Leadership & Excellence

Sri Eshwar
College of Engineering
Coimbatore | Tamilnadu
An Autonomous Institution
Affiliated to Anna University, Chennai

NAAC
NATIONAL ASSESSMENT AND ACCREDITATION COUNCIL
Accredited by NAAC with 'A' Grade

NBA
NATIONAL BOARD of ACCREDITATION
For CSE, ECE, EEE, MECH

nirf
National Institutional Ranking Framework

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

U23CS452–OBJECT-ORIENTED PROGRAMMING USING C++ LABORATORY RECORD

SRI ESHWAR COLLEGE OF ENGINEERING
KINATHUKADAVU,
COIMBATORE – 641202

SRI ESHWAR COLLEGE OF ENGINEERING

KINATHUKADAVU, COIMBATORE-641202

(An Autonomous Institution Affiliated to Anna University, Chennai)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by Mr. /Ms..…………………………………….
…………………………………………Register No:……………...………of I Year **B.E / B. Tech.** **……**in the **U23CS452–Object Oriented Programming using C++ Laboratory** during the **2nd Semester** of the academic year **2024– 25 (Even Semester).**

Signature of Staff In-charge **Head of the department**

Submitted for the End Semester Practical Examinations, held on…………….

Internal Examiner **External Examiner**

CONTENTS

	d		University Management System using class			
	e		Transport Management System			
	f		Multilevel Inheritance			
	II		**Polymorphism**			
	a		Polymorphic Shape in class concepts			
	b		Function Overloading			
	c		Dynamic dispatch mechanism			
	d		Runtime polymorphism			
	e		Hospital Appointment System using Virtual Functions			
	f		Concept on Virtual Base Class			
7			**Standard Template Library**			
	a		Employee Database linked list with dynamic memory allocation			
	b		Develop an Undo/Redo system using a stack			
	c		Queue Management System			
	d		Product Inventory using a vector			
	e		Contact list with a map			
	f		Find a unique elements in a list			
	g		Task Scheduler using Priority queue			
	h		Dictionary application using maps			
	i		Merge the lists using STL algorithms			
	j		Movie recommendation system using multimap			
8			**File Handling**			
	a		File Handling – R/W Operation			
	b		File handling – Count Words			
	c		File handling – Copy Content			
	d		File Handling – Data Appending			
	e		File Handling – Simple Log System			
	f		File handling – Merge file contents.			
	g		File Handling – Data Storage			
9			**Real-Time Concurrent Processing (Threads, Mutexes, and Asynchronous Programming)**			
	a		Multithreading			
	b		Thread Synchronization in mutex			
	c		Asynchronous Function Execution.			
	d		Thread-based Factorial Calculation			
	e		Simple Timer Using Threads.			
10			**Mini Project**			

Average Marks:

Average (in words)

Signature of the Faculty

Develop a program that converts temperature between Celsius, Fahrenheit, and Kelvin based on user input

Exp.No:1a	**Temperature Conversion**
Date:	

Aim
To develop a C++ program that converts temperature between Celsius, Fahrenheit, and Kelvin based on user input.

Algorithm
1. Start the program.
2. Display a menu with options to select the input temperature unit (Celsius, Fahrenheit, or Kelvin).
3. Ask the user to enter the temperature value.
4. Display a menu with options to select the output temperature unit (Celsius, Fahrenheit, or Kelvin).
5. Perform the required conversion using the following formulas:
   - Celsius to Fahrenheit: F = (C × 9/5) + 32
   - Celsius to Kelvin: K = C + 273.15
   - Fahrenheit to Celsius: C = (F - 32) × 5/9
   - Fahrenheit to Kelvin: K = (F - 32) × 5/9 + 273.15
   - Kelvin to Celsius: C = K - 273.15
   - Kelvin to Fahrenheit: F = (K - 273.15) × 9/5 + 32
6. Display the converted temperature.
7. Ask the user if they want to perform another conversion.
8. If yes, repeat from step 2; otherwise, exit.

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

```
double celsiusToFahrenheit(double c) {
    return (c * 9.0 / 5.0) + 32;
}

double celsiusToKelvin(double c) {
    return c + 273.15;
}

double fahrenheitToCelsius(double f) {
    return (f - 32) * 5.0 / 9.0;
}

double fahrenheitToKelvin(double f) {
    return (f - 32) * 5.0 / 9.0 + 273.15;
}

double kelvinToCelsius(double k) {
    return k - 273.15;
}

double kelvinToFahrenheit(double k) {
    return (k - 273.15) * 9.0 / 5.0 + 32;
}

int main() {
    int choice1, choice2;
    double temperature, convertedTemp;
    char repeat;

    do {
        cout << "Select the input temperature unit:" << endl;
        cout << "1. Celsius" << endl;
        cout << "2. Fahrenheit" << endl;
        cout << "3. Kelvin" << endl;
        cout << "Enter your choice: ";
        cin >> choice1;

        cout << "Enter the temperature value: ";
        cin >> temperature;

        cout << "Select the output temperature unit:" << endl;
        cout << "1. Celsius" << endl;
        cout << "2. Fahrenheit" << endl;
        cout << "3. Kelvin" << endl;
        cout << "Enter your choice: ";
        cin >> choice2;
```

```
            if (choice1 == 1 && choice2 == 2) {
            convertedTemp = celsiusToFahrenheit(temperature);
            cout << "Converted Temperature: " << convertedTemp << " °F" << endl;
        } else if (choice1 == 1 && choice2 == 3) {
            convertedTemp = celsiusToKelvin(temperature);
            cout << "Converted Temperature: " << convertedTemp << " K" << endl;
        } else if (choice1 == 2 && choice2 == 1) {
            convertedTemp = fahrenheitToCelsius(temperature);
            cout << "Converted Temperature: " << convertedTemp << " °C" << endl;
        } else if (choice1 == 2 && choice2 == 3) {
            convertedTemp = fahrenheitToKelvin(temperature);
            cout << "Converted Temperature: " << convertedTemp << " K" << endl;
        } else if (choice1 == 3 && choice2 == 1) {
            convertedTemp = kelvinToCelsius(temperature);
            cout << "Converted Temperature: " << convertedTemp << " °C" << endl;
        } else if (choice1 == 3 && choice2 == 2) {
            convertedTemp = kelvinToFahrenheit(temperature);
            cout << "Converted Temperature: " << convertedTemp << " °F" << endl;
        } else if (choice1 == choice2) {
            cout << "Same unit selected, temperature remains the same: " << temperature << endl;
        } else {
            cout << "Invalid selection. Please try again." << endl;
        }


        cout << "Do you want to convert another temperature? (y/n): ";
        cin >> repeat;

    } while (repeat == 'y' || repeat == 'Y');

    cout << "Program terminated." << endl;
    return 0;
}
```

Output:

Result:

Write a program to compute simple and compound interest for a given principal amount, rate, and time

Exp.No:1b	**Simple Interest & Compound Interest**
Date:	

Aim
To develop a C++ program to compute Simple Interest (SI) and Compound Interest (CI) for a given principal amount, rate of interest, and time.

Algorithm

1. Start the program.
2. Take input from the user for:
    - Principal amount (P)
    - Rate of interest (R%)
    - Time period in years (T)
    - Number of times interest is compounded per year (N) for compound interest calculation.
3. Compute Simple Interest (SI) using the formula:

    a. $$SI = \frac{P \times R \times T}{100}$$

4. Compute Compound Interest (CI) using the formula:

5. $$CI = P \times \left(1 + \frac{R}{100N}\right)^{N \times T} - P$$

6. Display the computed Simple Interest (SI) and Compound Interest (CI).
7. End the program.

Program:
```
#include <iostream>
#include <cmath>  // For pow() function
using namespace std;

double calculateSimpleInterest(double principal, double rate, double time) {
   return (principal * rate * time) / 100;
}
```

```
double calculateCompoundInterest(double principal, double rate, double time, int n) {
    return principal * pow((1 + rate / (100 * n)), n * time) - principal;
}

int main() {
    double principal, rate, time;
    int n;

    cout << "Enter the Principal Amount: ";
    cin >> principal;
    cout << "Enter the Rate of Interest (% per year): ";
    cin >> rate;
    cout << "Enter the Time Period (in years): ";
    cin >> time;
    cout << "Enter the Number of times interest is compounded per year: ";
    cin >> n;

    double simpleInterest = calculateSimpleInterest(principal, rate, time);

    double compoundInterest = calculateCompoundInterest(principal, rate, time, n);

    cout << "\nSimple Interest = " << simpleInterest << endl;
    cout << "Compound Interest = " << compoundInterest << endl;

    return 0;
}
```

Output:

Result:

Develop a BMI calculator that takes height and weight as input and categorizes the user's fitness level

Exp.No:1c	**BMI Calculator**
Date:	

Aim

To develop a **BMI (Body Mass Index) Calculator** in C++ that takes the user's height (in meters) and weight (in kilograms) as input and categorizes their fitness level.

Algorithm

1. Start the program.
2. Take user input for:
    - **Weight (kg)**
    - **Height (m)**
3. Compute **BMI** using the formula:

   BMI=Weight (kg)Height (m)2BMI = \frac{\text{Weight (kg)}}{\text{Height (m)}^2}BMI=Height (m)2Weight (kg)

4. Categorize the user based on their BMI:
    - **BMI < 18.5** → Underweight
    - **18.5 ≤ BMI < 24.9** → Normal weight
    - **25 ≤ BMI < 29.9** → Overweight
    - **BMI ≥ 30** → Obese
5. Display the **BMI value** and the corresponding **fitness category**.
6. End the program.

