Module 3 - Introduction to OOPS Programming

1.Introduction to C++

Q1. What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?

Aspect	Procedural Programming	Object-Oriented Programming (OOP)
Definition	A programming paradigm where the focus is on procedures or routines (functions) that operate on data.	A programming paradigm where the focus is on objects that encapsulate data and behavior together.
Approach	Top-down approach. The program is divided into functions.	Bottom-up approach. The program is built around objects.
Data and Functions	Data and functions are separate. Functions manipulate global data.	Data and functions are bundled together into objects (class instances).
Code Reusability	Less reusable because functions are tied to global data.	High reusability through inheritance and polymorphism.
Encapsulation	Not supported explicitly. Data is accessible by any function.	Strong support for encapsulation. Data is hidden and only accessible through methods.
Abstraction	Limited abstraction, functions operate on data structures directly.	Provides abstraction by hiding implementation details and exposing only necessary interfaces.
Inheritance	Not supported. Code duplication is common.	Supported. New classes can inherit properties and methods from existing classes.
Polymorphism	Not supported.	Supported. Objects can take many forms, making code more flexible and extensible.

Q2. List and explain the main advantages of OOP over POP.

1. **Encapsulation**
    - In OOP, data (attributes) and functions (methods) are bundled together into objects.
    - This hides the internal state of the object from the outside world and exposes only what is necessary, making the program more secure and less error-prone.
2. **Code Reusability**
    - OOP promotes reuse of code through inheritance and polymorphism.
    - Once a class is written, it can be used by other classes without rewriting code, which saves time and effort.
3. **Modularity**
    - Programs are broken down into objects that can be developed, tested, and debugged independently.
    - This makes it easier to manage large codebases and enhances collaboration in team projects.
4. **Maintainability**
    - OOP code is easier to maintain and extend because new functionality can be added by creating new classes or extending existing ones.
    - Debugging and updating are simpler as changes in one object don't heavily affect others.
5. **Abstraction**
    - OOP allows programmers to hide unnecessary details and complexity from the user.
    - Developers can focus on high-level problem-solving without worrying about underlying complexities.
6. **Inheritance**
    - OOP enables new classes to inherit properties and behaviors from existing classes.
    - This avoids duplication of code and promotes hierarchical class structures.
7. **Polymorphism**

- Objects can take multiple forms, meaning the same interface can be used for different underlying data types.
- This makes the code more flexible and allows easier integration of new components.

8. **Improved Productivity**
   - The modular and reusable nature of OOP helps developers build applications faster and with fewer bugs.
9. **Better Data Management**
   - By binding data and behavior together, OOP provides better data management, ensuring that only authorized functions access or modify the data.
10. **Real-world Mapping**
    - OOP mimics real-world entities more closely, making it easier to model complex systems, understand relationships, and design interactive applications.

Q3. Explain the steps involved in setting up a C++ development environment.

1. Choose and Install a Compiler

A compiler translates your C++ code into machine-readable format.

- Popular C++ Compilers:
  - GCC / G++ (GNU Compiler Collection)
    - Linux: Usually pre-installed or available via package managers (sudo apt install g++)
    - Windows: Install via MinGW or WSL (Windows Subsystem for Linux)
  - Clang – alternative compiler, available for macOS and Linux.
  - MSVC (Microsoft Visual C++) – comes with Visual Studio on Windows.

➞ For Windows:
Download and install MinGW:

1. Go to https://www.mingw-w64.org/
2. Download the installer and choose architecture (x86_64).

3. Install and set the path environment variable to access the compiler from the command line.

➞ For macOS:
Install via Homebrew:

brew install gcc

➞ For Linux:
Use your package manager:

sudo apt update

sudo apt install build-essential

2. Choose an Integrated Development Environment (IDE) or Text Editor

- IDEs with built-in compiler and debugger:
  - Visual Studio (Windows)
  - Code::Blocks (cross-platform)
  - CLion (JetBrains, paid with trial version)
  - Dev-C++ (Windows)
- Lightweight editors (with extensions):
  - Visual Studio Code
  - Sublime Text
  - Atom

➞ Recommended:
For beginners, Code::Blocks or Visual Studio Code are good choices.

