Module #3 Introduction to OOPS Programming

THEORY EXERCISE:

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

Procedural Programming	Object-Oriented Programming (OOP)
Focuses on functions and procedures	Focuses on objects and classes
Data and functions are separate	Data and methods are combined into objects
Uses a top-down approach	Uses a bottom-up approach
Less secure as data is easily accessible	More secure due to data hiding (encapsulation)
Code reusability is limited	High code reusability using inheritance
Difficult to manage large programs	Easier to manage large and complex programs
Examples: C, Pascal	Examples: C++, Java, Python

Explanation:

- **Procedural Programming** follows a step-by-step execution of instructions and is suitable for small and simple applications.
- **Object-Oriented Programming (OOP)** models real-world entities using objects and is suitable for large, complex, and scalable application

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

1. Data Security
   OOP provides better data security through *encapsulation*, which restricts direct access to data.
2. Code Reusability
   OOP supports *inheritance*, allowing existing classes to be reused and extended.
3. Modularity
   Programs are divided into objects, making the code easier to manage and understand.

4. Easy Maintenance
   Changes in one part of the program do not affect other parts, making maintenance easier.
5. Real-World Modeling
   OOP represents real-world entities using objects, making program design more natural and efficient.
6. Scalability
   OOP is suitable for large and complex applications due to its structured approach.

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

1. **Install a C++ Compiler**
   Install a compiler such as **GCC**, **MinGW**, or **Clang** to convert C++ code into executable programs.
2. **Install an IDE or Code Editor**
   Use an IDE like **Visual Studio**, **Code::Blocks**, or a code editor such as **VS Code** for writing and managing code.
3. **Configure the Compiler**
   Set the compiler path correctly so the system can access it from the command line or IDE.
4. **Create and Write a C++ Program**
   Create a .cpp file and write a simple C++ program.
5. **Compile and Run the Program**
   Compile the code using the compiler and execute the program to check the output.

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

Main Input/Output Operations in C++

In C++, input and output operations are performed using the iostream library.

```
#include <iostream>

using namespace std;
```

```
int main() {
   int x;
   cout << "Enter a number: ";
   cin >> x;
   cout << "You entered: " << x;
   return 0;
}
```

THEORY EXERCISE:

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

1. Basic (Primitive) Data Types

Used to store simple values.

- int – stores integers

```
int a = 10;
```

- float – stores decimal numbers

```
float b = 5.5;
```

- double – stores large decimal values

```
double c = 10.25;
```

- char – stores a single character

```
char ch = 'A';
```

- bool – stores true or false

```
bool isOn = true;
```

2. Derived Data Types

Derived from basic data types.

- Array

```
int arr[3] = {1, 2, 3};
```

- Pointer

```
int *p = &a;
```

- Function

3. User-Defined Data Types

Created by the user.

- struct

```
struct Student {
   int id;
   char name[20];
};
```

- union
- enum
- typedef

4. Void Data Type

Represents no value.

```
void display() {
  // no return value
}
```

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

1. Implicit Type Conversion

- Automatically performed by the compiler.
- Occurs when converting a smaller data type to a larger data type.
- No data loss in most cases.

Example:

```
int a = 10;
float b = a;   // implicit conversion
```

2. Explicit Type Conversion (Type Casting)

- Performed by the programmer manually.
- Used when converting a larger data type to a smaller data type.
- May cause data loss.

Example:

```
float x = 5.7;
int y = (int)x;   // explicit conversion
```

Key Differences:

Implicit Conversion	Explicit Conversion

Done automatically by compiler	Done manually by programmer
Safer	Risk of data loss
No casting syntax needed	Requires casting

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

1. Arithmetic Operators

Used to perform mathematical operations.

Operator	Example
+	a + b
-	a - b
*	a * b
/	a / b
%	a % b

2. Relational Operators

Used to compare two values.

Operator	Example
==	a == b
!=	a != b
<	a < b
>	a > b
<=	a <= b
>=	a >= b

3. Logical Operators

Used to combine conditions.

Operator	Example
&&	a > 0 && b > 0
`	
!	!a

4. Assignment Operators

Used to assign values to variables.

Operator	Example
=	a = 5
+=	a += 2
-=	a -= 2
*=	a *= 2
/=	a /= 2

5. Unary Operators

Operate on a single operand.

