = new int[n]; // allocate n integers
```

• What Happens:

- Memory for n integers is allocated in heap.
- new returns the starting address of that memory.
- arr is a pointer variable (stored in stack) holding that address.

```
cout << "Enter " << n << " numbers: ";
for (int i = 0; i < n; i++) {
    cin >> arr[i];
}

cout << "You entered: ";
for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
}</pre>
```

```
delete[] arr; // free memory
    return 0;
}
```

Q Output (if input is 3, then 5 6 7):

You entered: 5 6 7



1. 7. Common Mistakes to Avoid

Mistake What Happens

Not using delete Memory leak

Using normal delete for array Unexpected behavior

Using memory after delete Error / crash

Null pointer / garbage Forgetting new



📚 8. Summary Table

Keyword	Meaning
new	Allocates memory dynamically
delete	Frees memory from a pointer
new[]	Allocates an array dynamically

@ 9. Activities

Activity 1: Create a dynamic integer

```
int* p = new int;
*p = 45;
cout << *p;
delete p;</pre>
```

Activity 2: Allocate 3 marks for a student and print average

```
float* marks = new float[3];
marks[0] = 85.5;
marks[1] = 90.0;
marks[2] = 92.5;

float avg = (marks[0] + marks[1] + marks[2]) / 3;
cout << "Average: " << avg;

delete[] marks;</pre>
```

Activity 3: Create structure dynamically

```
struct Book {
    int id;
    char title[50];
};

Book* b = new Book;
b->id = 1;
strcpy(b->title, "C++ Basics");
cout << b->id << " " << b->title;
delete b;
```

Final Thoughts (for students):

- Use **new** when size is **not known at compile time**.
- Always delete what you create, to avoid memory waste.
- Use **pointers** with care and practice tracing memory.

Tab 19

***** Quiz: Dynamic Memory Allocation in C++

Part A: Multiple Choice Questions (MCQs)

Choose the correct answer.

- 1. What is the purpose of the new keyword in C++?
 - a) To create a new class
 - b) To declare a function
 - c) To allocate memory during runtime
 - d) To create static arrays
- 2. Which of the following is the correct way to dynamically allocate memory for one integer?

```
a) int p = new int;
b) int* p = new int;
c) int new = *p;
d) new int = p;
```

3. How do you free the memory allocated by new?

```
a) free(p);
b) delete p;
c) remove(p);
d) destroy p;
```

4. What is the correct way to deallocate memory allocated with new[] for an array?

```
a) delete arr;
b) free(arr);
c) delete[] arr;
d) clear arr;
```

5. What is the output of the following code?

```
int* p = new int;
*p = 10;
cout << *p;
delete p;</pre>
```

- a) Address of p
- b) 10
- c) Error
- d) Nothing

- 6. ____ The new keyword allows memory to be allocated at compile-time.
- 7. ___ Using delete[] is necessary for arrays created with new[].
- 8. ____ If you don't use delete, the memory will be automatically released.
- 9. ___ You can assign values to dynamically created arrays using arr[i].
- 10. ___ Dynamic memory is stored in the heap section of memory.

Part C: Fill in the Blanks

- 11. The keyword used to allocate memory dynamically is ______.
- 12. The keyword used to free memory created with new is _____.
- 13. When using new[], we must release memory using _____.
- 14. Dynamic memory is useful when the size is ______ at compile-time.
- 15. Using memory after delete causes _____ pointer issues.

Part D: Output-Based Questions

16. What will be the output?

```
int* arr = new int[3];
arr[0] = 5; arr[1] = 10; arr[2] = 15;
cout << arr[1];
delete[] arr;</pre>
```

- a) 0 b) 5
- c) 10
- d) 15
 - 17. What will happen if you forget to use delete[] for a dynamic array?
 - a) Compile-time error
 - b) Automatic memory release
 - c) Memory leak
 - d) Array will delete itself

Part E: Short Answer Questions

- 18. Write the code to create an array of n integers dynamically and delete it after use.
- 19. Why is delete[] important when working with dynamic arrays?
- 20. How is dynamic memory different from static memory?

Answer Key (for Teachers)

Part A (MCQ):

- 1. c
- 2. b
- 3. b
- 4. c
- 5. b

Part B (True/False):

- 6. False
- 7. True

