# Module #3 Introduction to OOPS Programming

## THEORY EXERCISE:

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

•The key difference between Procedural Programming and Object-Oriented Programming (OOP) is that procedural programming focuses on a step-by-step execution of functions to achieve a task, while OOP organizes code into "objects" which encapsulate data and the operations that can be performed on that data, representing real-world entities and their interactions, allowing for greater code reusability and flexibility.

### Key Point of Procedural Programming

**Structure**:

Procedural programming breaks code into functions, whereas OOP breaks code into objects, which are instances of classes containing data (attributes) and methods (functions).

**Data Handling:**

In procedural programming, data can move freely between functions, while in OOP, data is tightly bound to the object and accessed through controlled methods, promoting data protection.

**Approach:**

Procedural programming generally takes a top-down approach, defining a series of steps to follow, while OOP often uses a bottom-up approach, building complex systems from smaller, reusable objects.

### Key Point of OOP

Inheritance (creating new classes based on existing ones), Polymorphism (allowing objects to be used in different contexts), Encapsulation (hiding internal object details and providing controlled access).

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

Object-Oriented Programming (OOP) offers significant advantages over Procedural Programming (POP) primarily due to its ability to organize code into reusable, modular objects, promoting better code maintainability, scalability, and flexibility through features like inheritance, polymorphism, and encapsulation, which are largely absent in POP; making it more suitable for complex software projects.

**Key Advantages of OOP over POP**

**Code Reusability**

OOP allows developers to create reusable code components (classes and objects) that can be used across different parts of a project, reducing redundancy and development time.

**Modularity**

By breaking down complex systems into smaller, self-contained objects, OOP enhances code organization and makes it easier to understand and maintain.

**Inheritance**
This mechanism enables new classes to inherit properties and methods from existing classes, promoting code reuse and creating hierarchical relationships between objects.

**Polymorphism**
Different objects can respond to the same method call in different ways, allowing for greater flexibility and adaptability in code.

**Encapsulation**
Data within an object is protected by restricting direct access, enhancing data integrity and security.

**Better Maintainability**
With modular design and well-defined relationships between objects, OOP code is easier to modify and update without causing unintended side effects.

**Scalability**
OOP facilitates the addition of new features or functionalities to existing systems without major code overhauls, making it suitable for large and complex projects.

**Improved Problem Solving**
By modeling real-world entities as objects, OOP allows developers to approach problems from a more intuitive perspective, leading to cleaner and more logical code.

**Collaboration**
Multiple developers can work on different parts of a system simultaneously by creating separate classes and objects, improving team productivity.

**Important distinction**: While POP can be efficient for simple tasks with linear logic, it can become difficult to manage in larger projects due to its lack of modularity and reusability features.

**3. Explain the steps involved in setting up a C++ development environment.**
C++ is a powerful, high-performance programming language widely used in system software, game development, and competitive programming. Before diving into coding, it's essential to set up a proper development environment. This guide will walk you through installing a C++ compiler, choosing an IDE, and writing your first program.
**Step 1: Choose a C++ Compiler**

A compiler translates C++ code into machine-readable instructions. Some popular C++ compilers include:

GCC (GNU Compiler Collection) – Available for Linux, macOS, and Windows (via MinGW or Cygwin).

**Clang –** A modern compiler known for its speed and error diagnostics.

MSVC (Microsoft Visual C++) – Comes with Microsoft Visual Studio, widely used on Windows.

**Step 2: Install a Compiler**

For Windows Users

Option 1: Install MinGW-w64 (GCC for Windows)

Download MinGW-w64 from its official website and install it.

Add MinGW to the system PATH to allow it to run from the command prompt.

Option 2: Install Microsoft Visual Studio

Download and install Visual Studio from Microsoft's website.

Select C++ development tools during installation.

For macOS Users

Option 1: Install Xcode Command Line Tools

Open the terminal and install the Xcode command-line tools.

Option 2: Install GCC via Homebrew

Install Homebrew, then install GCC using the package manager.

For Linux Users

Most Linux distributions come with GCC pre-installed. If not, it can be installed using the system's package manager:

Ubuntu/Debian: Install GCC using the apt package manager.

Fedora: Install GCC using dnf.

Arch Linux: Install GCC using pacman.

**Step 3: Choose a Code Editor or IDE**

While a basic text editor can be used for writing C++ code, an Integrated Development Environment (IDE) enhances productivity with features like syntax highlighting, debugging, and auto-completion. Popular choices include:

Visual Studio Code (VS Code) – A lightweight editor with C++ support via extensions.
Code::Blocks – A simple and beginner-friendly IDE.
Dev-C++ – A classic IDE for C++ programming.

