Module - 3 Introduction to OOPS Programming

1. Introduction to C++

1) What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?
    - Procedural Oriented Programming
        - In procedural programming, the program is divided into small parts called functions.
        - Procedural programming follows a top-down approach.
        - There is no access specifier in procedural programming.
        - Adding new data and functions is not easy.
        - n procedural programming, overloading is not possible.
    - Object-Oriented Programming
        - In object-oriented programming, the program is divided into small parts called objects.
        - Object-oriented programming follows a bottom-up approach.
        - Object-oriented programming has access specifiers like private, public, protected, etc.
        - Adding new data and function is easy.
        - Overloading is possible in object-oriented programming.
2) List and explain the main advantages of OOP over POP.
    - Modularity and Reusability
        - OOP**:** Promotes code reuse through classes and objects. Once a class is created, it can be reused in other programs without modification.
        - POP**:** Functions can be reused, but the reuse isn't as flexible or organized as with classes in OOP.
    - Data Encapsulation
        - OOP**:** Encapsulation hides the internal state of objects and only exposes what is necessary, which helps protect the integrity of the data.
        - POP**:** Data is more exposed, and functions can modify it directly, which may lead to unintended side effects.
    - **I**nheritance
        - OOP**:** Inheritance allows new classes to be created based on existing classes, enabling code reuse and the creation of a hierarchical relationship between classes.

    - POP**:** There is no concept of inheritance. Each function must be written from scratch.
- Polymorphism
    - OOP**:** Polymorphism allows objects to be treated as instances of their parent class, providing flexibility in code and the ability to define methods that can operate on objects of different classes.
    - POP: Polymorphism is not inherent in procedural programming. Functions must be explicitly defined for each type.
- Maintenance and Scalability
    - OOP**:** The modular nature of OOP makes it easier to maintain and scale applications. Changes to one part of the code are less likely to affect other parts.
    - POP: Maintenance can be more challenging as the codebase grows, and changes in one part can have unforeseen effects elsewhere.
- Abstraction
    - OOP**:** Abstracts complexity by defining high-level interfaces and hiding the implementation details. This simplifies the development process and makes it easier to work with complex systems.
    - POP: Abstraction is less pronounced, and the developer must handle more details directly.

3) Explain the steps involved in setting up a C++ development environment.
    - Choose a Compiler:
        - GCC (GNU Compiler Collection): A popular choice for C++ development, especially
          on Linux and Windows (via MinGW).
    - Install an Integrated Development Environment (IDE) or Text Editor:
        - IDE Options**:**
            - Visual Studio**:** A powerful IDE for Windows with extensive features.
            - Code::Blocks: A free, open-source IDE that is easy to use.
        - Text Editor Options:
            - Visual Studio Code: Lightweight and customizable with extensions for C++.
    - Download and Install the Compiler:
        - For MinGW (on Windows):

        - Download the MinGW installer.
        - Run the installer and select the components you need (like g++ for C++).
        - After installation, add the MinGW bin directory to your system's PATH environment variable.
    - Set Up Your IDE:
        - Open your chosen IDE and configure it to use the installed compiler.
        - For example, in Visual Studio, you can create a new project and select the C++ template
    - Compile and Run Your Program:
        - Use the build/run commands in your IDE to compile and execute your program.

4) What are the main input/output operations in C++? Provide examples.
    - Input Stream: If the direction of flow of bytes is from the device (for example, Keyboard) to the main memory then this process is called input.
    - Syntax: cin >> variable;

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

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

  cout << "Age entered: " << age;
}
```

    - Output Stream: If the direction of flow of bytes is opposite, i.e. from main memory to device (display screen) then this process is called output.
    - Syntax: cout << value/variable;

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

main()
{
```

```
    cout << "Hello World";
}
```