```
8. False
9. True
10. True
Part C (Fill in the blanks):
11. new
12. delete
13. delete[]
14. unknown
15. dangling
Part D:
16. c (10)
17. c (Memory leak)
Part E (Sample Answers):
   18.
int n;
cin >> n;
int* arr = new int[n];
// ... use the array ...
delete[] arr;
```

- 19. Because it prevents memory leaks and frees all memory allocated for the array.
- 20. Dynamic memory is allocated during runtime (flexible), while static memory is fixed at compile-time (inflexible).

** Session 9: Arrays in Memory (Linear Arrays)

* Session 9: Representation of Arrays in Memory (Linear Arrays)

Objectives

By the end of this session, students will be able to:

- Understand how arrays are stored in memory.
- Learn how addresses are calculated for array elements.
- Visualize **linear (1D) arrays** in memory.
- Perform simple calculations of memory addresses.
- Gain confidence through diagrams and examples.

Why Learn This?

Before using arrays effectively in programs, it's important to understand **how they are stored in memory**. It helps you:

- Avoid errors
- Understand how data is accessed
- Improve performance of programs

1. What is a Linear Array?

A linear array is a collection of elements stored in a continuous (contiguous) block of memory.

Each element:

• Has the same data type

Is accessed using an index, starting from 0

Example:

```
int arr[4] = \{10, 20, 30, 40\};
```

This creates:

Index: 0 1 2 3 Value: 10 20 30 40

2. How Arrays Are Stored in Memory

Each element is stored right next to the previous one. This is called contiguous memory allocation.

Let's say:

- arr[0] is stored at address 1000
- Data type is int (size = 4 bytes)

Then:

Element	Index	Value	Address
arr[0]	0	10	1000
arr[1]	1	20	1004
arr[2]	2	30	1008
arr[3]	3	40	1012

★ Each next element is stored at:

Next Address = Previous Address + Size of data type

🧮 3. Formula to Calculate Address

Address of arr[i] = Base Address + (i × Size of Data Type)