Operator	Example
++	++a
--	--a

6. Bitwise Operators

Used for bit-level operati

Operator	**Example**
&	a & b
`	`
^	a ^ b
<<	a << 1
>>	a >> 1

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

Constants

- Constants are variables whose values cannot be changed once assigned.
- They are created using the const keyword.
- Used to protect data from modification and improve code readability.

Example:

```
const int MAX = 100;
```

Purpose of Constants:

- Prevent accidental modification of values
- Make programs safer and easier to maintain
- Improve code clarity

Literals

- Literals are fixed values written directly in the program code.
- They represent constant values of different data types.

Examples:

```
10        // integer literal
3.14      // floating-point literal
'A'       // character literal
"Hello"   // string literal
true      // boolean literal
```

Purpose of Literals:

- Provide direct values for calculations and assignments
- Make code simple and easy to understand

Difference (in brief):

- Constants are named fixed values.
- Literals are unnamed fixed values written directly in code.

THEORY EXERCISE:

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

1. if-else Statement

- Used to execute a block of code when a condition is true; otherwise, another block is executed.
- Suitable for checking conditions and ranges.

Syntax:

```
if (condition) {
  // true block
} else {
  // false block
}
```

Example:

```
int age = 18;
if (age >= 18)
  cout << "Eligible to vote";
else
  cout << "Not eligible";
```

2. switch Statement

- Used to select one block of code from multiple options.
- Works with constant values (cases).

Syntax:

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // code
}
```

Example:

```
int day = 2;
switch(day) {
  case 1: cout << "Monday"; break;
  case 2: cout << "Tuesday"; break;
  default: cout << "Invalid day";
}
```

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

Feature	for Loop	while Loop	do-while Loop
Condition check	Before loop	Before loop	After loop
Loop execution	May execute zero times	May execute zero times	Executes at least once
Use case	When iterations are known	When iterations are unknown	When loop must run once
Control type	Entry-controlled	Entry-controlled	Exit-controlled

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

break Statement

- Immediately terminates the loop.
- Control comes out of the loop.

Example:

```
for(int i = 1; i <= 5; i++) {
  if(i == 3)
    break;
  cout << i << " ";
}
```

Output: 1 2

continue Statement

- Skips the current iteration.
- Control moves to the next iteration of the loop.

Example:

```
for(int i = 1; i <= 5; i++) {
  if(i == 3)
    continue;
  cout << i << " ";
}
```

Output: 1 2 4 5

Difference (in short):

- break → exits the loop completely
- continue → skips only the current iteration

4. Explain nested control structures with an example.

Nested Control Structures in C++

Nested control structures mean using one control statement (if, for, while, etc.) inside another control statement.

Explanation:

- When a control structure is placed within another, it is called nesting.
- Used to check multiple conditions or perform repeated tasks inside another loop or condition.

Example:

```
for(int i = 1; i <= 3; i++) {
  for(int j = 1; j <= 2; j++) {
    cout << i << " " << j << endl;
  }
}
```

THEORY EXERCISE:

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

1. Function Declaration

- Tells the compiler about the function name, return type, and parameters.
- Also called a function prototype.

Example:

```
int add(int, int);
```

2. Function Definition

- Contains the actual code of the function.
- Defines what the function does.

Example:

```
int add(int a, int b) {
  return a + b;
}
```

3. Function Calling

- Executes the function by using its name and passing arguments.

Example:

```
int sum = add(10, 20);
```

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

Feature	Local Variable	Global Variable
Declaration	Inside a function/block	Outside all functions
Accessibility	Only inside the function/block	Anywhere in the program
Lifetime	Exists during function execution	Exists throughout the program
Default Value	Garbage (if not initialized)	0 (for basic types)

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

Recursion in C++

Recursion is a process in which a function calls itself to solve a problem.
It is used when a problem can be broken into smaller, similar sub-problems.

Key Points:

- A recursive function must have a base case to stop recursion.
- Recursive calls continue until the base case is reached.

Example: Factorial of a Number

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

// Recursive function
int factorial(int n) {
   if(n == 0)  // base case
      return 1;
   else
      return n * factorial(n - 1); // recursive call
}

int main() {
   int num = 5;
```

```
    cout << "Factorial of " << num << " is " << factorial(num);
    return 0;
}
```

Output:

```
Factorial of 5 is 120
```

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

Function Prototypes in C++

A function prototype is a declaration of a function that tells the compiler about:

- Function name
- Return type
- Parameters

It does not contain the body of the function.

