Module - 3 Introduction to OOPS Programming

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

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Top-down	Bottom-up
Focus	Focuses on **functions** and procedures	Focuses on **objects** and **classes**
Data Handling	Data is **global** and can be accessed by any function	Data is **encapsulated** within objects and accessed through methods
Modularity	Uses functions to organize code	Uses **classes and objects** to organize code
Security	Less secure, as data is accessible globally	More secure, due to **encapsulation** and access control (public, private, protected)
Code Reusability	Limited reuse; functions can be reused but not data	High reusability through **inheritance** and **polymorphism**
Real-world Modeling	Harder to model real-world entities directly	Easier to model real-world entities using objects
Examples	C, Pascal, Fortran	C++, Java, Python (with OOP), C#

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

1. Encapsulation (Data Hiding)

OOP: Combines data and functions into a single unit (class), hiding internal details from the outside.
Advantage: Increases security and prevents accidental data modification.

Example: You can make variables `private` and control access using `get()` and `set()` methods.

2. Reusability (Using Inheritance)

OOP: Allows new classes to reuse properties and behavior of existing classes using **inheritance**.
Advantage: Saves development time and reduces code duplication.

Example: A `Car` class can inherit from a `Vehicle` class.

3. Modularity

OOP: Programs are divided into small, independent, reusable objects.
Advantage: Easy to understand, maintain, and debug. Each object can be developed and tested independently.

Example: You can update the User class logic without touching the rest of the system.

4. Polymorphism

OOP: Allows one function name or operator to behave differently based on context.
Advantage: Makes code more flexible and extensible.

Example: A draw() function can draw a circle, square, or triangle depending on the object.

5. Real-world Modeling

OOP: Directly represents real-world entities as objects.
Advantage: Easier to visualize, design, and map real-life problems into code.

Example: Student, BankAccount, Employee classes mirror real entities.

6. Maintainability and Scalability

OOP: Code is better organized and modular.
Advantage: Easier to maintain, extend, and scale large software systems.

7. Avoids Global Data Issues

OOP: Limits data access using access modifiers (private, protected, public).
Advantage: Prevents unwanted changes and reduces bugs due to shared data.

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

Steps to Set Up a C++ Development Environment:

1. Install a C++ Compiler

- **Windows:**
  - Install **MinGW** or **TDM-GCC** (GCC for Windows)
  - Or install **Microsoft Visual Studio** (includes MSVC compiler)
- **Linux:**
  - GCC is usually pre-installed
  - If not:

    sudo apt update

sudo apt install g++

- **Mac:**
  - Install Xcode Command Line Tools:

    xcode-select --install

2. Choose and Install a Code Editor or IDE

You can choose either a **simple code editor** or a **full IDE**:

- **Popular Editors:**
  - **Visual Studio Code** (VS Code)
  - **Sublime Text**
  - **Atom**
- **Popular IDEs:**
  - **Code::Blocks**
  - **Dev-C++**
  - **Eclipse CDT**
  - **Microsoft Visual Studio** (not VS Code — the full IDE)

IDEs have built-in compiler support, debugging tools, and GUI project management.

3. Configure the Compiler in Your Editor

- **For VS Code:**
  - Install C/C++ extension (by Microsoft)
  - Create tasks.json to define how to compile
  - Create launch.json for debugging
- **For Code::Blocks / Dev-C++:**
  - Compiler is pre-configured; just write code and run.

4. Write and Run Your First Program

Example: hello.cpp

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

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

To Compile and Run (Terminal):

```
g++ hello.cpp -o hello
./hello
```

5. Test and Debug Your Code

- Use the built-in debugger in IDEs like **Code::Blocks** or **Visual Studio**.
- In VS Code, install the **C++ Debugger Extension** and set up launch.json.

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

In **C++**, the main **input/output (I/O) operations** are performed using:

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

1. cout – Console Output

Used to display output on the screen.

Example:

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

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

endl is used to move to a new line.

2. cin – Console Input

Used to take input from the user.

Example:

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

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

3. cerr – Error Output

Used to print **error messages**. Output is **unbuffered**, which means it's printed immediately.

