Module 3 Introduction to OOPS Programming

- Introduction to C++

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

Aspect	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Function-oriented	Object-oriented
Basic Unit	Functions/Procedures	Objects/Classes
Focus	On functions (what to do)	On data + methods (who is doing)
Data Handling	Data is global and shared by functions	Data is encapsulated inside objects
Security	Less secure (any function can access data)	More secure (data hiding with access specifiers)
Reusability	Low (code needs rewriting)	High (using inheritance, polymorphism)
Examples	C, Pascal	C++, Java, Python

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

   - Advantages of OOP over POP

i. **Modularity** – Code is divided into classes and objects.
ii. **Reusability** – Inheritance allows reuse of existing code.
iii. **Security** – Data hiding and encapsulation protect data.
iv. **Flexibility** – Polymorphism allows using same function in many ways.
v. **Easy Maintenance** – Programs are easier to modify and extend.
vi. **Real-world Mapping** – Objects represent real-life entities.

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

   - Here are the steps in setting up a C++ development environment:

1. **Install a C++ compiler** (e.g., GCC, MinGW, MSVC).
2. **Install an IDE or editor** (e.g., Code::Blocks, Dev C++, VS Code).
3. **Link/Configure the compiler** with the IDE.
4. **Write a C++ program** (save with .cpp extension).
5. **Compile the program** using IDE or command line.
6. **Run the program** to see the output.

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

In C++, input and output are performed using **streams**. The most commonly used objects are:

1. **cin** → Standard input (takes data from keyboard).
2. **cout** → Standard output (displays data on screen).

All are defined in the **<iostream>** header.

- **Output using cout:**

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

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

2. **Input Operation (**cin**)**

- Used to take input from the user.
- Syntax:

  cin >> variable;

- Example:

```
#include <iostream>

using namespace std;

int main() {

   int number;
```

```
    cout << "Enter a number: ";
    cin >> number;
    cout << "You entered: " << number << endl;
    return 0;
}
```

- Variables, Data Types, and Operators

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

- In C++, **data types** define the type of data that a variable can hold. They tell the compiler how much memory to allocate and what operations are allowed.
- Basic (Primitive) Data Types:

Data Type	Description	Example
int	Stores integer numbers (whole numbers)	int age = 25;
float	Stores decimal numbers (single precision)	float price = 99.99;
double	Stores decimal numbers (double precision, more accurate than float)	double distance = 1234.56789;
char	Stores a single character	char grade = 'A';
bool	Stores Boolean values: true or false	bool isPassed = true;
void	Represents no value or no return type (used in functions)	void display() { ... }

Example:

```
#include <iostream>
using namespace std;
int main() {
   int age = 25;
```

```
    float price = 99.99;
    double distance = 1234.56789;
    char grade = 'A';
    bool isPassed = true;
    cout << "Age: " << age << endl;
    cout << "Price: " << price << endl;
    cout << "Distance: " << distance << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << isPassed << endl;
    return 0;
}
```

2. Derived Data Types

These are created from basic types:

Array: Collection of elements of the same type

```
int marks[5] = {90, 85, 78, 92, 88};
```

Pointer: Stores the address of another variable.

```
int x = 10;
int *ptr = &x;
```

3. User-Defined Data Types

These allow you to define your own types:

Class/Struct: To define objects with attributes and methods.

```
struct Student {
    string name;
```

```
    int rollNo;
};
```

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

In C++, **type conversion** means changing one data type into another.

There are two types: implicit and explicit.

1. Implicit Type Conversion (Type Casting done by compiler)

- Also called Type Promotion / Type Casting automatically**.**
  - Done automatically by the compiler.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int a = 10;
    double b = a;       // int → double (implicit)
    cout << b;          // Output: 10
}
```

2. Explicit Type Conversion (Type Casting done by programmer)

- Also called Type Casting (manual)**.**
- Done manually by the programmer.
- Uses **type cast operator**.

```
#include <iostream>
using namespace std;
int main() {
```

```
    double pi = 3.14;
    int x = (int)pi;        // explicit cast double → int
    cout << x;              // Output: 3
}
```

- Difference Between Implicit and Explicit Type Conversion

Feature	Implicit Conversion	Explicit Conversion
Who performs it?	Compiler (automatic)	Programmer (manual)
Also called	Type promotion / type casting	Type casting (manual)
Syntax	No special syntax	`(datatype) variable`
Example	`int a = 10; double b = a;`	`int x = (int)3.14;`

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

- Operators are **symbols** used to perform operations on variables and values.

1. Arithmetic Operators

- Used for mathematical operations.

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

Operator	Description	Example
`--`	Decrement	`a--` or `--a`

EXAMPLE:

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

cout << a + b;  // 13 (Addition)

cout << a - b;  // 7  (Subtraction)

cout << a * b;  // 30 (Multiplication)

cout << a / b;  // 3  (Division - integer)

cout << a % b;  // 1  (Modulus)
```