Syntax:

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

Example:

```
int add(int, int); // function prototype
```

Purpose of Function Prototypes:

1. Inform the compiler about a function before its actual definition.
2. Allow functions to be called before they are defined.
3. Help type checking of function arguments during compilation.

Example with Prototype:

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

int add(int, int); // prototype

int main() {
    cout << add(5, 10);
    return 0;
}

int add(int a, int b) { // definition
    return a + b;
}
```

Output: 15

THEORY EXERCISE:

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

Arrays in C++

An array is a collection of elements of the same data type stored in contiguous memory locations.
It allows storing multiple values in a single variable.

Difference between 1D and

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	Single row of elements	Rows and columns (table-like)
Indexing	Single index (i)	Two or more indices (i, j)
Example	int arr[5]	int arr[2][3]
Use	Simple lists	Tables, matrices

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

String Handling in C++

In C++, **strings** are used to store **a sequence of characters**. Strings can be handled using:

1. **C-style strings** (character arrays)
2. **C++ string class** (from <string> library)

1. C-Style Strings

- Stored as **character arrays** ending with a **null character \0**.
- Can use **string functions** from <cstring> like strlen(), strcpy(), strcmp().

Example:

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

int main() {
   char str1[20] = "Hello";
   char str2[20];

   strcpy(str2, str1);      // copy string
   cout << "Length: " << strlen(str1); // length
   cout << "Copied String: " << str2;
   return 0;
}
```

2. C++ string Class

- Part of <string> library.
- Provides easy functions for string handling: .length(), .append(), .substr(), .find(), etc.

Example:

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

int main() {
   string str1 = "Hello";
   string str2 = " World";

   str1 += str2;  // concatenate strings
   cout << str1;  // Output: Hello World
   cout << "Length: " << str1.length();
   return 0;
}
```

THEORY EXERCISE:

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

Key Concepts of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on objects, which represent real-world entities. The main concepts of OOP are:

1. Class

- A blueprint or template for creating objects.
- Defines attributes (data members) and behaviors (methods).

Example:

```
class Car {
  public:
   string color;
   void start() { cout << "Car started"; }
};
```

2. Object

- An instance of a class.
- Represents a real-world entity with specific values.

Example:

```
Car myCar;
myCar.color = "Red";
myCar.start();
```

3. Encapsulation

- Combines data and functions into a single unit (class).
- Protects data from unauthorized access using access specifiers (private, public).

4. Inheritance

- Allows a class (child) to acquire properties and behaviors of another class (parent).
- Promotes code reusability.

Example:

```
class Vehicle { public: void fuel() {} };
class Car : public Vehicle {}; // Car inherits Vehicle
```

5. Polymorphism

- Ability of a function or object to take many forms.
- Two types:
  - Compile-time (function overloading, operator overloading)
  - Run-time (virtual functions)

6. Abstraction

- Hides implementation details and shows only essential features.
- Achieved using abstract classes and interfaces.

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

1. Class

- A class is a blueprint or template for creating objects.
- It defines attributes (data members) and behaviors (member functions).

Example:

```
class Car {
  public:
    string color;
    void start() {
        cout << "Car started";
    }
};
```

2. Object

- An object is an instance of a class.
- It represents a real-world entity with specific values.

Example:

```
Car myCar;        // creating object
myCar.color = "Red";
myCar.start();    // calling function
```

Output:

```
Car started
```

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

Inheritance in C++

Inheritance is an OOP concept where a child (derived) class acquires properties and behaviors of a parent (base) class.

- It promotes code reusability.
- The child class can also have its own members.

Syntax:

```
class DerivedClass : access_specifier BaseClass {
  // additional members
};
```

Example:

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

// Base class
class Vehicle {
 public:
  void fuel() {
    cout << "Vehicle is fueled\n";
  }
};

// Derived class
class Car : public Vehicle {
 public:
  void start() {
    cout << "Car started\n";
  }
};

int main() {
  Car myCar;
  myCar.fuel();  // inherited from Vehicle
```

```
    myCar.start(); // own function
    return 0;
}
```

Output:

```
Vehicle is fueled
Car started
```

Key Points:

- Single inheritance → one parent, one child
- Multiple inheritance → one child, multiple parents
- Inheritance helps reuse code and extend functionality without rewriting.