```
For int arr[4] = {10, 20, 30, 40};
If base address (arr[0]) = 1000, and size of int = 4 bytes
```

Then:

- $arr[0] = 1000 + 0 \times 4 = 1000$
- $arr[1] = 1000 + 1 \times 4 = 1004$
- $arr[2] = 1000 + 2 \times 4 = 1008$
- $arr[3] = 1000 + 3 \times 4 = 1012$

🧱 4. Practical Example in C++

Output (addresses may vary):

Value: 5, Address: 0x61ff08 Value: 10, Address: 0x61ff0c Value: 15, Address: 0x61ff10

Notice: Each address increases by 4 bytes (because int is 4 bytes)

5. Why Index Starts at 0?

Because it makes address calculation simple:

 $Address = Base + index \times size$

So:

- Index $0 \rightarrow Base$
- Index 1 → Base + 1 × size
- Index 2 → Base + 2 × size

6. Activity: Draw Array in Memory

Ask students to:

- 1. Assume $arr[4] = \{2, 4, 6, 8\}$
- 2. Base address = 1000
- 3. Data type = int = 4 bytes

Index	Value	Address
0	2	1000
1	4	1004
2	6	1008
3	8	1012

representation Let students draw this as blocks on paper or whiteboard.



7. Important Points

Concept Explanation

Contiguous memory Array elements are stored next to each other

Address calculation Formula: Base + index × size

Indexing Starts at 0 for easy calculation

Data type size Depends on type (int = 4, float = 4, char = 1)

Efficiency Fast access using index (O(1) time)

Summary

- Arrays are stored in contiguous memory blocks
- Use the formula: Base + index × size
- Index starts at 0
- Access time is constant for any index

Activities

- 1. Trace addresses using a given base address and array
- 2. **C++ Program** to print values and addresses
- 3. Fill the Table with missing values or addresses
- 4. **Draw memory blocks** on whiteboard or paper

Tab 23

Quiz: Representation of Arrays in Memory

Session 9 – Focus: Linear Arrays, Memory Mapping, Address Calculation

Part A: Multiple Choice Questions (MCQs)

Choose the correct option:

- Q1. How are elements of a linear array stored in memory?
- a) Randomly in free locations
- b) In linked memory blocks
- c) Contiguously in memory
- d) Backwards from top of memory

Q2. What does the formula Base Address + (index * size) calculate?

- a) Array size
- b) Number of elements
- c) Memory address of an element
- d) Number of swaps in sorting

Q3. If arr[0] is stored at address 2000 and each int takes 4 bytes, what is the address of arr[3]?

- a) 2003
- b) 2012
- c) 2016
- d) 2008

Q4. What is the base address in an array?

- a) The size of the array
- b) The address of the last element
- c) The total number of elements
- d) The memory address of the first element

Q5. Which data structure always stores elements in continuous memory locations?

- a) Stack
- b) Linked List

d) ⁻	Ггее			
•	Part B: True or False			
	Arrays in C++ are stored in non-contiguous memory.			
	7. The size of each element affects the memory address calculation.			
Q 8.	An array of 10 integers takes 40 bytes if each integer is 4 bytes.			
Q9.	Indexing in arrays starts from 1 in C++.			
Q10	Array memory layout is important for understanding performance.			
Q1 size Q1 Q1	. In memory, arrays are stored in locations. 2. The address of an element in an array is calculated using + (index * e). 3. Accessing an element using its index is called access. 4. Arrays provide fast access because they support access time. 5. The index of the first element in an array is			
✓ Par	Answer Key (For Teachers)			
	1. c			
	2. c			
	3. b (2000 + 3×4 = 2012)			
	4. d			
	5. c			

c) Array

Part B:

- 6. False
- 7. True
- 8. True
- 9. False (It starts from 0)
- 10. True

Part C:

- 11. contiguous
- 12. base address
- 13. direct (or random)
- 14. constant
- 15. 0

Topic: Structure in Programming (C/C++)

■ Topic: Structure in Programming (C/C++)

Learning Objectives

By the end of this lesson, students will be able to:

- Understand what a structure is.
- Declare and define structures.
- Create variables of structures.
- Access and manipulate structure members.
- Use structures with functions (pass by value and reference).
- Understand the use of nested structures and arrays of structures.

What is a Structure in C++?

A **structure** (**struct**) in C++ is a **user-defined data type** that allows you to combine data items of **different types** under one name.

Think of a structure as a **blueprint** or **template** for creating objects that group together related data, much like a real-world object. For example, a Student has a name (string), roll number (int), and marks (float).

TWhy Do We Need Structures?

In real-life applications, we often need to store **heterogeneous data** (different types of data) for one entity.

X Without Structures (Primitive Approach):

```
int roll = 1;
```

```
char name[50];
float marks = 89.5;
```

Not organized, difficult to manage for multiple students.

With Structures:

```
struct Student {
    int roll;
    char name[50];
    float marks;
};
```

All related data is bundled into a single unit.

🧱 Syntax of Structure in C++

```
struct StructureName {
    // member variables (data members)
    dataType member1;
    dataType member2;
    ...
};
```

Example:

```
struct Student {
    int roll;
    char name[50];
    float marks;
};
```

This doesn't allocate memory. It only defines the **structure type**.

X Creating Structure Variables

➤ After Structure Definition:

```
Student s1, s2;
```

Or:

```
struct Student s1;
```

In C++, the struct keyword is not required when declaring variables if the structure is already defined.

Accessing Structure Members

Use the **dot operator** . to access members:

```
s1.roll = 101;
strcpy(s1.name, "Alice");
s1.marks = 87.5;
```



⚠ Use #include <cstring> for strcpy().

Full Working Example

```
#include <iostream>
#include <cstring>
using namespace std;
struct Student {
    int roll;
    char name[50];
    float marks;
};
int main() {
    Student s1;
    s1.roll = 101;
    strcpy(s1.name, "Alice");
    s1.marks = 89.5;
```

```
cout << "Student Details:" << endl;</pre>
    cout << "Roll Number: " << s1.roll << endl;</pre>
    cout << "Name: " << s1.name << endl;</pre>
    cout << "Marks: " << s1.marks << endl;</pre>
    return 0;
}
```

Structures and Functions

Structures can be passed to functions by value or by reference.

📤 Pass by Value

```
void display(Student s) {
    cout << s.name << " has " << s.marks << " marks.\n";</pre>
}
```

Changes inside the function don't affect the original structure.

📥 Pass by Reference

```
void updateMarks(Student &s) {
    s.marks += 5;
}
```

Changes are reflected in the original variable.

Pointer to Structure

Syntax:

```
Student *ptr = &s1;
ptr->roll = 105;
```

Use **arrow operator (->)** when accessing through pointers.

Arrays of Structures

Used when dealing with multiple records.