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

double calculateBMI(double weight, double height) {
   return weight / (height * height);
}

string categorizeBMI(double bmi) {
   if (bmi < 18.5)
      return "Underweight";
   else if (bmi >= 18.5 && bmi < 24.9)
      return "Normal weight";
   else if (bmi >= 25 && bmi < 29.9)
      return "Overweight";
   else
```

```
        return "Obese";
}

int main() {
    double weight, height, bmi;

    cout << "Enter your weight (in kg): ";
    cin >> weight;
    cout << "Enter your height (in meters): ";
    cin >> height;

    bmi = calculateBMI(weight, height);

    cout << "\nYour BMI is: " << bmi << endl;
    cout << "Category: " << categorizeBMI(bmi) << endl;

    return 0;
}
```

Output :

Result:

Implement a program to check whether a given year is a leap year.

Exp.No:1d	**Leap Year**
Date:	

Aim:
To write a C++ program to check whether a given year is a leap year or not.

Algorithm:

1. Start the program.
2. Input a year from the user.
3. Check the following conditions:
   - If the year is divisible by 4, go to step 4. Otherwise, it's not a leap year.
4. If the year is divisible by 100, go to step 5. Otherwise, it's a leap year.
5. If the year is divisible by 400, then it's a leap year. Otherwise, it's not a leap year.
6. Display the result.
7. End the program.

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

int main() {
   int year;
   cout << "Enter a year: ";
   cin >> year;
   if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
      cout << year << " is a leap year." << endl;
   } else {
      cout << year << " is not a leap year." << endl;
   }
   return 0;
}
```

Output:

Result:

Write a program to determine if a given number is prime

Exp.No:1e	**Prime Number**
Date:	

Aim:
To write a C++ program to determine whether a given number is a prime number or not.

Algorithm:

1. Start the program.
2. Input a number from the user.
3. If the number is less than or equal to 1, it is not a prime number.
4. For numbers greater than 1, check divisibility from 2 to the square root of the number.
    - If the number is divisible by any of these, it is not a prime number.
    - If no divisor is found, the number is prime.
5. Display the result.
6. End the program.

Program:

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

int main() {
   int num;

   cout << "Enter a number: ";
   cin >> num;

   if (num <= 1) {
      cout << num << " is not a prime number." << endl;
   } else {
      bool isPrime = true;
      for (int i = 2; i <= sqrt(num); ++i) {
         if (num % i == 0) {
            isPrime = false;
            break;
         }
      }

      if (isPrime) {
         cout << num << " is a prime number." << endl;
      } else {
            cout << num << " is not a prime number." << endl;
        }
    }    return 0;}
```

Output:

Result:

Implement a Fibonacci sequence generator that outputs the first N terms based on user input.

Exp.No:1f	**Fibonacci sequence generator**
Date:	

Aim:
To write a C++ program that generates and prints the first N terms of the Fibonacci sequence, where N is provided by the user.

Algorithm:

1. Start the program.
2. Input the number N from the user, which represents how many terms of the Fibonacci sequence to generate.
3. Initialize the first two terms of the Fibonacci sequence:
   - First term (F(0)) = 0
   - Second term (F(1)) = 1
4. Print the first two terms (0 and 1).
5. Use a loop to calculate the subsequent terms by summing the previous two terms and print them.
6. Stop the program after printing the first N terms.
7. End the program.

Program

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

int main() {
    int N;

    cout << "Enter the number of terms for the Fibonacci sequence: ";
    cin >> N;

    if (N <= 0) {
        cout << "Please enter a positive integer greater than 0." << endl;
        return 0;
    }

    long long int a = 0, b = 1;

    cout << "Fibonacci Sequence: ";
    for (int i = 1; i <= N; ++i) {
        cout << a << " ";
        long long int nextTerm = a + b;
        a = b;
        b = nextTerm;
    }
    cout << endl;

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Develop a warehouse inventory management system that tracks incoming and outgoing stock.

Exp.No:2a	**Stocks Inventory Management System**
Date:	

Aim:

To develop a C++ program for a Warehouse Inventory Management System that tracks the incoming and outgoing stock, allowing the user to add stock to the inventory and remove stock from it.

Algorithm:

1. Start the program.
2. Declare a variable to store the inventory quantity.
3. Display a menu with the following options:
   - Add stock
   - Remove stock
   - View current stock
   - Exit
4. Based on the user's choice:
   - **Add stock**: Prompt the user for the quantity of stock to be added and update the inventory.
   - **Remove stock**: Prompt the user for the quantity of stock to be removed and update the inventory.
   - **View current stock**: Display the current inventory level.
   - **Exit**: End the program.
5. Loop the menu until the user chooses to exit.
6. End the program.

Program

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

class Warehouse {
private:
   int stock;  // Variable to hold the current stock level

public:
   Warehouse() {
      stock = 0;
   }
```

```
        void addStock(int quantity) {
        stock += quantity;
        cout << "Added " << quantity << " items to inventory.\n";
    }

    void removeStock(int quantity) {
        if (quantity > stock) {
            cout << "Error: Not enough stock to remove.\n";
        } else {
            stock -= quantity;
            cout << "Removed " << quantity << " items from inventory.\n";
        }
    }

    void displayStock() {
        cout << "Current stock: " << stock << " items.\n";
    }
};

int main() {
    Warehouse warehouse;
    int choice, quantity;

    while (true) {
        cout << "\nWarehouse Inventory Management System\n";
        cout << "1. Add stock\n";
        cout << "2. Remove stock\n";
        cout << "3. View current stock\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter quantity to add: ";
                cin >> quantity;
                warehouse.addStock(quantity);
                break;

            case 2:

                cout << "Enter quantity to remove: ";
                cin >> quantity;
                warehouse.removeStock(quantity);
                break;
            case 3:
                warehouse.displayStock();
                break;

            case 4:
                cout << "Exiting the program. Goodbye!\n";
                return 0;

            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }

    return 0;
}
```

Output:

Result:

Implement an ATM simulation with options for withdrawal, deposit, and balance inquiry

Exp.No:2 b	**ATM Simulation**
Date:	

Aim:
To implement a C++ program that simulates an ATM system, allowing the user to perform operations such as withdrawal, deposit, and balance inquiry.

Algorithm:

1. Start the program.
2. Initialize the account balance (starting with an arbitrary balance, for example, 1000).
3. Display the ATM menu with the following options:
   - **1**: Deposit money
   - **2**: Withdraw money
   - **3**: Balance inquiry
   - **4**: Exit
4. Based on the user input:
   - **Deposit**: Prompt the user to enter the amount to deposit and update the balance.
   - **Withdraw**: Prompt the user to enter the amount to withdraw, check if there is enough balance, and update the balance if the transaction is valid.
   - **Balance inquiry**: Display the current balance.
   - **Exit**: End the program.
5. Repeat the menu until the user chooses to exit.

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

class ATM {
private:
    double balance;  // Variable to store the balance

public:
    ATM(double initial_balance) {
        balance = initial_balance;
    }


    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            cout << "Successfully deposited " << amount << " units.\n";
        } else {
            cout << "Deposit amount must be greater than zero.\n";
        }
    }

    void withdraw(double amount) {
        if (amount > 0) {
            if (amount <= balance) {
                balance -= amount;
                cout << "Successfully withdrew " << amount << " units.\n";
            } else {
                cout << "Insufficient balance.\n";
            }
        } else {
            cout << "Withdrawal amount must be greater than zero.\n";
        }
    }


    void checkBalance() {
        cout << "Current balance: " << balance << " units.\n";
    }
};

int main() {
    double initial_balance = 1000.0;  // Starting balance
    ATM atm(initial_balance);         // Create ATM object with initial balance
    int choice;
    double amount;

    while (true) {
        cout << "\nATM Menu\n";
        cout << "1. Deposit\n";
        cout << "2. Withdraw\n";
        cout << "3. Balance Inquiry\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter amount to deposit: ";
                cin >> amount;
                atm.deposit(amount);
                break;

            case 2:
                cout << "Enter amount to withdraw: ";
                cin >> amount;
                atm.withdraw(amount);
                break;

            case 3:
                atm.checkBalance();
                break;

            case 4:

                cout << "Thank you for using the ATM. Goodbye!\n";
                return 0;

            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }

    return 0;
}
```

Output:

Result:

Create a traffic light simulation that displays different light transitions based on a timer.

Exp.No:2 c	Traffic Light Simualtion
Date:	

Aim:
To simulate a traffic light system in C++ that cycles through different traffic light colors (red, yellow, green) with a delay between each transition.

Algorithm:

1. **Initialize the program:**
   - Set the duration for each light color (for example, 3 seconds for green, 1 second for yellow, and 3 seconds for red).
2. **Start the cycle:**
   - Display the current light (red, yellow, or green) and wait for the designated duration for that light.
3. **Cycle through the lights:**
   - After each duration, transition to the next light in the cycle:
     - From **red** to **green**.
     - From **green** to **yellow**.
     - From **yellow** to **red**.
4. **Repeat the cycle:**
   - Continuously cycle through the lights indefinitely until the program is stopped.
5. **End the program.**

Program

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

void displayTrafficLight(const string& light) {
   cout << "Current light: " << light << endl;
}

int main() {
   int greenDuration = 3;  // Green light duration in seconds
   int yellowDuration = 1; // Yellow light duration in seconds
   int redDuration = 3;    // Red light duration in seconds

   while (true) {
       displayTrafficLight("Green");
      this_thread::sleep_for(chrono::seconds(greenDuration));
```

```
        displayTrafficLight("Yellow");
        this_thread::sleep_for(chrono::seconds(yellowDuration));

        displayTrafficLight("Red");
        this_thread::sleep_for(chrono::seconds(redDuration));
    }

    return 0;
}
```

Output:

Result:

Develop an employee payroll system that calculates net salary based on hours worked and tax deductions.

