Name: Prathamesh Arvind Jadhav

DSA Course

DAY-1:

CPP Basics:

1. Input and Output in C++

Header File Required

```
#include <iostream>

using namespace std;
```

Input

- Uses cin (Console Input) with the extraction operator >>

```
int age;

cin >> age; // User inputs a value for age
```

Output

- Uses cout (Console Output) with the insertion operator <<

```
cout << "Age is: " << age;
```

2. Variables in C++

- A **variable** is a container for storing data.
- Must be **declared with a data type** before using.

```
int age = 20;

float weight = 55.5;
```

Rules for Naming Variables

- Can contain letters, digits, and underscores.
- Cannot start with a digit.
- Cannot be a keyword (like int, float, etc.).

3. Data Types in C++

Type	Description	Example
int	Integer (4 bytes)	10
float	Floating point (4 bytes)	3.14
double	Double precision float(8 Byte)	3.14159
char	Character (1 byte)	'A'
bool	Boolean (true/false)	true, false
string	Text (needs <string>)	"Hello"

Example

```
#include <string>

string name = "John";
```

4. Operators in C++

Arithmetic Operators

```
+   // Addition

-   // Subtraction

*   // Multiplication

/   // Division

%   // Modulus (remainder)
```

Relational (Comparison) Operators

```
==  // Equal to
```

```
!=  // Not equal to
>   // Greater than
<   // Less than
>=  // Greater than or equal to
<=  // Less than or equal to
```

Logical Operators

```
&&  // Logical AND
||  // Logical OR
!   // Logical NOT
```

Assignment Operators

```
=   // Assign
+=  // Add and assign
-=  // Subtract and assign
*=  // Multiply and assign
/=  // Divide and assign
```

Increment/Decrement

```
++i; // Pre-increment
i++; // Post-increment
--i; // Pre-decrement
i--; // Post-decrement
```

5. Type Casting in C++

- **Type casting** converts a variable from one type to another.

Syntax

```
data_type(variable)
```

Example

```
int a = 10, b = 3;

float result = (float)a / b;  // Output: 3.33333
```

- **Implicit casting**: Automatic by compiler when safe.
- **Explicit casting**: Done by the programmer.

Conditional Statements (Decision Making)

Conditional statements help in making decisions based on certain conditions.

□ Types of Conditional Statements:

□ if Statement

- **Syntax:**

  ```
  if (condition) {
      // code to execute if condition is true
  }
  ```

- **Example:**

  ```
  int age = 20;
  if (age >= 18) {
      cout << "You are eligible to vote.";
  }
  ```

□ if-else Statement

- **Syntax:**

  ```
  if (condition) {
      // true block
  } else {
      // false block
  ```

```
}
```

- **Example:**

```
int marks = 45;
if (marks >= 50) {
    cout << "Passed";
} else {
    cout << "Failed";
}
```

□ else if Ladder

- Useful when checking multiple conditions.
- **Syntax:**

```
if (condition1) {
    // block1
} else if (condition2) {
    // block2
} else {
    // default block
}
```

- **Example:**

```
int score = 75;
if (score >= 90) {
    cout << "Grade A";
} else if (score >= 80) {
    cout << "Grade B";
} else if (score >= 70) {
    cout << "Grade C";
} else {
    cout << "Fail";
}
```

□ switch Statement

- Best for checking equality against multiple values.
- **Syntax:**

```
switch (expression) {
    case value1:
        // code block
        break;
    case value2:
        // code block
        break;
    default:
        // default block
}
```

- **Example:**

```
int day = 3;
switch (day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    case 3: cout << "Wednesday"; break;
    default: cout << "Invalid Day";
}
```

□ 2. Loops (Repetitive Tasks)

Loops allow executing a block of code multiple times.