2. Relational Operators

- Used to compare 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);     // 0 (false)

cout << (a != b);     // 1 (true)
```

```
cout << (a > b);     // 0

cout << (a < b);     // 1

cout << (a >= b);   // 0

cout << (a <= b);   // 1
```

3. Logical Operators

Used to combine multiple conditions. Result is **true (1)** or **false (0)**.

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 < b && a > 0); // 1 (true)

cout << !(a > b); // 1 (true)
```

4.Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	`a = b`
+=	Add and assign	`a += b` (a = a + b)
-=	Subtract and assign	`a -= b`
*=	Multiply and assign	`a *= b`
/=	Divide and assign	`a /= b`
%=	Modulus and assign	`a %= b`

Example:

```
int a = 10;
a += 5;   // 15
a -= 3;   // 12
a *= 2;   // 24
a /= 4;   // 6
a %= 5;   // 1
```

5.Increment & Decrement Operators

```
int a = 5;
cout << (++a) << endl; // Pre-increment → 6
cout << (a++) << endl; // Post-increment → 6 (a=7 now)
cout << (--a) << endl; // Pre-decrement → 6
cout << (a--) << endl; // Post-decrement → 6 (a=5 now)
```

5. Bitwise Operators

- Work on bits (0/1).

Example:

```
int a = 5, b = 3;                 // a=0101, b=0011
cout << (a & b) << endl;          // AND → 1
cout << (a | b) << endl;          // OR  → 7
cout << (a ^ b) << endl;          // XOR → 6
cout << (~a) << endl;             // NOT → -6
cout << (a << 1) << endl;         // Left shift → 10
cout << (a >> 1) << endl;         // Right shift → 2
```

6.Conditional (Ternary) Operator

Example:

```
int a = 10, b = 20;

int max = (a > b) ? a : b;

cout << "Maximum: " << max << endl;                    // 20
```

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

<u>Constants:</u>

- A constant is a value that cannot be changed during program execution.
- They are fixed values that remain the same throughout the program.
- Constants improve readability, reliability, and maintainability of code.

Types of Constants

1.Integer Constants:

Example:

const int age = 18;

2.Floating-point Constants:

Example:

const float pi = 3.14;

3. Character Constants:

Example:

const char grade = 'A';

Literals:

- A literal is the **actual fixed value** written directly in the program.
- Literals are used to represent constants directly in the code.

Types of Literals

1. Integer Literal → 10, -25, 0
2. Floating-point Literal → 3.14, -0.005, 2.5E3
3. Character Literal → 'A', '9', '$'

---

3. Control Flow Statements

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

- Conditional statements are used to make decisions in a program.
- They allow execution of certain blocks of code only if a given condition is true.
- They provide control over the flow of execution.

1. if-else Statement

Syntax:

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

- The if block runs when the condition is **true**.
- The else block runs when the condition is **false**.

2. if-else Ladder

When there are **multiple conditions**, we use else if.

Example:

```
if (marks >= 90)
   cout << "Grade A";
else if (marks >= 75)
   cout << "Grade B";
else if (marks >= 50)
   cout << "Grade C";
else
   cout << "Fail";
```

3. switch Statement

Syntax:

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

```
  ...
  default:
    // code block if no case matches
}
```

- The expression is evaluated once.
- It matches with one of the case constants.
- The corresponding block executes until a break is found.
- If no case matches, the default block runs.

---

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

➢ A loop is used to **execute a block of code repeatedly** until a condition is satisfied.

In C++, we mainly use:

- for loop
- while loop
- do-while loop

1. for Loop:

Syntax:

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

Working:

- Initialization → runs once at start.
- Condition → checked before every iteration.
- Update → executed after each iteration.

2. while Loop

Syntax:

```
while(condition) {
```

```
    // code to execute
}
```

3. do-while Loop

Syntax**:**

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

Working

- The body executes at least once**,** even if condition is false.
- After execution, condition is checked.

---

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

1. break statement

- The break statement is used to terminate the loop immediately.
- Control is transferred to the first statement after the loop.