CLion – A professional-grade IDE from JetBrains.

Eclipse CDT – A plugin-based IDE for C++ development.

Setting Up VS Code for C++ Development

Install VS Code from its official website.

Install the C/C++ extension from the Extensions Marketplace.

Configure the compiler and debugger settings for a seamless experience.

Step 4: Write and Compile Your First C++ Program

Open your chosen IDE or text editor.

Create a new C++ file and write a simple program.

Use the installed compiler to compile and run the program.

Step 5: Debugging Tools

For debugging, GDB (GNU Debugger) is widely used.

Install GDB through the system's package manager.

Compile programs with debugging symbols enabled.

Use GDB or an integrated debugger within the IDE for error detection.

**4. What are the main input/output operations in C++? Provide examples.**
In C++, the primary input/output operations are performed using the standard input stream (cin) to read data from the user (typically the keyboard) and the standard output stream (cout) to display data on the screen, both utilizing the extraction (>>) and insertion (<<) operators respectively; for example, to take an integer input from the user and print it back, you would use cin >> number; and cout << number;

**- cin (Standard Input):**

Used to read data from the keyboard.

- **cout (Standard Output):**

Used to display data on the screen.

**Stream Operators:**

>> (extraction operator): Reads data from an input stream into a variable.
<< (insertion operator): Writes data to an output stream.

# 2. Variables, Data Types, and Operators

1. **What are the different data types available in C++? Explain with examples.**
In C++, the primary data types include: integers (int), floating-point numbers (float, double), characters (char), Boolean values (bool), with each serving a specific purpose for storing different kinds of data; for example, an "int" stores whole numbers like 10, while a "char" stores a single character like 'a'.

Explanation with examples:

Integer (int): Used for whole numbers without decimal points.

Example: int age = 25;

Floating-point (float): Used for decimal numbers with limited precision.

Example: float pi = 3.14159;

Double-precision floating-point (double): Similar to float but with higher precision for decimal values.

Example: double gravity = 9.81;

Character (char): Used to store a single character like a letter, number, or symbol.

Example: char letter = 'A';

Boolean (bool): Represents logical values "true" or "false".

Example: bool isTrue = true;

Other important data types in C++:

Unsigned integer (unsigned int): Stores only positive integer values.

Example: unsigned int counter = 10;

Short integer (short): Smaller integer type, useful for memory optimization

Example: short smallNum = 100;

Long integer (long): Larger integer type for very large numbers

Example: long bigNum = 1234567890;

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

| **Implicit Type Conversion** | **Explicit Type Conversion** |
|---|---|
| An implicit type conversion is automatically performed by the compiler when differing data types are intermixed in an expression. | An explicit type conversion is user-defined conversion that forces an expression to be of specific type. |

| | |
|---|---|
| An implicit type conversion is performed without programmer's intervention. | An explicit type conversion is specified explicitly by the programmer. |
| Example:<br>a, b = 5, 25.5<br>c = a + b | Example:<br>a, b = 5, 25.5<br>c = int(a + b) |

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

| Operator | Name | Example |
|---|---|---|
| * | Multiplication | x * y |
| / | Division | x / y |
| % | Modulus | x % y |
| ++ | Increment | ++x |

**Assignment Operators**

Assignment operators are used to assign values to variables.

In the example below, we use the assignment operator (=) to assign the value 10 to a variable called x:

**Comparison Operators**

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either 1 or 0, which means true (1) or false (0). These values are known as Boolean values, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the greater than operator (>) to find out if 5 is greater than 3:

**Logical Operators**

As with comparison operators, you can also test for true (1) or false (0) values with logical operators.

Logical operators are used to determine the logic between variables or values:

| Operator | Name | Description | example |
|---|---|---|---|
| && | Logical and | Returns true if both statements are true | x < 5 && x < 10 |
| \|\| | Logical or | Returns true if one of the statements is true | x < 5 \|\| x < 4 |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |

**4. Explain the purpose and use of constants and literals in C++.**
constants" are named variables whose value cannot be changed once assigned, used to represent

fixed values within a program, while "literals" are directly written fixed values in the code, like numbers or characters, which can be used to initialize variables or directly in expressions; essentially, constants provide a meaningful name for a fixed value, while literals are the raw representation of that value within the code itself.