□ for Loop

- Best when you know how many times to loop.
- **Syntax:**

```
for (initialization; condition; update) {
    // loop body
}
```

- **Example:**

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

□ while Loop

- Condition is checked **before** execution.
- Best when the number of iterations is unknown.
- **Syntax:**

```
while (condition) {
    // code
}
```

- **Example:**

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

□ do-while Loop

- Condition is checked **after** the loop body.
- Executes at least once.
- **Syntax:**

```
do {
    // code
} while (condition);
```

- **Example:**

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

Name: Prathamesh Arvind Jadhav

□ Nested Loops

- A loop inside another loop.
- **Example:**

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        cout << "(" << i << "," << j << ") ";
    }
    cout << endl;
}
```

□ Loop Control Statements

These control the flow inside loops.

□ break:

- Exits the loop immediately.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;
    cout << i << " ";
}
```

□ continue:

- Skips the current iteration and moves to the next.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    cout << i << " ";
}
```

□ Summary Table

Name: Prathamesh Arvind Jadhav

Statement	Used For	Checks Condition When?	Executes At Least Once?
if, if-else, else-if	Conditional logic	N/A	Depends on condition
switch	Multi-value condition	N/A	Based on case match
for	Fixed iteration	Before loop	No
while	Unknown iteration	Before loop	No
do-while	Unknown iteration	After loop	Yes

Day-2:

Pattern Problems:

□ 1. Right-Angled Triangle of Stars

□ Pattern:

```
*
* *
* * *
* * * *
* * * * *
```

□ Code:

```
int n = 5;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        cout << "* ";
    }
    cout << endl;
}
```

□ 2. Inverted Triangle of Stars

□ Pattern:

```
* * * * *
* * * *
* * *
* *
*
```

□ Code:

```
int n = 5;
for (int i = n; i >= 1; i--) {
    for (int j = 1; j <= i; j++) {
        cout << "* ";
    }
    cout << endl;
}
```

□ 3. Pyramid Pattern

□ Pattern:

```
    *
   * *
  * * *
 * * * *
* * * * *
```

□ Code:

```
int n = 5;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n - i; j++) {
        cout << " ";
    }
    for (int k = 1; k <= i; k++) {
        cout << "* ";
    }
    cout << endl;
}
```

Name: Prathamesh Arvind Jadhav

□ 4. Number Triangle

□ Pattern:

```
1
1 2
1 2 3
1 2 3 4
```

□ Code:

```
int n = 4;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        cout << j << " ";
    }
    cout << endl;
}
```

□ 5. Floyd’s Triangle

□ Pattern:

```
1
2 3
4 5 6
7 8 9 10
```

□ Code:

```
int n = 4, num = 1;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        cout << num++ << " ";
    }
    cout << endl;
}
```

□ 6. Checkerboard (Using if condition)

Name: Prathamesh Arvind Jadhav

□ Pattern:

```
1 0 1 0
0 1 0 1
1 0 1 0
0 1 0 1
```

□ Code:

```
int n = 4;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if ((i + j) % 2 == 0)
            cout << "1 ";
        else
            cout << "0 ";
    }
    cout << endl;
}
```

□ 7. Hollow Rectangle

□ Pattern:

```
* * * * *
*       *
*       *
* * * * *
```

□ Code:

```
int rows = 4, cols = 5;
for (int i = 1; i <= rows; i++) {
    for (int j = 1; j <= cols; j++) {
        if (i == 1 || i == rows || j == 1 || j == cols)
            cout << "* ";
        else
            cout << "  ";
    }
    cout << endl;
```

}

□ 8. Diamond Pattern

□ Pattern:

```
  *
 * *
* * *
 * *
  *
```

□ Code:

```
int n = 3;
// Upper part
for (int i = 1; i <= n; i++) {
    for (int j = i; j < n; j++) cout << " ";
    for (int k = 1; k <= i; k++) cout << "* ";
    cout << endl;
}
// Lower part
for (int i = n - 1; i >= 1; i--) {
    for (int j = n; j > i; j--) cout << " ";
    for (int k = 1; k <= i; k++) cout << "* ";
    cout << endl;
}
```

DAY-3:

Functions:

□ What is a Function?

A **function** is a block of code that performs a specific task. It helps you **reuse code**, **reduce redundancy**, and **organize** your program efficiently.

□ Why Use Functions?

- **Modular Code**: Breaks large problems into smaller, manageable parts.
- **Reusability**: Write once, use many times.
- **Readability**: Easier to read and debug.
- **Avoid Redundancy**: No need to repeat the same code again.

□ Types of Functions

Type	Description
Built-in Functions	Provided by C++ like sqrt(), pow()
User-defined Functions	Functions created by the programmer

□ Syntax of a Function

