Name : Dhruvit Savaliya

Module – 3

Introduction to OOPS Programming

1.IntroductiontoC++

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

POP	OOP
In procedural programming, the program is divided into small parts called functions.	In object-oriented programming, the program is divided into small parts called objects.
Procedural programming follows a top-down approach.	Object-oriented programming follows a bottom-up approach.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.
there is no concept of data hiding and inheritance.	the concept of data hiding and inheritance is used.
the function is more important than the data.	data is more important than function.

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

➢ 1. Offers Security

Many developers use OOP because it ensures minimal exposure using encapsulation. In this method, developers bundle data to encapsulate information inside an object.

- 2. Improves Collaboration
  One of the top advantages of OOP is that it allows developers to divide a complex software system into small, manageable objects. Each object is responsible for a specific function.

- 3. Allows Reuse of Code
  The concept of inheritance allows OOP to promote the reuse of code. t reduces code duplication and saves development time significantly.

- 4. Makes Changes Seamlessly
  The abstraction of complex systems into simplified models is one of the pivotal benefits of object-oriented programming. This makes it useful for developers who want to make additional changes to the code or incorporate additions over time.

- 5. Locates and Fixes Problems Effortlessly
  The ability to easily pinpoint bugs is among the most vital advantages of object-oriented programming. The developers can isolate and test each object, which helps to identify and isolate problems.

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

- 1. Install a C++ Compiler
  The compiler translates C++ code into machine code that can be executed.
  Common C++ compilers include: GCC, Clang, or MSVC.

- 2. Choose a Development Environment
  IDE (Integrated Development Environment): Provides a complete suite for writing, debugging, and compiling code in one place.
  Text Editors with Extensions: If you prefer lightweight editors, you can use editors like: Visual Studio, CLion, VS Code.

- 3. Configure the Compiler in the IDE/Editor
  IDEs often detect the compiler automatically, but you may need to specify paths manually.

- 4. Set Up a Build System
  Build Systems automate compilation and linking. Common options include: CMake, Make, MSBuild.

- 5. Configure Debugging
  IDEs like Visual Studio and CLion come with built-in debugging tools.

- 6. Write and Run C++ Code
  After setting up, write C++ code in your IDE or editor.

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

- Standard output stream (cout)
  The C++ cout statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).

```
#include <iostream>
using namespace std;
int main()
{
   cout <<" C++ programmer";
}
```

- standard input stream (cin)
  C++ cin statement is the instance of the class istream and is used to read input from the standard input device which is usually a keyboard. The extraction operator(>>) is used along with the object cin for reading inputs.

```
#include <iostream>
using namespace std;
int main()
{
   int age;
   cout << "Enter your age:";
   cin >> age;
   cout << "\nYour age is : " << age;
}
```

2. Variables , Data Types and Operators

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

- Primitive Data type

1.) Integer - integer data types represent whole numbers without a fractional or decimal part.
int a = 123;

2.) Character - Character data types represent individual characters from a character set, like ASCII or Unicode. In C++, `char` is commonly used to represent characters.
Char ch = 'A';

3.) Boolean - Boolean data types represent binary values, typically used for true (1) or false (0) conditions. In C++, `bool` is used for Boolean data.
bool t = true;

4.) Floating Point - loating-point data types represent numbers with a fractional part. In C++, `float` is a single-precision floating-point type.
Float F = 3.14159f;

5.) Double Floating Point - ouble-precision floating-point data types are used to represent numbers with a larger range and higher precision compared to 'float'. In C++, 'double' is commonly used.
double D = 3.141592653589793;

➢ User-defined or Abstract Data Type

1. Class - A class is a user-defined data type that represents a blueprint for creating objects. It encapsulates data (attributes) and functions (methods) that operate on that data.

2. Array - An array is a derived data type that represents a collection of elements of the same data type, stored in contiguous memory locations. The elements can be accessed by their index.

3. Pointer - A pointer is a derived data type that stores the memory address of another data type. Pointers are often used for dynamic memory allocation and for accessing memory locations directly.

4. Structure - A structure is a user-defined data type that groups variables of different data types under a single name. Structures are useful for organizing related data elements.