**Key points about constants:**

Declared with "const" keyword:

To create a constant in C++, you use the const keyword before the variable declaration, like const int MAX_VALUE = 100.

**Improved code readability:**

Using constants with descriptive names makes code easier to understand and maintain, especially when the same value is used multiple times in the program.

**Compile-time checks:**

The compiler will catch attempts to modify a constant, preventing runtime errors.

**Key points about literals:**

Directly written in code:

Literals are values like numbers (e.g., 10), characters (e.g., 'a'), or strings (e.g., "Hello") that are directly written in the code without needing a separate variable declaration.

Different types of literals:

C++ supports various types of literals including integer literals, floating-point literals, character literals, string literals, and boolean literals.

Used for initialization:

Literals are commonly used to initialize variables or as operands in expressions.

# 3. Control Flow Statements

**1. What are conditional statements in C++? Explain the if-else and switch statements.**
Conditional statements in C++ are decision-making statements that control the flow of a program by evaluating conditions. They are also known as decision control statements.

**if-else: -**
Functionality: Executes a block of code if a specific condition is true, and optionally executes another block of code if the condition is false (using the "else" keyword).

**Syntax:**

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

**Switch statement:**

Functionality: Compares a variable's value against a list of constant values ("cases") and executes the corresponding code block for the matching case.

**Syntax:**

```
switch (expression) {

    case value1:

       // Code to execute if expression equals value1
       break;
    case value2:
       // Code to execute if expression equals 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 in Programming:**

The for loop is used when you know in advance how many times you want to execute the block of code.

It iterates over a sequence (e.g., a list, tuple, string, or range) and executes the block of code for each item in the sequence.

The loop variable (variable) takes the value of each item in the sequence during each iteration.

**For Loop Syntax:**

```
for (initialization; condition; increment/decrement) {
   // Code to be executed repeatedly
}
```

**While Loop in Programming:**

The while loop is used when you don't know in advance how many times you want to execute the block of code. It continues to execute as long as the specified condition is true.

It's important to make sure that the condition eventually becomes false; otherwise, the loop will run indefinitely, resulting in an infinite loop.

**While Loop Syntax:**

```
while (condition){
   # Code to be executed while the condition is true
}
```

**Do-While Loop in Programming:**

The do-while loop is similar to the while loop, but with one key difference: it guarantees that the block of code will execute at least once before checking the condition.

This makes it useful when you want to ensure that a certain task is performed before evaluating a condition for continuation.

The loop continues to execute as long as the specified condition is true after the first execution. It's crucial to ensure that the condition eventually becomes false to prevent the loop from running indefinitely, leading to an infinite loop.

**Syntax of do…while Loop:**

```
do {
   // body of do-while loop
} while (condition);
```

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

In programming, "break" and "continue" statements are used within loops to control the flow of execution, where "break" immediately exits the entire loop when a certain condition is met, while "continue" skips the remaining part of the current iteration and jumps to the next iteration of the loop.

**Example with "break":**

```
for i in range(10):
   if i == 5:
       break  # Exit the loop when i reaches 5
     print(i)
```

**Example with "continue":**

```
for i in range(10):

   if i % 2 == 0:  # Skip even numbers

      continue

   print(i)
```

**4. Explain nested control structures with an example.**

A nested control structure refers to placing one control structure (like an if-statement or loop) entirely within another control structure, essentially creating a layered decision-making process where the inner structure is executed based on the conditions of the outer structure; for example, using an "if" statement inside a "for" loop to check specific conditions within each iteration of the loop.

**Example:-**

```
for i in range(1, 11):  # Outer loop for rows

   for j in range(1, 11):  # Inner loop for columns

      print(i * j, end=" ")

   print()
```

**Key points about nested control structures:**

Increased complexity:

Nesting allows for more intricate logic by combining conditions and iterations, making your code handle more complex scenarios.

Readability is crucial:

Proper indentation and clear variable naming are important to understand the flow of nested structures.

Different combinations:

You can nest various control structures like "if-else" within "while" loops or "for" loops within "if" statements.

# 4. Functions and Scope

**1. What is a function in C++? Explain the concept of function declaration, definition, and calling.**
In C++, a function is a block of code that performs a specific task, allowing you to reuse code by calling it multiple times throughout your program; it can take input values (parameters) and optionally return a value as output; a function declaration specifies the function's name, return type, and parameters, while the definition includes the actual code that executes when the function is called; "calling" a function means using its name in your code to execute the code block it represents.

**Key points about functions in C++:**

**Function Declaration:**

A function declaration tells the compiler about the function's existence, including its name, return type, and parameter list, but does not include the actual code within the function.

Example: int calculateSum(int num1, int num2);

**Function Definition:**

The function definition provides the actual code that executes when the function is called, including the function body with the logic to perform the desired task.

Example:

Code