```
return_type function_name(parameter_list) {
    // body of function
    return value; // if return_type is not void
}
```

□ Example:

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

int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3);
    cout << "Sum = " << result;
    return 0;
}
```

□ Parts of a Function

Part	Description
return_type	Type of value the function returns (int, void, etc.)
function_name	Unique name for identification
parameter_list	Values passed into the function
function body	Code block that defines what the function does
return	Sends value back to the caller (if not void)

□ Function Declaration (Prototype)

- Used before main() to declare a function’s existence.

```
int add(int, int); // Declaration

int main() {
   cout << add(2, 3);
}

int add(int a, int b) {
   return a + b;
}
```

□ Calling a Function

- To **use** the function, you "call" it with arguments:

```
int result = add(4, 7); // Function call
```

□ Function with No Return (void)

```
void greet() {
   cout << "Hello, User!";
}
```

□ Types of User-defined Functions

Name: Prathamesh Arvind Jadhav

Type	Example
No arguments, no return	void greet()
With arguments, no return	void greet(string name)
No arguments, returns value	int getInput()
With arguments, returns value	int sum(int a, int b)

□ Example for All Types

1. No Argument, No Return

```
void showMessage() {
    cout << "Welcome!";
}
```

2. With Argument, No Return

```
void greet(string name) {
    cout << "Hello " << name;
}
```

3. No Argument, With Return

```
int giveNumber() {
    return 10;
}
```

4. With Argument, With Return

```
int square(int x) {
    return x * x;
}
```

□ Pass by Value vs Pass by Reference

□ Pass by Value (Copy is passed)

```
void change(int a) {
    a = 10;
}
```

□ Pass by Reference (Original is modified)

```
void change(int &a) {
    a = 10;
}
```

□ Recursive Functions

A **function calling itself**.

□ Example: Factorial

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

□ Inline Functions

- Used for **short** functions.
- Increases performance by replacing function call with the function code.
- Syntax:

```
inline int square(int x) {
    return x * x;
}
```

□ Default Arguments

- Allows **default values** for parameters.

```
int power(int base, int exp = 2) {
    return pow(base, exp);
}
```

Name: Prathamesh Arvind Jadhav

Summary Table

Concept	Example	Description
Declaration	int add(int, int);	Introduce function before use
Definition	int add(int a, int b) {}	Actual code of the function
Call	add(5, 3);	Using the function
Return type	int, void, float	Type of value returned
Recursion	factorial(n)	Function calling itself
Inline function	inline int square(int x)	Suggests compiler to expand inline
Default arguments	power(3)	Optional parameters
Reference vs Value	void func(int &x)	Modify original value

Day-4:

Pointers:

□ What is a Pointer?

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

□ Think of it like:

A pointer is a signboard that shows *where* a house (variable) is, not the house itself.

□ Why Use Pointers?

- To **access and modify** variables indirectly.
- To work with **arrays, functions, and dynamic memory**.
- To improve **performance** in large data operations.
- To **pass large data efficiently** to functions.

□ Pointer Syntax

data_type *pointer_name;

* means this variable is a pointer to the given data_type.

□ Example 1: Basic Pointer Usage

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

int main() {
    int x = 10;
    int *ptr = &x;  // ptr stores the address of x

    cout << "Value of x: " << x << endl;
    cout << "Address of x: " << &x << endl;
    cout << "Pointer ptr holds: " << ptr << endl;
    cout << "Value pointed by ptr: " << *ptr << endl;

    return 0;
}
```

□ Output Explanation:

- &x gives address of x.
- *ptr gives value **stored at** that address.

□ Pointer Terms

Term	Meaning
&	Address-of operator
*	Dereference operator (get value)
ptr	Pointer variable (holds address)
*ptr	Value at the address stored in ptr

□ Changing Values Using Pointers

```
int x = 5;
int *ptr = &x;

*ptr = 20;  // changes value of x directly

cout << x;  // Output: 20
```

□ Pointer to Different Data Types

```
int a = 5;
float b = 3.14;
char c = 'A';

int* p1 = &a;
float* p2 = &b;
char* p3 = &c;
```

□ Pointer and Arrays

```
int arr[] = {10, 20, 30};
int *ptr = arr;  // array name is the address of the first element

cout << *ptr;      // 10
cout << *(ptr+1);  // 20
cout << *(ptr+2);  // 30
```

*(ptr + i) gives the i-th element of the array.

□ Pointer to Pointer (Double Pointer)

```
int a = 10;
int *ptr = &a;
int **pptr = &ptr;

cout << **pptr;  // Output: 10
```

First * gets ptr, second * gets a.

Name: Prathamesh Arvind Jadhav

□ Functions and Pointers (Pass by Reference)

□ Example: Swapping two values

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    swap(&x, &y);
    cout << x << " " << y;  // Output: 10 5
}
```

Changes reflect in original variables because we're modifying their addresses.