2. Variables, Data Types, and Operators

1) What are the different data types available in C++? Explain with examples.
    - Primitive Data Types: These are the fundamental data types built into the language.
        - int: Used for storing whole number. Ex. int age = 30;
        - float: Used for storing single-precision floating-point number. Ex. float price = 99.99;
        - double: Used for storing double-precision floating-point number. Ex. double pi = 3.14159265359;
        - char: Used for storing single character. Ex. char initial = 'J';
        - bool: Used for storing boolean values (true or false). Ex. bool is_active = true;
    - Derived Data Types: These are data types derived from primitive data type
        - Array: A collection of elements of the same data type stored in contiguous memory locations. Ex. int numbers[5] = {1, 2, 3, 4, 5};
        - Pointer: A variable that stores the memory address of another variables. Ex. int num = 10;
          int* ptr = #
        - Function: A block of code that performs a specific task.
    - User-Defined Data Types: These are data types defined by the user.
        - Class: A blueprint for creating objects, which can contain both data and functions.
        - Structure: A collection of variables of different data types grouped together under a single name.
        - Union: A special data type that allows storing different data types in the same memory location.
2) Explain the difference between implicit and explicit type conversion in C++.
    - Implicit Type Conversion:
        - Also known as type coercion, implicit type conversion automatically converts one data type to another without requiring any specific syntax or code from the programmer.
        - Ex. int num = 10;
          double result = num;
    - Explicit Type Conversion:

- Also known as type casting, explicit type conversion requires the programmer to specify the type conversion using a cast operator.
- Ex. double pi = 3.14;
  int integer_pi = (int)pi;

3) What are the different types of operators in C++? Provide examples of each.
   - Arithmetic Operators
     - Perform basic mathematical calculations.
     - Examples:
       - + (Addition): int sum = 5 + 3; // sum = 8
       - - (Subtraction): int difference = 10 - 4; // difference = 6
       - * (Multiplication): int product = 2 * 7; // product = 14
       - / (Division): double quotient = 15 / 4; // quotient = 3.75
       - % (Modulo): int remainder = 17 % 5; // remainder = 2
   - Relational Operators
     - Compare values and return a boolean result (true or false).
     - Examples:
       - == (Equal to): bool isEqual = (5 == 5); // isEqual = true
       - != (Not equal to): bool isNotEqual = (10 != 15); // isNotEqual = true
       - > (Greater than): bool isGreater = (20 > 10); // isGreater = true
       - < (Less than): bool isLess = (5 < 12); // isLess = true
       - >= (Greater than or equal to): bool isGreaterOrEqual = (10 >= 10); // isGreaterOrEqual = true
       - <= (Less than or equal to): bool isLessOrEqual = (7 <= 7); // isLessOrEqual = true
   - Logical Operators
     - Combine boolean expressions.
     - Examples:
       - && (Logical AND): bool bothTrue = (true && true); // bothTrue = true
       - || (Logical OR): bool eitherTrue = (false || true); // eitherTrue = true
       - ! (Logical NOT): bool isFalse = !(true); // isFalse = false
   - Bitwise Operators
     - Operate on individual bits of data.
     - Examples:
       - & (Bitwise AND): int result = 12 & 5; // result = 4 (binary: 1100 & 0101 = 0100)

        - | (Bitwise OR): int result = 12 | 5; // result = 13 (binary: 1100 | 0101 = 1101)
        - ^ (Bitwise XOR): int result = 12 ^ 5; // result = 9 (binary: 1100 ^ 0101 = 1001)
        - ~ (Bitwise NOT): int result = ~12; // result = -13 (binary: ~1100 = 0011)
        - << (Left shift): int result = 5 << 2; // result = 20 (binary: 0101 << 2 = 10100)
        - >> (Right shift): int result = 12 >> 2; // result = 3 (binary: 1100 >> 2 = 0011)
    - Assignment Operators
        - Assign values to variables.
        - Examples:
            - = (Assignment): int age = 25;
            - += (Add and assign): age += 5; // age = 30
            - -= (Subtract and assign): age -= 2; // age = 28
            - *= (Multiply and assign): age *= 2; // age = 56
            - /= (Divide and assign): age /= 4; // age = 14
            - %= (Modulo and assign): age %= 3; // age = 2

4) Explain the purpose and use of constants and literals in C++.
    - Constants**:**
        - Purpose: Constants are variables whose values cannot be modified after they are initialized. Think of them as "read-only" variables.
        - const int MAX_USERS = 100; // constant using const keyword
          #define PI 3.14 // constant using #define
    - Literals:
        - Purpose: Literals represent fixed values that are directly used in the code. They are the actual values assigned to variables or constants.
        - int age = 25; // 25 is an integer literal
          double price = 19.99; // 19.99 is a floating-point literal
          char grade = 'A'; // 'A' is a character literal
          std::string message = "Hello!"; // "Hello!" is a string literal
          bool isActive = true; // true is a boolean literal