```
int calculateSum(int num1, int num2) {

    return num1 + num2;

}
```

**Calling a Function:**

To use a function in your code, you "call" it by writing its name followed by parentheses, which may include arguments (values passed to the function).

Example:

```
int result = calculateSum(5, 3); // This will call the calculateSum function
```

**2. What is the scope of variables in C++? Differentiate between local and global scope.**
In C++, the "scope of a variable" refers to the region of the program where that variable can be accessed and used; a variable declared within a specific block (like a function) has a "local scope" and

is only accessible within that block, while a variable declared outside any function has a "global scope" and can be accessed from anywhere in the program.

Key points about variable scope:

Local Scope:

Variables declared inside a function or block are considered local and can only be used within that block.

Once the function or block exits, the local variable ceases to exist.

This is generally preferred for variables that are only needed within a specific part of the code to improve code organization and avoid unintended modifications.

Global Scope:

Variables declared outside any function are global and can be accessed from anywhere in the program.

While convenient for sharing data across different parts of the program, excessive use of global variables can lead to code complexity and potential issues with debugging due to their wide accessibility.

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

In C++, recursion is a programming technique where a function calls itself repeatedly until a specific base case condition is met, effectively breaking down a complex problem into smaller, similar subproblems that are solved iteratively; a common example is calculating the factorial of a number, where the function calls itself with a decreasing value until it reaches 1, which is the base case.

**Example: Calculating Factorial using Recursion**

```
int factorial(int n) {

  if (n == 0) {

    return 1; // Base case: when n is 0, return 1

  } else {

    return n * factorial(n - 1); // Recursive case: call the function with n-1

  }

}
```

```
int main() {

   int number = 5;

   int result = factorial(number);

   std::cout << "Factorial of " << number << " is: " << result << std::endl;

   return 0;
```

**Key points about recursion**:

Base Case: Every recursive function needs a base case to prevent infinite recursion.

Recursive Call: The function calls itself with a modified input to gradually approach the base case.

Stack Management: When a function calls itself recursively, each call is stored on the call stack until the base case is reached.

**4. What are function prototypes in C++? Why are they used?**
function prototype in C++ is a declaration that specifies a function's name, return type, and parameter data types, essentially acting as a blueprint for the function without including the actual function body; it's used to inform the compiler about the function's signature before the function definition, allowing for better type checking and ensuring correct function calls throughout the code.

**Purpose:**

To provide the compiler with essential information about a function, including its name, return type, and parameter types, so it can validate function calls during compilation.

**Example of a function prototype:**

Code

```
int calculateSum(int num1, int num2);
```

# 5. Arrays and Strings

1. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.
An array can be defined as a group or collection of similar kinds of elements or data items that are stored together in contiguous memory spaces. All the memory locations are adjacent to each other, and the number of elements in an array is the size of the array.

Example:

**Single-dimensional array**
one-dimensional array stores elements in a single line.

Code

```
int numbers[5] = {1, 2, 3, 4, 5}; // Stores 5 integers in a single line
```

**To access the second element: numbers Multi-dimensional array (2D):-**

a multi-dimensional array stores them in a table format.

Code

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; // Represents a 2x3 matrix
```

**2. Explain string handling in C++ with examples.**
string handling refers to the process of manipulating and working with sequences of characters using the std::string class, which is part of the standard library and allows for easy declaration, assignment, and manipulation of strings through various member functions; to use strings, you need to include the <string> header file in your code.

| **Function** | **Syntax (or) Example** | **Description** |
|---|---|---|
| strset() | strset(string1, 'B') | Sets all the characters of string1 to given char B |
| strnset() | strnset(string1, 'B', 5) | Sets first 5 characters of string1 to given character 'B'. |
| strrev() | strrev(string1) | It reverses the value of string1 |

**3. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.**
In C++, arrays are initialized by declaring them with a data type, name, size, and then providing a list of values enclosed in curly braces, where each value corresponds to an element in the array; for a 1D array, simply list the values, while for a 2D array, use nested braces to represent rows and columns.