Exp.No:2 d	**Employee Payroll System**
Date:	

Aim:
To develop a C++ program that calculates an employee's net salary based on the number of hours worked, hourly wage, and applicable tax deductions.

Algorithm:

1. **Input:**
    - Prompt the user to enter the **number of hours worked** in a month.
    - Prompt the user to enter the **hourly wage**.
    - Input the **tax rate** as a percentage (e.g., 10%).
2. **Calculate Gross Salary:**
    - Gross Salary = hours worked * hourly wage
3. **Calculate Tax Deductions:**
    - Tax Deduction = gross salary * tax rate / 100
4. **Calculate Net Salary:**
    - Net Salary = gross salary - tax deduction
5. **Output:**
    - Display the **gross salary**, **tax deduction**, and **net salary**.
6. **End the program.**

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

class Employee {
private:
   double hoursWorked;
   double hourlyWage;
   double taxRate;

public:
   void inputDetails() {
      cout << "Enter the number of hours worked in a month: ";
      cin >> hoursWorked;
      cout << "Enter hourly wage: ";
      cin >> hourlyWage;
      cout << "Enter tax rate (in percentage): ";
      cin >> taxRate;
```

```
    }

    double calculateGrossSalary() {
        return hoursWorked * hourlyWage;
    }

    double calculateTaxDeduction(double grossSalary) {
        return grossSalary * taxRate / 100;
    }

    double calculateNetSalary(double grossSalary, double taxDeduction) {
        return grossSalary - taxDeduction;
    }


    void displaySalaryDetails() {
        double grossSalary = calculateGrossSalary();
        double taxDeduction = calculateTaxDeduction(grossSalary);
        double netSalary = calculateNetSalary(grossSalary, taxDeduction);

        cout << fixed << setprecision(2);  // Set to display 2 decimal points
        cout << "\nSalary Details:\n";
        cout << "Gross Salary: $" << grossSalary << endl;
        cout << "Tax Deduction: $" << taxDeduction << endl;
        cout << "Net Salary: $" << netSalary << endl;
    }
};

int main() {
    Employee emp;

    emp.inputDetails();

    emp.displaySalaryDetails();

    return 0;
}
```

Output:

Result:

Implement a Tic-Tac-Toe game for two players

Exp.No:2 e	**Tic Tac Toe**
Date:	

Aim:
To develop a Tic-Tac-Toe game in C++ that allows two players to play the game alternately on a 3x3 grid. The game should check for winning conditions, a draw, and display the board after each turn

Algorithm:

1. **Initialize the Board:**
   - Create a 3x3 grid, initially filled with numbers 1-9 to represent empty spots.
2. **Player Turn:**
   - Player 1 (X) and Player 2 (O) alternate turns.
   - Prompt the current player to choose an available spot on the grid.
3. **Check for Valid Move:**
   - Ensure the chosen spot is not already occupied.
4. **Update the Board:**
   - After each valid move, update the grid with the current player's symbol (X or O).
5. **Check for Win or Draw:**
   - After every move, check if the current player has won (three symbols in a row, column, or diagonal).
   - If all spots are filled and no player has won, it's a draw.
6. **Display the Board:**
   - Display the grid after each turn to show the current state of the game.
7. **End the Game:**
   - End the game when there is a winner or a draw.

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

class TicTacToe {
private:
   char board[3][3];
   char currentPlayer;
public:
   TicTacToe() {
      currentPlayer = 'X';
      int counter = 1;
      for (int i = 0; i < 3; i++) {
         for (int j = 0; j < 3; j++) {
            board[i][j] = '0' + counter++; // Initialize with numbers 1 to 9
         }
```

```
    }
}

void displayBoard() {
    cout << "\nTic-Tac-Toe Board:\n";
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

bool checkWin() {
    for (int i = 0; i < 3; i++) {
        if (board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0] != ' ')
            return true;  // Check rows
        if (board[0][i] == board[1][i] && board[1][i] == board[2][i] && board[0][i] != ' ')
            return true;  // Check columns
    }
    if (board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] != ' ')
        return true;
    if (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] != ' ')
        return true;

    return false;
}

bool checkDraw() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] != 'X' && board[i][j] != 'O') {
                return false;  // If there's an empty spot, no draw yet
            }
        }
    }
    return true;  // All spots are filled
}

bool makeMove(int position) {
    int row = (position - 1) / 3;
    int col = (position - 1) % 3;

    if (board[row][col] != 'X' && board[row][col] != 'O') {
        board[row][col] = currentPlayer;
        return true;
        }
        return false; // Spot already taken
    }

    void switchPlayer() {
        currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
    }

    void playGame() {
        int position;
        bool validMove = false;
        while (true) {
            displayBoard(); // Show the board

            cout << "Player " << currentPlayer << ", enter a position (1-9): ";
            cin >> position;

            validMove = makeMove(position);

            if (!validMove) {
                cout << "Invalid move, the position is already taken! Try again.\n";
                continue;
            }

            if (checkWin()) {
                displayBoard();
                cout << "Player " << currentPlayer << " wins!\n";
                break;
            }

            if (checkDraw()) {
                displayBoard();
                cout << "It's a draw!\n";
                break;
            }

            switchPlayer();
        }
    }
};

int main() {
    TicTacToe game;
    game.playGame();  // Start the game

    return 0;
```

```
}
```
Output:

Result:

Develop a Rock-Paper-Scissors game with computer vs. user play mode

Exp.No:2f	**Rock- Paper- Scissors Game**
Date:	

Aim:
To create a simple Rock-Paper-Scissors game where the user competes against the computer. The computer randomly selects one of the three options (Rock, Paper, or Scissors), and the user makes their choice. The program then determines the winner based on the rules of the game.

Algorithm:

1. **Display the Game Menu:**
   - Show the options (Rock, Paper, Scissors) to the user.
   - Prompt the user to select one of the options (1 for Rock, 2 for Paper, 3 for Scissors).
2. **Computer Selection:**
   - The computer randomly selects an option (Rock, Paper, or Scissors).
3. **Determine the Winner:**
   - If the user and the computer choose the same option, it's a draw.
   - If the user picks Rock and the computer picks Scissors, the user wins, and vice versa.
   - Use the rules:
     - Rock beats Scissors.
     - Scissors beats Paper.
     - Paper beats Rock.
4. **Display the Result:**
   - Display the choices made by both the user and the computer.
   - Announce whether the user won, lost, or if it's a draw.
5. **Repeat or End the Game:**
   - Ask the user if they want to play again.
   - If yes, repeat the process; if no, end the game.

Program

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

string determineWinner(int userChoice, int computerChoice) {
   if (userChoice == computerChoice) {
      return "It's a draw!";
   }
```

```
    if ((userChoice == 1 && computerChoice == 3) ||
        (userChoice == 2 && computerChoice == 1) ||
        (userChoice == 3 && computerChoice == 2)) {
        return "You win!";
    } else {
        return "You lose!";
    }
}

int main() {
    int userChoice, computerChoice;
    char playAgain;


    srand(time(0));

    do {

        cout << "\nRock-Paper-Scissors Game\n";
        cout << "1. Rock\n";
        cout << "2. Paper\n";
        cout << "3. Scissors\n";
        cout << "Enter your choice (1 for Rock, 2 for Paper, 3 for Scissors): ";
        cin >> userChoice;


        if (userChoice < 1 || userChoice > 3) {
            cout << "Invalid choice. Please select a number between 1 and 3.\n";
            continue;
        }

        computerChoice = rand() % 3 + 1;

        cout << "\nYou chose: ";
        if (userChoice == 1) cout << "Rock";
        else if (userChoice == 2) cout << "Paper";
        else cout << "Scissors";

        cout << "\nComputer chose: ";
        if (computerChoice == 1) cout << "Rock";
        else if (computerChoice == 2) cout << "Paper";
        else cout << "Scissors";
        cout << "\n" << determineWinner(userChoice, computerChoice) << endl;

        cout << "\nDo you want to play again? (y/n): ";
        cin >> playAgain;
    } while (playAgain == 'y' || playAgain == 'Y');

    cout << "\nThank you for playing!\n";

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Design a stock portfolio manager that calculates profit/loss after stock transactions

Exp.No:3 a	**Stock Portfolio**
Date:	

Aim:

To develop a **Stock Portfolio Manager** in C++ that helps manage stock transactions. The program will:

- Allow users to buy and sell stocks.
- Track the number of stocks owned and the price at which they were bought.
- Calculate the profit or loss after each transaction (selling price - buying price).

Algorithm:

1. **Stock Portfolio Structure:**
    - Store stock information such as **stock name**, **number of stocks owned**, and **purchase price**.
2. **Transaction Options:**
    - Provide the user with options to either **buy** or **sell** stocks.
    - For **buying**: Record the number of stocks bought and the price at which they were purchased.
    - For **selling**: Record the number of stocks sold and calculate the **profit/loss** (selling price - buying price).
3. **Profit/Loss Calculation:**
    - If the stock is sold for a price higher than the purchase price, the transaction results in a **profit**.
    - If the stock is sold for a price lower than the purchase price, the transaction results in a **loss**.
4. **Display Portfolio:**
    - After each transaction, show the current portfolio (number of stocks owned and the average purchase price).
5. **Repeat or End the Program:**
    - The user can continue making transactions or exit the program.