Example:

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

int main() {
   cerr << "An error occurred!" << endl;
   return 0;
}
```

4. clog – Log Output

Used for **logging/debugging messages**. Output is **buffered** (may not display immediately).

Example:

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

int main() {
   clog << "This is a log message." << endl;
   return 0;
}
```

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

1. Basic (Primitive) Data Types

Data Type	Description	Example
int	Stores integers	int a = 10;

Data Type	Description	Example
float	Stores decimal numbers (single precision)	float b = 3.14f;
double	Stores decimal numbers (double precision)	double c = 3.14159;
char	Stores a single character	char d = 'A';
bool	Stores Boolean values (true/false)	bool e = true;

2. Derived Data Types

Type	Description	Example
Array	Collection of fixed-size elements of same type	int arr[5] = {1, 2, 3, 4, 5};
Pointer	Stores memory address of another variable	int* ptr = &a;
Function	Block of code that performs a task	int sum(int x, int y);
Reference	Alias for another variable	int& ref = a;

3. User-Defined Data Types

Type	Description	Example
struct	Groups variables of different types	struct Student { int id; char name[20]; };
class	Blueprint for objects (OOP)	class Car { public: int speed; };
union	Stores one of many data types in same memory location	union Data { int i; float f; };
enum	Enumerated constants	enum Color { RED, GREEN, BLUE };

4. Void Type

- **void** means no value or no return type.
- Commonly used in functions that don't return anything.

Example:

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

5. Modifiers for Data Types

Modifiers alter the size or range of basic data types:

Modifier	Used with	Purpose
signed	int, char	Allows negative values (default)
unsigned	int, char	Only positive values
short	int	Smaller range
long	int, double	Larger range

Example:

```
unsigned int x = 100;
long double pi = 3.1415926535;
```

Example Code Using Various Types:

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

int main() {
   int age = 20;
   float height = 5.9f;
   char grade = 'A';
   bool passed = true;
   double marks = 85.67;

   cout << "Age: " << age << endl;
   cout << "Height: " << height << endl;
   cout << "Grade: " << grade << endl;
   cout << "Passed: " << passed << endl;
   cout << "Marks: " << marks << endl;

   return 0;
}
```

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

1.Implicit Type Conversion

Also known as **automatic type conversion**, it happens **automatically** when:

- A smaller data type is assigned to a larger data type.
- There's no risk of data loss.

Example:

```
int a = 10;
float b = a;  // Implicit conversion from int to float
```

The compiler converts int to float automatically.

Characteristics of Implicit Conversion:

- Performed by the compiler.
- No need for programmer action.
- Usually goes from **lower to higher** precision (e.g., int → float → double).
- **Safe**, but can cause **precision loss** if not handled carefully.

2. Explicit Type Conversion

This happens **manually** using **cast operators**. You tell the compiler what type to convert to.

Example:

```
float f = 5.75;
int x = (int)f;  // Explicit conversion (type casting)
```

Or using C++ style:

```
int x = static_cast<int>(f);
```

Characteristics of Explicit Conversion:

- Performed **manually** by the programmer.
- Needed when:
    - You want to force conversion between incompatible types.
    - You want to avoid implicit conversion that might cause unexpected results.
- May result in **data loss** (e.g., truncating decimals when converting float to int).

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

1. Arithmetic Operators

Used for basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

Example:

```
int a = 10, b = 3;
cout << a + b;  // Output: 13
```

2. Relational (Comparison) Operators

Used to compare values. Result is true or false.

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

3. Logical Operators

Used to combine or invert Boolean expressions.

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

4. Assignment Operators

Used to assign values.

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

5. Increment and Decrement Operators

Operator	Description	Example
++	Increment by 1	a++, ++a
--	Decrement by 1	a--, --a

++a is **pre-increment**, a++ is **post-increment**

6. Bitwise Operators

Operate at the binary level.

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

7. Conditional (Ternary) Operator

A shortcut for if-else.

(condition) ? value_if_true : value_if_false

Example:

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

8. Sizeof Operator

Returns the size (in bytes) of a variable or data type.

Example:

```
cout << sizeof(int);  // Output: usually 4
```

9. Type Cast Operator

Converts one type to another.

Example:

```
float f = 5.5;
int x = (int)f;  // Explicit casting
```

10. Pointer Operators

Used with pointers.

Operator	Description	Example
*	Dereference operator	*ptr
&	Address-of operator	&var

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

Constants are variables whose **value is fixed** and **cannot be changed** after initialization.

Why Use Constants?

- Improves **code readability**.
- Prevents **accidental changes**.
- Makes code **easier to maintain**.

Declaring Constants:

1. Using const keyword

```
const int MAX_USERS = 100;
```

Once declared, MAX_USERS cannot be changed.

2. Using #define preprocessor directive

```
#define PI 3.14159
```

Preprocessor replaces all instances of PI before compilation.

What are Literals?

Literals are **fixed values** directly used in the code.

Example:

```
int a = 10;       // 10 is an integer literal
char ch = 'A';    // 'A' is a character literal
float pi = 3.14f;  // 3.14f is a float literal
```

Types of Literals in C++

Type	Example	Description
Integer	10, 0x1F, 075	Decimal, hexadecimal, octal
Floating-point	3.14, 1.2e3	Decimal or exponential form
Character	'A', '9'	Enclosed in single quotes
String	"Hello"	Sequence of characters in double quotes
Boolean	true, false	Represents logical values
Null pointer	nullptr	Special literal for null pointer

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

conditional statements allow your program to **make decisions** and **execute certain blocks of code** based on conditions. They **control the flow** of the program based on conditions (expressions that evaluate to true or false).

1. if Statement

Executes a block of code 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." << endl;
```

2. if-else Statement

Executes one block if the condition is true, and another if it's false.

Syntax:

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

Example:

```
int marks = 40;
if (marks >= 50) {
   cout << "You passed." << endl;
} else {
   cout << "You failed." << endl;
}
```

4. switch Statement

Selects one of many code blocks to be executed based on a **single variable's value** (typically int or char).

Syntax:

```
switch (expression) {
   case value1:
      // Code
      break;
   case value2:
      // Code
      break;
   default:
      // Code if 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";
}
```

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

1. for Loop

Use When:

You know **exactly how many times** you want to loop.

Syntax:

```
for (initialization; condition; update) {
   // Code to repeat
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
   cout << i << " ";
}
// Output: 1 2 3 4 5
```

2. while Loop

Use When:

You **don't know** how many times to loop and want to **check the condition first**.

Syntax:

```
while (condition) {
   // Code to repeat
}
```

Example:

```
int i = 1;
```

```
while (i <= 5) {
   cout << i << " ";
   i++;
}
// Output: 1 2 3 4 5
```

3. do-while Loop

Use When:

You want the loop to run **at least once**, even if the condition is false at the beginning.

Syntax:

```
do {
   // Code to repeat
} while (condition);
```

Example:

```
int i = 1;
do {
   cout << i << " ";
   i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

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

1. break Statement

Purpose:

Used to **immediately exit** the loop or `switch` block, regardless of the condition.

Syntax:

```
if (condition) {
   break;
}
```

Example (inside a loop):

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

int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5)
         break;  // Loop exits when i == 5
      cout << i << " ";
   }
   return 0;
}
// Output: 1 2 3 4
```

2. continue Statement

Purpose:

Skips the current iteration of the loop and moves to the **next iteration**.

Syntax:

```
if (condition) {
   continue;
}
```

Example:

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

int main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3)
         continue;  // Skip when i == 3
      cout << i << " ";
   }
   return 0;
}
// Output: 1 2 4 5
```

(4)Explain nested control structures with an example.

Nested control structures are control statements (like if, for, while, switch) placed **inside one another**. This allows for **complex decision-making** or **multiple levels of iteration**.

Common Nested Structures:

- if inside if → **Nested if**
- if inside for, for inside for, etc.
- switch inside if, or vice versa

Example 1: Nested if Statement

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

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

   if (age >= 18) {
      if (gender == 'M')
         cout << "You are an adult male." << endl;
      else
         cout << "You are an adult female." << endl;
   } else {
      cout << "You are a minor." << endl;
   }
   return 0;
}
```

Example 2: Nested for Loops (for printing a pattern)

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

int main() {
   for (int i = 1; i <= 3; i++) {
      for (int j = 1; j <= 4; j++) {
         cout << "* ";
      }
      cout << endl;
   }
   return 0;
}
```

Output:

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

Example 3: if inside a for Loop

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

int main() {
   for (int i = 1; i <= 10; i++) {
      if (i % 2 == 0)
         cout << i << " is even" << endl;
      else
         cout << i << " is odd" << endl;
   }
   return 0;
}
```

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

A **function** is a **block of code** that performs a **specific task**. It helps make programs **modular**, **reusable**, and **easier to manage**.

