Module #3 Introduction to OOPS Programming

1.What are the key differences between Procedural Programming and Object Oriented Programming (OOP)?

-: Here are the key differences between Procedural Programming and Object-Oriented Programming (OOP):

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Top-down approach	Bottom-up approach
Focus	Focuses on functions or procedures	Focuses on objects (data + methods)
Data Handling	Data is exposed and can be accessed by any function	Data is encapsulated within objects and accessed through methods
Modularity	Program is divided into functions	Program is divided into objects and classes
Reusability	Limited code reusability	Promotes reusability through inheritance and polymorphism
Encapsulation	No concept of encapsulation	Supports encapsulation (hiding internal data)
Inheritance	Does not support inheritance	Supports inheritance (code sharing via parent-child relationships)
Polymorphism	No polymorphism	Supports polymorphism (same interface, different behavior)
Security	Less secure (data is globally accessible)	More secure (access modifiers like private, public)
Example Languages	C, Pascal, Fortran	C++, Java, Python, C#

2. List and explain the main advantages of OOP over POP.

-:Here are the main advantages of Object-Oriented Programming (OOP) over Procedural-Oriented Programming (POP), clearly explained:

---

1. Encapsulation (Data Hiding)

- OOP: Bundles data and functions into a single unit (class), restricting direct access to data.
- Advantage: Increases security by protecting internal object state and reducing unintended interference.
- POP: Data is often global and can be accessed/modified from any function, which is less secure.

---

2. Modularity

- OOP: Code is organized into reusable and independent classes.
- Advantage: Easier to debug, manage, and update individual parts of the code.
- POP: Code is divided into functions, but it's harder to isolate and reuse components cleanly.

---

3. Reusability via Inheritance

- OOP: Supports inheritance, allowing a class to inherit properties and behavior from another.
- Advantage: Reduces code duplication and speeds up development.
- POP: No inheritance; code must be rewritten or copied for reuse.

---

4. Polymorphism

- OOP: Supports polymorphism (e.g., method overloading and overriding).
- Advantage: The same function can work in different ways for different classes, making code flexible and extensible.
- POP: Functions are fixed in behavior and cannot be reused polymorphically.

---

5. Abstraction

- OOP: Allows hiding of complex implementation details and shows only the necessary features.
- Advantage: Simplifies code usage and makes interfaces cleaner.
- POP: Focuses on how tasks are performed, not what is being done, exposing more complexity to the user.

---

6. Improved Collaboration

- OOP: Classes and objects mimic real-world entities, making design more intuitive and easier to understand.
- Advantage: Helpful in team projects, especially for large-scale systems.
- POP: Logic and flow are more difficult to visualize in terms of real-world mapping.

---

7. Better Maintainability and Scalability

- OOP: Due to modular structure, updates or changes in code are easier and less risky.
- POP: Changes in one part can affect many others, increasing maintenance effort.

3. Explain the steps involved in setting up a C++ development environment.

-:Setting up a **C++ development environment** involves a few key steps to install the necessary tools and configure everything to write, compile, and run C++ programs efficiently.

Step-by-Step Guide to Set Up C++ Development Environment

Step 1: Install a C++ Compiler

A C++ compiler is required to convert your code into an executable. Common compilers:

- **Windows:** MinGW (GCC), MSVC (Microsoft Visual C++)
- **Mac:** Xcode Command Line Tools (includes clang)
- **Linux:** g++ (part of the GNU Compiler Collection)

For Windows (MinGW):

1. Download MinGW from https://www.mingw-w64.org.
2. Install it and **add the bin folder path to your System Environment Variables** (so you can use g++ in the terminal).
    - Example path: C:\Program Files\mingw-w64\bin

Step 2: Install a Text Editor or IDE

You need a good environment to write and manage your C++ code.

Popular options:

- **VS Code** (lightweight, powerful) — Recommended
- **Code::Blocks** (simple and built-in compiler)
- **Dev C++** (outdated but simple for beginners)
- **Visual Studio** (Microsoft's full-featured IDE)
- **CLion** (JetBrains' powerful IDE, paid)

Step 3: Set Up the Editor or IDE

For VS Code:

1. Download and install from https://code.visualstudio.com
2. Install extensions:
    - C/C++ by Microsoft
    - Code Runner (optional for easy execution)
3. Open a folder and create a .cpp file.
4. Configure tasks (tasks.json) and launch.json for build and run (if not using Code Runner).