3. Control Flow Statements

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

- The if-else statement is the most basic form of conditional statement. It checks a condition and executes a block of code if the condition is true, and optionally executes another block of code if the condition is false.
- if (condition)
  {
  // Code to execute if the condition is true
  }
  else
  {
  // Code to execute if the condition is false (optional)
  }

- switch Statements:
  - The switch statement is used when you need to choose from a set of multiple options. It compares a variable's value against a list of constant values.
  - switch (expression)
    {
    case value1:
    // Code to execute if expression matches value1
    break;
    case value2:
    // Code to execute if expression matches value2
    break;
    // ... more cases
    default:
    // Code to execute if no case matches
    }

2) What is the difference between for, while, and do-while loops in C++?
   - for Loop: The for loop is the most common and versatile loop. It's perfect for situations where you know exactly how many times you want to repeat a block of code.
     - for (initialization; condition; increment/decrement)
       {
       // Code to be repeated
       }
   - while Loop: The while loop is ideal for situations where you don't know in advance how many times you need to repeat a code block. It keeps executing the loop as long as a certain condition is true.
     - while (condition)
       {

```
        // Code to be repeated
    }
```

- do-while Loop: The do-while loop is similar to the while loop, but it guarantees that the code block will execute at least once, even if the condition is false initially.
    - ```
      do
      {
          // Code to be repeated
      } while (condition);
      ```

3) How are break and continue statements used in loops? Provide examples.
    - break Statement: The break statement is used to immediately exit the innermost loop it's inside. Think of it as a "stop" sign for the loop.
        - break;
        - ```
          Ex. for (int i = 1; i <= 10; i++)
              {
                  if (i == 5)
                  {
                      break; // Exit the loop when i reaches 5
                  }
                  cout << i << " ";
              }
          ```
    - continue Statement: The continue statement skips the remaining code within the current iteration of the loop and jumps to the next iteration. Think of it as a "skip" button for the loop.
        - continue;
        - ```
          Ex. for (int i = 1; i <= 10; i++)
          {
              if (i % 2 == 0)
              {
                  continue; // Skip even numbers
              }
              cout << i << " ";
          }
          ```

4) Explain nested control structures with an example.
    - A nested structure in C is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.
    - Outer Loop: The first for loop controls the number of rows in the triangle. It iterates from 1 to the number of rows entered by the user.
          ```

- Inner Loop: The second for loop is nested inside the first loop. It controls the number of stars printed in each row. It iterates from 1 to the current row number.
- Printing Stars: Inside the inner loop, the cout << "* "; statement prints a star followed by a space.
- Newline: After each row is complete, the cout << endl; statement moves the output to the next line.
- The inner loop executes completely for each iteration of the outer loop.
- Nested loops can be used to create patterns, process data in multiple dimensions, and implement more complex algorithms.
- You can nest any type of control structure (loops, if statements, switch statements) within each other.

4. Functions and Scope

1) What is a function in C++? Explain the concept of function declaration, definition, and calling.
    - Function Declaration: This is like a blueprint for your function. It tells the compiler the function's name, the type of data it returns (if any), and the types of data it expects as input (called parameters).
        - int addNumbers(int a, int b); // Declaration
    - Function Definition: This is where you actually write the code that the function will execute.
        - Ex. int addNumbers(int a, int b)
          {
              return a + b;
          } // Definition
    - Function Calling: This is how you use the function in your program. You simply write the function's name followed by parentheses containing the actual values (arguments) to be passed in.
        - Ex. int main()
          {
              int result = addNumbers(5, 3); // Calling the function
              cout << "The sum is: " << result << endl;
          }
2) What is the scope of variables in C++? Differentiate between local and global scope.
    - Local Scope: Inside a function or a block of code (defined by curly braces {}).

    - A local variable exists only within the function or block where it's declared. It's created when the function is called and destroyed when the function finishes executing.
    - Ex. int main()
      {
        int localVar = 10; // Local variable inside main function
        cout << localVar << endl;
      }
- Global Scope: Outside any function, typically at the beginning of your program file.
    - A global variable exists throughout the entire program's execution. It's created when the program starts and destroyed when the program ends.
    - Ex. int globalVar = 20; // Global variable
      int main()
      {
        cout << globalVar << endl;
      }
- Local Scope:
    - Declaration Inside a function or block.
    - Exists only within its function/block.
    - Accessible only within its function/block.
- Global Scope:
    - Declaration Outside any function.
    - Exists throughout the entire program.
    - Accessible from anywhere in the program.