3. Set Up Environment Variables (Optional but Recommended)

- Add the compiler's bin directory to your system's PATH so you can run it from any terminal.

For example, on Windows:

1. Find where g++.exe is installed (C:\mingw64\bin).
2. Right-click This PC → Properties → Advanced system settings → Environment Variables.
3. Edit Path and add the directory.

Q4.What are the main input/output operations in C++? Provide examples.

1. cout – Output Operation

Used to display text, variables, and results to the console.

Example:

```
#include <iostream>

using namespace std;

int main() {

   cout << "Hello, World!" << endl;

   int number = 10;

   cout << "The number is: " << number << endl;

   return 0;

}
```

2. cin – Input Operation

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 are " << age << " years old." << endl;

   return 0;

}
```

2. Variables, Data Types, and Operators

Q1. What are the different data types available in C++? Explain with examples.

1. Fundamental (Basic) Data Types

These are the most commonly used data types.

Data Type	Description	Example
int	Stores whole numbers (integers)	int age = 25;
float	Stores decimal numbers (single precision)	float pi = 3.14f;
double	Stores decimal numbers (double precision)	double e = 2.71828;
char	Stores a single character	char grade = 'A';
bool	Stores Boolean values (true or false)	bool isOpen = true;
void	Represents absence of value (used in functions)	void myFunction();

Example:

```
#include <iostream>
using namespace std;
int main() {
  int age = 25;
  float pi = 3.14f;
  double e = 2.71828;
  char grade = 'A';
  bool isOpen = true;
  cout << "Age: " << age << endl;
  cout << "Pi: " << pi << endl;
  cout << "e: " << e << endl;
  cout << "Grade: " << grade << endl;
  cout << "Is Open? " << isOpen << endl;
  return 0;
}
```

2. Modified Data Types

These are variations of the fundamental types to store larger or more specific values.

Data Type	Description	Example
short	Stores small integers	short x = 32000;
long	Stores large integers	long population = 7800000000;
unsigned int	Stores only non-negative integers	unsigned int score = 100;

Example:

```
#include <iostream>
using namespace std;
int main() {
   short smallNumber = 32000;
   long largeNumber = 1000000000;
   unsigned int positiveNumber = 4000000000;
   cout << "Short: " << smallNumber << endl;
   cout << "Long: " << largeNumber << endl;
   cout << "Unsigned: " << positiveNumber << endl;
   return 0;
}
```

3. Derived Data Types

These are types that are based on fundamental types.

Data Type	Description	Example
array	A collection of elements of the same type	int numbers[3] = {1, 2, 3};
pointer	Stores the address of a variable	int *ptr = &age;
reference	An alias for another variable	int &ref = age;

Data Type	Description	Example
function	Defines reusable blocks of code	int add(int a, int b);

Example (array and pointer):

```
#include <iostream>
using namespace std;
int main() {
   int numbers[3] = {10, 20, 30};
   int *ptr = numbers;
   cout << "First element: " << numbers[0] << endl;
   cout << "Pointer to first element: " << *ptr << endl;
   return 0;
}
```

4. User-Defined Data Types

These allow you to create your own structured types.

Data Type	Description	Example
class	Defines objects with properties and behaviors	class Car { public: string brand; };
struct	Similar to class, members are public by default	struct Point { int x, y; };
union	Shares memory for multiple variables	union Data { int i; float f; };
enum	Defines a list of named constants	enum Color { RED, GREEN, BLUE };

Example (class):

```
#include <iostream>
using namespace std;
class Car {
public:
   string brand;
};
int main() {
   Car car1;
   car1.brand = "Toyota";
   cout << "Brand: " << car1.brand << endl;
   return 0;
}
```

Example (enum):