```
Student students[3];
Input/Output:

for (int i = 0; i < 3; i++) {
    cin >> students[i].roll;
    cin >> students[i].name;
    cin >> students[i].marks;
}
```

Nested Structures

Structure inside another structure.

```
struct Date {
    int day, month, year;
};

struct Student {
    int roll;
    char name[50];
    Date dob; // Nested structure
};

Access:

s1.dob.day = 12;
s1.dob.month = 5;
s1.dob.year = 2001;
```

Structure vs Class (Important in C++)

Feature struct class

Default Access public private

Used for Data grouping Data + behavior

Inheritance Public/Private Public/Private

Methods Allowed? Yes Yes

In C++, both can have functions, constructors, and destructors. Only difference: default access modifier.

Practical Examples

1. Employee Record System

```
struct Employee {
    int id;
    string name;
    float salary;
};
```

2. Book Inventory

```
struct Book {
    string title;
    string author;
    float price;
};
```

Advanced Features (Optional for deeper understanding)

- Structures can have member functions in C++.
- You can define **constructors** inside structs.

• Can also use **dynamic memory allocation** with new and delete.

Practice Questions for Students

- 1. Create a structure to store information of 5 books.
- 2. Write a function to update the salary of employees by 10%.
- 3. Define a nested structure for a student's contact info (with address and phone).
- 4. Write a program to find students with the highest marks.
- 5. Demonstrate structure pointer with a function.

C++ Namespaces

C++ Namespaces

What Is a Namespace in C++?

A namespace in C++ is a container that allows you to group identifiers (like variables, functions, classes) under a name. It helps to avoid name conflicts when you have multiple libraries or code modules.

Think of a namespace like a **folder in a computer**. Two files with the same name can exist in different folders. Similarly, two variables or functions with the same name can exist in different namespaces.

X Why Are Namespaces Needed?

Let's say two libraries define a function named print().

- Without namespaces, it creates a **name collision** (compiler error).
- With namespaces, both can coexist without conflict.

▲ Example Without Namespace (Name Conflict):

```
void print() {
    cout << "Library A Print" << endl;
}

void print() {
    cout << "Library B Print" << endl; // X Error: redefinition of 'print'
}</pre>
```

Example With Namespaces:

```
namespace A {
    void print() {
        cout << "Library A Print" << endl;
    }
}</pre>
```

```
namespace B {
    void print() {
        cout << "Library B Print" << endl;
    }
}
int main() {
    A::print(); // Access using namespace A
    B::print(); // Access using namespace B
    return 0;
}</pre>
```

Syntax of Namespace

```
namespace namespace_name {
    // declarations
    int var;
    void func();
    class MyClass;
}
```

Basic Example

```
#include <iostream>
using namespace std;

namespace MySpace {
   int x = 10;

   void greet() {
      cout << "Hello from MySpace!" << endl;
   }
}

int main() {
   MySpace::greet(); // Access using scope resolution cout << "Value of x: " << MySpace::x << endl;</pre>
```

```
return 0;
}
```

Key Concepts of Namespaces

1. Access Using Scope Resolution Operator ::

```
namespace A {
    int val = 100;
}
int main() {
    cout << A::val;</pre>
}
```

In C++17 and above:

2. Using the using Keyword

```
using namespace A;
cout << val; // No need to use A::val</pre>
```

⚠ Only use this when you're sure no conflicts exist, especially in **small programs**.

3. Nested Namespaces (C++11 onwards)

```
namespace Outer {
    namespace Inner {
        void sayHi() {
            cout << "Hi from Inner!" << endl;</pre>
        }
    }
}
Outer::Inner::sayHi(); // Access nested namespace
```

```
namespace Outer::Inner {
    void sayHi() {
        cout << "Hi from C++17 nested namespace!" << endl;</pre>
    }
}
```

4. Anonymous (Unnamed) Namespace

Used for internal linkage (like static variables). It restricts visibility to that translation unit (file).