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

Implicit Conversion	**Explicit Conversion**

Automatic, done by the compiler	Manually specified by the programmer
No special syntax	Requires casting operators like static_cast
Only happens when it’s safe (usually no loss)	Can lead to data loss (e.g., double to int)
Less control, might lead to unexpected results	Complete control, safer for complex casts
Assigning smaller type to larger type	Converting between incompatible or specific types

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

- Arithmetic Operators
  Used to perform basic arithmetic operations.

```
int a = 10, b = 3;
int sum = a + b;     // Addition (+) → sum = 13
int diff = a - b;    // Subtraction (-) → diff = 7
int prod = a * b;    // Multiplication (*) → prod = 30
int quot = a / b;    // Division (/) → quot = 3
int rem = a % b;     // Modulus (%) → rem = 1
```

- Relational Operators
  Used to compare values.

```
int a = 5, b = 10;
bool result1 = (a == b);    // Equal to (==) → false
bool result2 = (a != b);    // Not equal to (!=) → true
bool result3 = (a < b);     // Less than (<) → true
bool result4 = (a > b);     // Greater than (>) → false
bool result5 = (a <= b);    // Less than or equal to (<=) → true
bool result6 = (a >= b);    // Greater than or equal to (>=) → false
```

- Logical Operators
  Used to combine conditional statements.

```
bool x = true, y = false;
bool result1 = (x && y);    // Logical AND (&&) → false
bool result2 = (x || y);    // Logical OR (||) → true
bool result3 = !x;          // Logical NOT (!) → false
```

- Bitwise Operators
  Used to perform operations on bits.

```
int a = 5;     // In binary: 0101
int b = 3;     // In binary: 0011
int andRes = a & b;  // Bitwise AND (&) → 0001 (1 in decimal)
int orRes = a | b;   // Bitwise OR (|) → 0111 (7 in decimal)
int xorRes = a ^ b;  // Bitwise XOR (^) → 0110 (6 in decimal)
int notRes = ~a;     // Bitwise NOT (~) → 1010 (2's complement)
int leftShift = a << 1; // Left Shift (<<) → 1010 (10 in decimal)
int rightShift = a >> 1; // Right Shift (>>) → 0010 (2 in decimal)
```

- Assignment Operators
  Used to assign values to variables.

```
int a = 5;       // Simple assignment (=)
a += 3;          // Add and assign (+=) → a = 8
a -= 2;          // Subtract and assign (-=) → a = 6
a *= 2;          // Multiply and assign (*=) → a = 12
a /= 3;          // Divide and assign (/=) → a = 4
a %= 2;          // Modulus and assign (%=) → a = 0
```

- Increment and Decrement Operators
  Used to increase or decrease the value by 1.

```
int a = 5;
int b = ++a;     // Pre-increment → a = 6, b = 6
int c = a--;     // Post-decrement → a = 5, c = 6
```

- Scope Resolution Operator
  Used to define a function outside its class or access global variables.

```
int x = 10;  // global variable
int main() {
    int x = 5;   // local variable
    std::cout << ::x; // Scope resolution to access global x
}
```

- Pointer Operators
  Used with pointers to access memory addresses.

```
int a = 5;
int* ptr = &a;   // Address-of operator (&)
int value = *ptr; // Dereference operator (*) → value = 5
```

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

- 1. Constants

  Constants are identifiers for fixed values that can't be altered once assigned. Constants improve code readability, help prevent unintended changes, and make the program easier to modify by centralizing fixed values in one place.

  Purpose:

  Ensure values don’t accidentally change during execution.
  Improve readability and maintainability.
  Centralize and control fixed values in a single place.

  Usage:

  ```
  const double TAX_RATE = 0.08;   // Constant tax rate
  int total = subtotal + (subtotal * TAX_RATE);
  ```

  Using const is preferred over #define for constants as it respects scope, type-checking, and can be used in functions and classes.

- 2. Literals

  Literals are fixed values written directly in the code. They represent exact values of specific types, like integers, characters, strings, and floating-point numbers.

  Purpose:

  Represent specific values directly in code.
  Used to initialize constants, variables, or as values in expressions.

  Types of Literals:

  Integer Literals, Floating-Point Literals, Character Literals, String Literals, Boolean Literals

3. Control Flow Statements

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

- In C++, conditional statements allow programs to make decisions based on conditions.

- hey help control the flow of the program by executing certain blocks of code only when specific conditions are met.

- The two primary conditional statements in C++ are if-else and switch statements.

- if-else Statement :

  The if-else statement executes a block of code if a specified condition evaluates to true. It can also include an else part to specify an alternative block of code to run when the condition is false.