```
#include <iostream>
using namespace std;
enum Color { RED, GREEN, BLUE };
int main() {
   Color favoriteColor = GREEN;
   cout << "Favorite color index: " << favoriteColor << endl;
   return 0;
}
```

Q2. Explain the difference between implicit and explicit type conversion in C++.

Aspect	Implicit Conversion	Explicit Conversion
Who performs it	Compiler automatically	Programmer manually specifies it
Control	Less control	Full control over how and when to convert
Safety	May lose data or precision silently	Programmer takes responsibility
Syntax	No syntax needed	Use casting operators like (type), static_cast<>
Example	int → double	double → int with (int)value

Q3.What are the different types of operators in C++? Provide examples of each.

1. Arithmetic Operators

These operators perform basic mathematical operations.

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

Example:

```
#include <iostream>
using namespace std;
int main() {
  int x = 10, y = 3;
  cout << "x + y = " << x + y << endl;
  cout << "x - y = " << x - y << endl;
  cout << "x * y = " << x * y << endl;
  cout << "x / y = " << x / y << endl;
  cout << "x % y = " << x % y << endl;
  return 0;
}
```

2. Relational (Comparison) Operators

These operators compare two values and return true or false.

Operator	Description	Example
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal	x >= y
<=	Less than or equal	x <= y

Example:

```
#include <iostream>
using namespace std;
int main() {
   int x = 5, y = 10;
   cout << (x == y) << endl;
   cout << (x != y) << endl;
   cout << (x > y) << endl;
   cout << (x < y) << endl;
   cout << (x >= y) << endl;
   cout << (x <= y) << endl;
   return 0;
}
```

3. Logical Operators

These operators combine multiple conditions.

Operator	Description	Example
&&	Logical AND	(x > 0 && y > 0)
`		`
!	Logical NOT	!(x > 0)

Example:

```
#include <iostream>
using namespace std;
int main() {
   int x = 5, y = -2;
   cout << ((x > 0) && (y > 0)) << endl;
   cout << ((x > 0) || (y > 0)) << endl;
   cout << !(x > 0) << endl;
   return 0;
}
```

4. Assignment Operators

These assign values to variables.

Operator	Description	Example
=	Assign	x = 5;
+=	Add and assign	x += 3;
-=	Subtract and assign	x -= 2;
*=	Multiply and assign	x *= 4;
/=	Divide and assign	x /= 2;
%=	Modulus and assign	x %= 3;

Example:

```
#include <iostream>
using namespace std;
int main() {
   int x = 10;
   x += 5;
   cout << "x += 5 → " << x << endl;
   x *= 2;
```

```
    cout << "x *= 2 → " << x << endl;
    return 0;
}
```

5. Increment and Decrement Operators

These operators increase or decrease the value by 1.

Operator	Description	Example
++	Increment	x++; or ++x;
--	Decrement	x--; or --x;

Example:

```
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    cout << "x++ → " << x++ << endl;
    cout << "++x → " << ++x << endl;
    return 0;
}
```

6. Bitwise Operators

Operate at the bit level.

Operator	Description	Example
&	Bitwise AND	x & y
`	`	Bitwise OR
^	Bitwise XOR	x ^ y
~	Bitwise NOT	~x
<<	Left shift	x << 2
>>	Right shift	x >> 1

Example:

```
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    int y = 3;
    cout << "x & y = " << (x & y) << endl;
    cout << "x | y = " << (x | y) << endl;
    cout << "x ^ y = " << (x ^ y) << endl;
    cout << "~x = " << (~x) << endl;
    cout << "x << 1 = " << (x << 1) << endl;
    cout << "x >> 1 = " << (x >> 1) << endl;
    return 0;
}
```

7. Conditional (Ternary) Operator

A compact way to write conditional statements.

Operator	Description	Example
?:	If-else shorthand	x > y ? x : y

Example:

```
#include <iostream>
using namespace std;
int main() {
    int x = 10, y = 20;
    int max = (x > y) ? x : y;
    cout << "Maximum: " << max << endl;
    return 0;
}
```

8. Type Casting Operators

Convert from one type to another.

Operator	Description	Example
static_cast<>	Compile-time cast	int(x)
dynamic_cast<>	Safe downcasting	Used in class hierarchies
const_cast<>	Removes constness	Used in advanced scenarios
reinterpret_cast<>	Low-level cast	Pointer reinterpretation

Example:

```
#include <iostream>

using namespace std;

int main() {

    double pi = 3.14;

    int intPi = static_cast<int>(pi);

    cout << "intPi = " << intPi << endl;

    return 0;

}
```

Q4.Explain the purpose and use of constants and literals in C++.

1. Constants

Definition:

A **constant** is a value that cannot be altered once it has been defined.

Purpose:

- Prevent accidental changes to values.
- Make code more readable and maintainable.
- Use meaningful names instead of hardcoded numbers.

Types of Constants:

1. **const keyword** – Declares a variable whose value cannot be changed.
2. **constexpr keyword** – Defines a value that is constant and must be evaluated at compile time.

3. **#define preprocessor directive** – Defines a symbolic constant.

2. Literals

Definition:

A **literal** is a fixed value written directly in the code.

Purpose:

- Represent constant values.
- Used wherever a fixed value is needed.

3.Control Flow Statements

Q1. What are conditional statements in C++? Explain the if-else and switch statements.

Conditional Statements in C++

Conditional statements are used to perform different actions based on whether a condition is true or false. They control the flow of the program by making decisions at runtime.

1. if-else Statement

Purpose:

The if-else statement executes a block of code if a condition is true, and optionally executes another block if the condition is false.

Syntax:

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

2. switch Statement

Purpose:

The switch statement selects one of many possible blocks of code to execute based on the value of a variable.

Syntax:

```
switch (expression) {
   case constant1:
       // Code block for case 1
       break;
   case constant2:
       // Code block for case 2
       break;
   ...
   default:
       // Code block if no case matches
}
```

Q2. What is the difference between for, while, and do-while loops in C++?

Feature	**for loop**	**while loop**	**do-while loop**
When condition is checked	Before each iteration	Before each iteration	After each iteration
Guaranteed to run at least once?	No	No	Yes
Initialization and update	In the loop header	Outside or inside	Outside or inside
Typical use case	Count-controlled loops	Condition-controlled loops	Loops that must run once before checking the condition

Q3.How are break and continue statements used in loops? Provide examples.

1. break Statement

Purpose:

The break statement is used to **exit the loop prematurely**, regardless of the loop's condition.

Use Cases:

- Stop the loop when a condition is met.
- Exit infinite loops when necessary.

Example:

```
#include <iostream>

using namespace std;

int main() {

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

       if (i == 5) {

           cout << "Breaking at i = " << i << endl;

           break;

       }

       cout << "i = " << i << endl;

   }

   return 0;

}
```

2. continue Statement

Purpose:

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

Use Cases:

- Skip unwanted or invalid values.
- Avoid deeply nested conditions.

Example:

```
#include <iostream>

using namespace std;

int main() {

   for (int i = 1; i <= 5; i++) {

       if (i == 3) {
           cout << "Skipping i = " << i << endl;
           continue;
       }
       cout << "i = " << i << endl;
   }
   return 0;
}
```

Q4.Explain nested control structures with an example.

A **nested control structure** means placing one control statement (like an if, for, while, etc.) **inside another**. This allows you to create more complex decision-making and iterative processes.

To handle multiple conditions or loops.

To process data in multi-dimensional forms (e.g., tables, matrices).

To create advanced algorithms where decisions depend on multiple factors.

EXAMPLE:-

```
#include <iostream>
using namespace std;
int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    if (number > 0) {
        cout << "The number is positive." << endl;
        if (number % 2 == 0) {
            cout << "It is even." << endl;
        } else {
            cout << "It is odd." << endl;
        }
    } else if (number < 0) {
        cout << "The number is negative." << endl;
    } else {
        cout << "The number is zero." << endl;
    }
    return 0;
}
```

4.Functions and Scope

Q1. What is a function in C++? Explain the concept of function declaration, definition, and calling.

A **function** in C++ is a block of code that performs a specific task. It allows you to group a set of instructions together and execute them when needed, making your program organized, reusable, and easier to maintain.

Function Declaration (or Prototype)

A **function declaration** tells the compiler about the function's name, return type, and parameters before its actual definition. It is usually written at the beginning or in a header file.

Syntax:

```
return_type function_name(parameter_list);
```

Function Definition

The **function definition** is where you write the actual code that executes when the function is called.

Syntax:

```
return_type function_name(parameter_list) {
    // statements to execute
}
```

Function Calling

You **call** a function when you want to execute its code. You pass actual values (arguments) to the function.

Syntax:

```
function_name(arguments);
```

Q2. What is the scope of variables in C++? Differentiate between local and global scope.

The **scope of a variable** refers to the part of the program where the variable is accessible or can be used. In C++, variables can be defined in different parts of a program, and their scope depends on where they are declared.

Feature	**Local Scope**	**Global Scope**
Definition	Inside a function or block	Outside all functions
Accessibility	Only within the block/function	Anywhere in the program
Lifetime	Exists only during the block's execution	Exists throughout the program
Usage	Used for temporary data	Used for data shared across functions
Memory	Allocated on the stack	Allocated in the global/static memory

Q3.Explain recursion in C++ with an example?

Recursion is a programming technique where a function calls itself in order to solve a problem. The problem is typically divided into smaller subproblems, and the function keeps calling itself until it reaches a base case, where the recursion stops.

Recursion is useful for problems that can be broken down into similar smaller problems, such as calculating factorials, Fibonacci numbers, and traversing data structures like trees.

```
#include <iostream>

using namespace std;

int factorial(int n) {

  if (n == 0)

     return 1;

  else

     return n * factorial(n - 1);

}
```

```
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   cout << "Factorial of " << num << " is: " << factorial(num) << endl;
   return 0;
}
```

Q4.What are function prototypes in C++? Why are they used?

A **function prototype** is a declaration of a function that specifies:

- The function's name,
- Its return type,
- The types (and optionally the names) of its parameters,

but **without providing the body** of the function.

It tells the compiler that the function exists and how it should be used, even if its definition appears later in the code.

1. **Allow calling functions before their definition:**
   - The prototype informs the compiler about the function's signature, so you can call the function in main() or other functions before its actual implementation is encountered.
2. **Enable modular programming:**
   - You can separate the function's interface from its implementation. The prototype can be placed in a header file, making it reusable across multiple source files.
3. **Help in type checking:**
   - The compiler can ensure that the arguments passed during the function call match the expected types and number of parameters.
4. **Improve readability and structure:**
   - Prototypes provide a quick overview of the functions in a program without diving into their implementation details.

5.Arrays and Strings

Q1. 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. It allows you to store multiple values using a single variable name and access each element using an index.

Feature	**Single-Dimensional Array**	**Multi-Dimensional Array**
Structure	Linear sequence	Grid or table format
Number of Indices	One index to access elements	Two or more indices required
Example	int arr[5];	int matrix[2][3];
Usage	Simple list of items	Complex data representation like matrices, tables
Access Pattern	arr[i]	matrix[i][j]

Q2. Explain string handling in C++ with examples.

In C++, strings are sequences of characters. There are two primary ways to work with strings:

1. C-style strings — arrays of characters ending with a null character '\0'.
2. C++ std::string class — a more powerful and flexible way to handle strings.

String handling involves operations like input/output, concatenation, comparison, searching, and modification.

1. C-style Strings

These are character arrays where the end of the string is marked with '\0'.

Example:

```
#include <iostream>

#include <cstring>

using namespace std;

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

    char str2[20] = "World";

        strcat(str1, str2); // str1 becomes "HelloWorld"

    cout << "Concatenated String: " << str1 << endl;

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

    strcpy(str2, "C++");

    cout << "Copied String: " << str2 << endl;

    if (strcmp(str1, str2) == 0)

        cout << "Strings are equal." << endl;

    else

        cout << "Strings are not equal." << endl;

    return 0;