Program

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

class StockPortfolioManager {
private:
    map<string, pair<int, double>> portfolio;  // Store stocks: stock_name -> (num_shares,
avg_price)

public:
    void buyStock(string stockName, int numShares, double pricePerShare) {
        if (portfolio.find(stockName) != portfolio.end()) {
            pair<int, double> currentStock = portfolio[stockName];
            int newTotalShares = currentStock.first + numShares;
            double newAveragePrice = ((currentStock.first * currentStock.second) + (numShares *
pricePerShare)) / newTotalShares;
            portfolio[stockName] = make_pair(newTotalShares, newAveragePrice);
        } else {
            portfolio[stockName] = make_pair(numShares, pricePerShare);
        }
        cout << "Bought " << numShares << " shares of " << stockName << " at $" <<
pricePerShare << " each." << endl;
    }

    void sellStock(string stockName, int numShares, double pricePerShare) {
        if (portfolio.find(stockName) == portfolio.end()) {
            cout << "You don't own any shares of " << stockName << endl;
            return;
        }

        pair<int, double> currentStock = portfolio[stockName];
        if (numShares > currentStock.first) {
            cout << "You do not have enough shares of " << stockName << " to sell." << endl;
            return;
        }


        double profitOrLoss = (pricePerShare - currentStock.second) * numShares;
        portfolio[stockName].first -= numShares;
        if (portfolio[stockName].first == 0) {
            portfolio.erase(stockName);
        }

        cout << "Sold " << numShares << " shares of " << stockName << " at $" << pricePerShare
<< " each." << endl;
        if (profitOrLoss >= 0) {
            cout << "You made a profit of $" << profitOrLoss << "." << endl;
        } else {
            cout << "You made a loss of $" << -profitOrLoss << "." << endl;
        }
    }
    void displayPortfolio() {
        cout << "\nCurrent Stock Portfolio:\n";
        if (portfolio.empty()) {
            cout << "No stocks in portfolio.\n";
        } else {
            for (auto& stock : portfolio) {
                cout << "Stock: " << stock.first << ", Shares: " << stock.second.first << ", Average
Price: $" << stock.second.second << endl;
            }
        }
    }
};

int main() {
    StockPortfolioManager manager;
    int choice;
    string stockName;
    int numShares;
    double pricePerShare;

    do {
        cout << "\nStock Portfolio Manager\n";
        cout << "1. Buy Stock\n";
        cout << "2. Sell Stock\n";
        cout << "3. Display Portfolio\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter the stock name: ";
            cin >> stockName;
            cout << "Enter number of shares to buy: ";
            cin >> numShares;
            cout << "Enter the price per share: ";
            cin >> pricePerShare;
            manager.buyStock(stockName, numShares, pricePerShare);
            break;

        case 2:
            cout << "Enter the stock name: ";
            cin >> stockName;
            cout << "Enter number of shares to sell: ";
            cin >> numShares;
            cout << "Enter the price per share: ";
            cin >> pricePerShare;
            manager.sellStock(stockName, numShares, pricePerShare);
            break;

        case 3:
            manager.displayPortfolio();
            break;

        case 4:
            cout << "Exiting the Stock Portfolio Manager.\n";
            break;

        default:
            cout << "Invalid choice! Please try again.\n";
        }

    } while (choice != 4);

    return 0;
}
```

Output:

Result:

Implement a program to calculate electricity bill based on unit consumption

Exp.No:3 b	**Electricity Bill Calculation**
Date:	

Aim:

To implement a program that calculates the electricity bill based on the number of units consumed by a customer. The rates for different ranges of consumption are predefined, and the program will calculate the total bill accordingly.

Algorithm:

1. **Input the Units Consumed:**
   - Prompt the user to input the number of units of electricity consumed during the billing period.
2. **Determine the Rate:**
   - The electricity rate is determined based on the units consumed:
     - **0-100 units**: A fixed rate, say, \$0.5 per unit.
     - **101-200 units**: A higher rate, say, \$0.75 per unit.
     - **201-300 units**: An even higher rate, say, \$1 per unit.
     - **Above 300 units**: The highest rate, say, \$1.5 per unit.
3. **Calculate the Bill:**
   - Based on the consumption, calculate the bill by multiplying the respective rate with the units consumed.
4. **Display the Total Bill:**
   - Output the total bill after calculating.

Program

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

double calculateBill(int units) {
   double bill = 0.0;


   if (units <= 100) {
      bill = units * 0.5;  // $0.5 per unit for 0-100 units
   } else if (units <= 200) {
      bill = 100 * 0.5 + (units - 100) * 0.75;  // $0.75 per unit for 101-200 units
   } else if (units <= 300) {
      bill = 100 * 0.5 + 100 * 0.75 + (units - 200) * 1.0;  // $1.0 per unit for 201-300 units
   } else {
      bill = 100 * 0.5 + 100 * 0.75 + 100 * 1.0 + (units - 300) * 1.5;  // $1.5 per unit for above
300 units
```

```
    }

    return bill;
}

int main() {
    int units;
    double totalBill;

    cout << "Enter the number of units consumed: ";
    cin >> units;

        if (units < 0) {
        cout << "Invalid input! Units cannot be negative." << endl;
        return 1;
    }

    totalBill = calculateBill(units);

    cout << "\nElectricity Bill Details:\n";
    cout << "Units Consumed: " << units << endl;
    cout << "Total Bill: $" << totalBill << endl;

    return 0;
}
‘
```

Output:

Result:

Write an age difference calculator that finds the difference between two given birthdates

Exp.No:3 c	**Difference between two birthdays**
Date:	

Aim:
To implement an Age Difference Calculator that finds the age difference between two given birthdates. The program will take two dates (birthdates of two individuals) and calculate the difference in years, months, and days

Algorithm:

1. **Input the Birthdates:**
   - Prompt the user to input two birthdates (one for each individual) in the format DD MM YYYY.
2. **Calculate the Difference:**
   - First, calculate the total difference in years.
   - Then, subtract the months and days appropriately if the second person's birthday hasn't occurred yet in the current year.
3. **Adjust the Difference:**
   - If the second person's birthdate is later than the first person's in the same year, adjust the months and days accordingly.
4. **Display the Age Difference:**
   - Output the difference in years, months, and days.

. Program

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

struct Date {
   int day;
   int month;
   int year;
};

void calculateAgeDifference(Date birthDate1, Date birthDate2) {
   int years = birthDate2.year - birthDate1.year;
   int months = birthDate2.month - birthDate1.month;
   int days = birthDate2.day - birthDate1.day;

   if (days < 0) {
      months--;
      days += 30;  // Assuming 30 days in a month
   }
```

```
        if (months < 0) {
        years--;
        months += 12;  // Adjust months to be in the correct range
    }

    cout << "\nAge Difference: " << years << " years, " << months << " months, and " << days <<
" days." << endl;
}

int main() {
    Date birthDate1, birthDate2;

    cout << "Enter the first person's birthdate (DD MM YYYY): ";
    cin >> birthDate1.day >> birthDate1.month >> birthDate1.year;

    cout << "Enter the second person's birthdate (DD MM YYYY): ";
    cin >> birthDate2.day >> birthDate2.month >> birthDate2.year;


    calculateAgeDifference(birthDate1, birthDate2);

    return 0;
}
```

Output:

Result:

Implement a basic arithmetic calculator supporting +, -, *, /, and modulus operations

Exp.No:3 d	**Arithmetic Calculator**
Date:	

Aim:
To implement a **basic arithmetic calculator** in C++ that performs basic operations such as:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)

The program will take two numbers and the operator as input, perform the selected operation, and display the result.

Algorithm:

1. **Input the two numbers and the operator:**
   - Prompt the user to enter two numbers and the operator for the arithmetic operation.
2. **Perform the operation:**
   - Based on the operator entered by the user, perform the corresponding operation:
     - If +, perform addition.
     - If -, perform subtraction.
     - If *, perform multiplication.
     - If /, perform division (handle division by zero).
     - If %, perform modulus (handle modulus by zero).
3. **Display the result:**
   - Output the result of the operation.
4. **Loop for multiple calculations:**
   - Allow the user to perform multiple calculations or exit the program.

Program

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

int main() {
   double num1, num2;
   char op;
   double result;
```

```
while (true) {
    cout << "Enter first number: ";
    cin >> num1;

    cout << "Enter operator (+, -, *, /, %): ";
    cin >> op;

    cout << "Enter second number: ";
    cin >> num2;

    switch (op) {
        case '+':
            result = num1 + num2;
            cout << "Result: " << num1 << " + " << num2 << " = " << result << endl;
            break;

        case '-':
            result = num1 - num2;
            cout << "Result: " << num1 << " - " << num2 << " = " << result << endl;
            break;

        case '*':
            result = num1 * num2;
            cout << "Result: " << num1 << " * " << num2 << " = " << result << endl;
            break;

        case '/':
            if (num2 != 0) {
                result = num1 / num2;
                cout << "Result: " << num1 << " / " << num2 << " = " << result << endl;
            } else {
                cout << "Error: Division by zero is not allowed." << endl;
            }
            break;

        case '%':
            if (num2 != 0) {
                result = static_cast<int>(num1) % static_cast<int>(num2);
                cout << "Result: " << num1 << " % " << num2 << " = " << result << endl;
            } else {
                cout << "Error: Modulus by zero is not allowed." << endl;
            }
            break;

        default:
            cout << "Error: Invalid operator." << endl;
            break;
    }

    char choice;
    cout << "Do you want to perform another calculation? (y/n): ";
    cin >> choice;
    if (choice != 'y' && choice != 'Y') {
        break;
    }
}

cout << "Exiting the calculator. Goodbye!" << endl;
return 0;
}
```

Output:

Result:

Write a program to check if three sides can form a valid triangle

Exp.No:3 e	**Valid Triangle or not?**
Date:	

Aim:
To implement a program that checks if three given sides can form a valid triangle. The program will verify whether the sum of any two sides is greater than the third side, which is the condition for a valid triangle.

