Module--3 Introduction to OOPS Programming

1. Introduction to C++:-

THEORY EXERCISE:

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

Ans. Procedural Oriented Programming:-

the program is divided into small parts called *functions*.

Procedural programming follows a *top-down approach*.

Adding new data and functions is not easy.

In procedural programming, overloading is not possible.

Examples: C, FORTRAN, Pascal, Basic, etc.

Object-Oriented Programming:-

the program is divided into small parts called *objects*.

Object-oriented programming follows a *bottom-up approach*.

Adding new data and functions is easy.

Overloading is possible in object-oriented programming.

Examples: C++, Java, Python, C#, etc.

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

Ans. oop(object-oriented programing):-

1. Modularity:-

Uses classes and objects, enabling modularity and reuse of code.

Once a class is written, it can be used multiple times without modification.

2. Encapsulation:-

Encapsulation bundles data and methods within classes.

3. Abstraction:-

It supports abstraction, allowing complex realities to be represented simply through classes and objects.

4. Polymorphism:-

Allows methods to be used in multiple forms, enhancing flexibility and integration with different modules.

Dynamic binding enables method overriding and runtime decision-making.

Pop(procedural oriented programming):-

1. Modularity:-

Functions are the primary building blocks, leading to less modularity and code reuse.

2. Encapsulation:-

Data is global or passed between functions, increasing the risk of accidental modifications and reducing security.

3. Abstraction:-

as functionality is implemented through sequential code, making modifications more complex and error-prone.

4. Polymorphism:-

Doesn't support polymorphism, leading to repetitive code and less flexibility in function usage

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

Ans. C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc.

1. Install a Compiler:-

GCC (GNU Compiler Collection): Common on Linux and available for Windows via MinGW or WSL.

2. Choose and Install an IDE or Text Editor:-

Visual Studio: Comprehensive IDE with MSVC integration (Windows).

Visual Studio Code: Lightweight and cross-platform, with C++ extensions.

3. Configure the Environment Path:-

Add the compiler's bin directory (e.g. C:\MinGW\bin) to the PATH environment variable.

4. Install Necessary Extensions:-

Visual Studio Code: Install the C/C++ extension by Microsoft for IntelliSense, debugging, and code navigation.

5. Verify the Installation:-

g++ --version.

6. Create and Compile a Test Program:-

```
#include <iostream>

using namespace std;

int main() {
   cout << "Hello, World!";
   return 0;
}
```

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

Ans. It is part of the std namespace and is connected to the standard output stream.

The insertion operator (<<) is used to send data to the output stream.

Example:-

```
#include <iostream>

using namespace std;
```

```
int main() {
   cout << "Hello,World!";
   int age = 25;
   cout << "Age=  " << age;
   return 0;
}
```

2. Variables, Data Types, and Operators:-

THEORY EXERCISE:

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

Ans. Data types specify the type of data that a variable can store.

1. **Primitive data types**
    - int: Stores whole numbers without decimal points
    - char: Stores individual characters, letters, or ASCII values
    - float: Stores decimal numbers with fewer digits
    - double: Stores decimal numbers with more digits
    - void: An incomplete type used for functions that do not return a value
2. **Derived data types**
    - Array: Derived from primitive data types
    - Function: Derived from primitive data types
    - Pointer: Derived from primitive data types
    - Reference: Derived from primitive data types
3. **User-defined data types**
    - Class: Defined by the user

- Structure: Defined by the user
- Union: Defined by the user

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

Ans.

Aspect	**Implicit Type Conversion**	**Explicit Type Conversion**
Trigger	Automatically done by the compiler	Done manually by the programmer
Safety	Safe if no data loss or precision issues	May cause loss of data or precision
Syntax	No special syntax is needed	Uses casting syntax (example: (type)variable)
Use cases	It happens with compatible types (example: int to float)	Used for type conversions that are not done automatically

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

Ans.

Arithmetic Operators: These operators are used to perform basic mathematical operations.