Example – using break:

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;                  // exit loop when i = 5
        }
        cout << i<<"\n" ;
```

```
    }

    return 0;

}
```

2. **continue statement**

- The continue statement is used to skip the current iteration of the loop.
- Control moves to the next iteration of the loop.
- It does not terminate the loop, only skips one pass.

Example – using continue:

```
#include <iostream>

using namespace std;

int main() {

    for (int i = 1; i <= 10; i++) {

        if (i == 5) {

            continue;               // skip when i = 5

        }

        cout << i << " ";

    }

    return 0;

}
```

---

4. Explain nested control structures with an example.

- A nested control structure means placing one control structure (like if, for, while, or switch) inside another.
- Example: An if inside another if, or a for loop inside a while loop.

Example 1: Nested if statements

```
#include <iostream>
using namespace std;
int main() {
   int age = 20;
   char citizen = 'Y';
   if (age >= 18) { // Outer if
      if (citizen == 'Y') { // Inner if
         cout << "You are eligible to vote." << endl;
      } else {
         cout << "You must be a citizen to vote." << endl;
      }
   } else {
      cout << "You must be at least 18 years old to vote." << endl;
   }
   return 0;
}
```

Example 2: Nested for loops (printing a pattern)

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 3; i++) {      // Outer loop (rows)
      for (int j = 1; j <= 3; j++) {   // Inner loop (columns)
         cout << i ;
      }
      cout << endl;  // Move to next row
   }
   return 0;
}
```

4. Functions and Scope

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

- A **function** in C++ is a **block of code** designed to perform a specific task.
- It increases **reusability** (write once, use many times).
- Makes code **modular, readable, and maintainable**.

- Parts of a Function

A function in C++ has **three main parts**:

1. Function Declaration (Prototype)

- Tells the compiler about the function's name, return type, and parameters.
- Ends with a semicolon ;.
- Written before **main()** (or in a header file).

Syntax:

returnType functionName(parameter1Type, parameter2Type, ...);

Example:

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

2. Function Definition

- Contains the actual code (body) of the function.
- Specifies what the function does.

Syntax:

returnType functionName(parameters) {

// function body

return value; // if returnType is not void

}

Example:

int add(int a, int b) {

```
    return a + b;

}
```

3. Function Call

- Used to invoke (execute) the function.
- Written inside main() or another function.
- Passes arguments (values) to the function.

Example:

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

---

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

The scope of a variable refers to the region of the program where the variable can be accessed/used**.**

In C++, variable scope is mainly of two types:

1. Local Scope

- A variable declared inside a function, block, or loop**.**
- Exists only within that block and is destroyed once the block ends.
- Cannot be accessed outside its block.

Example – Local Variable:

```
#include <iostream>

using namespace std;

int main() {

    int x = 10;          // local to main()

    cout << "Inside main, x = " << x << endl;

    if (true)

 {

        int y = 20;      // local to if-block
```

```
    cout << "Inside if-block, y = " << y << endl;
  }
  // cout << y;                    // Error: y is not accessible here
  return 0;
}
```

2. Global Scope

- A variable declared outside all functions (generally at the top of the program).
- Accessible from any function in the program.
- Exists throughout the lifetime of the program.

Example – Global Variable:

```
#include <iostream>
using namespace std;
int g = 100;   // global variable
int main() {
  cout << " g = " << g << endl;
  return 0;
}
```

---

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

- Recursion is a programming technique where a function calls itself (directly or indirectly) to solve a problem.

Types of Recursion

1. Direct Recursion → Function calls itself directly.
2. Indirect Recursion → Function calls another function, which then calls the first function.

Example 1: Factorial using Recursion

```
#include <iostream>

using namespace std;

// Recursive function to calculate factorial

int factorial(int n) {

   if (n == 0 || n == 1) {   // Base case

      return 1;

   } else {

      return n * factorial(n - 1); // Recursive call

   }

}

int main() {

   int num = 5;

   cout << "Factorial of " << num << " = " << factorial(num) << endl;

   return 0;

}
```

---

4. 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 name
- The return type
- The number and type of parameters

Syntax:

returnType functionName(parameterType1, parameterType2, ...);

Example:

```
#include <iostream>

using namespace std;

// Function prototype (declaration)

int add(int, int);

int main() {

   int result = add(5, 10);   // Function call

   cout << "Sum = " << result << endl;

   return 0;

}

// Function definition

int add(int a, int b) {

   return a + b;

}
```

Why are Function Prototypes Used?

- **Tell the compiler** about the function before it is used.
- **Enable type checking** (correct number and type of arguments).
- **Allow calling functions before their definition**.
- **Improve code organization** (keep prototypes in header files, definitions elsewhere).

---

5. Arrays and Strings

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

An **array** in C++ is a **collection of elements of the same data type** stored in **contiguous memory locations**.

Each element is accessed using an **index** (starting from 0).

