Module 3 : Introduction to oops programing

1. Introduction to C++

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

Ans :

Feature	**Procedural Programming**	**Object-Oriented Programming (OOP)**
Basic Concept	Program is divided into functions or procedures.	Program is divided into objects containing data and functions.
Approach	Focuses on **functions** (what to do).	Focuses on **objects** (who does it).
Data Handling	Data is usually **shared globally** among functions.	Data is **encapsulated** within objects and accessed through methods.
Data Security	Less secure — functions can access and modify any data.	More secure — data is hidden using **encapsulation** and **access specifiers** (private, public, protected).
Reusability	Code reuse is limited; functions can be reused but not easily extended.	High reusability — achieved through **inheritance** and **polymorphism**.
Examples	C, Pascal, FORTRAN	C++, Java, Python, C#
Program Structure	Sequence of instructions executed step-by-step.	Interaction between objects that represent real-world entities.
Main Elements	Functions, variables, and structures.	Classes, objects, inheritance, polymorphism, encapsulation, abstraction.
Example Code Style	void add(int a, int b) → performs an operation.	class Calculator { public: void add(); } → operation inside a class.

1) Procedural programing :

```
#include<iostream>

using namespace std;


int add(int a, int b) {

   return a + b;

}

int main() {
```

```
    int x = 5, y = 10;
  cout << "Sum = " << add(x, y);
  return 0;
}
```

2) Object oriented programing

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

class Calculator {
public:
  int add(int a, int b) {
    return a + b;
  }
};

int main() {
  Calculator c;      // Object creation
  cout << "Sum = " << c.add(5, 10);
  return 0;
}
```

2) List and explain the main advantages of OOP over POP?

Ans : Main Advantages of OOP (Object-Oriented Programming) over POP (Procedural-Oriented Programming.

1. Modularity (Code Organization)

- In OOP, the program is divided into **objects and classes**, making it more **organized**.
- Each class handles its own data and functions.
- This makes the code **easier to understand, test, and debug**.

Example: A Student class can handle all student-related details separately from a Teacher class.

2. Reusability

- Classes and objects can be **reused** in other programs or parts of the same program.
- **Inheritance** allows new classes to be created based on existing ones, reducing code duplication.

Example: A Vehicle class can be reused by Car, Bike, and Bus classes.

3. Data Security (Encapsulation)

- OOP hides data using **private and protected access specifiers**.
- Data can only be accessed through **public functions (methods)**.
- This prevents accidental changes and ensures **data integrity**.

Example: You can't directly change a student's marks; you must use a proper method like setMarks().

4. Flexibility through Polymorphism

- OOP allows the same function name to behave differently depending on the object.
- This makes programs **more flexible and easier to extend**.

Example: draw() can mean drawing a circle, rectangle, or triangle depending on the object.

5. Ease of Maintenance

- Because the program is divided into **independent objects**, changes can be made easily without affecting the whole system.
- Maintenance becomes **simpler and safer**.

Example: If you modify the Student class, it doesn't affect the Teacher class.

6. Real-world Modeling

- OOP is based on **real-world entities** like students, cars, or accounts.
- This makes design and understanding of the program **more natural and intuitive**.

Example: Instead of thinking about functions, you think about how real objects behave.

Summary Table

Advantage	**Description**
Modularity	Code organized into classes/objects
Reusability	Inheritance allows reuse of code
Data Security	Data hidden using encapsulation
Polymorphism	Same function, different behaviors
Maintenance	Easier to modify and fix
Real-world Modeling	Matches real-world structure

Let's include **POP (Procedural-Oriented Programming)** too — so you can clearly see **how OOP is better** than POP.

Procedural-Oriented Programming (POP)

Definition:
POP focuses on **functions (procedures)** that operate on data.
The main goal is to **perform tasks step-by-step**, following a **top-down approach**.

Features of POP:

1. Program is divided into **functions**.

2. Data is usually **shared globally** among all functions.
3. Emphasis is on **actions (what to do)**, not on data.
4. **Less secure**, since any function can modify data.
5. **Examples:** C, Pascal, FORTRAN.

Limitations of POP:

1. **Poor data security** — global data can be accidentally changed.
2. **Code reusability is low** — you can reuse functions but not easily extend them.
3. **Hard to maintain** — changes in one function can affect others.
4. **Not suitable for complex or large projects.**
5. **Real-world modeling is difficult** — it doesn't map naturally to real-world entities.

OOP vs POP (Comparison Table)

Feature	**POP (Procedural-Oriented Programming)**	**OOP (Object-Oriented Programming)**
Approach	Top-down (focus on functions)	Bottom-up (focus on objects)
Data Handling	Data shared among functions	Data hidden inside classes
Security	Less secure (global data)	More secure (encapsulation)
Reusability	Low	High (inheritance & polymorphism)
Maintenance	Difficult	Easier
Modeling	Hard to model real-world systems	Natural real-world modeling
Examples	C, Pascal	C++, Java, Python

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

Ans :

Step 1: Install a C++ Compiler

A compiler converts your C++ code into a form the computer can execute.

Popular compilers:

- **GCC (GNU Compiler Collection)** – for Windows, Linux, macOS
- **MinGW** – lightweight GCC for Windows

- **Clang** – used on macOS and Linux
- **MSVC (Microsoft Visual C++)** – built into Visual Studio

For Windows (MinGW/GCC):

1. Go to https://www.mingw-w64.org/
2. Download and install **MinGW-w64**.
3. During installation, select:
    - Architecture: x86_64
    - Threads: posix
    - Exception: seh
4. After installation, **add MinGW to the PATH**:
    - Open *System Properties → Environment Variables → Path → Add MinGW's bin folder path* (e.g., C:\mingw64\bin).

For Linux/macOS:

- Open Terminal → type:
- sudo apt install g++

or

xcode-select --install

(for macOS)

Step 2: Choose a Text Editor or IDE

An **IDE (Integrated Development Environment)** makes coding easier with features like syntax highlighting, auto-complete, and debugging.

Popular IDEs for C++:

- **Code::Blocks** – beginner-friendly and includes GCC.
- **Dev-C++** – lightweight and easy to use.
- **Visual Studio** – powerful IDE for Windows.
- **VS Code** – modern editor; works with extensions.
- **CLion** – advanced IDE for professional projects.

Step 3: Configure the IDE or Editor

- If you use **Code::Blocks** or **Dev-C++**, the compiler is usually auto-configured.
- If using **VS Code**:
    1. Install **C/C++ Extension (by Microsoft)**.
    2. Configure the **compiler path** in settings.

3. Create a new file → save as program.cpp.

Step 4: Write Your First Program

Open your IDE/editor and type:

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

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

Save it as hello.cpp.

Step 5: Compile the Program

- **Command Line:**
- g++ hello.cpp -o hello

Then run:

./hello

Step 6: Run and Test the Output

If everything is correct, you'll see:

Hello, World!

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

Ans :

- **cin** → for input
- **cout** → for output
- **cerr** → for error messages
- **clog** → for general log messages

To use these, we include the **header file**:

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

1. Output Operation (cout)

- cout stands for **console output**.
- It is used to **display information** on the screen.
- Uses the **insertion operator (<<)**.

Example:

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

int main() {
    cout << "Hello, World!" << endl;
    cout << "C++ is powerful!" << endl;
    return 0;
}
```