- + (Addition): Adds two operands.
- -(Subtraction): Subtracts the second operand from the first.
- *(Multiplication): Multiplies two operands.
- / (Division): Divides the numerator by the denominator.
- % (Modulus): Returns the remainder when one operand is divided by the other.

1) Arithmetic Operators:

+, -, *, /, %

Example:

```
#include<iostream>
using namespace std;
main()
{
	int a=18,b=26;
	cout<<"addition="<<a+b;
	cout<<"subtraction="<<a-b;
	cout<<"multiplication="<<a*b;
	cout<<"division="<<a/b;
	cout<<"modulus="<<a%b;
}
```

2) Assignment Operator: (Shorthand op)

	+=, -=, *=, /=, %=

	e. g x=x+y x+=y

```
#include<iostream>
using namespace std;
main()
{
	int a=18,b=26;
	a+=b;
	cout<<"a+=b :"<<a;
	a-=b;
	cout<<"a-=b :"<<a;
```

```
        a*=b;
        cout<<"a*=b :"<<a;
        a/=b;
        cout<<"a/=b :"<<a;
        }
```

3) Increment/Decrement Op. (by default it will take 1 only)

```
        ++, --
        a++, ++a, b--, --b
        a=5
        Prefix : ++a, --b
        Postfix : a++, b--
        Unary: a++ (one operand, one operator)
        binary: a+b (two operands, one operator)
```

4) Relational Operators: /Conditional/Comparision

```
        >, >=, <, <=, ==, !=
```

Example:

```
#include<iostream>
using namespace std;
main()
{
        int a=18,b=26;
        cout<<" a>b ="<<a>b;
        cout<<" a<b ="<<a<b;
        cout<<" a==b ="<<a==b;
        cout<<" a!=b ="<<a!=b;
```

}

5) Logical Operators

and - && - All the conditions have to be true.

or - || - One of the any conditions has to be true.

not -! - Logical making false result for true condition

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

Ans.

Aspects	**Constants**	**Literals**
Definition	A variable-like entity whose value cannot be changed	A fixed value directly used in the code
Mutability	Cannot be changed after initialization	Immutable by nature (example: 3.14 or 'A')
Usage	Used for names of fixed values (e.g., const int MAX = 10)	Directly used in the code (example: x = 3;)
Keywords	const, constexpr	No specific keyword; part of the value itself
Purpose	To make code more readable and prevent accidental changes	For representing fixed values in code
Example	const int max score = 100;	int x = 100; or float pi = 3.14159f;

3. Control Flow Statements:-

THEORY EXERCISE:

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

Ans.

1. if-else Statement:

The if-else statement is the most commonly used conditional statement.

It allows you to test a condition and execute one block of code if the condition is true, and another block if the condition is false.

Syntax:

```
if (condition)
{
    // if the condition is true
} else {
    // if the condition is false
}
```

2. switch Statement:

The switch statement provides an alternative to multiple if-else statements when you need to compare a single variable to many possible values.

It is more efficient and readable when you have a large number of conditions based on a single expression.

Syntax:

```
switch (expression) {
    case value1:
    // if expression == value1
    break;
    case value2:
    // if expression == value2
    break;
    case value3:
    // if expression == value3
```

```
break;
default:
// if the expression doesn't match any case
}
```

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

Ans.

Feature	**for Loop**	**while Loop**	**do-while Loop**
Condition check	Before each iteration	Before each iteration	After each iteration
Initialization	This can be done at the start of the loop	Done outside the loop	Done outside the loop
Increment/Decrement	Done in the loop itself	Done inside the loop	Done inside the loop
Use case	Iterating over a known range or count (e.g., array)	For conditions where you don't know the number of iterations	When at least one iteration is required, regardless of condition

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

Ans.

1. break Statement:

break immediately exits the loop, skipping any remaining iterations.

the break is executed, the loop terminates, and the program control moves to the first statement following the loop.

Example:

```
#include <iostream>
```

```
using namespace std;
main() {
	for (int i = 1; i <= 10; i++)
	{
	if (i == 8)
	{
		break;
	}
	cout << i<<" ";;
	}
	cout << "\n\n\t Loop terminated early";
return 0;
}
```

2. continue Statement:

Skips the current iteration and moves to the next iteration of the loop.

Example:

```
#include <iostream>
using namespace std;
main() {
	for (int i = 1; i <= 10; i++)
	{
	if (i == 8)
	{
		continue;
```

```
        }
        cout << i<<" ";;
        }
        cout << "\n\n\t Loop finished";
        return 0;
    }
```

4. Explain nested control structures with an example.

Ans.

Types of Nested Control Structures:

Nested if Statements: An if statement inside another if or else block.

Nested Loops: A loop inside another loop (either for, while, or do-while).

Mixed Nesting: A combination of conditionals inside loops, or loops inside conditionals

Example for nested if statement:

```
#include <iostream>
using namespace std;
main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;
    if (num > 0)
    {
```

```
            cout << "The number is positive.";
        }
        else
        {
            cout << "The number is not positive.";
        }
    }
```

Example for nested loop:

```
    #include <iostream>
    using namespace std;
    main()
    {
        int rows = 3;
        int cols = 4;
        for ( i = 1; i <= rows; i++)
        {
        for (int j = 1; j <= cols; j++)
        {
            cout << "* ";
        }
            cout <<"\n";
        }
    }
```

4. Functions and Scope:

THEORY EXERCISE:

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

Ans. A function in C++ is a block of code that performs a specific task.

Functions allow you to group related code, making programs more modular and easier to manage.

1. Function Declaration:

A function declaration tells the compiler about the function's name, return type, and parameters without providing the actual implementation.

Syntax:

```
return_type function_name(parameter1_type, parameter2_type, ...);
```

2. Function Definition:

A function definition provides the actual implementation of the function. It defines what the function does when called, including any computations or operations.

Syntax:

```
return_type function_name(parameter1_type, parameter2_type, ...)
{
    // Function body:
}
```

3. Function Calling:

Function calling means invoking a function in your program.

Invokes the function to execute its code in the program.

Syntax:

```
function_name(argument);
```

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

Ans. s

local and global scope:

Feature	**Local Scope**	**Global Scope**
Declaration	Inside a function, block, or loop.	Outside any function, generally at the top of the file.
Visibility	Accessible only within the block/function where declared.	Accessible throughout the entire program.
Lifetime	Exists only during the execution of the block or function.	Exists for the entire runtime of the program.
Access	Cannot be accessed by other functions.	Can be accessed by any function in the program.
Memory	Stored in the stack memory.	Stored in the data segment of the memory.

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

Ans. Recursion in C++ is a programming technique in which a function calls itself to solve a problem.

Calls itself to perform a task.

Has a base case, which is a condition that stops the recursion.

Example:

```
#include <iostream>

using namespace std;
```

```
int factorial(int n)
{
if (n == 0)
{
        return 1;
}
else
{
        return n * factorial(n - 1);
}
}
main()
{
        int num;
        cout << "Enter a number: ";
        cin >>num;
        int result = factorial(num);
        cout << "Factorial of " << num << " is: " << result;
}
```

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

Ans. At the top of the code, we declare the function prototypes for add(int, int) and greet().

This allows us to call these functions in main() before their actual definitions.

It is a declaration that allows the function to be called before its actual definition in the code.

Syntax:

return_type function_name(parameter1_type, parameter2_type, ...);

Why are they used:

1. To Inform the Compiler.

2. To Enable Forward Function Calls.

3. To Improve Code Organization.

4. For Type Checking.

5. Arrays and Strings:

THEORY EXERCISE:

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

Ans. An array is a collection of elements of the same data type stored in contiguous memory locations.

Arrays allow you to store multiple values in a single variable, simplifying data handling.

Basic Characteristics:

- Fixed size.
- All elements must be of the same data type.
- Elements are accessed using an index (starting from 0).

Differences Between Single-Dimensional and Multi-Dimensional Arrays:

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	A simple list of elements (linear).	An array of arrays (matrix, table, etc.).

Access Method	Accessed with one index (e.g., arr[0]).	Accessed with multiple indices (e.g., arr[0][0] for2D).
Usage	Used for storing simple lists of data.	Used for storing tables, matrices, or complex data structures.
Dimensionality	1 dimension.	More than 1 dimension (e.g., 2D, 3D).

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

Ans. String Handling in C++:

In C++, strings are sequences of characters.

In C++, strings are sequences of characters used to represent text.

The language provides two main ways to handle strings:

1. C-strings (null-terminated character arrays).
2. C++ string class (which is more modern and safer).

C-strings (Character Arrays):

Example:

```
#include <iostream>
using namespace std;
main()
{
    char s[]="hello";
    cout<< s<<endl;
}
```

C++ string Class (Standard Library):

Example:

```
#include <iostream>
using namespace std;
main()
{
        String str("hello");
        cout<< str;
}
```

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

Ans.

Initializing 1D Arrays in C++:

Syntax:

```
type array_name[size] = {value1, value2, value3, ...};
```

Example:

```
#include <iostream>
using namespace std;
main()
{
int arr[5] = {1, 2, 3, 4, 5};
for(int i = 0; i < 5; i++)
{
        cout <<"\n\n\t arr["<<i<<"]=" <<arr[i];
}
        cout <<"\n";
```

```
}
```

Initializing 2D Arrays in C++:

Syntax:

```
type array_name[rows][columns]
{
{value1, value2, ..., valueN},
{value1, value2, ..., valueN},
};
```

Example:

```
#include <iostream>
using namespace std;
main()
{
int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
for(int i = 0; i < 3; i++)
{
for(int j = 0; j < 3; j++)
{
        cout<<"\t"<<arr[i][j];
}
cout<<"\n";
}
}
```

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

Ans. C++ provides various ways to handle strings.

The language supports C-strings (arrays of characters) and the C++ string class from the Standard Library.

String:

group of characters. (%c-%s)

char c[20]; // string

c[0] c[1] c[2] .....

T O P C T E C H

Creating Strings:

Direct Initialization:

string str = "Hello, World!";

Default Initialization:

string str;

string functions

gets(): to read the string with white space.

puts(): to print the string with white spaces.

strlen(): to find out the length of the given string

strlwr(): to convert the given string into lower case

strupr(): to convert the given string into upper case

strrev(): to reverse the given string

strcpy(): To copy one string to another.

strcat(): to concatenate the given strings.

strcmp(): to compare the two strings.

stricmp(): to compare the two strings by ignoring the case.

6. Introduction to Object-Oriented Programming:

THEORY EXERCISE:

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

Ans.

1. Classes:

A class is a blueprint or template for creating objects.

It defines a data structure and functions that operate on the data.

Blueprint or template which is a collection data member & member function.

-data member: variable inside the class. (class member)

-member function: function (UDF) inside the class. (method)

2. object:

An object is an instance of a class.

Once a class is defined, you can create multiple objects of that class.

"Fruit" is a class, "apple" is an object.

3. Encapsulation:

A wrapping up of data & methods together in a single unit (class).

-Data hiding -Data protection

Encapsulation is implemented using access specifiers:

- **Public**: Members are accessible from outside the class.
- **Private**: Members are only accessible within the class.
- **Protected**: Members are accessible within the class and its derived classes

4. Inheritance:

Process to derive one class from another.

Parent Class / Super Class / Base Class

Child class / Subclass / Derived Class

Note:

Always make an object of a derived class.

Public derivation

Private derivation.

Types of Inheritance:

1) Single Inh. (Parent->Child)

2) Multilevel Inh. (GP -> Parent -> Child)

3) Multiple Inh. (Parent1, Parent2 -> Child)

4) Hierarchical Inh.

5) Hybrid Inh.

5. polymorphism:

One name has many forms.

Example:

add()

add(x, y)

add(x, y, z).

Types of Polymorphism :

1) Overloading (compile time poly. / Static binding):

-Method overloading/Function overloading

Function having the same name but can be differentiated by

arguments and data types.

2) Overriding (run time / Dynamic binding):

-Method overriding/function overriding :

-The same method is available in the base & derived class.

-Always with the concept of "Inheritance"

6. abstraction:

Data abstraction is one of the most essential features of object-oriented programming in C++.

Abstraction means displaying only essential information and ignoring the details.

Types of Abstraction:

1. Data abstraction – This type only shows the required information about the data and ignores unnecessary details.
2. Control Abstraction – This type only shows the required information about the implementation and ignores unnecessary details.

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

Ans.

1. Classes:

A class is a blueprint or template for creating objects.

It defines a data structure and functions that operate on the data.

Blueprint or template which is a collection data member & member function.

-data member: variable inside the class. (class member)

-member function: function (UDF) inside the class. (method)

2. object:

An object is an instance of a class.

Once a class is defined, you can create multiple objects of that class.

"Fruit" is a class, "apple" is an object.

Example:

```
#include<iostream>
```

```
using namespace std;
class employee
{
private:
	int id;
	char name[50];
	int salary;

public:
	void get_data()
	{
		cout<<"\n\n\t enter the id=";
		cin>>id;
		cout<<"\n\n\t enter the name=";
		cin>>name;
		cout<<"\n\n\t enter the salary=";
		cin>>salary;
	}

	void print_data()
	{
		cout<<"\n\n\t id="<<id;
		cout<<"\n\n\t name="<<name;
		cout<<"\n\n\t salary="<<salary;
```

```
        }
};
main()
{
        employee e;
        e.get_data();
        e.print_data();
}
```

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

Ans.

Inheritance:

Process to derive one class from another.

Parent Class / Super Class / Base Class

Child class / Subclass / Derived Class

Note:

Always make an object of a derived class.

Public derivation

Private derivation.

Types of Inheritance:

1) Single Inh. (Parent->Child)

2) Multilevel Inh. (GP -> Parent -> Child)

3) Multiple Inh. (Parent1, Parent2 -> Child)

4) Hierarchical Inh.

5) Hybrid Inh.

Example:

```
#include<iostream>
using namespace std;
class dept
{
        int d_id;
        string dname;
        string branch;
public:
void get_dept()
        {
                cout<<"\n\n\t enter the department id= ";
                cin>>d_id;
                cout<<"\n\n\t enter the department name= ";
                cin>>dname;
                cout<<"\n\n\t enter the department branch= ";
                cin>>branch;
        }

void print_dept()
        {
                cout<<"\n\n\t department id= "<<d_id;
                cout<<"\n\n\t department name= "<<dname;
                cout<<"\n\n\t department branch= "<<branch;
```

```
        }
};

class Employee : private dept
{
        int e_id;
        string ename;
public:
void get_emp()
        {
                cout<<"\n\n\t departments details----------";
                get_dept();
                cout<<"\n\n\t employee details----------";

                cout<<"\n\n\t enter the employee id= ";
                cin>>e_id;
                cout<<"\n\n\t enter the employee name= ";
                cin>>ename;
        }

void print_emp()
        {
                cout<<"\n\n\t departments details----------";
                print_dept();
```

```
                cout<<"\n\n\t employee details----------";


                cout<<"\n\n\t employee id= "<<e_id;
                cout<<"\n\n\t employee name= "<<ename;
        }


};


main()
{
        Employee E;
        E.get_emp();
        E.print_emp();
}
```

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

Ans.

Encapsulation is one of the fundamental concepts of Object-Oriented Programming (OOP).

A wrapping up of data & methods together in a single unit (class).

-Data hiding -Data protection

How Encapsulation is Achieved in C++ Classes:

In C++, encapsulation is achieved using access specifiers which control the access levels of class members (both attributes and methods).

There are three types of access specifiers in C++:

- **Public**: Members are accessible from outside the class.

- **Private**: Members are only accessible within the class.
- **Protected**: Members are accessible within the class and its derived classes.