```
namespace {
    int secret = 42;
    void show() {
        cout << "Secret = " << secret << endl;</pre>
    }
}
```

No need to name the namespace; can't access it from other files.

Real-Life Analogy for Students

Real World		C++ Analogy
Different folders		Different namespaces
file.txt C:/Docs	in	x in namespace A
file.txt D:/Data	in	x in namespace B
File access		A::x, B::x



Practical Classroom Use Case

Let's assume you and another student create a function called display() for a project. Without namespaces, you can't compile both functions in the same project.

With namespaces:

```
namespace Teacher {
    void display() {
        cout << "Display by Teacher\n";
    }
}
namespace Student {
    void display() {
        cout << "Display by Student\n";
    }
}
int main() {
    Teacher::display();
    Student::display();
}</pre>
```

Do's and Don'ts

✓ Good Practice ✓ Bad Practice

```
Use namespace to avoid Use global scope for everything clashes

Use A::function() explicitly Use using namespace std; in headers

Prefer local using inside Avoid namespace pollution main()
```

Interview Question Ideas

1. What is a namespace and why is it used?

- 2. Difference between using namespace and scope resolution?
- 3. What is an unnamed namespace?
- 4. Can you define a function with the same name in two namespaces?

Practice Exercises for Students

- 1. Create two namespaces Math and Physics, each with a function formula() print different outputs.
- 2. Create a nested namespace and access it.
- 3. Use an anonymous namespace to define a secret constant.
- 4. Explain what happens when you use using namespace std; globally.

Mini Quiz

Q1: What will this print?

```
namespace X {
    int x = 5;
}
namespace Y {
    int x = 10;
int main() {
    cout << X::x << " " << Y::x;
}
```

V Output: 5 10

◯ Summary

- Namespaces help organize code and avoid naming conflicts.
- Access members using :: operator.
- Use using namespace for convenience (with caution).
- Nested and anonymous namespaces provide more control.
- They're especially important in larger applications or multiple-file projects.

Would you like this in:

- PDF format for print?
- PowerPoint slides for classroom teaching?
- Live code examples or interactive C++ playgrounds?

Let me know how you'd like to present this to your students!

You said:

using namespace std; what is the meaning of this line

ChatGPT said:

Great question — this is one of the most common and important lines in C++ for beginners to understand clearly.



Explanation of using namespace

std;

What it means:

cpp
CopyEdit
using namespace std;

This line tells the compiler:

"I want to use everything from the std namespace directly, without writing std:: every time."

What is std?

- std stands for **standard**.
- It is the **standard namespace** that contains all the built-in C++ standard library features.

Examples:

- std::cout standard character output stream
- std::cin standard character input stream
- std::string standard string class
- std::vector, std::endl, std::sort, etc.

Without using namespace std;

You have to write std:: before every standard library item:

```
cpp
CopyEdit
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    std::string name = "Alice";
    std::cout << "Name: " << name << std::endl;
    return 0;
}</pre>
```

With using namespace std;

You can write it more simply:

```
срр
CopyEdit
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;</pre>
    string name = "Alice";
    cout << "Name: " << name << endl;</pre>
    return 0;
}
```

Why It's Not Always Recommended

While it's okay for small programs or beginners, it's not good in large projects or header files because:

Risk of Name Collisions

If your code or another library has a function or variable with the same name as one in std, it may cause conflicts.

Example:

```
срр
CopyEdit
namespace myspace {
    void sort() {
        // your custom sort function
    }
}
using namespace std; // std has its own sort()
sort(); // \times Now the compiler doesn't know which sort to call
```

☑ Best Practice (especially for professionals)

Instead of using:

```
cpp
CopyEdit
using namespace std;
Do this:
cpp
```

CopyEdit
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

Only bring in what you need, not everything.

Conclusion

using namespace std; Meaning

Imports everything from std So you don't need to write std::

everywhere

Easy for beginners Recommended for small, simple programs

 ** Session 10: Time and Space Complexity

* Session 10: Algorithm Analysis – Time and Space Complexity

Session Objectives

By the end of this session, your students will be able to:

- Understand what algorithm analysis is and why it's important.
- Grasp the concept of Time Complexity and Space Complexity.
- Learn how to analyze simple C++ code using Big-O Notation.
- Identify common patterns: O(1), O(n), O(n²), O(log n), O(n log n).
- Understand how algorithms scale with input size.
- Practice tracing loops and calculating time.

Why Algorithm Analysis?

Let's say you write two programs to solve the same problem. Both give correct results, but:

- One takes 2 seconds
- One takes 2 hours

Which one is better? ➤ The faster one!

- Algorithm analysis helps you:
 - Know which code is faster
 - Write **efficient** programs
 - Avoid slow programs that crash for big inputs



What is an Algorithm?

An **algorithm** is a step-by-step method or logic to solve a problem.

Example:

Finding the largest number in an array

Steps:

- 1. Assume the first number is the largest.
- 2. Check every other number.
- 3. Update the largest when needed.
- 4. Done!

Time Complexity – Explained Simply

> Definition:

Time complexity measures how many steps an algorithm takes as input size grows.

We do **not measure actual seconds**. We only count how many **operations** the algorithm performs.



More input = more steps = more time.

We represent time complexity using **Big-O notation**.

Big-O Notation (Time Complexity Patterns)

Notation	Meaning	Example	Performance
O(1)	Constant time	Accessing an array index	🔥 Fast
O(n)	Linear time	Single loop	Good

 O(n²)
 Quadratic time
 Nested loops
 ⚠ Slow

 O(log n)
 Logarithmic time
 Binary search
 ☒ Efficient

 O(n log n)
 Linearithmic time
 Merge sort, Quick sort
 ☒ Optimal

 O(2^n)
 Exponential time
 Recursion with multiple branches
 ☒ Very slow

X How to Find Time Complexity of Code

Rule 1: Count Loops

```
for (int i = 0; i < n; i++) {
   cout << i;
}</pre>
```

 \Rightarrow Runs **n times** \rightarrow **O(n)**

Rule 2: Nested Loops = Multiply Time

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      cout << i << j;
   }
}</pre>
```

ightharpoonup Runs $n \times n = n^2$ times $\rightarrow O(n^2)$

Rule 3: If statement doesn't change complexity

```
if (x > 0) {
   cout << "Hello";
}</pre>
```

 \Rightarrow Runs once \rightarrow O(1)

Rule 4: Drop constants and lower terms

```
for (int i = 0; i < n; i++) {
    cout << i;
}
for (int j = 0; j < n; j++) {
    cout << j;
}</pre>
```

 \rightarrow O(n) + O(n) = O(2n) \rightarrow drop constants \rightarrow O(n)

🔁 Real-Life Example

Task: Distribute sweets to students

- One student → 1 step
- 100 students → 100 steps
- \rightarrow Time increases **linearly** \rightarrow O(n)

Now imagine:

- For each student, you compare them with every other student
- \rightarrow n × n comparisons \rightarrow O(n²)

Space Complexity – How Much Memory Your Code Uses

> Definition:

Space complexity measures how much RAM your program needs as input grows.

Includes:

- Variables
- Arrays

Function call stack

• Examples:

```
int a; // 0(1)
int arr[n]; // 0(n)
int arr[n][n]; // 0(n^2)
```

Practical Examples

Example 1: Single loop

```
for (int i = 0; i < n; i++) {
   cout << i;
}</pre>
```

- Time complexity: O(n)
- Space complexity: **O(1)** (no extra space used)

Example 2: Nested loop

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      cout << i * j;
   }
}</pre>
```

- Time: O(n²)
- Space: **O(1)**

Example 3: Array allocation

```
int arr[n];  // or int* arr = new int[n];
```

• Time: depends on usage

• Space: O(n)

Simple Analogy to Explain Big-O

© Comparing Running Speeds:

Person	Steps Taken	Complexity
Alice (constant speed)	Always takes 1 step	O(1)
Bob (linear speed)	Takes 1 step per item	O(n)
Charlie (nested steps)	Compares each pair	O(n²)

Chart – Input Size vs Performance

Input Size (n)	O(1)	O(n)	O(n²)
5	1	5	25
10	1	10	100
100	1	100	10,000

⇒ Big-O matters more as n increases!

Summary

Concept	Description
Algorithm	Step-by-step solution
Time Complexity	Steps your code takes
Space Complexity	Memory your code uses
Big-O Notation	Describes growth of time/space

O(1) Fastest

O(n) Scales linearly

O(n²) Nested loops, slower

O(log n) Very efficient (binary search)

Activities for Students

Activity 1: Analyze this code