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

- switch Statement :

  The switch statement is another conditional statement used to execute one of many code blocks based on the value of a variable. It's often more efficient than a long if-else-if ladder when you are comparing a single variable to different constant values.

  Syntax:
  ```
       switch (expression) {
     case constant1:
        // code to execute if expression == constant1
        break;
     case constant2:
        // code to execute if expression == constant2
        break;
     // you can have any number of case statements
     default:
        // code to execute if none of the cases match
     }
  ```

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

FOR	**WHILE**	**DO-WHILE**
for (initialization; condition; increment/decrement) {}	while (condition) { }	do { } while (condition);
Checked before each iteration.	Checked before each iteration.	Checked after each iteration.
Declared within the loop structure and executed once at the beginning.	Declared outside the loop; should be done explicitly before the loop.	Declared outside the loop structure
Executed after each iteration.	Executed inside the loop; needs to be handled explicitly.	Executed inside the loop; needs to be handled explicitly.

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

- Break

  You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.
  The break statement can also be used to jump out of a loop.

```
int i;
for (i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  Cout<<I;
}
```

- Continue

  The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
int i;
for (i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  Cout<<I;
}
```

4. Explain nested control structures with an example.

- Nested control structures refer to the usage of control structures within each other.
- This allows us to create complex decision-making processes and iterate over multiple levels of data.
- Let's take a look at an example of nested control structures in C++:

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

int main() {
int age = 25;
bool isStudent = true;
```

```
    if (age >= 18) {    cout << "You are an adult.";
         if (isStudent) {  cout << "You are a student.";}
        els      cout << "You are not a student.";    }
     }
  else {   cout << "You are not an adult.";   }
   return 0;
 }
```

- In this example, we have nested an if statement inside another if statement. The outer if statement checks if the age is greater than or equal to 18.

4. Functions and Scope

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

- A function is a group of statements that together perform a task.
- Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.

- Defining a Function
  A function definition provides the actual body of the function.

  The general form of a C++ function definition is as follows –
```
return_type function_name( parameter list ) {
          body of the function
}
```

- Function Declarations
  A function declaration tells the compiler about a function's name, return type, and parameters. A function declaration tells the compiler about a function name and how to call the function.

  A function declaration has the following parts –
```
return_type function_name( parameter list );
```

- Calling a Function
  While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.
```
// calling a function to get max value.
         ret = max(a, b);
```

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

- In programming also the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with.
- There are mainly two types of variable scopes:
  Local Variables
  Global Variables

- Local Variables

  Variables defined within a function or block are said to be local to those functions. Anything between '{' and '}' is said to inside a block.

  Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

```
#include<iostream>
using namespace std;
void func()
{
        int age=18;
}
int main()
{
        cout<<"Age is: "<<age;
        return 0;
}
```

- Global Variables

  As the name suggests, Global Variables can be accessed from any part of the program. They are available through out the life time of a program.

  They are declared at the top of the program outside all of the functions or blocks.

```
#include<iostream>
using namespace std;
int global = 5;
void display()    {
  cout<<global<<endl;   }
int main()
{
        display();
        global = 10;
        display();
}
```

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

- Recursion in C++ is a technique where a function calls itself to solve a problem in smaller sub-problems.
- Each recursive call reduces the problem until it reaches a "base case," which is the simplest version of the problem that can be solved without further recursion.
- Once the base case is reached, the function begins to return values, and the recursive calls are resolved.
- Here's an example of recursion in C++ to calculate the factorial of a number . The factorial of a number n is defined as:

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

int factorial(int n) {
if (n <= 1) {
   return 1;
}
 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;
}
```

- In factorial(int n), when nnn is 0 or 1, it returns 1 because 0!=10! = 10!=1 and 1!=11! = 11!=1.
- Execution Flow: If we call factorial(5), the following recursive calls occur:
- factorial(5) = 5 * factorial(4)
- factorial(4) = 4 * factorial(3)
- factorial(3) = 3 * factorial(2)
- factorial(2) = 2 * factorial(1)
- factorial(1) = 1 (base case)
- Once the base case is reached, the function starts returning values, ultimately giving the result for factorial(5).

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

- If a function is called somewhere, but its body is not yet defined, that is defined after the current line; it may cause issues.
- The compiler cannot determine what the function is or what its signature is. In that scenario, we'll need function prototypes.
- The Function prototype has the following features:
  1) It specifies the data type that the function will return.
  2) It indicates how many parameters were passed to the function.
  3) It returns the data types of each of the passed inputs.
  4) It also indicates the sequence in which the inputs are passed to the function.

- Why Function Prototypes Are Used
  If a function is defined after it is called (e.g., in main), a prototype provides the compiler with the necessary information to compile the code without encountering an error.
  The compiler checks that arguments passed to the function match the expected types as per the prototype, helping prevent errors.
  Prototypes allow functions to be grouped or declared at the beginning of the file, improving code organization and making it easier for readers to understand which functions are used in the program.

5. Arrays and Strings

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

- In C++, an array is a collection of elements of the same data type stored in contiguous memory locations.
- Arrays provide a way to store multiple values in a single variable, which can be accessed using indices.

- Syntax of Array Declaration
  data_type array_name[array_size];

single –dimensional	multi dimensional
Store a single list of the element of a similar data type.	Store a 'list of lists' of the element of a similar data type.
Represent multiple data items as a list.	Represent multiple data items as a table consisting of rows and columns.
Structure Linear, one row of elements	Structure Grid (2D), Cube (3D), or higher
Common Usage is Simple lists	Common Usage isTables, matrices, grids
Uses a single index [i]	Uses multiple indices [i][j] (2D), [i][j][k] (3D)

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

- C++ provides two main ways to handle strings.
- using C-style character arrays and the std::string class from the Standard Library.

- 1. C-Style Strings
  C-style strings are arrays of characters ending with a null character \0.
  This is a common approach in C and still supported in C++ for low-level string manipulation.
  strlen(s): Returns the length of the string.
  strcpy(dest, src): Copies src into dest.
  strcat(dest, src): Concatenates src to the end of dest.

- 2. std::string Class
  The std::string class is more flexible and safer to use, as it automates memory management and provides many functions for manipulation.
  length(): Returns the number of characters in the string.
  append(str): Adds str to the end of the string.
  substr(pos, len): Returns a substring starting at pos with length len.
  find(str): Finds the first occurrence of str and returns its index (or std::string::npos if not found).
  replace(pos, len, str): Replaces part of the string with str.

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

- The initializer for an array is a comma-separated list of constant expressions enclosed in braces ({ }).
- The initializer is preceded by an equal sign (=). You do not need to initialize all elements in an array.
- If an array is partially initialized, elements that are not initialized receive the value 0 of the appropriate type.
- The same applies to elements of arrays with static storage duration.

- 1D Arrays
  A 1D array is a simple linear collection of elements, accessible via an index.
  ou can initialize it at the time of declaration or later.

```
int arr1[5] = {1, 2, 3, 4, 5};
int arr[3] = {10, 20, 30};
```

- 2D Arrays
  A 2D array is essentially an array of arrays, often used to represent a matrix or table.

Each element is accessed by two indices.

```
int matrix1[2][3] = {{1, 2, 3}, {4, 5, 6}};
int matrix[2][2] = {{1, 2}, {3, 4}};
```

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

- Concatenation

  Strings can be concatenated using the + operator or append() function.
  ```
  string str3 = str1 + " " + str2;
  string str4 = str1.append(" ").append(str2);
  ```

- Accessing Characters

  You can access individual characters using the [] operator or the .at() function.
  ```
  char ch1 = str1[0];
  char ch2 = str2.at(1);
  ```

- Finding Substrings

  The find() function locates the position of a substring within a string.
  ```
  size_t found = str1.find("el");
  ```

- Substring Extraction

  The substr() function extracts a portion of a string.
  ```
  string part = str1.substr(1, 3);
  ```

- nserting and Erasing

  You can insert or erase parts of a string with the insert() and erase() functions.
  ```
  str1.insert(0, "Say ");
  str1.erase(0, 4);
  ```

- Replacing Parts of a String

  The replace() function replaces part of the string with another string.
  ```
  str1.replace(0, 2, "Hi");
  ```

- Getting Length of a String

  The length() or size() function returns the length of the string.
  ```
  cout << "Length of str1: " << str1.length()
  ```

- Comparison

  Strings can be compared using ==, !=, <, >, or the compare() function.
  ```
  if (str1 == str2) {
      std::cout << "str1 and str2 are equal\n";
  } else if (str1.compare(str2) < 0) {
  ```