Step 4: Write a Simple C++ Program

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

int main() {
    cout << "Hello, World!";
    return 0;
}
```

Save as hello.cpp.

Step 5: Compile and Run the Code

Using Command Line (any OS)

g++ hello.cpp -o hello

./hello # or hello.exe on Windows

Step 6: (Optional) Debug Setup

- Most IDEs support debugging with breakpoints, variable watch, etc.
- For VS Code: Use the **Run and Debug** tab to start debugging, configure with launch.json.

4.What are the main input/output operations in C++? Provide examples

In **C++**, input and output operations are mainly performed using **cin** (for input) and **cout** (for output), which are part of the **iostream** library.

Main Input/Output Operations in C++

1. Output using cout

- Used to **display output** to the screen.
- Belongs to the **std** namespace.

Syntax:

cout << expression;

Example:

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

int main() {
    cout << "Hello, World!" << endl;
    cout << "The answer is: " << 42 << endl;
    return 0;
}
```

endl is used to move to a new line (like \n).

---

2. Input using cin

- Used to **take input** from the user.
- Also belongs to the **std** namespace.

Syntax:

```
cin >> variable;
```

Example:

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

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You entered: " << age << endl;
    return 0;
}
```

Multiple Inputs/Outputs

You can chain multiple variables:

Example:

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

int main() {
    string name;
    int age;

    cout << "Enter your name and age: ";
    cin >> name >> age;

    cout << "Hello, " << name << "! You are " << age << " years old." << endl;
```

```
    return 0;
}
```

5. What are the different data types available in C++? Explain with examples.

-: In C++, **data types** specify the type of data that a variable can hold. They are mainly categorized into:

---

1. Basic (Built-in) Data Types

Data Type	Description	Example
int	Integer numbers	int age = 25;
float	Floating point (decimal)	float pi = 3.14;
double	Double-precision float	double g = 9.81;
char	Single character	char grade = 'A';
bool	Boolean value (true/false)	bool isOn = true;

---

2. Derived Data Types

Data Type	Description	Example
array	Collection of elements of same type	int marks[5];
pointer	Stores memory address of another variable	int *ptr;
function	Represents functions returning values	int add(int a, int b);
reference	Alias for another variable	int& ref = x;

---

3. User-defined Data Types

Data Type	Description	Example
struct	Groups different types together	struct Student { int id; char name[50]; };
union	Like struct, but shares memory	union Data { int i; float f; };
enum	User-defined constants	enum Color { RED, GREEN, BLUE };
class	Blueprint for objects (OOP)	class Car { public: int speed; };

Data Type	Description	Example
typedef	Alias for data types	typedef int Marks;

---

4. Void Type

- void: Represents absence of type.
    - Used in functions that return nothing.
    - Example: void printMessage();

Example Program

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

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

int main() {
    int age = 20;
    float weight = 55.5;
    char grade = 'A';
    bool passed = true;
    Student s1 = {101, "Khushi"};

    cout << "Age: " << age << endl;
    cout << "Weight: " << weight << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << passed << endl;
    cout << "Student ID: " << s1.id << ", Name: " << s1.name << endl;
```

```
    return 0;
}
```

6.Explain the difference between implicit and explicit type conversion in C++.

-: In C++, type conversion refers to changing a variable from one data type to another. There are two main types:

---

1. Implicit Type Conversion (Type Promotion)

Also known as automatic type conversion, it is done automatically by the compiler when:

- Two different data types are used in an expression.
- No data is lost in the conversion (usually from lower to higher data type).

Example:

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

What happens:

- a (int) is converted to float → 10.0
- Result: float + float = float

---

2. Explicit Type Conversion (Type Casting)

Also known as manual type conversion, where the programmer explicitly specifies the type to convert to using type casting.

Syntax:

```
(type) expression
```

Example:

```
float num = 5.75;
int val = (int)num;  // explicit conversion from float to int
```

- val will be 5 (decimal part is truncated)

Comparison Table

Feature	Implicit Conversion	Explicit Conversion
Performed by	Compiler	Programmer
Syntax	Automatic	(type) before variable
Control	Less control	Full control
Risk of data loss	Low (upcasting)	Possible (downcasting)
Example	int + float → float	(int)5.75 → 5

Example Program Showing Both

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

int main() {
    int a = 10;
    float b = 3.5;

    // Implicit conversion
    float sum = a + b;  // a is implicitly converted to float

    // Explicit conversion
    int truncated = (int)b;  // decimal part of b is removed

    cout << "Implicit conversion result (sum): " << sum << endl;
    cout << "Explicit conversion result (truncated): " << truncated << endl;

    return 0;
}
```

7. What are the different types of operators in C++? Provide examples of each.

-: In C++, operators are special symbols used to perform operations on variables and values. They are categorized based on their functionality.