□ Pointers vs Normal Variables

Normal Variable	Pointer
Stores value	Stores address
int x = 5;	int *p = &x;
Access: x	Access: *p
Address: &x	Address: p

□ Null Pointer

Used to initialize a pointer when it is not assigned yet.

```
int *ptr = nullptr;  // Or NULL in older C++
```

□ Dangling Pointer

When a pointer refers to memory that has been freed or deleted.

```
int *ptr = new int(10);
delete ptr;       // Memory deleted
// Now ptr is dangling unless reset
ptr = nullptr;
```

□ Pointers and Dynamic Memory (new/delete)

□ Example: Allocating memory at runtime

```
int *ptr = new int;     // allocate memory
*ptr = 100;
cout << *ptr;

delete ptr;             // free memory
```

Always use delete to avoid memory leaks.

□ Summary of Pointer Operators

Operator	Name	Usage
*	Dereference	Get value at address
&	Address-of	Get address of variable
->	Member access	Access members from pointer to object/struct

---

Real-life Analogy

- Variable = **house**
- Address = **house number**
- Pointer = **someone holding the house number**
- *pointer = **going to that house and seeing what's inside**

Name: Prathamesh Arvind Jadhav

DAY-5:

Binary Number System:

□ What is the Binary Number System?

The **Binary Number System** is a **base-2** numeral system that uses only **two digits**:

0 and 1

□ Why Binary?

- Computers and digital devices use **binary logic**.
- Every value in memory (data, text, image, etc.) is ultimately stored as a series of **0s and 1s**.
- It is **efficient** and **easy** to represent with electronic signals (ON = 1, OFF = 0).

□ Basic Terminology

Term	Description
Bit	A single binary digit (0 or 1)
Nibble	4 bits
Byte	8 bits
LSB	Least Significant Bit (rightmost bit)
MSB	Most Significant Bit (leftmost bit)

□ Binary Positional Value System

Binary is a **positional number system**, just like decimal. Each digit has a **place value** which is a power of 2.

For a binary number 1011:

```
  1    0    1    1
 (8)  (4)  (2)  (1)   ← 2^3, 2^2, 2^1, 2^0
```

= 1×8 + 0×4 + 1×2 + 1×1 = 11 (in decimal)

□ Conversions Between Number Systems

□ Binary → Decimal

Method: Multiply each bit with 2 raised to its position (right to left) and sum them.

Example:

Binary: 1101
Decimal = $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$
$= 8 + 4 + 0 + 1 = 13$

□ Decimal → Binary

Method: Divide the number by 2 repeatedly and write remainders in reverse order.

Example: Convert 13 to binary

Division	Quotient	Remainder
13 ÷ 2	6	1
6 ÷ 2	3	0
3 ÷ 2	1	1
1 ÷ 2	0	1

Binary = 1101

□ Binary → Octal

Group binary digits in sets of 3 (right to left) and convert each to octal.

Example:

Binary: 110101

```
Grouped:  110 101
Octal:     6   5 → 65 (base 8)
```

□ Binary → Hexadecimal

Group digits in sets of 4 (right to left) and convert each to hexadecimal.