Example of an Array

```
#include <iostream>

using namespace std;

int main() {

   int marks[5] = {90, 85, 78, 92, 88};  // array of 5 integers

   cout << "First element = " << marks[0] << endl;

   cout << "Third element = " << marks[2] << endl;

   return 0;

}
```

Types of Arrays

1. Single-Dimensional Array

- A **linear list** of elements (like a row).
- Accessed using **one index**.
- Syntax:

  datatype arrayName[size];

2. Multi-Dimensional Array

- An array with more than one index (like a table, matrix, or cube).
- Most common is the two-dimensional array (rows × columns).

Syntax:

datatype arrayName[rows][columns];

- Difference Between Single-Dimensional and Multi-Dimensional Arrays:

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	Stores data in a linear form (like a list)	Stores data in a tabular/matrix form
Indexing	One index (arr[i])	More than one index (arr[i][j], arr[i][j][k])

Feature	Single-Dimensional Array	Multi-Dimensional Array
Example	int arr[5] = {1,2,3,4,5};	int mat[2][3] = {{1,2,3},{4,5,6}};
Use case	Useful for storing a simple list (marks, salaries)	Useful for storing tables, grids, matrices

---

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

A **string** in C++ is a **sequence of characters** used to represent text.

C++ provides **two ways** to handle strings:

1. **C-style strings** (character arrays, from C language).
2. **C++ strings** (using string class from Standard Template Library).

1. C-Style Strings (Character Arrays)

- Defined as an **array of characters** ending with a **null character '\0'**.

Example:

```
#include <iostream>

#include <cstring>

using namespace std;

int main() {

   char str1[20] = "Hello";

   char str2[20] = "World";

   // String handling functions

   cout << "Length of str1: " << strlen(str1) << endl;

   strcat(str1, str2);   // concatenation

   cout << "After concatenation: " << str1 << endl;

   cout << "Comparison: " << strcmp(str1, str2) << endl; // 0 if equal
```

```
    return 0;
}
```

2. C++ Strings (STL string class)

- Easier and safer than C-style strings.
- Defined in <string> header.
- Provides many built-in functions for string handling.

Example:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 = "Hello";
    string s2 = "World";
    cout << "Length of s1: " << s1.length() << endl;
    cout << "Concatenation: " << s1 + s2 << endl;
    cout << "Substring: " << s1.substr(1, 3) << endl;  // from index 1, length 3
    cout << "Comparison: " << (s1 == s2 ? "Equal" : "Not Equal") << endl;
    s1.append(" Everyone");
    cout << "After append: " << s1 << endl;
    return 0;
}
```

---

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

- **Initialization** means assigning values to an array **at the time of declaration** or later.

C++ supports initialization for both **1D (single-dimensional)** and **2D (multi-dimensional)** arrays.

1. Initialization of 1D Array

(a) Full Initialization

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

(b) Partial Initialization

Remaining elements are set to 0.

```
int arr[5] = {10, 20};   // {10, 20, 0, 0, 0}
```

2. Initialization of 2D Array

(a) Row-wise Initialization

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

(b) Single-Line Initialization

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

(c) Partial Initialization

- Remaining elements are set to 0.

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

---

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

- C++ string class (from <string> library)

(a) Declaration and Initialization

```
#include <iostream>
```

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


int main() {
   string s1 = "Hello";       // initialization
   string s2("World");        // another way
   string s3;                 // empty string
   s3 = "C++ Strings";        // assignment
   cout << s1 << " " << s2 << endl;
   cout << s3 << endl;
}
```

(b) Concatenation (+ and +=)

```
string a = "Good ";
string b = "Morning";
string c = a + b;    // Concatenation
a += b;              // Append
```

(c)Accessing Characters

```
string str = "C++";
cout << str[0];                // 'C'
```

(e) Comparison

```
string x = "apple", y = "banana";
if(x == y)
cout << "Equal";
else if(x < y)
```

```
    cout << "x comes first";
```

---

6. Introduction to Object-Oriented Programming

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

C++ is one of the most popular **OOP languages**.

The **four main pillars** of OOP are:

1. Encapsulation

a. Wrapping up of **data (variables)** and **functions (methods)** into a single unit called a **class**.
b. To **protect data** from outside interference and ensure controlled access.
c. Using **access specifiers** (private, protected, public).

EXAMPLE:

```
#include <iostream>

using namespace std;

class Student {

private:

   int marks;  // data hidden

public:

   void setMarks(int m) { marks = m; }   // controlled access

   int getMarks() { return marks; }

};



int main() {

   Student s;

   s.setMarks(95);      // encapsulated access
```

```
    cout << s.getMarks();

}
```

2. Abstraction

- Showing **only essential details** and hiding unnecessary implementation.
- To reduce complexity and increase clarity.
- Achieved using **abstract classes** and **interfaces (pure virtual functions)**.