```
        std::cout << "str1 is less than str2\n";
    } else {
        std::cout << "str1 is greater than str2\n";
    }
```

6.Introduction to Object-Oriented Programming

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

- 1. Class and Object

  Class: A class is a blueprint or template for creating objects. It defines a set of attributes and methods that the objects created from the class will have.

  Object: An object is an instance of a class. It represents a specific entity that has its own values for the attributes defined by the class.

- 2. Encapsulation

  Encapsulation is the bundling of data (attributes) and methods that operate on that data into a single unit (class) and restricting access to some components.

  It is achieved using access specifiers:
  Public: Members are accessible from outside the class.
  Private: Members are accessible only within the class.
  Protected: Members are accessible within the class and derived (inherited) classes.

- 3. Inheritance

  Inheritance is a mechanism by which one class (child class) acquires the properties and behaviors of another class (parent class).
  It promotes code reusability, allowing you to create a new class that extends or modifies the behavior of an existing class.

- 4. Polymorphism

  Polymorphism means "many forms." It allows the same function or method to behave differently based on the object calling it.

  Two types of polymorphism in C++:
  Compile-time (Static) Polymorphism: Achieved through function overloading and operator overloading.
  Run-time (Dynamic) Polymorphism: Achieved through virtual functions and inheritance.

- 5. Abstraction
  Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object.
  It can be achieved using abstract classes and interfaces (in C++, pure virtual functions are used to create abstract classes).

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

- What is a Class?
  A class is a blueprint or template for creating objects. It defines:
  Attributes (or data members) to represent the properties of the object.
  Methods (or member functions) to define the behaviors of the object.
  In C++, we define a class using the class keyword, followed by the class name and a set of curly braces {} containing its attributes and methods.

- What is an Object?
  An object is an instance of a class.
  When a class is defined, no memory is allocated until an object of that class is created.
  Each object has its own unique set of values for the attributes defined by its class.

- Example

```
#include <iostream>
#include <string>
using namespace std;
class Car {
public:
   string brand;
   string model;
   int year;
   void displayInfo() {
      cout << "Brand: " << brand << endl;
      cout << "Model: " << model << endl;
      cout << "Year: " << year << endl;
   }
   void start() {
      cout << brand << " " << model << " is starting." << endl;
   }
};

int main() {
   Car myCar;
```

```
    myCar.brand = "Toyota";
    myCar.model = "Corolla";
    myCar.year = 2020;
    myCar.displayInfo();
    myCar.start();

    return 0;
}
```

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

- Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows one class to inherit the properties and behaviors (attributes and methods) of another class.
- In C++, inheritance is a way to reuse the code from an existing class in a new class .
- This promotes code reusability and allows for creating more specialized versions of a class.

- Types of Inheritance in C++
  Single Inheritance: A derived class inherits from only one base class.
  Multiple Inheritance: A derived class inherits from more than one base class.
  Multilevel Inheritance: A class derives from a derived class.
  Hierarchical Inheritance: Multiple classes inherit from a single base class.
  Hybrid Inheritance: A combination of more than one type of inheritance.

- Example

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

class Animal {
public:
    string name;
    Animal(string n) : name(n) {}
    void eat() {
        cout << name << " is eating." << endl;
    }
};

class Dog : public Animal {
public:
    Dog(string n) : Animal(n) {}
    void bark() {
        cout << name << " is barking." << endl;
    }
};

int main() {
    Dog myDog("Buddy");
    myDog.eat();
    myDog.bark();
    return 0;
}
```

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

- Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP).
- It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class.
- dditionally, it hides the internal implementation details of the class from outside access and restricts direct modification of its attributes.

- Encapsulation serves several purposes:

  Data Hiding: It prevents direct access to an object's internal data, which helps protect the integrity of the data.

  Access Control: It provides a controlled way to access or modify data.

  Code Maintenance: By keeping data and functions together, and restricting direct access to the internal state, encapsulation makes it easier to maintain and modify the code.

- In C++, encapsulation is achieved through the use of access specifiers in a class. These are:

  Public (public): Members (variables and functions) declared as public are accessible from outside the class.

  Private (private): Members declared as private are not accessible from outside the class. They can only be accessed within the class itself.

  Protected (protected): Members declared as protected can be accessed within the class and by derived classes, but not from outside the class hierarchy.

- Advantages of Encapsulation

  Control: By using getters and setters, you can control how the data is accessed or modified. For instance, you can enforce constraints (like valid balance) when modifying data.

  Data Integrity: Internal object state can be protected from unintended changes by restricting direct access to its private members.

  Maintainability: Encapsulation allows you to change the internal implementation of a class (e.g., change how the balance is stored) without affecting the code that uses the class.