Types of Operators in C++ with Examples

1. Arithmetic Operators

Used to perform basic mathematical operations.

Operator	Description	Example (a = 10, b = 3)	Result
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3
%	Modulus (remainder)	a % b	1

2. Relational (Comparison) Operators

Used to compare two values.

Operator	Description	Example (a = 10, b = 3)	Result
==	Equal to	a == b	false
!=	Not equal to	a != b	true
>	Greater than	a > b	true
<	Less than	a < b	false
>=	Greater than or equal to	a >= b	true
<=	Less than or equal to	a <= b	false

3. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example (a=10, b=3)	Result
&&	Logical AND	(a > 5 && b < 5)	true
`	`		Logical OR
!	Logical NOT	!(a == b)	true

---

4. Assignment Operators

Used to assign values to variables.

Operator	Example	Equivalent To
=	a = 5	a = 5
+=	a += 2	a = a + 2
-=	a -= 3	a = a - 3
*=	a *= 4	a = a * 4
/=	a /= 2	a = a / 2
%=	a %= 2	a = a % 2

---

5. Increment and Decrement Operators

Operator	Description	Example (a = 5)	Result
++a	Pre-increment	++a	6
a++	Post-increment	a++	Returns 5, then a=6
--a	Pre-decrement	--a	4
a--	Post-decrement	a--	Returns 5, then a=4

---

6. Bitwise Operators

Used for bit-level operations.

Operator	Description	Example (a=5, b=3)	Binary	Result
&	Bitwise AND	a & b	0101 & 0011 = 0001	1
`	`	Bitwise OR	`a	b`
^	Bitwise XOR	a ^ b	0101 ^ 0011 = 0110	6
~	Bitwise NOT	~a	~0101	Depends on size
<<	Left Shift	a << 1	1010	10
>>	Right Shift	a >> 1	0010	2

---

7. Conditional (Ternary) Operator

Short-hand for if-else.

condition ? expr_if_true : expr_if_false;

Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
```

---

8. Sizeof Operator

Used to determine the size (in bytes) of a data type or variable.

```
int x = 5;
cout << sizeof(x);  // Outputs 4 on most systems
```

---

9. Scope Resolution Operator ::

Used to define or access a global variable when there is a local variable with the same name.

```
int x = 10;
int main() {
    int x = 20;
    cout << ::x; // prints global x i.e., 10
}
```

8. Explain the purpose and use of constants and literals in C++.

-: In C++, constants and literals are used to represent fixed values that do not change during the execution of a program.

---

1. Constants in C++

Constants are variables whose value is fixed (read-only) once declared.

Purpose:

- Prevent accidental value changes.
- Improve code safety and readability.
- Make programs easier to maintain.

Ways to Declare Constants

a) const Keyword

```
const float PI = 3.14;
PI = 3.1415; // ❌ Error: cannot modify a const variable
```

b) #define Preprocessor Directive

```
#define MAX_SIZE 100
cout << MAX_SIZE; // Outputs: 100
```

#define doesn't do type checking or scope limiting, so const is generally preferred in C++.

Example:

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

const float GRAVITY = 9.8;

int main() {
    float mass = 50;
    float weight = mass * GRAVITY;  // GRAVITY is a constant
    cout << "Weight is " << weight << " N";
    return 0;
}
```

---

2. Literals in C++

Literals are fixed values used directly in the code.

Types of Literals with Examples:

Literal Type	Example	Description
Integer	10, -42	Whole numbers
Floating-point	3.14, -0.005	Decimal numbers

Literal Type	Example	Description
Character	'A', '5'	Enclosed in single quotes
String	"Hello"	Enclosed in double quotes
Boolean	true, false	Used in logical conditions
Octal	012	Starts with 0 (octal for 10)
Hexadecimal	0x1A	Starts with 0x (hex for 26)

Example with Literals:

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

int main() {
    int age = 25;              // 25 is an integer literal
    char grade = 'A';          // 'A' is a character literal
    float pi = 3.14;           // 3.14 is a float literal
    string name = "Khushi";    // "Khushi" is a string literal
    bool passed = true;        // true is a boolean literal

    cout << name << " scored grade " << grade << " at age " << age;
    return 0;
}
```

9. What are conditional statements in C++? Explain the if-else and switch statements.

-:In C++, conditional statements are used to make decisions in your program based on whether a condition is true or false.

They allow your program to choose different paths of execution.

Types of Conditional Statements:

1. if statement

2. if-else statement
3. else if ladder
4. switch statement

---

1. if Statement

Used to execute a block of code only if a condition is true.

Syntax:

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

Example:

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

---

2. if-else Statement

Used to execute one block if the condition is true, and another if false.

Syntax:

```
if (condition) {
    // code if condition is true
} else {
    // code if condition is false
}
```

Example:

```
int age = 16;
if (age >= 18) {
    cout << "You can vote.";
} else {
    cout << "You cannot vote.";
```

```
}
```

---

3. else if Ladder

Used to check multiple conditions in sequence.

Example:

```
int marks = 85;

if (marks >= 90) {
    cout << "Grade A";
} else if (marks >= 75) {
    cout << "Grade B";
} else if (marks >= 60) {
    cout << "Grade C";
} else {
    cout << "Fail";
}
```

---

4. switch Statement

Used when you have multiple constant values to compare a variable against.

Syntax:

```
switch (expression) {
    case constant1:
        // code
        break;
    case constant2:
        // code
        break;
    ...
    default:
        // code
```

```
}
```

break is important to stop further execution of cases
default is optional (used when no case matches)

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";
}
```

10. What is the difference between for, while, and do-while loops in C++?

-: In C++, for, while, and do-while loops are used to execute a block of code repeatedly based on a condition. The key differences lie in syntax, control flow, and when the condition is checked.

---

1. for Loop

Purpose:

Used when the number of iterations is known in advance.

Syntax:

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

}

Example:

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

Output:
1 2 3 4 5

---

2. while Loop

Purpose:

Used when the number of iterations is unknown, and you want to check the condition before the loop runs.

Syntax:

```
while (condition) {
    // loop body
}
```

Example:

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

Output:
1 2 3 4 5

---

3. do-while Loop

Purpose:

Used when the loop must run at least once, even if the condition is false initially.

Syntax:

```
do {
    // loop body
```

```
} while (condition);
```

Example:

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

Output:
1 2 3 4 5

Comparison Table

Feature	for loop	while loop	do-while loop
Condition check	Before loop body	Before loop body	After loop body
Runs at least once?	No	No	Yes
Use case	Known number of iterations	Unknown iterations (pre-check)	Unknown iterations (post-check)
Syntax compactness	Most compact (all in one line)	Needs manual setup	Similar to while + ensures 1 run

---

Example Showing All Together

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

int main() {
    cout << "Using for loop:\n";
    for (int i = 1; i <= 3; i++) {
        cout << i << " ";
    }

    cout << "\nUsing while loop:\n";
```

```
    int j = 1;
    while (j <= 3) {
        cout << j << " ";
        j++;
    }

    cout << "\nUsing do-while loop:\n";
    int k = 1;
    do {
        cout << k << " ";
        k++;
    } while (k <= 3);

    return 0;
}
```

11. How are break and continue statements used in loops? Provide examples.

-: In C++, break and continue statements are used to control the flow of loops (for, while, do-while).

---

1. break Statement

Purpose:

The break statement immediately exits the loop, even if the condition has not become false.

Syntax:

```
for (int i = 0; i < 10; i++) {
    if (i == 5)
        break; // exits the loop
}
```

Example:

```
#include <iostream>
```

```
using namespace std;


int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 6)
            break; // exit loop when i is 6
        cout << i << " ";
    }
    return 0;
}
```

Output:
1 2 3 4 5

---

2. continue Statement

Purpose:

The continue statement skips the current iteration and moves to the next iteration of the loop.

Syntax:

```
for (int i = 0; i < 10; i++) {
    if (i == 5)
        continue; // skip this iteration
}
```

Example:

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


int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3)
            continue; // skip when i is 3
        cout << i << " ";
```

```
    }
    return 0;
}
```

Output:
1 2 4 5

Summary Table

Statement	Behavior	Use Case Example
break	Exits the loop immediately	Stop searching when item is found
continue	Skips current iteration, continues	Skip processing invalid data