Example:

```
#include <iostream>

using namespace std;

class Shape {

public:

    virtual void draw() = 0;  // Pure virtual function

};

class Circle : public Shape {

public:

    void draw() { cout << "Drawing Circle"; }

};

int main() {

        Circle c1;

        C1.draw();

}
```

3. Inheritance

The process by which one class **(**child/derived class) acquires the properties and behaviors of another class **(**parent/base class**).**

To re-use code and create a hierarchy of classes.

Types in C++**:**

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

4. Polymorphism

The ability of a function or object to take **many forms**.

To **achieve flexibility** in code.

Types:

- **Compile-time (Static) Polymorphism** → Function overloading, Operator overloading.
- **Run-time (Dynamic) Polymorphism** → Achieved using **virtual functions**.

Example 1: Function Overloading

```
#include <iostream>

using namespace std;

class Math {

public:

   int add(int a, int b)

{

 return a+b;

 }

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

};

int main() {

   Math m;
```

```
    cout << m.add(5, 3) << endl;
    cout << m.add(2.5, 3.1);
}
```

Example 2: Runtime Polymorphism

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void sound() { cout << "Animal sound\n"; }
};
class Dog : public Animal {
public:
    void sound() { cout << "Bark\n"; }
};
int main() {
    Animal* a;
    Dog d;
    a = &d;
    a->sound();   // Runtime polymorphism → Bark
}
```

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

1. Class

- A **class** is a **blueprint or template** for creating objects.

- It defines **data members (variables)** and **member functions (methods)** that operate on the data.

Syntax:

```
class ClassName {
 access_specifier:
   // data members (variables)
   // member functions (methods)
};
```

2.Object

An **object** is a **real instance** of a class.

Each object has **its own copy of the data members** defined in the class.

Example

```
#include <iostream>
using namespace std;
// Class definition
class Student {
public:
   string name;
   int age;
   void display() {         // Member function
      cout << "Name: " << name << endl;
      cout << "Age: " << age << endl;
   }
};
int main() {
   // Creating objects
   Student s1;
   Student s2;
```

```
    // Assign values to object s1
    s1.name = "Shifa";
    s1.age = 22;
    // Assign values to object s2
    s2.name = "Aman";
    s2.age = 20;
    // Display object data
    s1.display();
    s2.display();
    return 0;
}
```

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

- **Inheritance** is the process by which one class (**derived/child class**) **acquires properties and behaviors** of another class (**base/parent class**).
- It allows **code reusability** and **hierarchical classification**.
- **Base class (Parent class)** → The class whose properties are inherited.
- **Derived class (Child class)** → The class that inherits from the base class.

Types of Inheritance in C++

1. **Single inheritance** → One base, one derived class.
2. **Multiple inheritance** → One derived class inherits from multiple base classes.
3. **Multilevel inheritance** → Chain of inheritance (A → B → C).
4. **Hierarchical inheritance** → Multiple derived classes from one base.
5. **Hybrid inheritance** → Combination of above types.

Example of Single Inheritance:

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

// Base class
class Vehicle {
```

```
public:
    string brand = "Ford";
    void honk() {
        cout << "Beep beep!" << endl;
    }
};
// Derived class
class Car : public Vehicle {        // 'public' inheritance
public:
    string model = "Mustang";
};
int main() {
    Car myCar;
    // Accessing members of the base class
    cout << myCar.brand << endl;  // Output: Ford
    myCar.honk();                 // Output: Beep beep!
    // Accessing member of the derived class
    cout << myCar.model << endl;   // Output: Mustang
    return 0;
}
```

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

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

- Access to data is controlled using **access specifiers**:

- private: Only accessible within the class.
- protected: Accessible within the class and derived classes.

- public: Accessible from outside the class.

Example:

```
#include <iostream>
using namespace std;
class BankAccount {
private:
   double balance;  // private data member
public:
   // Setter function to set balance
   void setBalance(double amount) {
      if(amount >= 0) {
         balance = amount;
      } else {
         cout << "Invalid amount!" << endl;
      }
   }
   // Getter function to get balance
   double getBalance() {
      return balance;
   }
};
int main() {
   BankAccount account;
   account.setBalance(5000);  // Setting balance using setter
```

```
    cout << "Account balance: " << account.getBalance() << endl;  // Accessing using getter

    account.setBalance(-100);  // Invalid amount, cannot directly modify private data

    return 0;

}
```
