**Module 3 Introduction to OOPS Programming**

**1. Introduction to C++**

**LAB EXERCISES:**

1. First C++ Program: Hello World

Write a simple C++ program to display "Hello, World!".

**#include<iostream> using namespace std; Int main() { cout << “Hello World”; return 0;**

**}**

1. Basic Input/Output

Write a C++ program that accepts user input for their name and age and then displays a personalized greeting.

:- #include <iostream

#include <string>

int main() {

std::string name;

int age;

std::cout << " enter your name: ";

std::getline(std::cin, name);

std::cout << "Please enter your age: ";

// Read the integer input for the age

std::cin >> age;

std::cout << "Hello, " << name << "! You are " << age << " years old." << std::endl;

return 0;

}

THEORY EXERCISE:

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

|  |  |
| --- | --- |
| **Procedural Oriented Programming (POP)** | **Object-Oriented Programming (OOP)** |
|  |  |
| In POP, the program is divided into small parts called ***functions***. | In OOP, the program is divided into small parts called ***objects***. |
|  |  |
| There is no access specifier in procedural programming. | Object-oriented programming has access specifiers like private, public, protected, etc. |
|  |  |
| Procedural programming does not have any proper way of hiding data so it is ***less secure***. | Object-oriented programming provides data hiding so it is ***more secure***. |
|  |  |
| In procedural programming, overloading is not possible. | Overloading is possible in object-oriented programming. |
|  |  |
| In procedural programming, there is no concept of data hiding and inheritance. | In object-oriented programming, the concept of data hiding and inheritance is used. |
| **Procedural Oriented Programming (POP)** | **Object-Oriented Programming (OOP)** |
|  |  |
| In procedural programming, the function is more important than the data. | In object-oriented programming, data is more important than function. |
|  |  |
| Procedural programming is based on the ***unreal world***. | Object-oriented programming is based on the ***real world***. |
|  |  |
| **Examples:** C, FORTRAN, Pascal, Basic, etc. | **Examples:** C++, Java, Python, C#, etc |

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

|  |  |
| --- | --- |
| **Advantage** | **Why It Matters** |
| Encapsulation | Protects data from unintended access |
| Inheritance | Promotes code reuse |
| Polymorphism | Enables flexibility and interface consistency |
| Abstraction | Simplifies code interaction and hides complexity |
| Modularity | Makes programs easier to manage and debug |
| Maintainability | Simplifies changes and bug fixes |
| Real-world modeling | Bridges software and real-world concepts |
| Scalability | Supports large-scale team development |

1. **the steps Explain involved in setting up a C++ development environment.**
2. Choose a Text Editor or IDE
3. Install a C++ Compiler
4. Install and Set Up an IDE
5. Write First C++ Program
6. Compile and Run

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

**1. Input with cin** cin is used to read user input. It uses the extraction operator (>>).

**Example:**

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

cout << "You entered: " << age << endl; return 0;

}

1. **Output with cout**

cout is used to display output to the console. It uses the insertion operator (<<).

**Example:**

#include <iostream> using namespace std; int main() { cout << "Hello, World!"

<< endl;

return 0;

}

2. Variables, Data Types, and Operators:

LAB EXERCISES:

1. Variables and Constants

Write a C++ program that demonstrates the use of variables and constants. Create variables of different data types and perform operations on them.

Ans:- #include <iostream>

using namespace std;

int main() {

int a = 10;

float fVariable = 20.5;

char charVar = 'A';

bool booleanVar = true;

const float PI = 3.14;

const double GRAVITY = 9.81;

cout << "Integer Variable: " << a << endl;

cout << "Float Variable: " << fVariable << endl; cout << "Character Variable: " << charVar << endl; cout << "Boolean Variable: " << booleanVar << endl; cout << "Value of PI: " << 3.14 << endl; cout << "Value of Gravity: " << GRAVITY << endl;

a = a + 5; fVariable = fVariable \* 2;

cout << "Updated Integer Variable: " << a << endl;

cout << "Updated Float Variable: " << fVariable << endl; if (booleanVar) { cout << "Boolean Variable is true" << endl;

} else { cout << "Boolean Variable is false" << endl;

}

}

1. Type Conversion

Write a C++ program that performs both implicit and explicit type conversions and prints the results.

Ans:-

#include <iostream>

using namespace std;

int main() {

int intVar = 10;

double doubleVar = intVar;

cout << "Implicit Type Conversion:" << endl;

cout << "Integer Variable: " << intVar << endl;

cout << "Double Variable (after implicit conversion): " << doubleVar << endl; double doubleValue = 20.99;

int intValue = (int)doubleValue;

cout << "\nExplicit Type Conversion:" << endl;

cout << "Double Value: " << doubleValue << endl;

cout << "Integer Value (after explicit conversion): " << intValue << endl;

}

3. Operator Demonstration

Write a C++ program that demonstrates arithmetic, relational, logical, and bitwise operators. Perform operations using each type of operator and display the results.

#include <iostream> using namespace std; int main() {

int a = 10, b = 3;

cout << "Arithmetic Operators:" << endl; cout << "a + b = " << a + b << endl;

cout << "a - b = " << a - b << endl;

cout << "a \* b = " << a \* b << endl;

cout << "a / b = " << a / b << endl; cout << "a % b = " << a % b << endl; cout << endl;

cout << "Relational Operators:" << endl; cout << "a == b: " << (a == b) << endl; cout << "a != b: " << (a != b) << endl; cout << "a > b: " << (a > b) << endl;

cout << "a < b: " << (a < b) << endl; cout << "a >= b: " << (a >= b) << endl; cout << "a <= b: " << (a <= b) << endl; cout << endl;

"Logical Operators:" << endl;

cout << "x && y = " << (x && y) << endl; cout << "x || y = " << (x || y) << endl; cout << "!x = " << (!x) << endl;

cout << "!y = " << (!y) << endl; cout << endl;

**THEORY EXERCISE:**

1. What are the different data types available in C++? Explain with examples. **1. Basic (Primitive) Data Types**

|  |  |
| --- | --- |
| **Type Description** | **Example** |
| int Integer values (whole numbers) | int age = 25; |
| float Floating-point numbers (decimal) | float height = 5.9f; |
| double Double-precision floating-point | double pi = 3.14159; |
| char Single character | char grade = 'A'; |
| bool Boolean values (true or false)    **2. Derived Data Types** | bool isReady = true; |
| **Type Description** | **Example** |
| array Collection of elements of same type | int numbers[5] = {1,2,3,4,5}; |
| pointer Stores memory address of a variable | int\* ptr = &age; |
| function Reusable block of code | int add(int a, int b); |
| reference Alias for another variable | int& ref = age; |

**3. User-defined Data Types**

**Type Description Example**

struct Group of related variables struct Person {string name; int age;};

|  |  |
| --- | --- |
| **Type Description** | **Example** |
| class Blueprint for objects (OOP) | class Car {public: void drive(); }; |
| union Memory-sharing data types | union Data {int i; float f;}; |
| enum Enumerated list of constants | enum Color {RED, GREEN, BLUE}; |
| typedef Alias for data types    **4. Void Type** | typedef unsigned int uint; |
| **Type Description** | **Example** |
| void Indicates no value or return type | void display(); |

**5. Modifiers with Data Types**

Modifiers like signed, unsigned, short, and long are used with integral types.

|  |  |  |
| --- | --- | --- |
| **Modified Type** | **Description** | **Example** |
| short int | Smaller range than int | short int a = 100; |
| long int | Larger range than int | long int b = 100000L; |
| unsigned int | Only positive integers | unsigned int x = 500; |
| long double | More precision than double | long double pi = 3.14159; |

**Example** #include <iostream> using namespace std; int main() { int a = 10; float fVariable = 20.5;

char charVar = 'A';

bool booleanVar = true; const float PI = 3.14; const double GRAVIT = 9.81;

cout << "Integer Variable: " << a << endl; cout << "Float Variable: " << fVariable << endl; cout << "Character Variable: " << charVar << endl;

cout << "Boolean Variable: " << booleanVar << endl;

cout << "Value of PI: " << 3.14 << endl;

cout << "Value of Gravity: " << GRAVITY << endl;

a = a + 5; fVariable = fVariable \* 2;

cout << "Updated Integer Variable: " << a << endl; cout << "Updated Float Variable: " << fVariable << endl; if (booleanVar) {

cout << "Boolean Variable is true" << endl;

} else {

cout << "Boolean Variable is false" << endl;

}

}

**1. Arithmetic Operators**

Used to perform basic mathematical operations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator Description** | | | **Example** | |
| + Addition | | | a + b | |
| - Subtraction | | | a - b | |
| \* Multiplication | | | a \* b | |
| / Division | | | a / b | |
| % Modulus (remainder)  Example:- int a = 10, b = 3; cout << a + b << endl; cout << a % b << endl;  2. Relational (Comparison) Operators  Used to compare two values. | | | a % b | |
| **Operator Description** | | | **Example** | |
| == Equal to | | | a == b | |
| **Operator** | **Description** | **Example** |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater than or equal | a >= b |
| <= | Less than or equal | a <= b |

Example:-

if (a > b) { cout << "a is greater than b" << endl;

}

1. **Logical Operators**

Used to perform logical operations (usually in conditions).

**Operator Description Example**

&& Logical AND a > 0 && b > 0

` `

! Logical NOT !flag

Example :- if (a > 0 && b > 0) { cout << "Both are positive" << endl;

}

1. **Assignment Operators**

Used to assign values to variables.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assignment | a = 5 |
| += | Add and assign | a += 2 |
| -= | Subtract and assign | a -= 2 |
| **Operator** | **Description** | **Example** |
| \*= | Multiply and assign | a \*= 3 |
| /= | Divide and assign | a /= 2 |
| %= | Modulus and assign | a %= 2 |

1. **Increment and Decrement Operators**

Used to increase or decrease values by 1.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ++ | Increment | ++a, a++ |
| -- | Decrement | --a, a-- |

1. **Bitwise Operators**

Operate on binary representations.

# Operator Description Example

|  |  |  |
| --- | --- | --- |
| & | AND | a & b |
| ` | ` | OR |
| ^ | XOR | a ^ b |
| ~ | NOT | ~a |
| << | Left shift | a << 1 |
| >> | Right shift | a >> 1 |

**3. Control Flow Statements**

**LAB EXERCISES:**

**1. Grade Calculator**:-

o Write a C++ program that takes a student’s marks as input and calculates the grade based on if-else conditions.

Ans:-

#include <iostream>

using namespace std;

int main() {

int marks;

cout << "Enter the student's marks (0-100): ";

cin >> marks;

if (marks < 0 || marks > 100) {

cout << "Invalid marks. Please enter a value between 0 and 100." <<endl;

} else {

if (marks >= 90) {

cout << "Grade: A+" <endl; } else if (marks >= 80) {

cout << "Grade: A" << endl; } else if (marks >= 70) {

cout<<"Grade:B"<<endl;

} else if (marks >= 60) { cout << "Grade: C" << endl;

} else if (marks >= 50) { cout << "Grade: D" << endl;

} else if (marks >= 33){ cout << "Grade: E" << endl;

} else {cout<<"Grade:F(Fail)" << endl;

}

}

return 0;

}

**2. Number Guessing Game**

Write a C++ program that asks the user to guess a number between 1 and 100. The program should provide hints if the guess is too high or too low. Use loops to allow the user multiple attempts.

#include <iostream>

using namespace std;

int main() {

int secretNumber = 42;

int userGuess;

cout << "Guess the number (between 1 and 100): ";

while (true) {

cin >> userGuess;

if (userGuess < 1 || userGuess > 100) {

cout << "Invalid input. Please enter a number between 1 and 100: ";

continue;

}

if (userGuess < secretNumber) { cout << "Too low! Try again: ";

} else if (userGuess > secretNumber) { cout << "Too high! Try again: ";

} else {

cout << "Congratulations! You guessed the correct number: " << secretNumber << endl;

break;

}

}

return 0;

}

3. Multiplication Table

Write a C++ program to display the multiplication table of a given number using a for loop.

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter a number to display its multiplication table: "; cin >> number;

cout << "Multiplication Table of " << number << ":" << endl; for (int i = 1; i <= 10; i++) {

cout << number << " x " << i << " = " << number \* i << endl;

}

return 0;

}

4. Nested Control Structures

Write a program that prints a right-angled triangle using stars (\*) with a nested loop.

#include <iostream>

using namespace std;

int main() {

int rows = 5;

for (int i = 1; i <= rows; i++) { for (int j = 1; j <= i; j++) { cout << "\* ";

}

cout << endl;

}

return 0;

}

THEORY EXERCISE:

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

**Conditional statements** allow a C++ program to make decisions based on whether a condition is true or false. These are used to control the flow of the program.

1. if and if-else Statement

* If the condition is true, the block under if executes.
* If the condition is false, the block under else executes. Syntax :- if (condition) {

} else {

}

Example :- int number = 10; if (number > 0) {

cout << "Positivenumber";

} else {

cout << "Non-positive number";

}

2. switch Statement

Evaluates the expression.

Matches the result with one of the case values.

Executes the matching block.

break prevents fall-through to other cases.

default is optional and runs if no case matches Syntax :- switch (expression) {

case value1:

break;

case value2:

break;

default:

}

Example :-

int day = 3; switch (day) { case 1:

cout<<"mon;

break;

case 2:

cout <<"Tuesday";

break;

case 3:

cout << "Wednesday";

break; default:

cout << "Invalid";

}

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **For Loop** | **while loop** | **do-while loop** |
| **Syntax** | for (initialization; condition;  increment/decrement)  {} | while (condition) { } | do { } while (condition); |
| **Initialization** | Declared within the loop structure and executed once at the beginning. | Declared outside the loop; should be done explicitly before the loop. | Declared outside the loop structure |
| **Condition** | Checked before each iteration. | Checked before each iteration. | Checked after each iteration. |
| **Use Cases** | Suitable for a known number of iterations or when looping over ranges. | Useful when the number of  iterations is not known in advance or based on a condition. | Useful when the loop block must be executed at least once, regardless of the initial condition. |
| **Initialization and**  **Update Scope** | Limited to the loop body. | Scope extends beyond the loop; needs to be handled explicitly. | Scope extends beyond the loop; needs to be handled explicitly. |

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

**break Statement**

Terminates the loop immediately.

When you want to exit a loop based on a specific condition.

**Example:**

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 10; ++i) { if (i == 5) { break;

}

cout << i << " ";

}

return 0;

}

**continue Statement :-**

Skips the current iteration and continues with the next one.

When you want to skip specific values or conditions without breaking the loop.

**Example:**

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 10; ++i) { if (i == 5) {

continue;

}

cout << i << " ";

}

return 0;

}

4. Explain nested control structures with an example.

**Nested control structures** refer to placing one control structure. like if, for, while, switch . This is useful when decisions or iterations depend on other conditions or loops.

**Example :** #include <iostream>using namespace std;

int main() {

int marks;

cout << "Enter your marks: "; cin >> marks;

if (marks >= 50) {

cout << "You passed." << endl; if (marks >= 90) {

cout << "Excellent!" << endl;

}

} else { cout<<"Youfailed."<< endl;

}

return 0;

}

3. Variable Scope

Write a program that demonstrates the difference between local and global variables in C++. Use functions to show scope. #include <iostream> using namespace std; // Global variable int globalVar = 10;

displayAndModify() {

int localVar = 5;

cout << "Inside displayAndModify() function:" << endl;

cout << " Local variable: " << localVar << endl;

cout << " Global variable before modification: " << globalVar << endl;

globalVar += 5; cout << " Global variable after modification: " << globalVar << endl;

}

main() {

int localVar = 20;

cout << "Inside main() function:" << endl;

cout << " Local variable: " << localVar << endl; cout << " Global variable: " << globalVar << endl;

displayAndModify();

cout << "Back in main() function after calling displayAndModify():" << endl; cout << " Local variable: " << localVar << endl;

cout << " Global variable: " << globalVar << endl;

return 0;

}

THEORY EXERCISE:

**What is the scope of variables in C++? Differentiate between local and globalscope.**

**Scope** refers to the region of a program where a variable is **declared**, **accessible**, and **valid**. In C++, variable scope helps control visibility and lifetime of variables.

|  |  |
| --- | --- |
| **Feature** | **Local Variable Global Variable** |
| **Declared in** | Outside all functions (top of  Inside a function or block  file) |
| **Accessible in** | Only within that function/block All functions in the same file |
| **Lifetime** | Created when block starts, destroyed when Exists for entire program  it ends duration |
| **Memory Use** | Stack Data segment |
| **Name**  **Conflicts** | Can be hidden by local  Can hide global variable with same name  variables |
| **Best Use** | Shared data across multiple  Temporary, function-specific data  functions |

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

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

* The function name
* Its return type
* The number and type of parameters
* It ends with a semicolon and does not contain the body of the function.

|  |  |
| --- | --- |
| **Reason** | **Explanation** |
| **Allows calling functions before definition** | Lets you call a function in main() even if it's defined later in the code. |
| **Helps with type checking** | Compiler ensures the correct number and types of arguments are passed. |
| **Improves modularity** | Useful when separating code into multiple files or headers. |
| **Prevents errors** | Catch errors like wrong return types or argument mismatches early. |

5. Arrays and Strings

LAB EXERCISES:

1. Array Sum and Average

Write a C++ program that accepts an array of integers, calculates the sum and average, and displays the results.

#include<iostream>

using namespace std;

int main(){

int num;

cout<< "Enter the number of element: ";

cin >> num;

int arr[num];

cout << "Enter " << num << " integers:" << endl;

for(int i=0;i<num;i++){

cin >> arr[i];

}

int sum=0;

for(inti=0;i<num;i++){ sum += arr[i];

}

double average = (double)sum/num; cout << "Sum:" << sum << endl;

cout << "Average:" << average << endl; return 0;

}

3. String Palindrome Check

Write a C++ program to check if a given string is a palindrome (reads the same forwards and backwards).

#include <iostream> #include <string> using namespace std; int main() { string str;

cout << "Enter a string: "; cin >> str;

int start = 0;

int end = str.length() - 1;

bool is Palindrome = true; while (start < end) {

if (str[start] != str[end]) {

is Palindrome = false; break;

}

start++; end--;

}

if (isPalindrome) {

cout << "The string is a palindrome." << endl;

} else {

cout << "The string is not a palindrome." << endl;

}

return 0;

}

THEORY EXERCISE:

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

**Single-Dimensional Arrays**

A **single-dimensional array** also called a 1D array stores elements in a **linear form** — like a list.

**Example:-**

int marks[3] = {85, 90, 78};

**Multidimensional Arrays**

A **multidimensional array** is an array of arrays. The most common form is the **twodimensional array** (2D), which can be thought of like a table or matrix.

**Example of 2D Array:**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

|  |  |
| --- | --- |
| **Feature 1D Array**  Shape Linear list  Declaration int arr[5];  Access arr[i] | **Multidimensional Array**  Table (2D), cube (3D), etc.  int arr[2][3];    arr[i][j] |
| Use Case Simple lists | Matrices, grids, tables |

Memory Single row of elements Rows and columns (or more dimensions)

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

Declaring and Initializing Strings:

std::string str1 = "Hello";

std::string str2("World");

Operations:

1. Concatenation: Joining two or more strings together.

std::string str1 = "Hello";

std::string str2 = "World";

std::string result = str1 + " " + str2; "Hello World"

1. Substring: Extracting a part of a string.

std::string str = "Hello World";

std::string substr = str.substr(6, 5); "World"

1. Find: Searching for a substring within a string.

std::string str = "Hello World";

size\_t pos = str.find("World");

1. Replace: Replacing a substring with another string.

std::string str = "Hello World";

str.replace(6, 5, "Universe"); "Hello Universe"

1. Compare: Comparing two strings for equality or lexicographical order.

std::string str1 = "Hello";

std::string str2 = "World"; if (str1 == str2) { std::cout << "Strings are equal." << std::endl;

} else {

std::cout << "Strings are not equal." << std::endl;

}

String Functions:

1. length(): Returns the length of a string.

std::string str = "Hello";

int length = str.length();

2. empty(): Checks if a string is empty.

std::string str = ""; if (str.empty()) { std::cout << "String is empty." << std::endl;

}

1. at(): Accesses a specific character in a string. std::string str = "Hello";

char c = str.at(0);

Examples:

1. String Reversal: std::string str = "Hello"; std::string rev = str;

std::reverse(rev.begin(), rev.end());

"olleH"

1. String Case Conversion

: std::string str = "Hello World";

for (char& c : str) {

c = std::toupper(c);

"HELLO WORLD"

}

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

In C++, arrays can be initialized in several ways. Below are examples of how to initialize **1D (one-dimensional)** and **2D (two-dimensional)** arrays.

**1D Array Initialization**

#include <iostream>

using namespace std;

int main() {

int numbers[5] = {1, 2, 3, 4, 5};

int zeros[5] = {};

for (int i = 0; i < 5; ++i) {

cout << "numbers[" << i << "] = " << numbers[i] << endl;

}

return 0;

}

**2D Array Initialization** #include <iostream> using namespace std; int main() {

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

int zeroMatrix[2][3] = {};

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

cout << "matrix[" << i << "][" << j << "] = " << matrix[i][j] << " ";

}

cout << endl;

}

return 0;

}

6. Introduction to Object-Oriented Programming

LAB EXERCISES:

1. Class for a Simple Calculator

Write a C++ program that defines a class Calculator with functions for addition, subtraction, multiplication, and division. Create objects to use these functions.

#include <iostream>

using namespace std; class Calculator {

public:

double add(double a, double b) {

return a + b;

}

double subtract(double a, double b) {

return a - b;

}

double multiply(double a, double b) {

return a \* b;

}

double divide(double a, double b) {

if (b == 0) {

cout << "Error: Division by zero!" << endl; return 0;

}

return a / b;

}

};

int main() {

Calculator calc;

double num1, num2;

cout << "Enter two numbers: ";

cin >> num1 >> num2;

cout << "Addition: " << calc.add(num1, num2) << endl;

cout << "Subtraction: " << calc.subtract(num1, num2) << endl; cout << "Multiplication: " << calc.multiply(num1, num2) << endl; cout << "Division: " << calc.divide(num1, num2) << endl;

return 0;

}

2. Class for Bank Account

Create a class BankAccount with data members like balance and member functions like deposit and withdraw. Implement encapsulation by keeping the data members private.

#include <iostream>

using namespace std; class BankAccount { private:

double balance; public:

BankAccount(double initialBalance) {

if (initialBalance < 0) {

cout << "Initial balance cannot be negative. Setting to 0." << endl; balance = 0;

} else {

balance = initialBalance;

}

}

void deposit(double amount) {

if (amount > 0) {

balance += amount;

cout << "Deposited: " << amount << endl;

} else {

cout << "Deposit amount must be positive!" << endl;

}

}

void withdraw(double amount) {

if (amount > balance) {

cout << "Insufficient balance!" << endl;

} else if (amount <= 0) {

cout << "Withdrawal amount must be positive!" << endl;

} else {

balance -= amount;

cout << "Withdrawn: " << amount << endl;

}

}

double getBalance() const { return balance;

}

};

int main() {

Bank Account my Account(5000);

My Account. deposit(1500);

My Account. withdraw(1000);

My Account. withdraw(500);

cout << "Current Balance: " << my Account. get Balance () << endl; return 0;

}

THEORY EXERCISE:

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

The key concepts of OOP are:

1. Classes and Objects :-

A class is a blueprint or template that defines the properties and behavior of an object. An object is an instance of a class, which has its own set of attributes (data) and methods .

1. Inheritance :-

Inheritance allows one class to inherit the properties and behavior of another class. The child class inherits all the attributes and methods of the parent class and can also add new attributes and methods or override the ones inherited from the parent class.

1. Polymorphism :-

Polymorphism is the ability of an object to take on multiple forms. It can be achieved through method overloading (multiple methods with the same name but different parameters) or method overriding (a child class provides a different implementation of a method already defined in its parent class).

1. Encapsulation :-

Encapsulation is the concept of hiding the implementation details of an object from the outside world and only exposing the necessary information through public methods. It helps to protect the internal state of an object and ensures that it is not accidentally modified.

1. Abstraction :-

Abstraction is the concept of showing only the necessary information to the outside world while hiding the internal implementation details. It helps to reduce complexity and improve modularity by exposing only the essential features of an object or system.

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

Achieving Encapsulation in Classes:

In C++, encapsulation is achieved in classes by using access specifiers to control access to the class members. The three main access specifiers are:

1. private: Members declared as private are accessible only within the class and are hidden from the outside world.
2. protected: Members declared as protected are accessible within the class and its derived classes.
3. public: Members declared as public are accessible from anywhere in the program.