Purpose of Functions:

- Avoid code repetition
- Make code cleaner and easier to debug
- Divide a large problem into smaller manageable parts

Types of Functions:

1. **Predefined (Built-in) Functions** → e.g., cout, sqrt(), strlen()
2. **User-defined Functions** → Functions created by the programmer

Three Main Parts of a Function:

1. Function Declaration (Prototype)

Tells the compiler about the function's **name**, **return type**, and **parameters** (no body).

return_type function_name(parameter_list);

Example:

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

2. Function Definition

Contains the **actual code** of the function.

```
return_type function_name(parameter_list) {
   // Function body
}
```

Example:

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

3. Function Call

Used to **execute** the function from main() or another function.

```
function_name(arguments);
```

Example:

```
int result = add(5, 3);
```

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

Scope refers to the **region of the program** where a variable is **accessible** (visible and usable). In C++, variable scope determines how and where variables can be used.

Types of Scope in C++:

1. Local Scope

2. Global Scope

Also includes **block scope**, **function scope**, and **class scope**, but the focus here is local vs global.

1. Local Variables

- Declared **inside a function**, block ({}), or loop.

- **Only accessible within that block.**
- Destroyed when the block ends.

Example:

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

void show() {
   int x = 5;  // local to function 'show'
   cout << "Local x: " << x << endl;
}
```

x is **not accessible** outside show().

2. Global Variables

- Declared **outside all functions**, usually at the top.
- Accessible from **anywhere in the program** after declaration.
- Lives throughout the program's life.

Example:

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

int x = 10;  // global variable

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

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

Both display() and main() can access x.

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

Recursion is a programming technique where a **function calls itself** to solve smaller instances of the same problem.

Recursion = **Breaking down a big problem into smaller sub-problems**.

Key Concepts:

1. **Base Case** – Condition to **stop** recursion (must have!)
2. **Recursive Case** – The part where the function **calls itself**

Example: Factorial Using Recursion

Code:

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

// Function to calculate factorial recursively
int factorial(int n) {
   if (n == 0)     // Base case
      return 1;
   else
      return n * factorial(n - 1);  // Recursive case
}

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

Output:

```
Factorial of 5 is 120
```

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

A **function prototype** is a **declaration** of a function **before its actual definition**. It tells the compiler:

- The **function name**
- **Return type**
- The **number and type of parameters**

But it **doesn't contain the body** of the function.

Syntax of a Function Prototype:

return_type function_name(parameter_list);

Example:

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

Why Are Function Prototypes Used?

Purpose	Explanation
Inform the compiler	Allows calling a function **before it's defined** in the code
Enable modular programming	Keeps main code clean and organized
Type checking	Ensures correct number and type of arguments
Supports multiple files	Useful in large projects with multiple .cpp and .h files

Example Without Function Prototype:

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

int main() {
   cout << add(5, 3);  // Error: 'add' not declared yet
   return 0;
}

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

Example With Function Prototype:

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

// Function Prototype
```

```
int add(int, int);

int main() {
   cout << add(5, 3);  // OK
   return 0;
}

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

Output:

8

5.(1) 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**. Arrays allow storing **multiple values** under a single variable name, using **indexing**.

Declaring an Array

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

Example:

```
int numbers[5];        // declares an array of 5 integers
float marks[3] = {90.5, 85.2, 88.0};  // initializes array with values
```

Accessing Array Elements

Array elements are accessed using **zero-based indexing**:

```
cout << marks[0];  // prints 90.5
marks[1] = 92.3;   // updates second element
```

Types of Arrays

1. Single-Dimensional Array

2. Multi-Dimensional Array

1. Single-Dimensional Array

- A **linear list** of elements.
- One index is used to access elements.

Example:

```
int arr[4] = {10, 20, 30, 40};
cout << arr[2];  // Output: 30
```

2. Multi-Dimensional Array

- Stores data in **tables**, **matrices**, or **grids**.
- Most common is the **2D array** (rows and columns).
- Accessed using **two or more indices**.

Declaration:

```
int matrix[2][3];  // 2 rows and 3 columns
```

Initialization:

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

Access:

```
cout << matrix[1][2];  // Output: 6
```

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

In C++, **strings** are used to **store and manipulate text**. You can handle strings in **two ways**:

2. C++ string Class (Recommended)

Modern and easier way to handle strings using the **<string>** header.

Declaration:

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

Input/Output:

```
string fullName;
getline(cin, fullName);   // reads full line with spaces
cout << fullName;
```

Useful string Member Functions:

Function	Description	Example
.length() or .size()	Length of string	str.length()
.append()	Add to end	str.append("XYZ")
.substr()	Get substring	str.substr(2, 3)
.find()	Find position	str.find("abc")
.replace()	Replace part	str.replace(0, 3, "Hi")
.compare()	Compare strings	str1.compare(str2)

Example: C++ string Class

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

int main() {
   string name = "C++ Programming";

   cout << "Length: " << name.length() << endl;
   cout << "Substring: " << name.substr(4, 6) << endl;
   cout << "Position of 'gram': " << name.find("gram") << endl;

   return 0;
}
```

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

1. One-Dimensional (1D) Arrays

Syntax:

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

Examples:

1.1 Initializing with values:

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

1.2 Partial initialization (rest become 0):

```
int scores[5] = {90, 80};  // scores[2], [3], [4] = 0
```

1.3 Letting compiler decide size:

```
int data[] = {1, 2, 3};  // size = 3 automatically
```

Accessing 1D Array:

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

2. Two-Dimensional (2D) Arrays

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

Syntax:

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

Examples:

2.1 Full initialization:

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

2.2 Partial initialization:

```
int table[2][3] = {
   {1},    // rest auto-filled with 0
   {7, 8}
};
```

Accessing 2D Array:

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

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

2. C++ string Class

Header: #include <string>

Declaration:

string name = "OpenAI";

Useful String Operations:

Operation	Function / Syntax	Example
Length	str.length() or str.size()	name.length() → 6
Concatenation	+ or .append()	str1 + str2 or str1.append(str2)
Substring	str.substr(pos, len)	name.substr(0, 4) → "Open"
Find	str.find("text")	name.find("AI") → 4
Replace	str.replace(pos, len, "new")	Replace part of string
Compare	str1.compare(str2)	0 if equal, >0 or <0 if different
Access character	str[index]	name[0] → O
Clear	str.clear()	Empties the string

Operation	Function / Syntax	Example
Empty	str.empty()	Returns true if string is empty

Example:

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

int main() {
   string text = "C++ Programming";

   cout << "Length: " << text.length() << endl;
   cout << "Substring (4–6): " << text.substr(4, 6) << endl;
   cout << "Find 'gram': " << text.find("gram") << endl;

   text.replace(0, 3, "Java");  // Replace "C++" with "Java"
   cout << "After Replace: " << text << endl;

   return 0;
}
```

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

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **"objects"**, which contain **data (attributes)** and **functions (methods)**. OOP makes code more **modular, reusable, and maintainable**.

Four Main Pillars of OOP:

1. Encapsulation

Bundling of data and functions into a single unit (class), and **restricting direct access** to some components.

- Achieved using **classes** and **access specifiers** like private, public, and protected.
- Protects data from outside interference.

Example:

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

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

2. Abstraction

Hiding complex internal details and showing **only the necessary features**.

- Achieved using **classes**, **access specifiers**, and **abstract classes**.
- Simplifies usage and enhances security.

Example:
You drive a car using a steering wheel and pedals, but don't need to know how the engine works internally.

3. Inheritance

One class (**child**) can inherit **properties and methods** from another class (**parent**).

- Promotes **code reuse**.
- Types: single, multilevel, multiple, hierarchical, hybrid.

Example:

```
class Animal {
public:
   void eat() { cout << "Eating..."; }
};

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

4. Polymorphism

Same function or method behaves **differently** based on context.