3) Explain recursion in C++ with an example.
    - Recursion in C++ occurs when a function calls itself directly or indirectly to solve a problem.
    - It's often used for problems that can be broken down into smaller, similar subproblems.
    - One of the classic examples of recursion is calculating the factorial of a number.
    - Base Case: A recursive function must have a base case, which is a condition that stops the recursion. Without a base case, the function would call itself infinitely, leading to a stack overflow.
    - Recursive Step: The recursive step is where the function calls itself with a modified input, bringing the problem closer to the base case.
4) What are function prototypes in C++? Why are they used?

- A function prototype is a declaration that tells the compiler about a function's existence, its return type, and the types of its parameters. It's like a preview of the function before you actually define it.
- Ex. int add(int a, int b); // Function prototype
- Why Use Function Prototypes:
    - Forward Declaration: Function prototypes allow you to use a function before it's fully defined. This is especially useful when you have multiple functions that call each other, as it allows you to define them in any order without encountering errors.
    - Compile-Time Error Checking: Prototypes help the compiler catch errors related to function calls, such as incorrect parameter types or missing parameters.
    - Code Readability: Prototypes make your code more readable by providing a clear overview of the functions available in your program.

5. Arrays and Strings
    1) What are arrays in C++? Explain the difference between single-dimensional and multi- dimensional arrays.
        - Arrays in C++ are like containers that hold a fixed-size collection of elements of the same data type. Imagine them as rows of boxes, each box holding a value.
        - Single-Dimensional Arrays: A single-dimensional array stores elements in a linear sequence, like a list.
            - data_type array_name[size];
            - You access elements using an index, starting from **0**.
            - numbers[0] = 10; // Assigns the value 10 to the first element.
        - Multi-Dimensional Arrays: Multi-dimensional arrays allow you to store data in a tabular format, like a spreadsheet.
            - data_type array_name[size1][size2]...[sizeN];
            - int matrix[3][4]; // Declares a 2D array named matrix with 3 rows and 4 columns.
    2) Explain string handling in C++ with examples.
        - Declaration and Initialization:
            - Declaration: std::string myString; // Declares an empty string
            - Initialization:

        - std::string greeting = "Hello, world!"; // Initializes with a string literal
        - std::string name = "Alice"; // Initializes with a string variable
    - Accessing Characters:
        - Using the [] operator: char firstChar = myString[0]; // Accesses the first character
        - Using the at() method: char secondChar = myString.at(1); // Accesses the second character (more robust, throws an exception if out of bounds)
    - String Length**:** size_t length = myString.length(); // Returns the length of the string (number of characters)
    - Concatenation:
        - myString += "!"; // Appends an exclamation mark to the end
        - std::string combined = myString + " " + name; // Combines strings with spaces
    - Finding Substrings:
        - size_t pos = myString.find("world"); // Finds the first occurrence of "world" and returns its position (0-based index)
        - size_t pos = myString.rfind("world"); // Finds the last occurrence of "world"
    - Replacing Substrings:
        - myString.replace(pos, 5, "universe"); // Replaces 5 characters starting at position pos with "universe"
    - Comparing Strings:
        - if (myString == "Hello, world!") { ... } // Checks for equality
        - if (myString < "Hello, universe!") { ... } // Lexicographical comparison (alphabetical order)
    - Converting to Uppercase/Lowercase:
        - std::transform(myString.begin(), myString.end(), myString.begin(), ::toupper); // Converts to uppercase
        - std::transform(myString.begin(), myString.end(), myString.begin(), ::tolower); // Converts to lowercase
3) How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.
    - Initializing 1D Arrays: You can assign values to elements during declaration.
        - int numbers[5] = {1, 2, 3, 4, 5}; // Initialize with 5 elements
          char letters[4] = {'A', 'B', 'C', 'D'}; // Initialize with 4 characters

    - You can initialize only some elements, the rest will be set to 0 (for numeric types) or null characters ('\0') for character arrays.
    - int scores[10] = {90, 85}; // Scores[2] to scores[9] will be
- Initializing 2D Arrays: Similar to 1D arrays, but with nested curly braces for each row.
    - int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; // 2x3 matrix
      char names[3][10] = {{"Alice", "Bob"}, {"Charlie", "David"}, {"Eve", "Frank"}}; // 3x10 matrix
    - You can initialize only some rows or elements within rows. The rest will be set to 0 or null characters.
    - int table[3][4] = {{1, 2}, {3, 4}}; // Only the first two rows are initialized