---

Bonus: Example with while Loop

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

int main() {
    int i = 0;
    while (i < 6) {
        i++;
        if (i == 4)
            continue;
        if (i == 6)
            break;
        cout << i << " ";
    }
    return 0;
}
```

Output:
1 2 3 5

12. Explain nested control structures with an example.

-: Nested Control Structures in C++

Nested control structures occur when you place one control structure **(like if, for, while, etc.) inside another**. They allow you to handle **more complex decision-making or looping** situations.

- **Types of Nesting:**
  - if inside if
  - loop inside another loop
  - loop inside if, and vice versa

Example 1: Nested if Statements

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

int main() {
    int age = 20;
    char gender = 'F';

    if (age >= 18) {
        if (gender == 'F') {
            cout << "Eligible woman voter";
        } else {
            cout << "Eligible man voter";
        }
    } else {
        cout << "Not eligible to vote";
    }

    return 0;
}
```

Explanation:

- Outer if: checks if age is 18 or more.
- Inner if: checks if the person is female or male.

Example 2: Nested Loops

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

int main() {
    for (int i = 1; i <= 3; i++) {          // outer loop
        for (int j = 1; j <= 2; j++) {      // inner loop
            cout << "i = " << i << ", j = " << j << endl;
        }
    }
    return 0;
}
```

Output:

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```

13. What is a function in C++? Explain the concept of function declaration, definition, and calling.

-: A function in C++ is a block of reusable code that performs a specific task. It helps in:

- Modular programming (breaking code into smaller pieces)
- Code reuse (write once, use many times)
- Improved readability and maintenance

Types of Functions

1. Library/Predefined Functions – e.g., cout, sqrt(), strlen()
2. User-defined Functions – functions created by the programmer

---

Function Components in C++

To use a user-defined function, you need:

1. Function Declaration (Prototype)

Tells the compiler about the function name, return type, and parameters.

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

2. Function Definition

Contains the actual code (body) of the function.

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

3. Function Call

Executes the function when needed.

```
int result = add(5, 3);  // Calling the function
```

Full Example:

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

// 1. Function Declaration
int add(int, int);

int main() {
    // 3. Function Call
    int sum = add(10, 20);
    cout << "Sum = " << sum;
    return 0;
```

```
}

// 2. Function Definition
int add(int a, int b) {
    return a + b;
}
```

Output:
Sum = 30

Function Syntax Breakdown

```
return_type function_name(parameter_list) {
    // body of the function
}
```

Example:

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

14. What is the scope of variables in C++? Differentiate between local and global scope.

-: What is the Scope of Variables in C++?

In C++, the scope of a variable refers to the portion of the program where the variable can be accessed or used.

Types of Scope

Scope Type	Description
Local	Accessible only inside the block or function where declared
Global	Declared outside all functions and accessible throughout the program
Function/Parameter	Variables passed as arguments to a function – available only inside that function

Scope Type	Description
Class/Member	For object-oriented C++, class members have class-level scope

---

1. Local Scope

- Declared inside a function, loop, or block ({ })
- Exists only during the execution of that block
- Cannot be accessed from outside

Example:

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

void show() {
    int x = 10;  // local variable
    cout << "Inside function: " << x << endl;
}
int main() {
    // cout << x;   Error: x is not accessible here
    show();
    return 0;
}
```

---

2. Global Scope

- Declared outside all functions
- Can be accessed from any function in the file

Example:

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

int x = 100;  // global variable
```

```
void display() {
    cout << "Inside display(): " << x << endl;
}

int main() {
    cout << "Inside main(): " << x << endl;
    display();
    return 0;
}
```

Output:

Inside main(): 100

Inside display(): 100

15.Explain recursion in C++ with an example

-: Recursion is a programming technique where a function calls itself in order to solve a problem.

- It is used when a problem can be broken down into smaller sub-problems of the same type.
- Every recursive function must have:
    1. A base case to stop recursion.
    2. A recursive case to call itself.

General Syntax

```
return_type function_name(parameters) {
    if (base_condition)
        return result;
    else
        return function_name(smaller_problem);
}
```

Example: Factorial Using Recursion

The factorial of a number n is:
n! = n × (n - 1) × (n - 2) × ... × 1

Recursive Formula:

factorial(n) = n × factorial(n - 1)
with base case: factorial(0) = 1

C++ Code Example:

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

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0)        // base case
        return 1;
    else
        return n * factorial(n - 1);  // recursive call
}

int main() {
    int number = 5;
    cout << "Factorial of " << number << " is " << factorial(number);
    return 0;
}
```

Output:

Factorial of 5 is 120

How it Works (Step by Step for factorial(5)):

factorial(5)

= 5 * factorial(4)

= 5 * 4 * factorial(3)

= 5 * 4 * 3 * factorial(2)

= 5 * 4 * 3 * 2 * factorial(1)

= 5 * 4 * 3 * 2 * 1 * factorial(0)

= 5 * 4 * 3 * 2 * 1 * 1

= 120

Advantages of Recursion

- Simplifies code for problems like factorial, Fibonacci, tree traversal, etc.
- Makes code more elegant and readable.

Disadvantages

- Uses more memory (function call stack).
- Risk of stack overflow if base case is missing or wrong.

16. What are function prototypes in C++? Why are they used?

-: A function prototype is a declaration of a function that tells the compiler:

- The function's name
- Return type
- Parameter types

But without defining the body of the function.

Syntax of Function Prototype:

return_type function_name(parameter_list);

Example:

int add(int, int); // function prototype

Why Are Function Prototypes Used?

Reason	Explanation
Tells the compiler	Helps the compiler understand the function before it is used (e.g., in main()).
Allows flexible ordering	You can call a function before its definition in the code.

Reason	Explanation
Enables type checking	Ensures that correct argument types are passed during function calls.
Improves readability	Gives a quick overview of all functions in a program.

Example with Function Prototype

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

//  Function Prototype
int multiply(int a, int b);

int main() {
    int result = multiply(4, 5);  // function called before definition
    cout << "Product = " << result;
    return 0;
}

//  Function Definition
int multiply(int x, int y) {
    return x * y;
}
```

Output:
Product = 20

17. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

-: An array is a collection of elements of the same data type stored in contiguous memory locations.
Each element can be accessed using an index.

Why Use Arrays?

- To store multiple values of the same type in one variable
- Easy to manage collections like marks, scores, items, etc.
- Indexed access (fast lookup)

---

1. Single-Dimensional Array

Definition:

An array with one row of elements.

Syntax:

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

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
cout << numbers[2];  // Outputs: 30
```

- Index starts from 0
- numbers[0] = 10, numbers[4] = 50

---

2. Multi-Dimensional Array

Definition:

An array with more than one dimension, commonly 2D (like a table or matrix).

Syntax:

```
data_type array_name[row][column];
```

Example (2D array):

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
cout << matrix[1][2];  // Outputs: 6
```

- matrix[0][0] = 1
- matrix[1][2] = 6

Comparison Table

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	List or line	Table, matrix (2D), or cube (3D)
Declaration	int a[5];	int a[2][3];
Access Syntax	a[i]	a[i][j]
Use Case Example	Storing marks of 5 students	Storing marks for 3 students in 2 subjects

---

Example Program Using Both

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

int main() {
    // Single-dimensional
    int marks[3] = {85, 90, 78};
    cout << "Mark 1: " << marks[0] << endl;

    // Two-dimensional
    int table[2][2] = {{1, 2}, {3, 4}};
    cout << "Element at [1][1]: " << table[1][1] << endl;

    return 0;
}
```

Output:

less

Mark 1: 85

Element at [1][1]: 4

18.Explain string handling in C++ with examples.

-: String Handling in C++

In C++, strings are used to store and manipulate sequences of characters. C++ provides two ways to handle strings:

---

1. C-style Strings (Character Arrays)
   - Based on arrays of char
   - Terminated with a null character '\0'

Declaration:

```
char name[10] = "Alice";
```

Example:

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

int main() {
    char name[20] = "Khushi";
    cout << "Hello " << name << "!" << endl;
    return 0;
}
```

Common Functions for C-strings (from <cstring>):

Function	Purpose
strlen(s)	Length of the string
strcpy(d, s)	Copy string s to d
strcat(d, s)	Concatenate s to d
strcmp(s1, s2)	Compare two strings

Example with C-string functions:

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

int main() {
```

```
    char str1[20] = "Hello";
    char str2[20] = "World";

    strcat(str1, str2);  // str1 becomes "HelloWorld"
    cout << str1 << endl;

    cout << "Length: " << strlen(str1) << endl;
    return 0;
}
```

---

2. C++ string Class (from <string> library)

- Easier and safer than character arrays
- Comes with many built-in functions

Declaration:

```
#include <string>
string name = "Khushi";
```

Example:

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

int main() {
    string first = "Hello";
    string last = "World";
    string full = first + " " + last;

    cout << full << endl;
    cout << "Length: " << full.length() << endl;
    return 0;
}
```

Useful Functions of string Class

Function	Description
length() / size()	Returns length of string
append(str)	Adds str to end
substr(pos, len)	Returns substring
find(str)	Finds index of substring
replace(pos, len, str)	Replaces part with another string
compare(str)	Compares strings
clear()	Empties the string
empty()	Checks if string is empty

Example:

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

int main() {
    string message = "Hello C++";

    message.append(" World");
    cout << message << endl;

    cout << "Substring: " << message.substr(6, 3) << endl;
    cout << "Found at: " << message.find("C++") << endl;

    return 0;
}
```

Comparison: C-strings vs string Class

Feature	C-style Strings (char[])	C++ string Class
Header File	<cstring>	<string>
Null Terminator	Required	Handled internally
Safer to use	❌ Prone to buffer overflows	✅ Exception-safe
Functions	strlen, strcpy, etc.	.length(), .find(), etc.

---

19. . How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

-: Arrays in C++ can be initialized in several ways at the time of declaration. The syntax depends on whether it's a 1D (one-dimensional) or 2D (two-dimensional) array.

---

1. One-Dimensional (1D) Array Initialization

Syntax:

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

Examples:

a) Full initialization:

```
int marks[5] = {90, 80, 70, 60, 50};
```

b) Partial initialization (rest are set to 0):

```
int numbers[5] = {1, 2};  // becomes {1, 2, 0, 0, 0}
```

c) Automatic size detection:

```
int scores[] = {10, 20, 30};  // size automatically set to 3
```

d) Default initialization (set all to 0):

```
int arr[5] = {0};  // becomes {0, 0, 0, 0, 0}
```

---

2. Two-Dimensional (2D) Array Initialization

A 2D array is like a table (rows × columns).

Syntax:

```
data_type array_name[rows][cols] = {
    {row1_values},
```

```
    {row2_values},
    ...
};
```

Examples:

a) Full initialization:

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

b) Partial initialization (rest elements set to 0):

```
int grid[2][3] = {
    {1},
    {4}
};  // grid = {{1, 0, 0}, {4, 0, 0}}
```

c) Flat list initialization (row-wise):

```
int table[2][2] = {1, 2, 3, 4};  // same as {{1, 2}, {3, 4}}
```

Example Program with 1D and 2D Arrays:

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

int main() {
    // 1D array
    int nums[4] = {5, 10, 15, 20};

    cout << "1D Array: ";
    for (int i = 0; i < 4; i++) {
        cout << nums[i] << " ";
    }
```

```
    // 2D array
    int matrix[2][2] = {
        {1, 2},
        {3, 4}
    };

    cout << "\n2D Array:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

1D Array: 5 10 15 20

2D Array:

1 2

3 4

20. Explain string operations and functions in C++

-: String Operations and Functions in C++

C++ supports two types of strings:

1. C-style strings (character arrays)
2. C++ string class (from the <string> library – easier and safer)

Let's explore string operations and functions for both types.

---

1. C-Style Strings (Character Arrays)

These are arrays of characters terminated with a null character ('\0').

Example:

```
char str[10] = "Hello";
```

- Common C-String Functions (from <cstring>):

Function	Description	Example
strlen(str)	Returns length of string	strlen("Hi") → 2
strcpy(dest, src)	Copies one string to another	strcpy(b, a)
strcat(dest, src)	Concatenates two strings	strcat(a, b)
strcmp(s1, s2)	Compares two strings	strcmp("a", "b") → -1

Example Code:

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

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";

    strcat(str1, str2);  // str1 becomes "HelloWorld"
    cout << "Concatenated: " << str1 << endl;
    cout << "Length: " << strlen(str1) << endl;

    return 0;
}
```

---

2. C++ string Class (from <string>)

Much more powerful and safer than character arrays.

Declaration:

```
#include <string>
string name = "Khushi";
```

21. . Explain the key concepts of Object-Oriented Programming (OOP).

-: Object-Oriented Programming (OOP) is a programming paradigm centered around objects, which are instances of classes. It helps organize code, increase reusability, and make programs easier to scale and maintain.

The 4 Core Concepts of OOP:

---

1. Encapsulation

Wrapping data and functions into a single unit (class), and restricting direct access to some of the object's components.

- Purpose: Protects data from outside interference (data hiding).
- Achieved by: Using private, public, and protected access specifiers.

Example:

```
class Student {
private:
    int marks;  // hidden from outside

public:
    void setMarks(int m) {
        marks = m;
    }

    int getMarks() {
        return marks;
    }
};
```

---

2. Abstraction

Hiding complex implementation details and showing only the essential features of an object.

- Purpose: Reduces complexity.
- Achieved by: Using classes, access specifiers, and abstract classes.

Example:

```
class Car {
public:
    void startEngine() {  // hides internal combustion logic
        cout << "Engine Started" << endl;
    }
};
```

---

3. Inheritance

Allows a class (child/derived class) to acquire properties and behaviors from another class (parent/base class).

- Purpose: Code reusability and hierarchical classification.
- Types: Single, Multiple, Multilevel, Hybrid, Hierarchical

Example:

```
class Animal {
public:
    void sound() {
        cout << "Makes sound" << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Barks" << endl;
    }
};
```

---

4. Polymorphism

Ability to use one function or operator in different ways.

- Types:

- Compile-time polymorphism (Function/Operator Overloading)
- Run-time polymorphism (Function Overriding via Virtual Functions)

Example: Function Overloading

```
class Print {
public:
    void show(int a) {
        cout << "Integer: " << a << endl;
    }
    void show(string s) {
        cout << "String: " << s << endl;
    }
};
```

22. What are classes and objects in C++? Provide an example.

-:In C++, classes and objects are fundamental concepts of Object-Oriented Programming (OOP).

What is a Class?

A class is a blueprint or template that defines the properties (data members) and behaviors (member functions) of objects.

Syntax:

```
class ClassName {
public:
    // data members
    // member functions
};
```

What is an Object?

An object is a real-world instance of a class.
Once a class is defined, you can create multiple objects of that class.

Example: Class and Object in C++

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

// Defining the class
class Student {
public:
    // Data members (attributes)
    string name;
    int roll;

    // Member function (behavior)
    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "Roll No: " << roll << endl;
    }
};

int main() {
    // Creating an object of class Student
    Student s1;

    // Assigning values to object
    s1.name = "Khushi";
    s1.roll = 101;

    // Calling member function
    s1.displayInfo();

    return 0;
```

}

- Output:

Name: Khushi

Roll No: 101

23. What is inheritance in C++? Explain with an example.

-: Inheritance is an Object-Oriented Programming concept that allows one class (derived/child class) to acquire properties and behaviors (data members and member functions) of another class (base/parent class).

Purpose of Inheritance:

- Code reusability (write once, reuse in multiple classes)
- Improved organization and modularity
- Supports hierarchical relationships (e.g., Animal → Dog, Cat)

Syntax of Inheritance:

```
class DerivedClass : access_specifier BaseClass {
    // new members or overridden members
};
```

- access_specifier is usually public, protected, or private.

Example: Single Inheritance

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

// Base class
class Animal {
public:
    void eat() {
        cout << "Animal is eating." << endl;
    }
```

```
};

// Derived class
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking." << endl;
    }
};

int main() {
    Dog d1;
    d1.eat();   // Inherited from Animal
    d1.bark();  // Defined in Dog
    return 0;
}
```

- Output:

```
Animal is eating.
Dog is barking.
```

24. What is encapsulation in C++? How is it achieved in classes?

-: Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP).
It refers to the bundling of data and the functions that operate on that data into a single unit (class), while restricting direct access to some components.

Key Goals of Encapsulation:

- Data hiding: Keep internal details private.
- Controlled access: Use public methods to access or modify private data.
- Improved security, modularity, and maintainability.

How is Encapsulation Achieved in C++?

1. Declare class variables as private or protected
2. Provide public methods (getters/setters) to access and modify those variables.

Example of Encapsulation:

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

class Student {
private:
    int rollNumber;       // private = hidden from outside
    string name;

public:
    // Setter functions
    void setRollNumber(int r) {
        rollNumber = r;
    }

    void setName(string n) {
        name = n;
    }

    // Getter functions
    int getRollNumber() {
        return rollNumber;
    }

    string getName() {
        return name;
```

```
    }
};

int main() {
    Student s;

    // Accessing private members through public methods
    s.setRollNumber(101);
    s.setName("Khushi");

    cout << "Roll Number: " << s.getRollNumber() << endl;
    cout << "Name: " << s.getName() << endl;

    return 0;
}
```

- Output:

```
Roll Number: 101
Name: Khushi
```