**Example of a 1D array:**

Code

```
int numbers[5] = {10, 20, 30, 40, 50}; // Declares an array named "numbers"
```

**Example of a 2D array:**

Code

```
int matrix[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}; // A 2D array
```

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

In C++, string operations refer to manipulating text data (strings) using built-in functions provided by the <string> header file, allowing actions like concatenating strings, finding substrings, comparing strings, extracting characters, and modifying their content; common string functions include find(), substr(), append(), compare(), length(), and erase() which can be used on std::string objects to perform these operations.

Key points about string operations in C++:

String type: C++ uses the std::string class to represent strings, which is a more convenient and robust way to handle text compared to C-style character arrays.

Header file: To use string functions, include the <string> header file in your code.

**Basic string operations:**

**Accessing characters:**

string[index]: Accesses the character at the specified index.

string.at(index): Similar to [], but throws an exception if the index is out of bounds.

Concatenation:

string1 += string2: Appends string2 to string1

string1.append(string2): Another way to append strings

Length:

string.length(): Returns the number of characters in the string

**Common string functions and their functionalities:**

find(substring): Returns the index of the first occurrence of a substring within the string

rfind(substring): Returns the index of the last occurrence of a substring

Substrings:

substr(start, length): Extracts a substring starting at a given index with a specified length

Comparison:

compare(string): Compares two strings, returning a negative value if the current string is less than the argument, 0 if equal, and a positive value if greater

Modification:

erase(start, length): Removes a substring from the string

insert(position, string): Inserts a new string at a specific position

# 6. Introduction to Object-Oriented Programming

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

Object-Oriented Programming (OOP) revolves around the concept of "objects," which are entities that encapsulate data (attributes) and behavior (methods), and key concepts include: abstraction, encapsulation, inheritance, and polymorphism; essentially allowing developers to design software by modeling real-world entities and their interactions, promoting code reusability and maintainability.

**Key Concepts:**

**Class:**

A blueprint or template that defines the properties and behaviors (methods) that an object will have.

**Object:**

An instance of a class, meaning a concrete representation of the data and methods defined in the class.

**Abstraction:**

Focusing on the essential features of an object and hiding unnecessary details, providing a simplified interface for users.

**Encapsulation:**

The practice of bundling data (attributes) with the methods that operate on them within a class, protecting data integrity by restricting direct access to internal variables.

**Inheritance:**

The ability of a new class to inherit properties and methods from an existing class (parent class), allowing for code reuse and creating hierarchical relationships between classes.

**Polymorphism:**

The ability of an object to take on multiple forms, allowing the same method to behave differently depending on the context or object type.

**2. What are classes and objects in C++? Provide an example.**
A "class" is a blueprint or template that defines the properties (data members) and behaviors (member functions) of a group of objects, while an "object" is an actual instance of that class, meaning a concrete entity with specific values for the data members and the ability to perform the defined actions.

**Class syntax: -**

```
class ClassName {
    access_specifier:
    // Body of the class
};
```

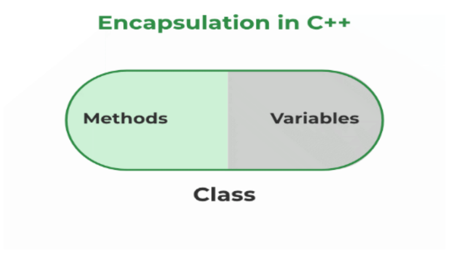
**Object : -**

```
ClassName ObjectName;
```

3. What is inheritance in C++? Explain with an example.
Inheritance is a mechanism that allows a new class (called a derived class or child class) to inherit properties and methods from an existing class (called a base class or parent class), essentially enabling code reuse by creating a hierarchical relationship between classes; this means a derived class can access and extend the functionality of the base class while adding its own unique features.

```
class Animal {

public:

   string name;
   string color;

   void eat() {

       cout << name << " is eating." << endl;
   }
};

class Dog : public Animal {
public:
 void bark() {

       cout << name << " barks!" << endl;
   }
};

int main() {

   Dog myDog;

   myDog.name = "Max";

   myDog.color = "Brown";

   myDog.eat();  // Access inherited method from Animal

   myDog.bark(); // Specific Dog behavior

   return 0;

}
```

4.What is encapsulation in C++? How is it achieved in classes?
encapsulation is the practice of bundling data members (variables) within a class as "private" and providing controlled access to them through public member functions (methods), effectively hiding the internal details of an object and ensuring data integrity by only allowing modifications through defined operations.


Encapsulation in C++
Methods
Variables
Class