Output:

Hello, World!
C++ is powerful!

Explanation:

endl means "end line" — it moves the cursor to a new line.

2. Input Operation (cin)

- cin stands for **console input**.
- It is used to **take input from the user**.
- Uses the **extraction operator (>>)**.

Example:

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

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You are " << age << " years old.";
    return 0;
}
```

Output (example run):

Enter your age: 20
You are 20 years old.

Explanation:

cin >> age reads user input and stores it in the variable age.

2) . Variables, Data Types, and Operators

1) What are the different data types available in C++? Explain with examples

Ans :

Data Type	Description	Example
int	Stores whole numbers (no decimal)	int age = 20;

Data Type	Description	Example
float	Stores decimal numbers (single precision)	float marks = 89.5;
double	Stores decimal numbers (double precision)	double pi = 3.14159;
char	Stores a single character	char grade = 'A';
bool	Stores true or false values	bool pass = true;
void	Represents “no value” (used in functions)	void display();
string	Stores a sequence of characters (text)	string name = "Bharat";

Input :

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

int main() {
   int roll = 101;
   float marks = 92.5;
   double average = 90.12345;
   char grade = 'A';
   bool pass = true;
   string name = "Bharat";

   cout << "Name: " << name << endl;
   cout << "Roll No: " << roll << endl;
   cout << "Marks: " << marks << endl;
   cout << "Average: " << average << endl;
   cout << "Grade: " << grade << endl;
   cout << "Pass: " << pass << endl;
   return 0;
}
```

Output :

Name: Bharat

Roll No: 101

Marks: 92.5

Average: 90.1235

Grade: A

Pass: 1

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

Ans :

Type Conversion in C++

Type conversion is the process of converting a value from **one data type to another**.

There are two types:

1. **Implicit Type Conversion (Type Casting / Type Promotion / Automatic Conversion)**
2. **Explicit Type Conversion (Type Casting / Manual Conversion)**

1. Implicit Type Conversion (Type Promotion)

- Also called **automatic type conversion**.
- The **compiler automatically converts** a smaller or compatible data type to a larger or compatible one.
- Usually happens **when different types are used in an expression**.
- **Safe**, but sometimes precision can be lost in certain cases.

Example 1: Integer to Float

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

int main() {
   int x = 10;
   double y;

   y = x;  // int automatically converted to double
   cout << "y = " << y << endl;

   return 0;
}
```

Output:

```
y = 10
```

Example 2: Mixed Expression

```
int a = 5;
float b = 2.5;
float c;

c = a + b;  // 'a' is automatically converted to float
cout << "c = " << c;
```

Output:

c = 7.5

2. Explicit Type Conversion (Type Casting)

- Also called **manual type conversion**.
- The **programmer explicitly tells the compiler** to convert a value from one type to another.
- Done using **casting operators**:
    1. C-style cast → (type)value
    2. C++ style casts → static_cast<type>(value)

Example 1: C-Style Casting

```
int x = 10;
double y;

y = (double)x;  // explicitly converting int to double
cout << "y = " << y;
```

Example 2: C++ Style Casting

```
int a = 7, b = 2;
double result;

result = static_cast<double>(a) / b;  // convert 'a' to double
cout << "Result = " << result;
```

Output:

Result = 3.5

Without casting, integer division would give 3.

Key Differences Between Implicit and Explicit Conversion

Feature	**Implicit Conversion**	**Explicit Conversion**
Also called	Type promotion / Automatic conversion	Type casting / Manual conversion
Who does t	Done automatically by compiler	Done manually by programmer
Syntax	No special syntax	(type)value or static_cast<type>(value)
Safety	Generally safe	May lose data if not used carefully
Use case	Small to large type conversions ıutomatically	Force conversion to avoid loss or control ›behavior

- **Implicit** → compiler converts automatically.
- **Explicit** → programmer forces the conversion.

3) What are the different types of operators in C++? Provide examples of each ?

Ans :

An **operator** is a symbol that tells the compiler to perform a **specific operation** on one or more operands (variables or values).

C++ operators can be broadly classified into **8 main types**:

1. Arithmetic Operators

Used to perform **mathematical operations**.

Operator	Description	Example
+	Addition	int sum = 5 + 3;
-	Subtraction	int diff = 5 - 3;
*	Multiplication	int prod = 5 * 3;
/	Division	int div = 10 / 2;
%	Modulus (remainder)	int rem = 10 % 3;
++	Increment by 1	int x = 5; x++;
--	Decrement by 1	int x = 5; x--;

Example Program:

```
int a = 10, b = 3;
cout << a + b << endl; // 13
cout << a % b << endl; // 1
```

2. Relational Operators

Used to **compare two values**. Returns **true (1)** or **false (0)**.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Example:

```
int a = 5, b = 10;
cout << (a < b); // 1 (true)
```

3. Logical Operators

Used to perform **logical operations** (true/false).

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
||	Logical OR	(a>0 || b>0)
!	Logical NOT	!(a > b)

Example:
int a = 5, b = 10;
cout << (a > 0 && b > 0); // 1 (true)

4. Assignment Operators
Used to **assign values** to variables.

Operator	Description	Example
=	Simple assignment	a = 5
+=	Add and assign	a += 3; // a = a + 3
-=	Subtract and assign	a -= 2; // a = a - 2
*=	Multiply and assign	a *= 2; // a = a * 2
/=	Divide and assign	a /= 2; // a = a / 2
%=	Modulus and assign	a %= 3; // a = a % 3

5. Bitwise Operators
Used to perform operations on **bits** of an integer.

Operator	Description	Example
&	Bitwise AND	a & b
|	Bitwise OR	a|b
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

6. Conditional (Ternary) Operator
- A shorthand for **if-else** statements.
- Syntax: condition ? value_if_true : value_if_false

Example:
int a = 10, b = 20;
int max = (a > b) ? a : b;
cout << "Maximum: " << max;
Output:
Maximum: 20

7. Increment/Decrement Operators
- Already part of arithmetic operators but often listed separately.
- **Prefix (++a)** → increment before use

- **Postfix (a++)** → increment after use

8. Miscellaneous Operators

Operator	Description	Example
sizeof()	Returns memory size of data type	sizeof(int)
&	Address-of operator	int *ptr = &a;
*	Pointer dereference	cout << *ptr;
,	Comma operator	int a = (1,2,3); // a=3

Summary Table

Type	Purpose	Example
Arithmetic	Math operations	+, -, *, /, %
Relational	Compare values	==, !=, >, <
Logical	Logical conditions	`&&,
Assignment	Assign values	=, +=, -=
Bitwise	Operate on bits	`&,
Conditional	Shorthand if-else	?:
Miscellaneous	Size, address, pointer	sizeof(), &, *

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

Ans :

Constants in C++

Definition:

A **constant** is a **value that cannot be changed** during program execution.
It is used when you want **fixed values** that should remain the same

Purpose of Constants:

1. **Prevent accidental changes** in variables.
2. **Make code more readable** and maintainable.
3. **Represent fixed values** like π, maximum limits, or configuration values.

Ways to Define Constants:

a) Using const keyword

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

int main() {
   const double PI = 3.14159;  // constant
   // PI = 3.14;  // Error: cannot change a constant
   cout << "Value of PI: " << PI;
   return 0;
}
```

b) Using #define preprocessor directive

```
#include <iostream>
#define MAX 100  // constant using define

int main() {
   cout << "Max value: " << MAX;
   return 0;
}
```

c) Using enum for integer constants

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

enum { RED = 1, GREEN = 2, BLUE = 3 };

int main() {
   cout << "Red color code: " << RED;
   return 0;
}
```

2. Literals in C++

Definition:

A **literal** is a **fixed value** that is directly written in the code.
It can be of various types: integer, floating-point, character, boolean, or string.

Types of Literals:

Integer Literals

```
int a = 100;  // decimal

int b = 0xFF; // hexadecimal

int c = 077;  // octal
```

Floating-Point Literals

```
float pi = 3.14f;    // float literal

double e = 2.718;    // double literal

long double l = 3.14L; // long double literal
```

Character Literals

```
char ch = 'A';       // single character

char newline = '\n'; // escape character
```

String Literals

```
string name = "Bharat";  // sequence of characters
```

Boolean Literals

```
bool pass = true;   // true

bool fail = false;  // false
```

Purpose of Literals:

1. Represent **fixed values** in the code.
2. Used in **assignments, calculations, and comparisons**.
3. Make the program **more readable** than using raw numbers or characters everywhere.

Example Using Constants and Literals Together

```
#include <iostream>

using namespace std;


int main() {

   const double PI = 3.14159;  // constant

   int radius = 5;            // literal value
```

```
    double area = PI * radius * radius;


    cout << "Area of circle: " << area;
    return 0;
}
```

Output:

Area of circle: 78.5397

Explanation:

- PI → constant (cannot be changed).
- 5 → literal (direct value in code).

Constants → fixed **named values** (cannot change).

Literals → fixed **unnamed values** written directly in the program.

3) . Control Flow Statements

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

Ans :

Definition:
Conditional statements allow the program to **make decisions** and execute certain code **only if a condition is true**.

C++ provides two main types of conditional statements:

1. **if-else statement**
2. **switch statement**

---

1. if-else Statement

Syntax:

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

Explanation:

- The if part checks a **condition**.
- If the condition is **true**, the if block executes.
- If the condition is **false**, the else block executes.
- else is optional.

Example:

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

int main() {
   int marks;
   cout << "Enter your marks: ";
   cin >> marks;

   if (marks >= 50) {
      cout << "You passed the exam.";
   } else {
      cout << "You failed the exam.";
   }

   return 0;
}
```

Output (example run):

```
Enter your marks: 65
You passed the exam.
```

Nested if-else Example:

```
if (marks >= 75)
   cout << "Grade A";
else if (marks >= 50)
   cout << "Grade B";
else
   cout << "Grade C";
```

2. switch Statement

Syntax:

```
switch (expression) {
   case value1:
      // code to execute if expression == value1
      break;
   case value2:
      // code to execute if expression == value2
      break;
   ...
   default:
      // code to execute if no case matches
}
```

Explanation:

- switch evaluates an **integer or character expression**.
- Executes the **matching case**.
- break is used to **exit the switch** after a case executes.
- default executes if **no case matches**.

Example:

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

int main() {
   int day;
   cout << "Enter day number (1-7): ";
   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";
    }

    return 0;
}
```

Output (example run):

Enter day number (1-7): 3

Wednesday

Key Differences: if-else vs switch

Feature	**if-else**	**switch**
Type of condition	Boolean expression (any type)	Integer, char, or enum
Flexibility	Very flexible (complex conditions)	Less flexible (only equality check)
Number of choices	Works for few or many conditions	Best for multiple discrete choices
Syntax	if (condition) { } else { }	switch(expression) { case value: … }

if-else → use for general conditions and ranges.

switch → use for fixed discrete values like menu options or days.

2) **What is the difference between for, while, and do-while loops in C++?**

Ans :

Definition:

Loops are used to **execute a block of code repeatedly** as long as a condition is true.

C++ has three main types of loops:

1. **for loop**
2. **while loop**
3. **do-while loop**

1. for Loop

Syntax:

```
for(initialization; condition; increment/decrement) {
   // code to execute
}
```

Explanation:

- **Initialization** → executed once before the loop starts.
- **Condition** → checked before each iteration.
- **Increment/Decrement** → executed after each iteration.

Example:

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

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

Output:

1 2 3 4 5

Use when the number of iterations is known.

2. while Loop

Syntax:

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

Explanation:

- Condition is **checked before** each iteration.
- If condition is false initially, **loop may not execute even once**.

Example:

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

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

Output:

1 2 3 4 5

Use when the number of iterations is unknown or depends on a condition.

3. do-while Loop

Syntax:

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

Explanation:

- **Code executes first**, then condition is checked.
- Guaranteed to execute **at least once**.

Example:

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

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

}

Output:

1 2 3 4 5

Use when you want the loop to execute at least once.

Key Differences Between for, while, and do-while

Feature	**for loop**	**while loop**	**do-while loop**
Initialization	Inside loop statement	Before loop	Before loop
Condition checked	Before each iteration	Before each iteration	After each iteration
Execution guarantee	May not execute if condition false	May not execute if condition false	Executes at least once
Use case	Known number of iterations	Unknown iterations	At least one execution needed
Syntax	Compact	Flexible	Flexible, ends with while(condition);

- **for** → best for loops with a known number of repetitions.
- **while** → best for loops controlled by a condition.
- **do-while** → best when code must run at least once before checking the conditi

3) **How are break and continue statements used in loops? Provide examples.**

Ans :

- **break** → Immediately **terminates the loop** and transfers control **outside the loop**.
- **continue** → **Skips the current iteration** of the loop and continues with the **next iteration**.

These are used inside for, while, and do-while loops.

1. Break Statement

Purpose: Stop the loop **before its normal end** when a condition is met.

Example 1: Using break in a for loop

```
#include <iostream>

using namespace std;
```

```
int main() {
   for(int i = 1; i <= 10; i++) {
      if(i == 5) {
         break;  // exit the loop when i is 5
      }
      cout << i << " ";
   }
   return 0;
}
```

Output:

1 2 3 4

Explanation:

- The loop stops when i == 5.
- Statements after break are **not executed** in the loop.

2. Continue Statement

Purpose: Skip the **current iteration** and move to the **next iteration** of the loop.

Example 2: Using continue in a for loop

```
#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

Explanation:

- When i == 3, continue skips the cout statement.
- Loop continues with i = 4.

Key Points

Statement	Effect on Loop	Use Case
break	Terminates the loop completely	Stop loop when a condition is met
continue	Skips current iteration	Skip specific values or conditions

Example in while loop

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

int i = 0;
while(i < 5) {
   i++;
   if(i == 2) continue;  // skip 2
   if(i == 4) break;     // stop loop at 4
   cout << i << " ";
}
```

Output:

1 3

In Short:

- **break** → exit the loop immediately.
- **continue** → skip **only the current iteration**, continue looping.

4) Explain nested control structures with an example.

Ans :

A **nested control structure** is when **one control structure is placed inside another**.
Control structures include:

- Conditional statements (if, if-else, switch)

- Loops (for, while, do-while)

Purpose:

- To handle **more complex logic**
- Allows **multiple levels of decision-making** or repeated operations.

Example 1: Nested if-else

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

int main() {
   int marks;
   cout << "Enter your marks: ";
   cin >> marks;

   if (marks >= 50) {
      // Nested if
      if (marks >= 90) {
         cout << "Grade: A+";
      } else if (marks >= 75) {
         cout << "Grade: A";
      } else {
         cout << "Grade: B";
      }
   } else {
      cout << "Failed";
   }

   return 0;
}
```

Explanation:

- The outer if checks if the student **passed**.

- The inner if-else determines the **grade**.

Sample Output:

Enter your marks: 82

Grade: A

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

Explanation:

- **Outer loop (i)** runs 3 times.
- **Inner loop (j)** runs 2 times **for each iteration of outer loop**.

🔹 **Key Points**

1. Nested structures can be **any combination** of if-else, for, while, do-while, or switch.
2. Helps to **break complex logic into smaller decisions**.
3. Avoid excessive nesting — can make code **hard to read**.

Nested if-else → decision inside a decision

Nested loops → loop inside a loop

Topic – 4 Function And Scope

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

Ans :

Definition:
A **function** is a **block of code** that performs a specific task and can be **reused multiple times** in a program.

Purpose of Functions:

1. Reduce code duplication.
2. Improve readability and organization.
3. Make debugging easier.
4. Allow modular programming.

Parts of a Function

A function generally has three components:

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

Function Declaration (Prototype)

Definition:
Tells the compiler **about the function name, return type, and parameters** before it is used.
It **does not contain the function body**.

Syntax:

return_type function_name(parameter_list);

Example:

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

Function Definition

Definition:
Provides the **actual body of the function** — the statements that execute when the function is called.

Syntax:

return_type function_name(parameter_list) {

```
    // statements
    return value;  // if return_type is not void
}
```

Example:

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

Function Call

Definition:
Executes the function. You pass the required **arguments**, and the function returns a result (if not void).

Example:

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

// Function declaration
int add(int, int);

int main() {
    int x = 10, y = 20;
    int result = add(x, y);  // function call
    cout << "Sum = " << result;
    return 0;
}

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

Output:

```
Sum = 30
```

Term	Explanation
Function Declaration	Introduces the function to the compiler; usually at the top of the program.
Function Definition	Contains the actual code (body) of the function.
Function Call	Executes the function with actual arguments.
Return Type	Specifies the type of value returned (int, float, void, etc.).
Parameters	Variables that the function takes as input.
Arguments	Actual values passed to the function when calling it.

Example with void function (no return value)

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

void greet(string name) {  // function definition
   cout << "Hello, " << name << "!" << endl;
}

int main() {
   greet("Bharat");  // function call
   greet("Dost");
   return 0;
}
```

Output:

```
Hello, Bharat!
Hello, Dost!
```

1. **Declaration** → tells the compiler about the function.
2. **Definition** → contains the actual code.
3. **Call** → executes the function in the program.

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

Ans :

Scope of Variables in C++

Definition:
The **scope** of a variable is the **region or part of the program** where the variable can be **accessed or modified**.

- Variables can have **different scopes** depending on where they are declared.
- Proper scope management helps **avoid errors and unexpected behavior**.

1. Local Scope (Local Variables)

Definition:
A variable declared **inside a function, block, or loop** is called a **local variable**.

- **Accessible only within that function or block**.
- **Destroyed automatically** when the function or block ends.

Example:

```
#include <iostream>

using namespace std;


int main() {
   int x = 10;  // local variable
   cout << "x inside main: " << x << endl;
   {
      int y = 20;  // local to this block
      cout << "y inside block: " << y << endl;
   }
   // cout << y; // Error: y is not accessible here
   return 0;
}
```

Output:

```
x inside main: 10
y inside block: 20
```

2. Global Scope (Global Variables)

Definition:
A variable declared **outside all functions** is called a **global variable**.

- **Accessible by all functions** in the program (unless shadowed by a local variable).
- Exists **throughout the lifetime of the program**.

Example:

```
#include <iostream>
using namespace std;
int x = 100;  // global variable
void display() {
   cout << "x inside function: " << x << endl;
}
int main() {
   cout << "x inside main: " << x << endl;
   display();
   return 0;
}
```

Output:

```
x inside main: 100
x inside function: 100
```

Key Differences: Local vs Global Variables

Feature	Local Variable	Global Variable
Declared	Inside a function, block, or loop	Outside all functions
Scope	Only within the block/function	Entire program
Lifetime	Exists only during block execution	Exists until program ends
Accessibility	Cannot be accessed outside the block	Accessible from any function
Default Value	Garbage value if not initialized	Zero-initialized if not explicitly initialized
Memory	Stored in stack	Stored in data segment

Shadowing Example (Local vs Global)

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

int x = 50;  // global
```

```
int main() {
   int x = 10;  // local shadows global
   cout << "Local x: " << x << endl;   // 10
   cout << "Global x: " << ::x << endl; // 50 (::x accesses global)
   return 0;
}
```

Output:

Local x: 10

Global x: 50

Local variable → temporary, exists only inside function/block.

Global variable → permanent, accessible throughout the program.

3**) Explain recursion in C++ with an example.**

Ans :

Recursion in C++

Definition:
Recursion is a programming technique where a **function calls itself** to solve a problem.

Purpose:

- Used to **break complex problems into smaller, simpler problems**.
- Commonly used in **factorial calculation, Fibonacci series, tree traversal, etc.**

Key Points of Recursion

1. **Base Case:**
    - A condition that **stops the recursion** to prevent infinite calls.
2. **Recursive Case:**
    - The part of the function where the **function calls itself**.

Without a **base case**, recursion leads to **stack overflow**.

Example 1: Factorial using Recursion

Factorial of a number n is n! = n × (n-1) × (n-2) × ... × 1.

Code:

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

```
// Function to calculate factorial
int factorial(int n) {
   if(n == 0 || n == 1)   // base case
      return 1;
   else                   // recursive case
      return n * factorial(n - 1);
}

int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;

   int result = factorial(num);  // function call
   cout << "Factorial of " << num << " is " << result;
   return 0;
}
```

Sample Output:

```
Enter a number: 5
Factorial of 5 is 120
```

How It Works:

If n = 5, the function calls happen like this:

factorial(5) -> 5 * factorial(4)

factorial(4) -> 4 * factorial(3)

factorial(3) -> 3 * factorial(2)

factorial(2) -> 2 * factorial(1)

factorial(1) -> 1 (base case reached)

Then the results **multiply back up the call stack**:

factorial(2) = 2*1 = 2

factorial(3) = 3*2 = 6

factorial(4) = 4*6 = 24

factorial(5) = 5*24 = 120

Advantages of Recursion:

1. Simplifies complex problems.
2. Reduces code size.
3. Useful in problems like **tree/graph traversal, sorting, searching**.

Disadvantages of Recursion:

1. Uses more **memory** (stack memory for each function call).
2. Can be **slower** than loops for simple problems.
3. Risk of **stack overflow** if base case is missing.

4) What are function prototypes in C++? Why are they used?

Ans :

Definition:
A **function prototype** is a **declaration of a function** that tells the compiler **the function's name, return type, and parameters before it is defined**.

Syntax:

return_type function_name(parameter_list);

Example:

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

Why Function Prototypes are Used

1. **Inform the Compiler in Advance:**
    - The compiler knows **how the function will be called** before it encounters the function definition.
2. **Enable Calls Before Definition:**
    - You can **call a function in main()** or another function **even if the definition appears later** in the program.
3. **Type Checking:**
    - Ensures that **correct number and type of arguments** are passed during function calls.
4. **Better Program Structure:**
    - Allows organizing **large programs** with functions defined **after main()** or in separate files.

Example Using Function Prototype

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

// Function prototype
int add(int a, int b);

int main() {
   int sum;
   sum = add(10, 20);  // function call before definition
   cout << "Sum = " << sum;
   return 0;
}
// Function definition
int add(int a, int b) {
   return a + b;
}
```

Output:

Sum = 30

Explanation:

- int add(int a, int b); tells the compiler the **function exists**.
- The function is **defined later**, but the call in main() works fine.

5. Arrays and Strings

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

Ans :

Definition:
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**.
- Arrays help **store multiple values under a single variable name**.

Syntax for Array Declaration:

data_type array_name[size];

Example:

int marks[5]; // array of 5 integers

1. Single-Dimensional Arrays

Definition:

- A **single-dimensional array** stores elements in a **linear form** (like a list).
- Each element is accessed using **one index**.

Example:

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

int main() {
   int marks[5] = {85, 90, 78, 92, 88};  // single-dimensional array

   cout << "Marks of student 3: " << marks[2]; // index starts from 0
   return 0;
}
```

Output:

Marks of student 3: 78

2. Multi-Dimensional Arrays

Definition:

- A **multi-dimensional array** stores elements in **rows and columns** (like a table or matrix).
- **2D arrays** are the most common, but 3D or higher dimensions are also possible.

Syntax for 2D Array:

data_type array_name[rows][columns];

Example (2D Array):

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

int main() {
```

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


    cout << "Element at row 2, column 3: " << matrix[1][2]; // index starts from 0
    return 0;
}
```

Output:

Element at row 2, column 3: 6

Key Differences: Single vs Multi-Dimensional Arrays

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	Linear, like a list	Table, matrix, or cube
Indexing	One index	Multiple indices (e.g., [row][col])
Use Case	Storing a simple list (marks, names, etc.)	Storing tables, matrices, grids, or multi-level data
Memory	Contiguous block for all elements	Contiguous block but arranged in rows and columns

Summary:

- **Array** → stores multiple values of same type in one variable.
- **Single-dimensional** → one index, linear storage.
- **Multi-dimensional** → multiple indices, like rows & columns.

2**) Explain string handling in C++ with examples.**

Ans :

String Handling in C++

Definition:
A **string** is a sequence of characters.
C++ provides **two ways to handle strings**:

1. **C-style strings** → using **character arrays** (char[])
2. **C++ string class** → using **std::string** from the Standard Library

1. C-Style Strings (Character Arrays)

Declaration:

```
char str[20];  // array of 20 characters
```

Example:

```
#include <iostream>
#include <cstring> // for string functions
using namespace std;

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

    strcpy(str2, str1);         // copy str1 to str2
    strcat(str2, " World");     // concatenate strings
    int len = strlen(str2);     // length of string

    cout << "String: " << str2 << endl;
    cout << "Length: " << len << endl;

    return 0;
}
```

Output:

```
String: Hello World
Length: 11
```

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

Function	Description
strcpy(dest, src)	Copy src to dest
strcat(dest, src)	Concatenate src to dest
strlen(str)	Returns length of string
strcmp(str1, str2)	Compares two strings (0 if equal)

2. C++ string Class

Advantages:

- Easier and safer than C-style strings
- Supports **concatenation, comparison, and many built-in functions**

Declaration:

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

string str = "Hello";
```

Example:

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

int main() {
    string str1 = "Hello";
    string str2 = "World";

    string str3 = str1 + " " + str2;  // concatenation
    cout << "Concatenated String: " << str3 << endl;

    cout << "Length: " << str3.length() << endl;  // string length

    str3.append("!");  // append string
    cout << "After append: " << str3 << endl;

    cout << "First character: " << str3[0] << endl; // access character

    return 0;
}
```

Output:

```
Concatenated String: Hello World
```

Length: 11

After append: Hello World!

First character: H

Common string Class Functions

Function	**Description**
length() or size()	Returns length of string
append(str)	Adds str at the end
substr(pos, len)	Returns substring from position pos of length len
find(str)	Finds first occurrence of str
erase(pos, len)	Deletes part of string
replace(pos, len, str)	Replaces part of string with str
c_str()	Converts string to C-style string

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

Ans :

Array initialization is the process of **assigning values to the elements** of an array at the time of declaration or later.

C++ supports both **1-dimensional (1D)** and **multi-dimensional (2D, 3D, …)** arrays.

1. Single-Dimensional (1D) Arrays

Syntax:

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

Example 1: Initialize at declaration

```
#include <iostream>

using namespace std;


int main() {

   int marks[5] = {85, 90, 78, 92, 88};  // 1D array initialization


   cout << "Marks of student 3: " << marks[2] << endl;
```

```
    return 0;
}
```

Output:

Marks of student 3: 78

Example 2: Partial initialization

```
int arr[5] = {1, 2}; // remaining elements set to 0
```

- arr → {1, 2, 0, 0, 0}

Example 3: Initialization later

```
int arr[3];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
```

2. Multi-Dimensional (2D) Arrays

Syntax:

```
data_type array_name[rows][columns] = { {row1_values}, {row2_values}, ... };
```

Example 1: Initialize at declaration

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

int main() {
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };  // 2 rows, 3 columns

    cout << "Element at row 2, column 3: " << matrix[1][2] << endl;
    return 0;
}
```

Output:

Element at row 2, column 3: 6

Example 2: Partial initialization

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

Example 3: Initialization later

```
int mat[2][2];

mat[0][0] = 10;

mat[0][1] = 20;

mat[1][0] = 30;

mat[1][1] = 40;
```

Key Points

1. Array indices start from **0**.
2. If fewer values are provided than the size, remaining elements are **initialized to 0**.
3. Multi-dimensional arrays are stored in **row-major order** in memory.
4. Arrays can also be initialized **using loops** at runtime.

Array Type	Initialization Example
1D	int arr[5] = {1,2,3,4,5};
2D	int mat[2][3] = {{1,2,3},{4,5,6}};

4) Explain string operations and functions in C++.

Ans :

Common String Operations in C++ (std::string)

1. **Concatenation** → + or .append()
2. **Access Characters** → [] or .at()
3. **Length** → .length() or .size()
4. **Substring** → .substr(pos, len)
5. **Find** → .find(str)
6. **Replace** → .replace(pos, len, str)
7. **Erase** → .erase(pos, len)
8. **Compare** → .compare(str)
9. **Convert to C-style string** → .c_str()

Example Program Demonstrating All Operations

```
#include <iostream>

#include <string>
```

```
using namespace std;

int main() {
    string str1 = "Hello";
    string str2 = "World";

    // 1. Concatenation
    string str3 = str1 + " " + str2;
    cout << "Concatenated: " << str3 << endl;

    // 2. Access Characters
    cout << "First character of str1: " << str1[0] << endl;
    cout << "Second character of str2: " << str2.at(1) << endl;

    // 3. Length
    cout << "Length of str3: " << str3.length() << endl;

    // 4. Substring
    cout << "Substring of str3 (6,5): " << str3.substr(6,5) << endl;

    // 5. Find
    int pos = str3.find("World");
    cout << "'World' found at position: " << pos << endl;

    // 6. Replace
    str3.replace(pos, 5, "C++");
    cout << "After replace: " << str3 << endl;

    // 7. Erase
    str3.erase(5, 4); // remove 4 characters starting at index 5
    cout << "After erase: " << str3 << endl;
```

```
    // 8. Compare
    string str4 = "Hello";
    cout << "Compare str1 and str4: " << str1.compare(str4) << endl; // 0 means equal

    // 9. Convert to C-style string
    const char* cstr = str3.c_str();
    cout << "C-style string: " << cstr << endl;

    return 0;
}
```

Sample Output

```
Concatenated: Hello World
First character of str1: H
Second character of str2: o
Length of str3: 11
Substring of str3 (6,5): World
'World' found at position: 6
After replace: Hello C++
After erase: Hello++
Compare str1 and str4: 0
C-style string: Hello++
```

6) Introduction to Object-Oriented Programming?

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

Ans :

Definition:
OOP is a **programming paradigm** that uses **objects** and **classes** to design and organize programs.

- It focuses on **data (attributes)** and **behavior (methods/functions)** together.
- Makes programs more **modular, reusable, and maintainable**.

Key Concepts of OOP

Class

- A **blueprint or template** for creating objects.
- Defines **attributes (data members)** and **methods (functions)**.

Example:

```
class Student {
public:
   string name;
   int age;

   void display() {
      cout << "Name: " << name << ", Age: " << age << endl;
   }
};
```

Object

- An **instance of a class**.
- Each object has its **own copy of data members**.

Example:

```
Student s1;   // object s1
s1.name = "Bharat";
s1.age = 20;
s1.display();
```

Encapsulation

- **Wrapping data and functions together** in a class.
- **Hides internal details** from outside (data hiding).
- Access controlled by **access specifiers**: private, public, protected.

Example:

```
class Account {
```

```
private:
   double balance;   // hidden from outside

public:
   void setBalance(double b) { balance = b; }
   double getBalance() { return balance; }
};
```

Abstraction

- **Hiding unnecessary details** and showing only the **essential features**.
- Achieved using **classes and interfaces**.

Example:

- User interacts with ATM interface without knowing **internal workings**.

Inheritance

- **Deriving a new class (child)** from an **existing class (parent)**.
- Child class **inherits properties and methods** of parent class.
- Supports **code reusability**.

Example:

```
class Person {
public:
   string name;
};

class Student : public Person {   // Student inherits from Person
public:
   int rollNo;
};
```

Polymorphism

- Ability of an object or function to **take many forms**.

Types:

1. **Compile-time (Method Overloading / Operator Overloading)**
2. **Run-time (Virtual Functions / Method Overriding)**

Example:

```
class Print {
public:
   void show(int i) { cout << "Integer: " << i << endl; }
   void show(string s) { cout << "String: " << s << endl; }  // same function name, different parameters
};
```

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

Ans :

Class

Definition:
A **class** is a **blueprint or template** that defines the **data (attributes)** and **functions (methods)** for objects.

- It **does not occupy memory** until an object is created.

Syntax:

```
class ClassName {
   // data members (attributes)
   // member functions (methods)
};
```

Object

Definition:
An **object** is an **instance of a class**.

- Each object has its **own copy of the data members** defined in the class.
- Objects **occupy memory**.

Syntax:

```
ClassName objectName;
```

Example of Class and Object

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

```
// Class definition
class Student {
public:
    string name;   // data member
    int age;       // data member

    // member function
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    // Object creation
    Student s1;

    // Accessing data members
    s1.name = "Bharat";
    s1.age = 20;

    // Calling member function
    s1.display();

    // Another object
    Student s2;
    s2.name = "Dost";
    s2.age = 22;
    s2.display();

    return 0;
```

}

Output:

Name: Bharat, Age: 20

Name: Dost, Age: 22

Explanation:

1. **Class Student** → defines **name, age**, and a **display function**.
2. **Objects s1 and s2** → separate instances with their own **data values**.
3. **Member function display()** → shows the values of data members for each object.

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

Ans :

Definition:
Inheritance is an **OOP concept** where a **derived (child) class inherits properties and methods** from a **base (parent) class**.

Purpose:

- Promotes **code reuse**.
- Helps in creating **hierarchical relationships**.
- Makes programs more **modular and maintainable**.

Syntax (Single Inheritance):

```
class Base {
  // base class members
};


class Derived : access_specifier Base {
  // derived class members
};
```

- access_specifier → public, protected, private (usually public)

Single Inheritance

Definition:
A derived class inherits from **one base class only**.

Example:

```
#include <iostream>
```

```
using namespace std;

class Person {      // Base class
public:
   string name;
   void displayName() {
      cout << "Name: " << name << endl;
   }
};

class Student : public Person {  // Derived class
public:
   int rollNo;
   void displayRoll() {
      cout << "Roll Number: " << rollNo << endl;
   }
};

int main() {
   Student s1;
   s1.name = "Bharat";   // inherited from Person
   s1.rollNo = 101;      // own member

   s1.displayName();     // inherited function
   s1.displayRoll();     // own function
   return 0;
}
```

Output:

Name: Bharat

Roll Number: 101

Explanation:

- Student inherits name and displayName() from Person.
- Student can use both inherited and own members.

Multiple Inheritance

Definition:
A derived class inherits from **more than one base class**.

Example:

```
#include <iostream>

using namespace std;


class Person {

public:

   string name;

};


class Marks {

public:

   int score;

};


class Student : public Person, public Marks {  // inherits from 2 base classes

public:

   void display() {

      cout << "Name: " << name << ", Score: " << score << endl;

   }

};


int main() {

   Student s1;

   s1.name = "Dost";

   s1.score = 95;
```

```
    s1.display();
    return 0;
}
```

Output:

Name: Dost, Score: 95

Explanation:

- Student inherits **name** from Person and **score** from Marks.
- Multiple inheritance allows combining features from multiple classes.

Multilevel Inheritance

Definition:
A class is derived from a derived class, forming a **chain**.

Example:

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

class Person {  // Base class
public:
    string name;
};

class Student : public Person {  // Derived from Person
public:
    int rollNo;
};

class Result : public Student {  // Derived from Student
public:
    float percentage;
    void display() {
        cout << "Name: " << name
```

```
        << ", Roll No: " << rollNo
        << ", Percentage: " << percentage << "%" << endl;
    }
};

int main() {
    Result r1;
    r1.name = "Bharat";
    r1.rollNo = 101;
    r1.percentage = 88.5;
    r1.display();
    return 0;
}
```

Output:

Name: Bharat, Roll No: 101, Percentage: 88.5%

Explanation:

- Result inherits from Student, which inherits from Person.
- Access to name, rollNo, and own member percentage is possible.

Hierarchical Inheritance

Definition:
Multiple derived classes inherit from **one base class**.

Example:

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

class Person {  // Base class
public:
    string name;
};
```

```
class Student : public Person {
public:
    int rollNo;
};

class Teacher : public Person {
public:
    string subject;
};

int main() {
    Student s1;
    s1.name = "Dost";
    s1.rollNo = 101;

    Teacher t1;
    t1.name = "Mr. Sharma";
    t1.subject = "Math";

    cout << "Student Name: " << s1.name << ", Roll No: " << s1.rollNo << endl;
    cout << "Teacher Name: " << t1.name << ", Subject: " << t1.subject << endl;

    return 0;
}
```

Output:

Student Name: Dost, Roll No: 101

Teacher Name: Mr. Sharma, Subject: Math

Explanation:

- Both Student and Teacher inherit from Person.
- Shows how multiple derived classes can reuse the base class.

Key Points About Inheritance

1. **Access specifier** determines visibility in derived class.
2. **Code reusability** → avoids rewriting common features.
3. **"is-a" relationship** → derived class is a type of base class.
4. Derived classes can have **their own members** in addition to inherited members.

- **Single Inheritance** → one base class
- **Multiple Inheritance** → multiple base classes
- **Multilevel Inheritance** → chain of inheritance
- **Hierarchical Inheritance** → one base class, multiple derived classes

4) What is encapsulation in C++? How is it achieved in classes?

Ans :

Definition:
Encapsulation is an **OOP concept** that **wraps data (variables) and functions (methods) together into a single unit** — usually a **class**.

- It also **restricts direct access** to some of the object's data, providing **data hiding**.
- Only **authorized functions (public methods)** can access or modify private data.

Purpose:

1. **Data Hiding** → Protect internal data from outside interference.
2. **Controlled Access** → Provide getter and setter functions.
3. **Improved Security** → Prevent invalid or unwanted changes.

How Encapsulation is Achieved in C++

1. Using Access Specifiers in Classes:

Access Specifier	Description
private	Members cannot be accessed outside the class. Only accessible via class methods.
public	Members can be accessed from anywhere.
protected	Members accessible in derived classes.

- **Encapsulation** is achieved by **keeping data members private** and **using public member functions** to read or modify them.

Example of Encapsulation

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

class Account {
private:
   double balance;  // private data member

public:
   // Setter function to set balance
   void setBalance(double b) {
      if (b >= 0) {  // only allow positive balance
         balance = b;
      } else {
         cout << "Invalid balance!" << endl;
      }
   }

   // Getter function to get balance
   double getBalance() {
      return balance;
   }
};

int main() {
   Account acc1;

   acc1.setBalance(5000);        // set balance using setter
   cout << "Balance: " << acc1.getBalance() << endl;

   acc1.setBalance(-1000);       // invalid, cannot directly access private variable
```

```
    return 0;
}
```

Output:

Balance: 5000

Invalid balance!

Explanation:

1. balance is **private**, so it **cannot be accessed directly** outside the class.
2. setBalance() is a **public function** that **controls how balance is modified**.
3. getBalance() allows **safe access** to the balance value.
4. This ensures **data safety and integrity**.

Key Points

- **Encapsulation = Data + Methods in a class**
- Protects **data from unwanted access**.
- Achieved by **private/protected data members + public getter/setter methods**.
- Supports **modularity, maintainability, and security**.