- **Compile-time (static)** polymorphism → Function overloading, Operator overloading

- **Run-time (dynamic)** polymorphism → Function overriding using virtual functions

Example:

```
class Shape {
public:
   virtual void draw() { cout << "Drawing shape\n"; }
};

class Circle : public Shape {
public:
   void draw() override { cout << "Drawing circle\n"; }
};
```

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

In **C++**, **classes and objects** are the core building blocks of **Object-Oriented Programming (OOP)**.

Class

A **class** is a **user-defined data type**. It acts as a **blueprint** for creating **objects**. It defines the **data members (variables)** and **member functions (methods)** that operate on that data.

Syntax:

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

Object

An **object** is an **instance of a class**. It has its own copy of class data and can access class functions.

Example: Class and Object in C++

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

// Class Declaration
class Student {
```

```
public:
   // Data members
   string name;
   int age;

   // Member function
   void displayInfo() {
      cout << "Name: " << name << endl;
      cout << "Age: " << age << endl;
   }
};

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

   // Assigning values
   s1.name = "Ravi";
   s1.age = 20;

   // Function call using object
   s1.displayInfo();

   return 0;
}
```

Output:

```
Name: Ravi
Age: 20
```

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

Inheritance in C++ is a key concept of **Object-Oriented Programming (OOP)** that allows a class (called a **derived class** or **child class**) to acquire the properties and behaviors (data members and member functions) of another class (called a **base class** or **parent class**).

This promotes **code reusability**, **hierarchical classification**, and **extensibility**.

Syntax of Inheritance:

```
class BaseClass {
```

```
  // members of base class
};

class DerivedClass : accessSpecifier BaseClass {
  // members of derived class
};
```

- accessSpecifier can be public, private, or protected.
    - Most commonly used: public

Example of Inheritance:

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

// Base class
class Animal {
public:
   void eat() {
      cout << "This animal eats food." << endl;
   }
};

// Derived class
class Dog : public Animal {
public:
   void bark() {
      cout << "The dog barks." << endl;
   }
};

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

Output:

```
This animal eats food.
The dog barks.
```

Types of Inheritance in C++:

1. **Single Inheritance** – One base and one derived class.
2. **Multiple Inheritance** – One derived class inherits from multiple base classes.
3. **Multilevel Inheritance** – A class is derived from a class which is also derived from another class.
4. **Hierarchical Inheritance** – Multiple derived classes from a single base class.
5. **Hybrid Inheritance** – Combination of multiple types.

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

Encapsulation is one of the fundamental concepts in **Object-Oriented Programming (OOP)**. It refers to the **binding of data (variables)** and **functions (methods)** that operate on that data into a **single unit (class)**, and restricting direct access to some of the object's components.

This helps to:

- **Hide internal details** of how an object works (data hiding).
- **Protect data** from unauthorized access or modification.
- Promote **modularity and maintainability** of code.

How is Encapsulation Achieved in C++?

Encapsulation is achieved using **classes** and **access specifiers**:

- **private**: Members are accessible only within the class.
- **public**: Members are accessible from outside the class.
- **protected**: Members are accessible in the class and its derived classes.

By keeping data **private** and providing **public methods (getters/setters)** to access or modify it, encapsulation is implemented.

Example of Encapsulation:

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

class Student {
private:
   int rollNo;
   string name;

public:
   // Setter for rollNo
```

```
    void setRollNo(int r) {
        rollNo = r;
    }

    // Getter for rollNo
    int getRollNo() {
        return rollNo;
    }

    // Setter for name
    void setName(string n) {
        name = n;
    }

    // Getter for name
    string getName() {
        return name;
    }
};

int main() {
    Student s;
    s.setRollNo(101);
    s.setName("Vrushti");

    cout << "Roll No: " << s.getRollNo() << endl;
    cout << "Name: " << s.getName() << endl;

    return 0;
}
```

Output:

```
Roll No: 101
Name: Vrushti
```

Key Benefits of Encapsulation:

- **Data Security**: Prevents accidental changes to critical data.
- **Controlled Access**: Access to data is provided through public methods.
- **Code Maintainability**: Internal code can change without affecting external code using the class.