}
```

2. C++ std::string Class

The std::string class from the C++ Standard Library makes string handling much easier and safer.

Example:

```
#include <iostream>

#include <string>

using namespace std;

int main() {

    string str1 = "Hello";

    string str2 = "World";

        string result = str1 + " " + str2;

    cout << "Concatenated String: " << result << endl;

    cout << "Length: " << result.length() << endl;

    cout << "Substring (0, 5): " << result.substr(0, 5) << endl;
    size_t pos = result.find("World");
    if (pos != string::npos)
        cout << "'World' found at position: " << pos << endl;
    result.replace(6, 5, "C++");
    cout << "After Replace: " << result << endl;
    return 0;
}
```

Q3. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

In C++, arrays are fixed-size collections of elements of the same data type. Arrays can be initialized when they are declared, and depending on whether they are one-dimensional (1D) or two-dimensional (2D), the syntax varies slightly.

Example of 1D Array

```
#include <iostream>
using namespace std;
int main() {
    int numbers[5] = {10, 20, 30, 40, 50};
    for (int i = 0; i < 5; i++) {
        cout << "Element at index " << i << ": " << numbers[i] << endl;
    }
    return 0;
}
```

Example of 2D Array

```
#include <iostream>
using namespace std;
int main() {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << "Element at [" << i << "][" << j << "]: " << matrix[i][j] << endl;
        }
    }
    return 0;
}
```

Q4. Explain string operations and functions in C++.

Initialization

You can initialize strings in multiple ways:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "Hello";
    string str2("World");
    string str3;
    str3 = "C++";
    cout << str1 << " " << str2 << " " << str3 << endl;
    return 0;
}
```

Concatenation

Use the + operator to combine strings:

```
string str1 = "Hello";
string str2 = "World";
string result = str1 + " " + str2;
cout << result << endl;  // Output: Hello World
```

Accessing Characters

You can access individual characters using [] or at():

```
string str = "Programming";
cout << str[0] << endl;  // Output: P
cout << str.at(3) << endl; // Output: g
```

Length of a String

Find the length using length() or size():

```
string str = "Hello";
cout << "Length: " << str.length() << endl;  // Output: 5
```

Comparison

Compare strings using operators like ==, !=, <, etc.:

```
string str1 = "Apple";
string str2 = "Banana";
if (str1 < str2)
    cout << str1 << " comes before " << str2 << endl;
```

Common String Functions

append()

Add more text at the end:

```
string str = "Hello";
str.append(" World");
cout << str << endl;
```

insert()

Insert text at a specific position:

```
string str = "Hello!";
str.insert(5, " World");
```

```
cout << str << endl;
```

erase()

Remove a portion of the string:

```
string str = "Hello World!";

str.erase(5, 6);

cout << str << endl;
```

replace()

Replace part of the string with new text:

```
string str = "Hello World!";

str.replace(6, 5, "C++");

cout << str << endl;
```

find()

Search for a substring:

```
string str = "Hello World!";

size_t pos = str.find("World");

if (pos != string::npos)

    cout << "Found at index " << pos << endl;
```

substr()

Extract a portion of the string:

```
string str = "Hello World!";

string sub = str.substr(6, 5);

cout << sub << endl;
```

c_str()

Convert to C-style string:

```
string str = "Hello";

const char* cstr = str.c_str();

cout << cstr << endl;
```

getline()

Read a line of input including spaces:

```
string str;

getline(cin, str);

cout << "You entered: " << str << endl;
```

6.Introduction to Object-Oriented Programming

Q1. Explain the key concepts of Object-Oriented Programming (OOP).

Key Concepts of Object-Oriented Programming (OOP) – Theoretical Explanation

Object-Oriented Programming (OOP) is a programming paradigm that focuses on designing software using **objects** rather than procedures or functions. This approach closely models real-world entities and their interactions, making programs easier to design, manage, and extend.

Here are the fundamental concepts of OOP explained in theory:

Class

- A **class** is a blueprint or template that defines the structure and behavior of objects.
- It specifies the attributes (data fields or properties) and methods (functions or behaviors) that the objects created from it will have.
- Classes are abstract representations; they do not consume memory until objects are instantiated from them.