Example:

```
Binary:  11010110

Grouped:  1101 0110
Hex:       D     6 → D6 (base 16)
```

Binary Arithmetic

1. Binary Addition Rules

A	B	A + B	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Example:

```
  1101
+ 1011
------
11000
```

---

2. Binary Subtraction Rules

A	B	A − B	Borrow
0	0	0	0

A	B	A − B	Borrow
1	0	1	0
1	1	0	0
0	1	1	1

Example:

```
  1010
− 0011
-------
  0111
```

□ Binary Number Representation

Type	Example	Range (for 8 bits)
Unsigned Binary	00001101	0 to 255
Signed Binary (2’s)	11110011	-128 to +127 (2’s complement)
Binary Coded Decimal	0001 0011 (for 13)	0–9 per 4 bits

Real-Life Applications

- Memory addressing
- Networking (IP addressing)
- Machine-level programming
- Digital circuit design
- Encryption & data storage

Bitwise Operators in C++

Bitwise operators are used to perform operations on **bits** (binary representations).

Operator	Description	Example (a = 5, b = 3)	Result (in binary)
&	AND	a & b (5 & 3)	0101 & 0011 = 0001 (1)
`	`	OR	`a
^	XOR	a ^ b (5 ^ 3)	0101 ^ 0011 = 0110 (6)

Operator	Description	Example (a = 5, b = 3)	Result (in binary)
~	NOT (1's complement)	~a (~5)	~0101 = 1010 (in 2's complement = -6)
<<	Left Shift	a << 1	0101 << 1 = 1010 (10)
>>	Right Shift	a >> 1	0101 >> 1 = 0010 (2)

□ Use cases: Encryption, Graphics, Low-level programming.

□ 2. Data Type Modifiers in C++

Used to **change the size** or **sign** of the data types.

Modifier	Purpose	Example
signed	Can hold both +ve and -ve	signed int a = -10;
unsigned	Only +ve (doubles max value)	unsigned int a = 10;
short	Smaller range than int	short int a = 100;
long	Larger range than int	long int a = 100000;
long long	Very large integer	long long int a = 1e18;

□ unsigned int x = -1; → Causes wraparound.

□ 3. Type Conversion (Type Casting)

- **Implicit**: Automatically done by compiler.

  ```
  int a = 5;
  float b = a; // int to float automatically
  ```

- **Explicit**: Manual casting.

  ```
  float a = 5.5;
  int b = (int)a; // truncates decimal
  ```

□ 4. Storage Classes in C++

Define scope, lifetime, and linkage of variables.

Storage Class	Scope	Lifetime	Default Value	Keyword
auto	Local	Function block	Garbage	auto
register	Local (in CPU)	Fast access	Garbage	register
static	Local	Entire program	Zero	static
extern	Global	Entire program	Zero	extern

□ 5. Constants and Macros

- **const**: Constant variable.

  ```
  const int x = 10;
  ```

- **#define**: Preprocessor macro.

  ```
  #define PI 3.14159
  ```

DAY-6:

Arrays:

What is an Array?

An **array** is a **collection of elements** of the **same data type** stored in **contiguous memory locations** and accessed using **indexing**.

Syntax:

```
data_type array_name[size];
```

□ Types of Arrays

1. **One-Dimensional Array**
2. **Two-Dimensional Array (Matrix)**
3. **Multi-Dimensional Array**

Name: Prathamesh Arvind Jadhav

□ 1. One-Dimensional Array

Declaration:

```
int arr[5];  // Uninitialized array of 5 integers
```

Initialization:

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

Accessing Elements:

```
cout << arr[2];  // Outputs 3
```

Input and Output:

```
for (int i = 0; i < 5; i++)
    cin >> arr[i];

for (int i = 0; i < 5; i++)
    cout << arr[i] << " ";
```

□ 2. Two-Dimensional Array

A 2D array is like a **matrix**: rows × columns.

Declaration:

```
int mat[3][4];  // 3 rows, 4 columns
```

Initialization:

```
int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Access:

```
cout << mat[1][2];  // Outputs 6
```

Input/Output:

```
for (int i = 0; i < 2; i++)
    for (int j = 0; j < 3; j++)
        cin >> mat[i][j];
```

□ 3. Multi-Dimensional Arrays

- 3D arrays example:

```
int cube[2][3][4];
```

- Think of it as: 2 layers of 3×4 matrices.

□ Array Memory Allocation

- Arrays are stored in **contiguous memory**.
- Indexing starts at **0**.
- arr[i] refers to *(base_address + i × size_of_datatype)*

□ Common Operations

Operation	Description
Traversal	Loop through elements
Insertion	Insert at index (manual shifting required)
Deletion	Remove element by shifting
Searching	Find element using linear/binary search
Sorting	Bubble, Selection, Insertion, Merge, Quick etc.

□ Limitations of Arrays

- **Fixed size**: Cannot resize after declaration.
- **No bounds checking**: Accessing out-of-bounds is undefined behavior.
- **Inflexible insertion/deletion**

Name: Prathamesh Arvind Jadhav

□ Advantages of Arrays

- Fast access via index
- Efficient in memory for homogeneous data
- Easy to implement static data structures

□ Array Example Program

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

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

    int arr[n];
    cout << "Enter elements:\n";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

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

    return 0;
}
```

□ Array with Functions

```
void displayArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
}
```

Name: Prathamesh Arvind Jadhav

Vectors:

What is a Vector?

A **vector** in C++ is a part of the **STL (Standard Template Library)** that acts as a **dynamic array** — it can **resize itself automatically** when elements are added or removed.

Vectors are more powerful and flexible than traditional arrays.

□ Header File

```
#include <vector>
```

You can also use:

```
using namespace std;
```

□ Declaration and Initialization

```
vector<int> v;                    // Empty vector of int
vector<int> v(5);                 // Vector with 5 default-initialized elements (0)
vector<int> v(5, 10);             // Vector with 5 elements, each initialized to 10
vector<int> v2 = {1, 2, 3, 4};    // Initialization using list
```

□ Common Member Functions

Function	Description
v.size()	Returns number of elements
v.push_back(x)	Adds element x at the end
v.pop_back()	Removes last element

Function	Description
v.front()	Returns first element
v.back()	Returns last element
v[i] or v.at(i)	Access element at index i
v.clear()	Removes all elements
v.empty()	Returns true if vector is empty
v.insert(pos, val)	Inserts val at specified position
v.erase(pos)	Erases element at position
v.begin() / v.end()	Returns iterator to start/end of vector
v.resize(n)	Resizes vector to n elements
v.swap(v2)	Swaps contents with another vector
v.assign(n, val)	Assigns n copies of val to vector

□ Example: Basic Vector Usage

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

int main() {
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";  // Output: 10 20 30

    v.pop_back();  // Removes 30

    cout << "\nFront: " << v.front();  // 10
    cout << "\nBack: " << v.back();    // 20

    return 0;
}
```

□ Accessing Elements

- Using [] operator:

```
cout << v[1];  // Fast but unsafe (no bounds checking)
```

- Using at():

```
cout << v.at(1);  // Safe (throws out_of_range if invalid)
```

□ Traversing Vectors

➤ ***Using index-based loop:***

```
for (int i = 0; i < v.size(); i++)
    cout << v[i];
```

➤ ***Using auto and range-based loop:***

```
for (auto x : v)
    cout << x << " ";
```

➤ ***Using iterator:***

```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << " ";
```

□ Sorting a Vector

```
#include <algorithm>
sort(v.begin(), v.end());           // Ascending
sort(v.rbegin(), v.rend());         // Descending
```

□ Useful Vector Tricks

Copying a vector:

```
vector<int> v2 = v;  // Shallow copy
```

Removing duplicate elements:

```
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

Find an element:

```
if (find(v.begin(), v.end(), 20) != v.end())
    cout << "Found";
```

□ Vector of Pairs

```
vector<pair<int, int>> vp;
vp.push_back({1, 2});
vp.push_back(make_pair(3, 4));

for (auto p : vp)
    cout << p.first << " " << p.second << endl;
```

□ 2D Vectors (Vector of Vectors)

```
vector<vector<int>> matrix(3, vector<int>(4, 0));  // 3x4 matrix with 0s

matrix[1][2] = 5;
```

Input/output in 2D vector:

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
        cin >> matrix[i][j];

for (auto row : matrix) {
    for (auto val : row)
        cout << val << " ";
    cout << "\n";
}
```

□ Advantages of Vectors over Arrays

Feature	Vector	Array
Size	Dynamic	Fixed
Bounds Check	at() supports it	Not available
Functions	Rich STL support	None
Easy to insert	push_back(), insert()	Manual shifting
Safe & flexible	Yes	No

□ When NOT to Use Vectors

- When fixed-size arrays are sufficient and performance is extremely critical (e.g., embedded systems).
- When you need raw pointers and memory control.

DAY-7: