1. Introduction to C++

C++ is a powerful, fast, and widely-used **object-oriented programming language**. It is commonly used for software development, game development, and system programming.

Header Files of C++

A **header file** provides **input/output** functions and other important features.

✓ `#include <iostream>`

- Used to enable **input (cin)** and **output (cout)**.
- Must be written at the top of the program.

```
#include <iostream>
```

Blank Line

- A **blank line** adds space between lines.
- It improves readability but has **no effect** on the code execution.

```
cout << "Hello";

cout << "World";  // Blank line in between is OK
```

Void Spaces (Whitespaces)

- Whitespaces are **spaces, tabs, or blank lines**.
- They are used to **separate tokens** (words, symbols) in code.

```
int    x   =    5;  // Extra spaces are allowed
```

---

int main()

- The `main()` function is where the program **starts running**.
- `int` means the function returns an **integer** (usually 0).

```
int main() {
    // code here
    return 0;
}
```

Functions and Keywords

✅ Function Definition

- A **function** is a block of code that performs a task.

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

✅ Function Calling

- To run a function, you must **call it**.

```
greet();  // This will print "Hello!"
```

✅ Keywords in C++

Keywords are **reserved words** with special meaning in C++. You **cannot use them as variable names**.

Examples:

`int, float, return, if, else, for, while, switch, void`

✅ Semicolon Usage (;)

- Every **C++ statement must end** with a semicolon `;`.

```
int a = 10;     // correct
cout << a;      // correct
```

Missing a semicolon gives a **compilation error**.

Example: Full Simple Program

```
#include <iostream>  // Header file
using namespace std;


void greet() {       // Function definition
    cout << "Welcome to C++!" << endl;
}
```

```
int main() {          // Starting point
    greet();          // Function call
    return 0;         // End of program
}
```

Output:

```
Welcome to C++!
```

➗ 2. Basic Arithmetic Operations in C++

In C++, arithmetic operations are performed using **operators**.

✅ Arithmetic Operators

Operator	Name	Description	Example (a = 10, b = 3)
+	Addition	Adds two numbers	a + b = 13
-	Subtraction	Subtracts second number from first	a - b = 7
*	Multiplication	Multiplies two numbers	a * b = 30
/	Division	Divides first number by second	a / b = 3 (integer division)
%	Modulus	Gives remainder of division	a % b = 1

Important Notes:

- If both operands are integers, **division /** gives only the **whole number** part (e.g., `7 / 2 = 3`).
- Use `float` or `double` for decimal division (e.g., `7.0 / 2 = 3.5`).

Example: Perform Basic Arithmetic Operations on Two Numbers

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


int main() {
    int num1, num2;


    // Input
    cout << "Enter first number: ";
    cin >> num1;
    cout << "Enter second number: ";
    cin >> num2;

    // Operations
    cout << "Addition: " << num1 + num2 << endl;
    cout << "Subtraction: " << num1 - num2 << endl;
    cout << "Multiplication: " << num1 * num2 << endl;

    // Prevent divide-by-zero error
    if (num2 != 0) {
        cout << "Division: " << num1 / num2 << endl;
        cout << "Modulus: " << num1 % num2 << endl;
    } else {
        cout << "Cannot divide by zero!" << endl;
    }

    return 0;
}
```

Sample Output:

If user inputs: `num1 = 10` `num2 = 3`

Then output:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3
Modulus: 1
```

4. Scope and Type Conversion

✓ Scope of a Variable

Scope refers to **where a variable can be accessed** in your program.

Types of Scope:

1. Local Scope

- A variable declared **inside a function or block**.
- Accessible only within that function/block.

```
void myFunction() {
    int x = 10; // local to myFunction
    cout << x;
}
```

2. Global Scope

- A variable declared **outside all functions**.
- Accessible from anywhere in the program.

```
int x = 100; // global

int main() {
    cout << x;
}
```

3. Block Scope

- A variable declared inside {} brackets.
- Limited to that block only.

```
int main() {
    {
        int x = 5;
        cout << x; // OK
    }
    // cout << x; // Error: x not in scope here
}
```

Type Conversion

Type conversion is the process of **changing one data type to another**.

---

1. Type Coercion

- *Happens **automatically** by the compiler.*
- *Example:*

```
int a = 10;
float b = 5.5;
float result = a + b; // a is converted to float automatically
```

---

2. Promotion and Demotion

- ***Promotion**: Converting a **smaller type to a larger type** (safe).*

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

- ***Demotion**: Converting a **larger type to a smaller type** (may lose data).*

```
float a = 5.7;
int b = a;    // float demoted to int => b becomes 5
```

---

3. Rules for Type Conversion

- ***Higher type wins** in expressions:*

```
int a = 3;
double b = 4.5;
auto result = a + b; // a promoted to double
```

- *Conversion Order:* `bool → char → int → float → double`
- *If both operands are of different types, **the lower type is converted to the higher type** before the operation.*

Overflow and Underflow

Overflow

- Happens when a value is **too large** for the data type.
- Example:

```
unsigned char x = 255;
x = x + 1;  // x becomes 0 (wraps around)
```

Underflow

- Happens when a value is **too small** for the data type (e.g., below 0 for unsigned types).
- Example:

```
unsigned int x = 0;
x = x - 1;  // x becomes a large value (wraps around)
```

Example Code Demonstrating Type Conversion and Overflow

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


int main() {
    int a = 10;
    float b = 3.5;
    float result = a + b;  // int converted to float


    cout << "Result = " << result << endl;


    unsigned char x = 255;
    x = x + 1;  // overflow


    cout << "After overflow, x = " << (int)x << endl;


    return 0;
}
```

5. Switch Statement and Menus in C++

The `switch` statement is used to **choose between multiple options** based on the value of a variable or expression.

✓ Syntax of Switch Statement

```
switch (expression) {
    case value1:
        // Code for value1
        break;
    case value2:
        // Code for value2
        break;
    ...
    default:
        // Code if no case matches
}
```

- `break` *stops the execution of further cases.*
- `default` *is optional and runs if no case matches.*

1. Arithmetic Operations with Switch

You can use `switch` to perform operations like **add**, **subtract**, etc., based on user choice.

Example: Taking Two Numbers as Input and Performing Arithmetic Operations

```
#include <iostream>
using namespace std;
int main() {
    int num1, num2, choice;
    cout << "Enter first number: ";
    cin >> num1;
    cout << "Enter second number: ";
    cin >> num2;
    cout << "\nSelect Operation:\n";
    cout << "1. Addition\n";
    cout << "2. Subtraction\n";
    cout << "3. Multiplication\n";
    cout << "4. Division\n";
    cout << "Enter your choice: ";
    cin >> choice;
    switch (choice) {
        case 1:
            cout << "Result = " << num1 + num2 << endl;
            break;
        case 2:
            cout << "Result = " << num1 - num2 << endl;
            break;
        case 3:
            cout << "Result = " << num1 * num2 << endl;
            break;
        case 4:
            if (num2 != 0)
                cout << "Result = " << num1 / num2 << endl;
            else
                cout << "Division by zero not allowed.\n";
            break;
        default:
            cout << "Invalid choice.\n";
    }
    return 0;
}
```

2. Displaying a Menu (Press 1 to Add, 2 to Subtract, etc.)

This is already shown in the previous example: you display options and use `switch` to respond to the user's choice.

3. Display Day of the Week Using Switch

```
#include <iostream>
using namespace std;
int main() {
    int day;
    cout << "Enter day number (1 to 7): ";
    cin >> day;
    switch (day) {
        case 1:
            cout << "Monday";
            break;
        case 2:
            cout << "Tuesday";
            break;
        case 3:
            cout << "Wednesday";
            break;
        case 4:
            cout << "Thursday";
            break;
        case 5:
            cout << "Friday";
            break;
        case 6:
            cout << "Saturday";
            break;
        case 7:
            cout << "Sunday";
            break;
        default:
            cout << "Invalid day number.";
    }
    return 0;
}
```

Data Types

Data types define **what kind of data** a variable can store. In C++, they are divided into several categories:

✅ *1. Integer (int)*

- Stores **whole numbers** (no decimals).
- Example:

```
int age = 20;
```

✅ *2. Floating Point Types*

Used to store **decimal (fractional) numbers**.

Type	Size	Example
float	4 bytes	float pi = 3.14;
double	8 bytes	double largePi = 3.1415926535;

✅ *3. Character (char)*

- Stores a **single character** in single quotes.
- Example:

```
char grade = 'A';
```

✅ *4. Derived Types*

- Formed using basic types.
- Examples:
  - **Arrays**: `int numbers[5];`

  - **Pointers**: `int* ptr;`
  - **Functions**: that return data types.

✓ *5. User-Defined Identifiers*

- These are **names you give** to variables, functions, classes, etc.
- Must:
  - Start with a letter or underscore
  - Not be a **reserved word**
  - Example:

```
 int studentMarks;  // valid
```

Variables

A **variable** is a named space in memory used to **store data**.

Variable Declaration

- Tells the compiler the **name and type** of the variable.
- Example:

```
    int marks;
float percentage;
```

Variable Initialization

- Assigning a **starting value** to a variable.
- Example:

```
        int marks = 95;
    float pi = 3.14;
```

Reserved Words (Keywords)

- Special words that **have fixed meaning** in C++.
- ❌ You **cannot use them** as variable names.

Examples:

`int, float, if, else, while, return, void, for, switch, break`

Valid variable: `int score;` ❌ Invalid: `int int;` (because `int` is a keyword)

Example Code: Using All the Above Concepts

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


int main() {
    int age = 18;              // Integer
    float height = 5.8;        // Float
    char grade = 'A';          // Character


    cout << "Age: " << age << endl;
    cout << "Height: " << height << endl;
    cout << "Grade: " << grade << endl;


    return 0;
}
```

Functions and Modular Programming

What is Modular Programming?

Modular programming means breaking a large program into smaller, manageable, and reusable parts called **modules** or **functions**. Each function performs a specific task. This makes the program:

- Easier to understand.
- Easier to test.
- Easier to fix errors.
- Easier to reuse.

Function Overview

A **function** is a block of code written to perform a specific task when it is called.

Example:

- You want to add two numbers. You can write a function named `addNumbers()` to do this task.

In C++, there are two types of functions:

4. **Library Functions** — Built-in functions provided by C++ (like `sqrt()`, `pow()`, `cin`, `cout`).
5. **User-Defined Functions** — Functions you create according to your program's needs.

Function Components

A C++ function has these parts:

Part	Purpose
Return Type	What type of value the function will return (int, float, etc.)
Function Name	The name you give to the function

Part	Purpose
Parameters (Arguments)	Data you pass into the function
Function Body	The actual set of instructions the function performs
Return Value	The final result sent back by the function after execution

Return Type

The **Return Type** is the data type of the value the function will send back to the calling part of the program.

Common Return Types:

- `int` — Returns an integer value.
- `float` — Returns a decimal value.
- `char` — Returns a character.
- `void` — Returns nothing.

Example:

```
int addNumbers() // returns an integer
{
    return 5;
}
```

`void displayMessage()` → means this function won’t return any value.

Function Name

The **Function Name** is an identifier you give to your function. Rules for naming:

- Must begin with a letter or underscore.
- Can contain letters, numbers, and underscores.
- Should be meaningful (e.g., `calculateArea()`).

Example:

```
void greetUser()
{
    cout << "Hello!";
}
```

Parameters (or Arguments)

Parameters are variables listed inside the parentheses when you define a function. They receive values when the function is called.

Example:

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

Here, `name` is a parameter.

Function Body

The **Function Body** is the group of statements written inside curly braces `{}`. This is where the actual task is written.

Example:

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

Inside `{}` is the function body.

Return Value of a Function

When a function finishes its task, it can **return a value** to the calling code using the `return` keyword.

Example:

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

Here, `a + b` is returned when the function is called.

If the function is of type `void`, it won't return anything.

Function Prototyping

Function Prototyping is a way to tell the compiler about a function **before** its actual definition appears in the code. It includes:

- Return Type
- Function Name
- Parameters (if any)
- Semicolon **;** at the end

Purpose:

- It ensures that the function is declared before it is called.
- Helps the compiler check the correctness of function calls.

Example:

```
int addNumbers(int, int); // Function prototype


int main()
{
    // function call
}


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

If you don't use a function prototype, the function definition must be placed **above `main()`** in the program.

Summary Table

Part	Example	Description
Return Type	`int, float, void`	Type of value returned by the function

Part	Example	Description
Function Name	`addNumbers`	Name given to the function
Parameters	`(int a, int b)`	Data sent to the function
Function Body	`{ return a + b; }`	Code inside the function
Return Value	`return a + b;`	Final result returned by the function
Function Prototyping	`int addNumbers(int, int);`	Declaration before calling/defining

➤ Increment and Decrement Operators

These operators are used to **increase or decrease the value of a variable** by 1.

✅ Increment Operator (++)

Used to **add 1** to a variable.

```
int a = 5;
a++;  // Now a is 6
```

✅ Decrement Operator (--)

Used to **subtract 1** from a variable.

```
int b = 5;
b--;  // Now b is 4
```

Types of Increment/Decrement

There are **two forms** of each:

Operator	Name	Action
`i++`	Post-increment	Use `i`, then increment it
`++i`	Pre-increment	Increment `i`, then use it
`i--`	Post-decrement	Use `i`, then decrement it
`--i`	Pre-decrement	Decrement `i`, then use it

Example 1: Difference Between Post and Pre Increment

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


int main() {
    int a = 5;
    int b = a++;  // Post-increment: b = 5, a = 6

    int x = 5;
    int y = ++x;  // Pre-increment: x = 6, y = 6

    cout << "Post-increment: a = " << a << ", b = " << b << endl;
    cout << "Pre-increment: x = " << x << ", y = " << y << endl;

    return 0;
}
```

Output:

```
Post-increment: a = 6, b = 5
Pre-increment: x = 6, y = 6
```

Example 2: Using Decrement Operators

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

int main() {
    int a = 10;

    cout << "a-- = " << a-- << endl;  // prints 10, then a becomes 9
    cout << "--a = " << --a << endl;  // a becomes 8, then prints 8

    return 0;
}
```

Use in Loops

Increment and decrement are very commonly used in **loops**:

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

Loops in Programming

Loops are used to **repeat a block of code** multiple times until a certain condition is met.

✅ Types of Loops

1. *while Loop*

- Checks the condition **before** running the code.

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

2. *do-while Loop*

- Runs the code **at least once**, and then checks the condition.

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

3. *for Loop*

- Has initialization, condition, and update all in one line.

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

Assignments Using `while` Loop

---

1. Display Counting from 1 to 10 Using while Loop

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


int main() {
    int i = 1;
    while (i <= 10) {
        cout << i << " ";
        i++;
    }
    return 0;
}
```

---

2. Display 5 Numbers Using while Loop

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

int main() {
    int i = 1;
    while (i <= 5) {
        cout << i << " ";
        i++;
    }
    return 0;
}
```

---

3. Input Starting and Ending Point and Display Even Numbers in Range

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

int main() {
    int start, end;
    cout << "Enter starting point: ";
    cin >> start;
    cout << "Enter ending point: ";
    cin >> end;

    while (start <= end) {
        if (start % 2 == 0) {
            cout << start << " ";
        }
        start++;
    }
    return 0;
}
```

---

4. Input a Number and Display its Multiplication Table

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

```
int main() {
    int num, i = 1;
    cout << "Enter a number: ";
    cin >> num;

    while (i <= 10) {
        cout << num << " x " << i << " = " << num * i << endl;
        i++;
    }
    return 0;
}
```

6. **NumPy** – Helps do math with big lists of numbers.
7. **Pandas** – Helps organize and work with table-like data.
8. **Matplotlib** – Lets you draw charts and graphs.
9. **Seaborn** – Makes prettier graphs and charts.
10. **SciPy** – Helps with scientific math and calculations.
11. **Scikit-learn** – Lets you create simple machine learning models.
12. **TensorFlow** – Builds smart programs that learn (AI).
13. **Keras** – Makes deep learning easier to build and understand.
14. **PyTorch** – Another tool to build and train AI models.
15. **OpenCV** – Helps work with pictures and videos.
16. **NLTK** – Lets programs understand and use human language.
17. **spaCy** – Another tool for working with text and language.
18. **BeautifulSoup** – Pulls data out of web pages (HTML).
19. **Scrapy** – Helps collect data from websites.
20. **Requests** – Lets your program talk to websites easily.
21. **Flask** – Helps you make small websites or web apps.
22. **Django** – Helps you build full websites quickly.
23. **SQLAlchemy** – Connects your code with databases.
24. **Pygame** – Lets you make games with Python.
25. **Tkinter** – Helps you make simple windows and buttons (GUIs).
26. **PyQt** – Builds apps with windows and menus.
27. **Plotly** – Makes cool, interactive graphs.
28. **Dash** – Makes data dashboards (web apps with graphs).
29. **SymPy** – Does algebra and math with symbols like x and y.
30. **Statsmodels** – Used for statistics and finding patterns.
31. **NetworkX** – Helps draw and study networks (like social networks).
32. **Joblib** – Speeds up big programs and saves results.
33. **Pillow** – Lets you edit and change pictures.
34. **Pytest** – Helps test if your code works correctly.
35. **Bokeh** – Makes interactive charts for the web.

Temperature Conversion Table

Problem:

Display a table of equivalent temperatures from **50 to 100 Fahrenheit** in increments of **5**, converting them to **Celsius**.

Formula:

$$C = \frac{5}{9} \times (F - 32)$$

Example Program:

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


int main() {
    float celsius;


    cout << "Fahrenheit to Celsius Table" << endl;
    cout << "Fahrenheit\tCelsius" << endl;


    for (int f = 50; f <= 100; f += 5) {
        celsius = (5.0 / 9.0) * (f - 32);
        cout << f << "\t\t" << celsius << endl;
    }


    return 0;
}
```

Output:

```
Fahrenheit to Celsius Table
Fahrenheit      Celsius
50              10
55              12.7778
60              15.5556
...             ...
100             37.7778
```

Nested for Loops

A **nested loop** means having **one loop inside another**.

Syntax:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        // statements
    }
}
```

Example: Triangle Pattern

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

Program:

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


int main() {
    int n = 5;


    for (int i = 1; i <= n; i++) {  // outer loop for rows
        for (int j = 1; j <= i; j++) {  // inner loop for columns
            cout << "*";
        }
        cout << endl;
    }


    return 0;
}
```

Example: Diamond Pattern

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

```
  ***
   *
```

Program:

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


int main() {
    int n = 3;


    // Upper Half
    for (int i = 1; i <= n; i++) {
        for (int j = i; j < n; j++)
            cout << " ";
        for (int k = 1; k <= (2 * i - 1); k++)
            cout << "*";
        cout << endl;
    }


    // Lower Half
    for (int i = n - 1; i >= 1; i--) {
        for (int j = n; j > i; j--)
            cout << " ";
        for (int k = 1; k <= (2 * i - 1); k++)
            cout << "*";
        cout << endl;
    }


    return 0;
}
```

Output:

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

Summary Table

Concept	Use	Example
Simple `for` Loop	Run code repeatedly with increment or decrement	`for (int i = 0; i < 5; i++)`
Nested `for` Loop	One loop inside another (for patterns or tables)	See triangle and diamond examples
Temperature Conversion	Convert Fahrenheit to Celsius in a loop	`C = (5.0 / 9.0) * (F - 32)`

What are Pointers?

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

In simple words:

- Normal variables hold data values.
- Pointers hold the **address (location) of variables** in computer memory.

Example:

If `int a = 10;` is stored at memory address `2000`, a pointer can hold this address `2000`.

Why Use Pointers?

- To work directly with memory.
- To create **dynamic memory allocation**.
- To pass large data (like arrays) to functions efficiently.
- To create complex data structures like **linked lists, stacks, queues**.
- To manage **arrays and strings**.