```
for (int i = 0; i < n; i++) {
   cout << "Hello";
}</pre>
```

→ O(n)

Activity 2: Spot the complexity

```
int a = 10; // 0(1)
int arr[n]; // 0(n)
```

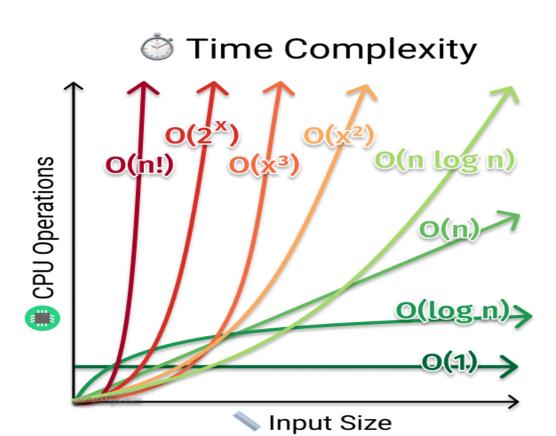
Activity 3: Write code with each of these:

- O(1)
- O(n)
- O(n²)

Example:

```
// 0(1)
cout << "Hi";
// 0(n)
for (int i = 0; i < n; i++) cout << i;</pre>
```

```
// 0(n²)
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    cout << i + j;</pre>
```



Tab 22

* Session 10 Quiz: Algorithm Analysis– Time and Space Complexity

✓ Part A: Multiple Choice Questions (MCQs)

Choose the correct answer.

- 1. What does **Time Complexity** measure?
 - a) How much memory your code uses
 - b) How many lines of code are written
 - c) How fast your code runs depending on input size
 - d) How many variables are declared
- 2. Which of the following is the **best** time complexity?
 - a) O(n²)
 - b) O(n)
 - c) O(1)
 - d) O(log n)
- 3. What is the **Time Complexity** of this loop?

```
for (int i = 0; i < n; i++) {
    cout << i;
}

a) O(1)
b) O(n)
c) O(n²)
d) O(log n)</pre>
```

4. What is the **Space Complexity** of the following code?

```
int arr[n];
```

```
a) O(1)
b) O(n²)
c) O(n)
d) O(log n)
```

5. What is the **Time Complexity** of nested loops?

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << i << j;
    }
}

a) O(n)
b) O(n²)
c) O(log n)
d) O(1)</pre>
```

🔽 Part B: True or False

- 6. ___ Space complexity is about how fast your code runs.
- 7. ___ O(1) means the code runs in constant time.
- 8. \bigcirc O(n²) is faster than O(n).
- 9. ___ Binary search runs in O(log n) time.
- 10. ___ O(n log n) is better than $O(n^2)$.

Part C: Fill in the Blanks

- 11. The Big-O notation describes how _____ increases with input size.
- 12. The time complexity of a loop that runs n times is ______.
- 13. A nested loop usually results in _____ time complexity.

- 14. If memory usage depends on n, space complexity is ______.
- 15. Searching a sorted array using binary search takes _____ steps in O(log n) time.

Part D: Output & Analysis Questions

16. What is the time complexity of this function?

```
void show(int n) {
    for (int i = 0; i < n; i++) {
        cout << i;
    }
}
a) O(n)
b) O(1)
c) O(n²)
d) O(log n)</pre>
```

17. What is the space complexity of this code?

```
int a = 5;
int b = 10;

a) O(n)
b) O(1)
c) O(n²)
d) O(log n)
```

18. Choose the function with the highest time complexity:

- a) Accessing a single element in an array
- b) A loop from 1 to n
- c) Nested loops each from 1 to n
- d) A loop that divides n by 2 each time

Answer Key (For Teachers)

Part A:

- 1. c
- 2. c
- 3. b
- 4. c
- 5. b

Part B:

- 6. False
- 7. True
- 8. False
- 9. True
- 10. True

Part C:

- 11. time
- 12. O(n)
- 13. O(n²)
- 14. O(n)
- 15. log n steps

Part D:

- 16. a
- 17. b
- 18. c