4) Explain string operations and functions in C++.
    - String Operations:
        - **Concatenation:** Combining two or more strings into a single string.
        - string firstName = "Alice";
          string lastName = "Smith";
          string fullName = firstName + " " + lastName; // fullName will be "Alice Smith"
    - Substring Extraction: Getting a portion of a string.
        - Substring Extraction: Getting a portion of a string.
        - string message = "Hello, world!";
          string greeting = message.substr(0, 5); // greeting will be "Hello"
    - **C**omparison**:** Comparing strings for equality, inequality, or lexicographical order.
        - string str1 = "apple";
          string str2 = "banana";
          if (str1 == str2) {
            // ...
          } else if (str1 < str2) {
            // ...
          }
    - String Functions:
        - find(): Locates the first occurrence of a substring within a string.
            - string sentence = "The quick brown fox jumps over the lazy dog.";

        - size_t position = sentence.find("fox"); // position will be 16 (index of 'f' in "fox")
    - rfind(): Locates the last occurrence of a substring within a string
        - string text = "Hello, world! Hello again.";
          size_t lastPosition = text.rfind("Hello"); // lastPosition will be 21 (index of 'H' in the second "Hello")
    - erase(): Removes a substring from a string.
        - string message = "Hello, world!";
          message.erase(5, 7); // Removes "world!"

6. Introduction to Object-Oriented Programming

1) Explain the key concepts of Object-Oriented Programming (OOP).
    - Objects:
        - Think of objects as self-contained units that represent things in the real world.
        - They have data (attributes, like a person's name and age) and behavior (actions they can perform, like walking or talking).
        - In code, objects are instances of classes.
    - Classes:
        - Classes are blueprints or templates for creating objects.
        - They define the structure (data) and behavior (functions) that all objects of that class will share.
        - Think of a class as a cookie cutter – it defines the shape, but the actual cookies (objects) are created from it.
    - Encapsulation:
        - This is about bundling data and methods (functions) together within an object.
        - It controls access to the object's data, protecting it from unauthorized changes.
        - Imagine a safe – the data is inside, and only the methods (like a lock and key) can access it.
    - Inheritance:
        - This is like creating a new class based on an existing one.
        - The new class inherits all the data and methods from the parent class, but can add its own unique features.
        - It's like building on a foundation – you get the basic structure, but you can add your own design elements.
    - Polymorphism:

    - This means "many forms." It allows objects of different classes to be treated in a uniform way.
    - Imagine having a group of animals – you can call a "makeSound()" method on each, and they'll all respond appropriately, even though they make different sounds.

2) What are classes and objects in C++? Provide an example.
    - Class: The blueprint. It defines the structure (data) and behavior (functions) of a particular type of object. It's like a template for creating objects.
    - Object: The actual house built from the blueprint. It's an instance of a class, meaning it has the specific data and functions defined by the class.
3) What is inheritance in C++? Explain with an example.
    - Inheritance in C++ allows you to create new classes (called derived classes) that inherit properties and behaviors from existing classes (called base classes).
    - This promotes code reusability and helps create a hierarchical structure for your program.
    - Code Reusability: Inheritance allows you to reuse code from existing classes, saving time and effort.
    - Polymorphism: Inheritance enables polymorphism, where objects of different classes can be treated as objects of a common base class.
4) What is encapsulation in C++? How is it achieved in classes?
    - Encapsulation is like a protective shell around your data. It keeps your data safe and organized, allowing you to control how it's accessed and modified. In C++, encapsulation is achieved through classes.
    - Data Hiding: Classes in C++ allow you to declare data members (variables) as private. This means they can only be accessed and modified within the class itself. This prevents direct access from outside the class, protecting the data's integrity.
    - Access Control: You use public, private, and protected access specifiers to control how members of a class can be accessed.
        - Public members are accessible from anywhere
        - Private members are only accessible within the class itself.
        - Protected members are accessible within the class itself and by derived classes.
    - Member Functions: You create member functions (methods) within the class to provide controlled access to the private data.

These functions act as the interface to the data, ensuring that it's manipulated in a safe and consistent manner.