Pointer Syntax and Declaration

Syntax:

```
data_type *pointer_name;
```

- `data_type` → Type of data the pointer will point to (like int, float, char)
- `*` → Asterisk symbol to declare a pointer
- `pointer_name` → Name of the pointer variable

Example:

```
int *p;
float *q;
char *ch;
```

Here:

- `p` is a pointer to an integer.
- `q` is a pointer to a float.
- `ch` is a pointer to a char.

How to Use Pointers

There are two important operators for pointers:

Operator	Name	Purpose
&	Address of Operator	Gives the address of a variable.
*	Value at Address (Dereference)	Gives the value stored at a memory address pointed by the pointer.

Example Program:

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


int main() {
    int a = 10;
    int *p;    // declaring pointer
    p = &a;    // storing address of a in pointer p


    cout << "Value of a: " << a << endl;
    cout << "Address of a: " << &a << endl;
    cout << "Value of p (address of a): " << p << endl;
    cout << "Value at address p points to: " << *p << endl;


    return 0;
}
```

Output:

```
Value of a: 10
Address of a: 0x61ff08  (this will vary on your system)
Value of p (address of a): 0x61ff08
Value at address p points to: 10
```

Summary Table

Concept	Example	Description

Concept	Example	Description
Pointer Declaration	`int *p;`	Declares a pointer to an int
Assigning Address to Pointer	`p = &a;`	Pointer p holds the address of variable a
Address of Operator (&)	`&a`	Returns the address of a
Dereferencing Operator (*)	`*p`	Returns the value stored at address p

What are Structures?

A **Structure** in C++ is a **user-defined data type** that allows you to combine **different types of variables** under a single name.

In simple words: It’s like a **custom-made data type** where you can group related information of different data types.

Example:

If you want to store information about a student (name, roll number, marks):

- `name → string`
- `roll number → int`
- `marks → float`

You can group these together using a structure.

Why Use Structures?

- To group related data items.
- To manage complex data (like student records, employee details, etc.)
- Easier to pass grouped data to functions.
- Cleaner and more organized code.

Defining a Structure with `struct` Keyword

Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
    // more members...
};
```

Example:

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

Explanation:

- `Student` is the name of the structure.
- `rollNo`, `name`, `marks` are its members (variables of different types).

Declaration and Nexus (Using Structure Variables)

After defining a structure, you can declare variables of that structure type.

Syntax:

```
structure_name variable_name;
```

Example:

```
Student s1, s2;
```

Accessing Structure Members

Use the **dot (.) operator** to access members.

Example:

```
s1.rollNo = 101;
s1.name = "Ali";
s1.marks = 89.5;

cout << "Name: " << s1.name;
```

Structure and Functions

You can pass structure variables to functions in two ways:

36. **By Value**
37. **By Reference**

Example: Passing Structure to Function

Define a Function That Takes a Structure:

```
void display(Student s) {
    cout << "Roll No: " << s.rollNo << endl;
    cout << "Name: " << s.name << endl;
    cout << "Marks: " << s.marks << endl;
}
```

Call the Function:

```
Student s1;
s1.rollNo = 101;
s1.name = "Ali";
s1.marks = 89.5;

display(s1);
```

Example: Passing by Reference (More Efficient)

```
void display(Student &s) {
    cout << "Roll No: " << s.rollNo << endl;
    cout << "Name: " << s.name << endl;
    cout << "Marks: " << s.marks << endl;
}
```

Note: `&s` means reference — so the function works on the original structure, not a copy.

Complete Program Example

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

struct Student {
    int rollNo;
    string name;
    float marks;
};

void display(Student s) {
    cout << "Roll No: " << s.rollNo << endl;
    cout << "Name: " << s.name << endl;
    cout << "Marks: " << s.marks << endl;
}
```

```
int main() {
    Student s1;

    s1.rollNo = 101;
    s1.name = "Ali";
    s1.marks = 89.5;

    display(s1);

    return 0;
}
```

Summary Table

Concept	Example	Description
Structure Definition	`struct Student { int rollNo; };`	Defines a custom data type
Structure Variable Declaration	`Student s1, s2;`	Declares structure variables
Access Structure Members	`s1.rollNo = 101;`	Uses dot operator to access members
Pass Structure to Function	`display(s1);`	Passes structure variable to a function
Pass by Reference	`void display(Student &s)`	Passes reference to avoid copying data

What is a Counter in a Loop?

A **counter** is a variable we use inside a loop to:

- **Count how many times** the loop runs
- **Keep track of numbers** while the loop runs

Usually, it increases or decreases on each loop cycle.

Using Counter Variable in a `while` Loop

Example:

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


int main() {
    int counter = 1;


    while (counter <= 5) {
        cout << "Counter is: " << counter << endl;
        counter++;  // Increase counter by 1
    }


    return 0;
}
```

Assignments (Programs)

✓ 1. Print Sum of Odd Numbers Between 1 and 100 Using `for` Loop

Odd numbers = 1, 3, 5, 7, ..., 99

Program:

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


int main() {
    int sum = 0;

    for (int i = 1; i <= 100; i += 2) {
        sum += i;  // Add odd number to sum
    }

    cout << "Sum of odd numbers between 1 and 100 is: " << sum << endl;

    return 0;
}
```

✅ 2. Display Product of All Odd Numbers Between 1 and 10 Using `for` Loop

> Odd numbers = 1, 3, 5, 7, 9

Program:

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

int main() {
    int product = 1;

    for (int i = 1; i <= 10; i += 2) {
        product *= i;  // Multiply odd numbers
    }

    cout << "Product of odd numbers between 1 and 10 is: " << product << endl;

    return 0;
}
```

✅ 3. Input an Integer and Display its Multiplication Table in Descending Order

> Example: If user enters 5 Print: 5 × 10 = 50, 5 × 9 = 45, ..., 5 × 1 = 5

Program:

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

```
int main() {
    int num;

    cout << "Enter a number: ";
    cin >> num;

    for (int i = 10; i >= 1; i--) {
        cout << num << " x " << i << " = " << num * i << endl;
    }

    return 0;
}
```

Summary Table

Task	Loop Type	Counter Usage
Count iterations	`while`	`counter++` inside loop
Sum of odd numbers 1–100	`for`	Increase by 2, `sum += i`
Product of odd numbers 1–10	`for`	Increase by 2, `product *= i`
Multiplication table in descending order	`for`	Decrease by 1, from 10 to 1

What is an Array?

An **array** is a collection of variables **of the same data type** stored together under a **single name**. It is used to store **multiple values** in a single variable instead of creating separate variables for each value.

Example: If you want to store the marks of 5 students:

- Without array: `int m1, m2, m3, m4, m5;`
- With array: `int marks[5];`

Why Use Arrays?

- To store multiple values of the same type.
- To reduce the number of variables.
- To easily access and manipulate large sets of data using loops.
- To make code organized and clean.

Types of Arrays in C++

38. **Single-Dimensional Array**
39. **Two-Dimensional Array**
40. (Multi-Dimensional Arrays — advanced, optional)

Single-Dimensional Array

A **Single-Dimensional Array** is a list of elements stored in a single row (like a list).

Declaration:

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

Example:

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

This creates an integer array of size 5 (can store 5 integers).

Initialization:

You can assign values at the time of declaration:

```
int numbers[5] = {10, 20, 30, 40, 50};
```

Or assign values individually:

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

Note: Array indexing starts from **0**.

Accessing Elements:

To access array elements:

```
cout << numbers[0]; // Displays 10
```

Example Program:

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


int main() {
    int marks[3] = {90, 85, 78};


    for (int i = 0; i < 3; i++) {
        cout << "Mark " << i+1 << ": " << marks[i] << endl;
    }


    return 0;
}
```

Two-Dimensional Array

A **Two-Dimensional Array** is like a table with **rows and columns** (like a matrix).

Declaration:

```
data_type array_name[row_size][column_size];
```

Example:

```
int marks[2][3];
```

This creates a 2D array with 2 rows and 3 columns.

Initialization:

You can initialize it like this:

```
int marks[2][3] = {
    {90, 85, 78},
    {88, 92, 80}
};
```

Accessing Elements:

To access elements:

```
cout << marks[0][1]; // Displays 85
```

Note:

- `marks[0][0]` → first row, first column
- `marks[0][1]` → first row, second column

Example Program:

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


int main() {
    int marks[2][3] = {
        {90, 85, 78},
        {88, 92, 80}
    };


    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << "marks[" << i << "][" << j << "] = " << marks[i][j] <<
endl;
        }
    }
    return 0;
}
```

Differences Between Single & Two-Dimensional Arrays

Single-Dimensional Array	Two-Dimensional Array
Stores values in a single list	Stores values in a table (rows & columns)
Accessed using one index	Accessed using two indexes
Example: `int a[5];`	Example: `int a[2][3];`

Summary Table

Concept	Syntax Example	Description
Declare 1D array	`int a[5];`	A list of 5 integers
Initialize 1D array	`int a[3] = {1, 2, 3};`	Assign values during declaration
Access 1D element	`a[0]`	Access first element
Declare 2D array	`int a[2][3];`	2 rows, 3 columns array
Initialize 2D array	`int a[2][2] = {{1,2},{3,4}};`	Set values in a table format
Access 2D element	`a[1][0]`	Access first element of second row

What is File Handling?

File handling means **reading data from files** and **writing data to files** using a C++ program.

It allows us to **store data permanently** (unlike variables that lose data when the program ends).

Why Use File Handling?

- To save output/results for future use.
- To read and process large data from files.
- To store records in files (like student records, marksheets, etc.)

C++ File Handling Classes

C++ provides three main classes for file handling (from the `<fstream>` header file):

Class	Purpose
`ifstream`	To **read** data from files
`ofstream`	To **write** data to files
`fstream`	To **read and write** both

Types of File Handling

1 Text File Handling

Text files store data in a **readable format (like .txt files)**.

Text File Operations

Operation	Function
Open File	`open()`
Write to File	`<<`

Operation	Function
Read from File	>> or getline()
Close File	close()

Example: Writing to a Text File

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


int main() {
    ofstream myFile("example.txt"); // Create and open a file


    myFile << "Hello, this is a text file."; // Write to the file


    myFile.close(); // Close the file


    return 0;
}
```

Example: Reading from a Text File

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


int main() {
    string line;
    ifstream myFile("example.txt"); // Open the file for reading


    while (getline(myFile, line)) {
        cout << line << endl; // Display each line
    }


    myFile.close(); // Close the file


    return 0;
}
```

2 Binary File Handling

Binary files store data in **binary (machine) format**. It's **faster and more secure** than text files.

Used when storing structured data like images, videos, or custom data records.

Binary File Operations

Operation	*Function*
Open File	`open()`
Write to File	`write()`
Read from File	`read()`
Close File	`close()`

Note: `write()` and `read()` work with memory addresses.

Example: Writing to a Binary File

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


int main() {
    ofstream myFile("data.dat", ios::binary); // Open binary file for writing


    int num = 100;
    myFile.write((char*)&num, sizeof(num)); // Write integer to file


    myFile.close(); // Close file
    return 0;
}
```

Example: Reading from a Binary File

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main() {
    ifstream myFile("data.dat", ios::binary); // Open binary file for reading

    int num;
    myFile.read((char*)&num, sizeof(num)); // Read integer from file

    cout << "Value from file: " << num << endl;

    myFile.close(); // Close file
    return 0;
}
```

File Opening Modes

When opening files, you can specify modes:

Mode	Meaning
`ios::in`	Open for reading
`ios::out`	Open for writing (overwrite if exists)
`ios::app`	Open for appending at the end
`ios::binary`	Open in binary mode

Summary Table

Operation	Text File Example	Binary File Example
Open for writing	`ofstream f("file.txt");`	`ofstream f("file.dat", ios::binary);`
Write data	`f << "Hello";`	`f.write((char*)&num, sizeof(num));`
Open for reading	`ifstream f("file.txt");`	`ifstream f("file.dat", ios::binary);`

Operation	*Text File Example*	*Binary File Example*
Read data	`getline(f, line);` *or* `f >> x;`	`f.read((char*)&num, sizeof(num));`
Close file	`f.close();`	`f.close();`