Object

- An **object** is an instance of a class, created during program execution.
- It contains actual values for the attributes and can perform actions through methods.
- Objects encapsulate both data and the functions that manipulate that data.
- Multiple objects can be created from the same class but with different data.

Encapsulation

- **Encapsulation** is the concept of bundling data (attributes) and methods that operate on that data into a single unit – the object.
- It restricts direct access to some of an object's components, which helps in protecting the internal state from unintended interference or misuse.

- Access to data is controlled through public methods (getters/setters), ensuring better control and validation.

Inheritance

- **Inheritance** is a mechanism that allows a new class (called the derived or child class) to inherit the attributes and methods from an existing class (called the base or parent class).
- It promotes **code reuse**, reduces redundancy, and enables hierarchical relationships.
- The child class can extend or override the functionalities of the parent class as needed.

Polymorphism

- **Polymorphism** refers to the ability of different objects to respond to the same message (method call) in different ways.
- It allows methods with the same name to behave differently depending on the object's type.
- This supports flexibility and scalability in program design, as new object types can be integrated with minimal changes to existing code.

Abstraction

- **Abstraction** is the process of hiding complex implementation details and showing only the essential features to the user.
- It helps in reducing complexity by focusing on what an object does rather than how it does it.
- Abstract classes or interfaces can define methods that must be implemented by derived classes, providing a structured way to enforce behavior.

Q2. What are classes and objects in C++? Provide an example.

Class

- A **class** in C++ is a user-defined data type that acts as a blueprint for creating objects.
- It defines **attributes** (data members) and **functions** (methods) that describe the properties and behavior of the objects.
- Classes provide abstraction by grouping related variables and functions into a single unit.

Object

- An **object** is an instance of a class.
- It represents an actual entity with real values assigned to its attributes.
- Multiple objects can be created from a single class, each having its own set of data.

Example in C++

Here’s a simple example of a class and object in C++:

```
#include <iostream>
using namespace std;
class Car {
public:
   string brand;
   int speed;
   void display() {
       cout << "Brand: " << brand << endl;
       cout << "Speed: " << speed << " km/h" << endl;
   }
};
int main() {
   Car myCar;
   myCar.brand = "Toyota";
   myCar.speed = 120;
   myCar.display();
   return 0;
}
```

Q3. What isinheritance in C++? Explain with an example.

Inheritance is a fundamental concept of **Object-Oriented Programming (OOP)** in C++ that allows one class (called the **derived class**) to inherit the properties and behaviors of another class (called the **base class**).

Types of Inheritance

C++ supports various forms of inheritance:

1. Single inheritance – one base class and one derived class.
2. Multiple inheritance – derived class inherits from more than one base class.
3. Multilevel inheritance – inheritance chain goes deeper.
4. Hierarchical inheritance – multiple derived classes inherit from a single base class.

Example of Single Inheritance

```
#include <iostream>
using namespace std;
class Vehicle {
public:
   string brand = "Generic Vehicle";
   void honk() {
      cout << "Beep Beep!" << endl;
   }
};
class Car : public Vehicle {
public:
   string model = "Sedan";
   void display() {
      cout << "Brand: " << brand << endl;
      cout << "Model: " << model << endl;
   }
};
int main() {
   Car myCar;
```

```
    myCar.honk();

    myCar.display();

    return 0;

}
```

Q4. What is encapsulation in C++? How isit achieved in classes?

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP) that involves bundling data and methods that operate on that data into a single unit, i.e., a class.

Encapsulation in C++ is achieved by using **access specifiers** to control the visibility of class members (data and functions).

Access Specifiers in C++

1. **private**
   - Members are accessible only within the class.
   - They cannot be accessed from outside the class.
2. **public**
   - Members are accessible from outside the class.
3. **protected**
   - Accessible within the class and its derived classes.

Typically, data members are declared as **private**, and access functions like getters and setters are declared as **public** to control how the data is accessed and modified.