Algorithm:

1. **Input the three sides of the triangle:**
   - Prompt the user to input the lengths of the three sides.
2. **Check the triangle inequality theorem:**
   - The three sides can form a valid triangle if:
     - Sum of side1 and side2 > side3
     - Sum of side1 and side3 > side2
     - Sum of side2 and side3 > side1
3. **Display the result:**
   - If all three conditions are true, print that the sides form a valid triangle.
   - Otherwise, print that the sides cannot form a valid triangle.

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

bool isValidTriangle(int side1, int side2, int side3) {

   if (side1 + side2 > side3 && side1 + side3 > side2 && side2 + side3 > side1) {
      return true;
   }
   return false;
}

int main() {
   int side1, side2, side3;


   cout << "Enter the first side: ";
   cin >> side1;

   cout << "Enter the second side: ";
   cin >> side2;
```

```
    cout << "Enter the third side: ";
    cin >> side3;


    if (isValidTriangle(side1, side2, side3)) {
        cout << "The sides can form a valid triangle." << endl;
    } else {
        cout << "The sides cannot form a valid triangle." << endl;
    }

    return 0;
}
```

Output:

Result:

Develop a program to perform arithmetic operations on complex numbers

Exp.No:3 f	Arithmetic operation on complex numbers
Date:	

Aim:

To implement a program that performs arithmetic operations (addition, subtraction, multiplication, and division) on complex numbers. The program will allow the user to input two complex numbers and perform the operations.

Algorithm:

1. **Input the complex numbers:**
   - The user will input two complex numbers, each consisting of a real and an imaginary part.
2. **Perform the arithmetic operations:**
   - For **addition**, add the real parts and the imaginary parts separately.
   - For **subtraction**, subtract the real parts and the imaginary parts separately.
   - For **multiplication**, apply the distributive property (i.e., expand the product of the two complex numbers).
   - For **division**, use the formula for dividing complex numbers, which involves multiplying both the numerator and denominator by the conjugate of the denominator.
3. **Display the result:**
   - Display the result of each operation in the form a + bi.

Program

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

class Complex {
public:
    double real, imag;

    Complex(double r = 0, double i = 0) {
        real = r;
        imag = i;
    }

    void display() {
        if (imag >= 0)
            cout << real << " + " << imag << "i" << endl;
        else
            cout << real << " - " << -imag << "i" << endl;
    }
    Complex add(Complex c2) {
        return Complex(real + c2.real, imag + c2.imag);
    }

    Complex subtract(Complex c2) {
        return Complex(real - c2.real, imag - c2.imag);
    }

    Complex multiply(Complex c2) {
        return Complex(real * c2.real - imag * c2.imag, real * c2.imag + imag * c2.real);
    }

    Complex divide(Complex c2) {
        double denom = c2.real * c2.real + c2.imag * c2.imag;
        double realPart = (real * c2.real + imag * c2.imag) / denom;
        double imagPart = (imag * c2.real - real * c2.imag) / denom;
        return Complex(realPart, imagPart);
    }
};

int main() {
    double r1, i1, r2, i2;
    cout << "Enter the real and imaginary parts of the first complex number: ";
    cin >> r1 >> i1;
    Complex c1(r1, i1);

    cout << "Enter the real and imaginary parts of the second complex number: ";
    cin >> r2 >> i2;
    Complex c2(r2, i2);

   Complex result;

    cout << "\nFirst complex number: ";
    c1.display();

    cout << "Second complex number: ";
    c2.display();

    result = c1.add(c2);
    cout << "\nAddition: ";
    result.display();

    result = c1.subtract(c2);
    cout << "Subtraction: ";
    result.display();
    result = c1.multiply(c2);
    cout << "Multiplication: ";
    result.display();

    result = c1.divide(c2);
    cout << "Division: ";
    result.display();

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Implement a bank management system with C++ functions to handle transactions, account balances, and user authentication, thereby enhancing modularity, code reusability, and the overall efficiency of the program.

Exp.No:4 a	**Bank Management System**
Date:	

Aim:

To implement a **Bank Management System** using functions to handle:

- **Account balance management**
- **Transactions** (deposit, withdrawal)
- **User authentication** (login system)

The program will enhance **modularity**, **code reusability**, and **efficiency** by breaking down the functionalities into functions, making it easier to maintain and extend.

Algorithm:

1. **Define the Account Structure:**
    - Create a structure to hold account details such as account number, name, and balance.
2. **User Authentication:**
    - Create a login system with user authentication (using a simple username and password system).
3. **Transactions:**
    - Implement functions for:
        - **Deposit:** Increase the account balance.
        - **Withdraw:** Decrease the account balance, ensuring no overdrafts.
        - **Display Balance:** Show the current balance.
4. **Menu System:**
    - Display a menu with options for the user to choose:
        - Deposit
        - Withdraw
        - Display balance
        - Exit
5. **Main Functionality:**
    - The program should loop, allowing the user to perform multiple transactions until they choose to exit.

Program

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

```
struct Account {
   string name;
   string accountNumber;
   double balance;
};

bool authenticateUser(const string &username, const string &password) {
   const string correctUsername = "user";
   const string correctPassword = "pass";
   return (username == correctUsername && password == correctPassword);
}

void displayMenu() {
   cout << "\nBank Management System\n";
   cout << "1. Deposit\n";
   cout << "2. Withdraw\n";
   cout << "3. Check Balance\n";
   cout << "4. Exit\n";
   cout << "Please choose an option: ";
}

void deposit(Account &acc) {
   double amount;
   cout << "Enter amount to deposit: ";
   cin >> amount;
   if (amount > 0) {
      acc.balance += amount;
      cout << "Deposited " << amount << " successfully.\n";
   } else {
      cout << "Invalid amount. Please enter a positive value.\n";
   }
}

void withdraw(Account &acc) {
   double amount;
   cout << "Enter amount to withdraw: ";
   cin >> amount;
   if (amount > 0 && amount <= acc.balance) {
      acc.balance -= amount;
```

```
        cout << "Withdrawn " << amount << " successfully.\n";
    } else if (amount > acc.balance) {
        cout << "Insufficient funds.\n";
    } else {
        cout << "Invalid amount. Please enter a positive value.\n";
    }
}

void checkBalance(const Account &acc) {
    cout << "Current balance: " << acc.balance << endl;
}

int main() {
    Account account1 = {"John Doe", "123456789", 1000.0};  // Example account
    string username, password;
    int choice;
    bool isAuthenticated = false;

    cout << "Enter username: ";
    cin >> username;
    cout << "Enter password: ";
    cin >> password;

    if (authenticateUser(username, password)) {
        isAuthenticated = true;
        cout << "Authentication successful!\n";
    } else {
        cout << "Authentication failed. Exiting program.\n";
        return 0;
    }

    while (isAuthenticated) {
        displayMenu();
        cin >> choice;

        switch (choice) {
            case 1:
                deposit(account1);
                break;
            case 2:
                withdraw(account1);
                break;
            case 3:
                checkBalance(account1);
                break;
            case 4:
                cout << "Exiting the system. Goodbye!\n";
                return 0;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }

    return 0;
}
```

Output:

Result:

Develop a C++ program that swaps two integer values using pointers. Use type casting to access and manipulate the values through pointers

Exp.No:4 b	**Swapping of two numbers using pointers**
Date:	

Aim:
To develop a C++ program that swaps two integer values using pointers. The program will demonstrate how pointers can be used to manipulate variable values and also use type casting to work with pointers.

Algorithm:

1. **Input the two integer values:**
   - Prompt the user to input two integer values.
2. **Declare pointers to the integers:**
   - Declare two pointer variables that will point to the integer values.
3. **Perform type casting:**
   - Use type casting to access and manipulate the values of the integers through the pointers.
4. **Swap the values:**
   - Swap the values using the pointers by dereferencing them.
5. **Display the results:**
   - Display the values of the integers before and after swapping.

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

void swapUsingPointers(int* ptr1, int* ptr2) {
   int temp = *ptr1;
   *ptr1 = *ptr2;
   *ptr2 = temp;
}

int main() {
   int a, b;

   cout << "Enter the first integer: ";
   cin >> a;
   cout << "Enter the second integer: ";
   cin >> b;

   cout << "\nBefore swapping:\n";
```

```
    cout << "a = " << a << ", b = " << b << endl;

    swapUsingPointers(&a, &b);

    cout << "\nAfter swapping:\n";
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

Output:

Result:

Develop a C++ program to implement a student grade tracker using arrays. Create an array to store the grades of multiple students and implement functions to calculate the average, find the highest and lowest grades, and display the overall performance

Exp.No:4 c	**Student Grade tracker using Arrays**
Date:	

Aim:

To implement a **Student Grade Tracker** that:

- Stores the grades of multiple students in an array.
- Provides functions to calculate the average grade, find the highest and lowest grades, and display the overall performance.

Algorithm:

1. **Input the grades of students:**
   - Prompt the user to input the number of students.
   - Store the grades in an array.
2. **Implement functions:**
   - **Calculate Average:** Sum the grades and divide by the total number of students.
   - **Find Highest Grade:** Traverse the array to find the highest grade.
   - **Find Lowest Grade:** Traverse the array to find the lowest grade.
   - **Display Performance:** Based on the highest and lowest grades, display a performance report.
3. **Display the results:**
   - After calculating the average, highest, and lowest grades, display the results.

Program

```
#include <iostream>
#include <limits.h>
using namespace std;

double calculateAverage(int grades[], int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += grades[i];
   }
   return static_cast<double>(sum) / n;
}

int findHighestGrade(int grades[], int n) {
   int highest = INT_MIN;
   for (int i = 0; i < n; i++) {
      if (grades[i] > highest) {
         highest = grades[i];
```

```
        }
    }
    return highest;
}

int findLowestGrade(int grades[], int n) {
    int lowest = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (grades[i] < lowest) {
            lowest = grades[i];
        }
    }
    return lowest;
}

void displayPerformance(int grades[], int n) {
    double average = calculateAverage(grades, n);
    int highest = findHighestGrade(grades, n);
    int lowest = findLowestGrade(grades, n);

    cout << "\nOverall Performance:\n";
    cout << "Average Grade: " << average << endl;
    cout << "Highest Grade: " << highest << endl;
    cout << "Lowest Grade: " << lowest << endl;
}

int main() {
    int n;

    // Input the number of students
    cout << "Enter the number of students: ";
    cin >> n;

    // Declare an array to store the grades
    int grades[n];

    // Input the grades of each student
    for (int i = 0; i < n; i++) {
        cout << "Enter grade for student " << i + 1 << ": ";
        cin >> grades[i];
    }

    // Display the performance of the class
    displayPerformance(grades, n);

    return 0;
```

```
}
```

Output:

Result:

Write a function that takes an integer array and finds the maximum and minimum values without using built-in functions.

Exp.No:4 d	**Finding Maximum and Minimum values in an array**
Date:	

Aim:
To write a C++ function that accepts an integer array and finds the maximum and minimum values manually (without using built-in functions), demonstrating array traversal and condition-based comparison.

Algorithm:

1. **Input:**
    - Accept the size of the array and its elements from the user.
2. **Initialize max and min:**
    - Assign the first element of the array to both max and min.
3. **Traverse the array:**
    - Loop through the rest of the array.
    - If the current element is greater than max, update max.
    - If the current element is less than min, update min.
4. **Output:**
    - Return or print the max and min values.

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

void findMaxMin(int arr[], int size, int &max, int &min) {
    max = arr[0]; // Initialize max
    min = arr[0]; // Initialize min

    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
        if (arr[i] < min) {
            min = arr[i];
        }
    }
}

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[n];


    cout << "Enter " << n << " integers:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int maxVal, minVal;

    findMaxMin(arr, n, maxVal, minVal);

    cout << "Maximum value: " << maxVal << endl;
    cout << "Minimum value: " << minVal << endl;

    return 0;
}
```

Output:

Result:

Develop a program that multiplies two matrices using functions for input, processing, and output

Exp.No:4 e	Matrix Multiplication using Functions
Date:	

Aim:
To develop a C++ program that performs matrix multiplication by:

Taking two matrices as input,

Multiplying them using a separate processing function,

Displaying the result using an output function

Algorithm:

1. **Input:**
    - Get the dimensions of Matrix A (rowsA x colsA) and Matrix B (rowsB x colsB).
    - Ensure that colsA == rowsB for valid multiplication.
    - Input the elements of both matrices.
2. **Processing (Multiplication):**
    - Initialize the result matrix with zeros.
    - For each element in the result matrix, compute the sum of products of corresponding elements from A and B.
3. **Output:**
    - Display the resulting product matrix.

Program

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

void inputMatrix(int matrix[10][10], int rows, int cols, char name) {
    cout << "Enter elements of Matrix " << name << " (" << rows << "x" << cols << "):\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << "Element [" << i + 1 << "][" << j + 1 << "]: ";
            cin >> matrix[i][j];
        }
    }
}

void multiplyMatrices(int A[10][10], int B[10][10], int result[10][10],
                      int rowsA, int colsA, int colsB) {
    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsB; j++) {
            result[i][j] = 0; // Initialize to 0
            for (int k = 0; k < colsA; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void displayMatrix(int matrix[10][10], int rows, int cols, const string& title) {
    cout << title << ":\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << matrix[i][j] << "\t";
        }
        cout << endl;
    }
}

int main() {
    int A[10][10], B[10][10], result[10][10];
    int rowsA, colsA, rowsB, colsB;

    cout << "Enter number of rows and columns of Matrix A: ";
    cin >> rowsA >> colsA;

    cout << "Enter number of rows and columns of Matrix B: ";
    cin >> rowsB >> colsB;

    if (colsA != rowsB) {
        cout << "Matrix multiplication not possible. Columns of A must equal rows of B.\n";
        return 1;
    }

    inputMatrix(A, rowsA, colsA, 'A');
    inputMatrix(B, rowsB, colsB, 'B');

    multiplyMatrices(A, B, result, rowsA, colsA, colsB);

    displayMatrix(A, rowsA, colsA, "\nMatrix A");
    displayMatrix(B, rowsB, colsB, "\nMatrix B");
    displayMatrix(result, rowsA, colsB, "\nResultant Matrix (A x B)");

    return 0;
}
```

Output:

Result:

Write a function that reverses a string using pointers instead of indexing.

Exp.No:4 f	**Reverse String using pointers**
Date:	

Aim:
To write a C++ function that reverses a given string using pointers instead of array indexing, demonstrating understanding of pointer arithmetic and memory access.

Algorithm:

1. **Input the string.**
2. **Use two pointers:**
   - One pointing to the beginning of the string.
   - Another pointing to the end of the string.
3. **Swap characters** at the two pointers.
4. Move the start pointer forward and the end pointer backward.
5. Repeat until both pointers meet or cross.
6. Display the reversed string.

Program

```
#include <iostream>
#include <cstring> // for strlen
using namespace std;
void reverseString(char* str) {
   char* start = str;
   char* end = str + strlen(str) - 1;

   while (start < end) {
            char temp = *start;
      *start = *end;
      *end = temp;

      // Move pointers
      start++;
      end--;
   }
}

int main() {
   char str[100];

   cout << "Enter a string: ";
   cin.getline(str, 100);

      reverseString(str);
```

```
    cout << "Reversed string: " << str << endl;

    return 0;
}
```

Output:

Result:

Implement Bubble Sort using function pointers to allow sorting in ascending or descending order dynamically.

Exp.No:4 g	**Bubble sort using function pointer**
Date:	

Aim:
To implement the Bubble Sort algorithm using function pointers so that the user can choose ascending or descending sorting dynamically at runtime.

Algorithm:

1. **Input the array** elements from the user.
2. Define two comparator functions:
   - One for ascending order.
   - One for descending order.
3. Use a **function pointer** to dynamically select the comparison logic.
4. Perform **Bubble Sort**, comparing elements using the chosen function pointer.
5. Display the sorted array.

Program

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

bool ascending(int a, int b) {
   return a > b;  // Swap if first is greater
}

bool descending(int a, int b) {
   return a < b;  // Swap if first is smaller
}

void bubbleSort(int arr[], int size, bool (*compare)(int, int)) {
   for (int i = 0; i < size - 1; i++) {
      for (int j = 0; j < size - i - 1; j++) {
         if (compare(arr[j], arr[j + 1])) {
            // Swap
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}

void displayArray(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;
}

int main() {
   int arr[100], n, choice;

   cout << "Enter number of elements: ";
   cin >> n;

   cout << "Enter " << n << " elements:\n";
   for (int i = 0; i < n; i++) {
      cin >> arr[i];
   }

   cout << "Choose sorting order:\n1. Ascending\n2. Descending\nEnter choice: ";
   cin >> choice;

   bool (*compare)(int, int);
   if (choice == 1)
      compare = ascending;
   else
      compare = descending;

   bubbleSort(arr, n, compare);

   cout << "Sorted array:\n";
   displayArray(arr, n);

   return 0;
}
```

Output:

Result:

Implement a function that checks if an array is a palindrome using pointers

Exp.No:4 h	**Palindrome Array**
Date:	

Aim:

To implement a C++ function that checks whether an array is a palindrome using pointer manipulation instead of array indexing.

Algorithm:

1. **Input** the array elements from the user.
2. Initialize two pointers:
   - start pointing to the first element of the array.
   - end pointing to the last element of the array.
3. **Compare elements** pointed by start and end.
4. If at any point the elements are not equal, the array is **not a palindrome**.
5. Move start forward and end backward, and repeat until they meet or cross.
6. If all comparisons pass, the array is a **palindrome**.

Program

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

bool isPalindrome(int* arr, int size) {
    int* start = arr;
    int* end = arr + size - 1;

    while (start < end) {
        if (*start != *end) {
            return false;
        }
        start++;
        end--;
    }

    return true;
}

int main() {
    int arr[100], n;

    cout << "Enter the number of elements: ";
    cin >> n;

    cout << "Enter " << n << " integers:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    if (isPalindrome(arr, n)) {
        cout << "The array is a palindrome.\n";
    } else {
        cout << "The array is not a palindrome.\n";
    }

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Create a program that generates a personalized greeting message based on user input

Exp.No:5 a	**Generate Greetings**
Date:	

Aim:
To create a C++ program that takes user input (like name and time of day) and generates a personalized greeting message.

Algorithm:

1. Start the program and prompt the user for their **name**.
2. (Optional) Ask for the **time of day** or any other detail to personalize the message more.
3. Concatenate the input into a **custom greeting**.
4. Display the **personalized message**.

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

int main() {
   string name;
   int hour;

   cout << "Enter your name: ";
   getline(cin, name);

   cout << "Enter the current hour (0-23): ";
   cin >> hour;

   string greeting;

   if (hour >= 0 && hour < 12) {
      greeting = "Good morning";
   } else if (hour >= 12 && hour < 18) {
      greeting = "Good afternoon";
   } else if (hour >= 18 && hour < 24) {
      greeting = "Good evening";
   } else {
      greeting = "Hello";
   }

   cout << greeting << ", " << name << "! Have a great day!" << endl;

   return 0;
}
```

Output:

Result:

Develop a secure access card generator using string manipulation techniques

Exp.No:5 b	Access card Generator
Date:	

Aim:
To develop a C++ program that generates a secure access card by creating a unique access code using string manipulation, based on user details like name, ID, and department.

Algorithm:
1. Ask the user to input:
   - Full Name
   - Employee/User ID
   - Department
2. Extract initials from the name.
3. Combine initials, ID, and department code to generate a unique **access code**.
4. Optionally apply string transformations like:
   - Convert to uppercase/lowercase
   - Add random digits or special characters for security
5. Display a formatted **access card** with user details and the generated access code.

Program
```
#include <iostream>
#include <string>
#include <cstdlib>   // For rand()
#include <ctime>     // For time()
#include <cctype>    // For toupper()

using namespace std;

string getInitials(const string& name) {
   string initials = "";
   bool takeNext = true;
   for (char ch : name) {
      if (takeNext && isalpha(ch)) {
         initials += toupper(ch);
         takeNext = false;
      }
      if (ch == ' ') {
         takeNext = true;
      }
   }
   return initials;
}

string generateRandomDigits(int length) {
   string digits = "";
   for (int i = 0; i < length; i++) {
      digits += to_string(rand() % 10);
   }
   return digits;
}

int main() {
   string name, id, department;

   srand(time(0));

   cout << "Enter full name: ";
   getline(cin, name);
   cout << "Enter ID: ";
   getline(cin, id);
   cout << "Enter department: ";
   getline(cin, department);

   string initials = getInitials(name);
   string depCode = "";
   for (char ch : department) {
      if (isalpha(ch)) {
         depCode += toupper(ch);
      }
   }
   if (depCode.length() > 3) depCode = depCode.substr(0, 3);

   string accessCode = initials + id.substr(0, 3) + depCode + generateRandomDigits(3);

   cout << "\n==== Secure Access Card ====" << endl;
   cout << "Name       : " << name << endl;
   cout << "ID         : " << id << endl;
   cout << "Department : " << department << endl;
   cout << "Access Code: " << accessCode << endl;
   cout << "============================" << endl;

   return 0;
}
```

Output:

Result:

Create a sentence word counter that determines the number of words in an input text.

Exp.No:5 c	**Word Counter**
Date:	

Aim:
To create a C++ program that counts the number of words in an input sentence using string manipulation techniques.

Algorithm:

1. Prompt the user to **input a sentence**.
2. Initialize a **word counter**.
3. Traverse the sentence character by character:
    - Increment the word counter each time a space is followed by a non-space character.
4. Account for the **first word** and avoid counting extra spaces.
5. Display the total number of words.

Program

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

int countWords(const string& sentence) {
   int count = 0;
   bool inWord = false;

   for (char ch : sentence) {
      if (!isspace(ch)) {
         if (!inWord) {
            count++;
            inWord = true;
         }
      } else {
         inWord = false;
      }
   }

   return count;
}

int main() {
   string sentence;

   cout << "Enter a sentence: ";
   getline(cin, sentence);
```

```
    int wordCount = countWords(sentence);

    cout << "Number of words: " << wordCount << endl;

    return 0;
}
```

Output:

Result:

Develop a substring search program to find words within sentences.

Exp.No:5 d	**Find the Substring**
Date:	

Aim:
To develop a C++ program that searches for a substring (word or phrase) inside a given sentence and reports whether it was found and at which position

Algorithm:

1. Accept a **sentence** (main string) from the user.
2. Accept a **word/substring** to search for.
3. Use the **find()** function of the C++ string class to look for the substring.
4. If found:
   - Return the starting **index** of the first occurrence.
5. If not found:
   - Inform the user the substring doesn't exist in the sentence.

Program

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

int main() {
    string sentence, word;

    cout << "Enter a sentence: ";
    getline(cin, sentence);

    cout << "Enter the word to search: ";
    getline(cin, word);


    size_t position = sentence.find(word);

    if (position != string::npos) {
        cout << "The word \"" << word << "\" was found at position " << position << "." << endl;
    } else {
        cout << "The word \"" << word << "\" was not found in the sentence." << endl;
    }

    return 0;
}
```

Output:

Result:

Implement a word frequency counter that analyzes text.

Exp.No:5 e	**Word Frequency Counter**
Date:	

Aim:
To develop a C++ program that analyzes a block of input text and counts the frequency of each word, displaying how many times each word appears.

Algorithm:

1. Prompt the user to input a **block of text**.
2. Split the text into **individual words** (by spaces or punctuation).
3. Use a **map** (associative array) to store each word and its count.
4. Traverse each word:
    - If it exists in the map, **increment** the count.
    - Otherwise, **insert** the word with count 1.
5. Display all words with their frequencies

Program

```
#include <iostream>
#include <string>
#include <map>
#include <sstream>
#include <algorithm>
using namespace std;

string toLowerCase(const string& str) {
   string lower = str;
   transform(lower.begin(), lower.end(), lower.begin(), ::tolower);
   return lower;
}

string cleanWord(string word) {
   word.erase(remove_if(word.begin(), word.end(), ::ispunct), word.end());
   return toLowerCase(word);
}

int main() {
   string text;
   map<string, int> wordCount;

   cout << "Enter a block of text:\n";
   getline(cin, text);

   stringstream ss(text);
   string word;
```

```
    while (ss >> word) {
        word = cleanWord(word); // Clean punctuation and convert to lowercase
        if (!word.empty()) {
            wordCount[word]++;
        }
    }

    cout << "\nWord Frequencies:\n";
    for (const auto& pair : wordCount) {
        cout << pair.first << " : " << pair.second << endl;
    }

    return 0;
}
```

Output:

Result:

Write a program to check if two words are anagrams.

Exp.No:5 f	**Anagrams.**
Date:	

Aim:
To write a C++ program that checks whether two given words are anagrams — i.e., they contain the same characters in the same frequency, regardless of order.

Algorithm:

1. Input two strings from the user.
2. Remove any whitespace and convert both strings to lowercase for uniformity.
3. Check if the lengths are the same:
   - If not, they can't be anagrams.
4. Sort both strings.
5. Compare the sorted strings:
   - If they match, they are anagrams.
   - Otherwise, they are not.

Program

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

string normalize(const string& str) {
    string cleaned = "";
    for (char ch : str) {
        if (!isspace(ch)) {
            cleaned += tolower(ch);
        }
    }
    return cleaned;
}

bool areAnagrams(string str1, string str2) {
    str1 = normalize(str1);
    str2 = normalize(str2);

    if (str1.length() != str2.length()) {
        return false;
    }

    sort(str1.begin(), str1.end());
    sort(str2.begin(), str2.end());
    return str1 == str2;
}

int main() {
    string word1, word2;

    cout << "Enter first word: ";
    getline(cin, word1);
    cout << "Enter second word: ";
    getline(cin, word2);

    if (areAnagrams(word1, word2)) {
        cout << "The words are anagrams.\n";
    } else {
        cout << "The words are not anagrams.\n";
    }

    return 0;
}
```

Output:

Result:

Implement an algorithm to find the longest common subsequence between two strings

Exp.No:5 g	**Longest common subsequence**
Date:	

Aim:
To implement a C++ program that computes the Longest Common Subsequence (LCS) between two input strings using dynamic programming.

Algorithm:

1. **Input** two strings str1 and str2.
2. Create a **2D table dp[][]** of size (m+1) x (n+1) where:
   - m = length of str1
   - n = length of str2
3. Fill the dp table such that:
   - If characters match:
     dp[i][j] = dp[i-1][j-1] + 1
   - If characters don't match:
     dp[i][j] = max(dp[i-1][j], dp[i][j-1])
4. The length of the LCS is dp[m][n].
5. To **construct the LCS**, backtrack through the dp table from dp[m][n].

Program

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

string longestCommonSubsequence(string str1, string str2) {
    int m = str1.length(), n = str2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }

    string lcs = "";
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (str1[i - 1] == str2[j - 1]) {
            lcs = str1[i - 1] + lcs;
            i--; j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    return lcs;
}

int main() {
    string str1, str2;

    cout << "Enter first string: ";
    getline(cin, str1);
    cout << "Enter second string: ";
    getline(cin, str2);

    string lcs = longestCommonSubsequence(str1, str2);

    cout << "\nLongest Common Subsequence: " << lcs << endl;
    cout << "Length of LCS: " << lcs.length() << endl;

    return 0;
}
```

Output:

Result:

Develop a valid parentheses checker for expressions

Exp.No:5 h	Parentheses checker for expressions
Date:	

Aim:
To develop a C++ program that checks whether the parentheses in a given expression are valid and balanced. This includes checking (), {}, and [].

Algorithm:

1. Input an expression (string) from the user.
2. Create a **stack** to keep track of opening brackets.
3. Traverse each character in the string:
   - If it's an **opening bracket** ((, {, [), **push** it onto the stack.
   - If it's a **closing bracket**:
     - Check if the stack is **not empty** and if the **top** of the stack **matches** the current closing bracket.
     - If not, return invalid.
     - Else, **pop** the matched opening bracket from the stack.
4. After the traversal:
   - If the stack is **empty**, the expression is valid.
   - If not, it means there are unmatched brackets → invalid.

Program

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

bool isMatchingPair(char open, char close) {
   return (open == '(' && close == ')') ||
          (open == '{' && close == '}') ||
          (open == '[' && close == ']');
}

bool isValidExpression(const string& expr) {
   stack<char> stk;

   for (char ch : expr) {
      if (ch == '(' || ch == '{' || ch == '[') {
         stk.push(ch);
      } else if (ch == ')' || ch == '}' || ch == ']') {
         if (stk.empty() || !isMatchingPair(stk.top(), ch)) {
            return false;
         }
         stk.pop();
```

```
        }
    }

    return stk.empty();
}

int main() {
    string expression;

    cout << "Enter an expression: ";
    getline(cin, expression);

    if (isValidExpression(expression)) {
        cout << "The expression has valid and balanced parentheses." << endl;
    } else {
        cout << "The expression has invalid or unbalanced parentheses." << endl;
    }

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Develop a class hierarchy for different types of bank accounts

Exp.No:6.1 a	**Class hierarchy for different types of bank accounts.**
Date:	

Aim:
To develop a class hierarchy that represents different types of bank accounts, such as SavingsAccount and CheckingAccount, with basic functionalities like deposit, withdrawal, and account balance checking.

Algorithm:

1. **Define a base class BankAccount:**
   - This class will have member variables like accountHolderName, accountBalance, and methods for deposit, withdrawal, and checking balance.
2. **Derive classes for specific types of accounts:**
   - **SavingsAccount**: A subclass of BankAccount with additional functionality like interest calculation.
   - **CheckingAccount**: Another subclass of BankAccount that may include a method for handling overdraft protection.
3. **Inherit functionalities:**
   - Both derived classes inherit basic functionality (deposit, withdraw, and balance checking) from BankAccount.
4. **Demonstrate polymorphism**:
   - Create instances of different types of bank accounts and interact with them using polymorphism

Program

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

class BankAccount {
protected:
   string accountHolderName;
   double accountBalance;

public:
      BankAccount(string name, double balance = 0) {
      accountHolderName = name;
      accountBalance = balance;
   }

   void deposit(double amount) {
      accountBalance += amount;
      cout << "Deposited: " << amount << "\n";
```

```
    }

    bool withdraw(double amount) {
        if (amount > accountBalance) {
            cout << "Insufficient funds.\n";
            return false;
        }
        accountBalance -= amount;
        cout << "Withdrew: " << amount << "\n";
        return true;
    }

    double getBalance() const {
        return accountBalance;
    }

    virtual void displayInfo() const {
        cout << "Account Holder: " << accountHolderName << "\n";
        cout << "Balance: " << accountBalance << "\n";
    }
};

class SavingsAccount : public BankAccount {
private:
    double interestRate; // Interest rate for the savings account

public:
    SavingsAccount(string name, double balance, double rate)
        : BankAccount(name, balance), interestRate(rate) {}

    void applyInterest() {
        double interest = accountBalance * interestRate / 100;
        accountBalance += interest;
        cout << "Interest applied: " << interest << "\n";
    }

    void displayInfo() const override {
        BankAccount::displayInfo();  // Calling base class display
        cout << "Interest Rate: " << interestRate << "%\n";
    }
};

class CheckingAccount : public BankAccount {
private:
    double overdraftLimit; // Overdraft limit for checking account
```

```
public:
    CheckingAccount(string name, double balance, double limit)
        : BankAccount(name, balance), overdraftLimit(limit) {}

    bool withdraw(double amount) override {
        if (amount > (accountBalance + overdraftLimit)) {
            cout << "Exceeds overdraft limit.\n";
            return false;
        }
        accountBalance -= amount;
        cout << "Withdrew: " << amount << "\n";
        return true;
    }

    void displayInfo() const override {
        BankAccount::displayInfo();  // Calling base class display
        cout << "Overdraft Limit: " << overdraftLimit << "\n";
    }
};

int main() {
    SavingsAccount savings("Alice", 1000, 5);
    savings.displayInfo();
    savings.deposit(500);
    savings.applyInterest();
    savings.withdraw(200);
    savings.displayInfo();

    cout << "\n";

    CheckingAccount checking("Bob", 500, 200);
    checking.displayInfo();
    checking.deposit(300);
    checking.withdraw(600);  // Should work within overdraft limit
    checking.withdraw(700);  // Should exceed overdraft limit
    checking.displayInfo();

    return 0;
}
```

Output:

Result:

Create an Employee management system with a base Employee class and derived classes for different job roles

Exp.No:6.1.b	**Employee Management System using Class** .
Date:	

Aim:
To create a C++ program that models an Employee Management System where employees are represented by a base class Employee and specific job roles are derived from it, like Manager, Developer, and Intern. This system will allow us to manage employees, store their information, and display their details.

Algorithm:

1. **Base Class - Employee**:
   - Include attributes like name, id, salary, and jobRole.
   - Define methods for input and displaying employee details.
2. **Derived Classes** (Manager, Developer, Intern):
   - Inherit from Employee class and add role-specific attributes (like teamSize for Manager, programmingLanguages for Developer, etc.).
   - Implement any specific behavior or methods for each role (e.g., approveLeave() for Manager).
3. **Manage Employees**:
   - Create an array or list to store different employee objects.
   - Use polymorphism to display employee information based on their role.

Program

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

class Employee {
protected:
    string name;
    int id;
    double salary;

public:
    Employee(string n, int i, double s) : name(n), id(i), salary(s) {}

    virtual void displayDetails() const {
        cout << "Employee Name: " << name << endl;
        cout << "Employee ID: " << id << endl;
        cout << "Salary: $" << salary << endl;
    }

    virtual ~Employee() {}
};

class Manager : public Employee {
private:
    int teamSize;

public:
    Manager(string n, int i, double s, int t) : Employee(n, i, s), teamSize(t) {}

    void displayDetails() const override {
        Employee::displayDetails();
        cout << "Team Size: " << teamSize << endl;
    }


    void approveLeave() {
        cout << name << " has approved the leave." << endl;
    }
};

class Developer : public Employee {
private:
    string programmingLanguages;

public:
    Developer(string n, int i, double s, string p) : Employee(n, i, s), programmingLanguages(p) {}

    void displayDetails() const override {
        Employee::displayDetails();
        cout << "Programming Languages: " << programmingLanguages << endl;
    }


    void writeCode() {
        cout << name << " is writing code." << endl;
    }
};

class Intern : public Employee {
private:
    string collegeName;
```

```
public:
    Intern(string n, int i, double s, string c) : Employee(n, i, s), collegeName(c) {}

    void displayDetails() const override {
        Employee::displayDetails();
        cout << "College: " << collegeName << endl;
    }


    void workOnTask() {
        cout << name << " is working on a task." << endl;
    }
};

void displayEmployeeDetails(const Employee& emp) {
    emp.displayDetails();
    cout << "-------------------------------\n";
}

int main() {
    Manager mgr("Alice", 101, 75000, 10);
    Developer dev("Bob", 102, 65000, "C++, Java");
    Intern intern("Charlie", 103, 20000, "XYZ University");

    vector<Employee*> employees;
    employees.push_back(&mgr);
    employees.push_back(&dev);
    employees.push_back(&intern);

    for (const auto& emp : employees) {
        displayEmployeeDetails(*emp);
    }

    mgr.approveLeave();
    dev.writeCode();
    intern.workOnTask();

    return 0;
}
```

Output:

Result:

Implement a vehicle hierarchy with a base Vehicle class and derived classes for Cars and Bikes.

Exp.No:6.1. c	**Vehicle Hierarchy**
Date:	

Aim:
To implement a vehicle hierarchy where a base class Vehicle contains common attributes and functions for all vehicles, and derived classes Car and Bike represent specific types of vehicles with additional functionalities.

Algorithm:

1. **Define the base class Vehicle:**
   - The base class will have attributes like brand, model, year, and price.
   - It will include general functions for displaying vehicle details.
2. **Define derived classes Car and Bike:**
   - Both derived classes inherit from the Vehicle class.
   - The Car class will include additional attributes like numDoors and hasSunroof.
   - The Bike class will include attributes like typeOfHandlebars and hasGears.
3. **Use polymorphism** to handle different vehicle types dynamically.
4. **Display the details** of different vehicle types using the displayDetails() function.

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

class Vehicle {
protected:
    string brand;
    string model;
    int year;
    double price;

public:
    Vehicle(string b, string m, int y, double p) : brand(b), model(m), year(y), price(p) {}

    virtual void displayDetails() const {
        cout << "Brand: " << brand << endl;
        cout << "Model: " << model << endl;
        cout << "Year: " << year << endl;
        cout << "Price: $" << price << endl;
    }

    virtual ~Vehicle() {}
};
class Car : public Vehicle {
private:
    int numDoors;
    bool hasSunroof;

public:
    Car(string b, string m, int y, double p, int d, bool s)
        : Vehicle(b, m, y, p), numDoors(d), hasSunroof(s) {}

    void displayDetails() const override {
        Vehicle::displayDetails(); // Calling base class display
        cout << "Number of Doors: " << numDoors << endl;
        cout << "Has Sunroof: " << (hasSunroof ? "Yes" : "No") << endl;
    }
};

class Bike : public Vehicle {
private:
    string typeOfHandlebars;
    bool hasGears;

public:
    Bike(string b, string m, int y, double p, string h, bool g)
        : Vehicle(b, m, y, p), typeOfHandlebars(h), hasGears(g) {}

    void displayDetails() const override {
        Vehicle::displayDetails(); // Calling base class display
        cout << "Type of Handlebars: " << typeOfHandlebars << endl;
        cout << "Has Gears: " << (hasGears ? "Yes" : "No") << endl;
    }
};

void displayVehicleDetails(const Vehicle& v) {
    v.displayDetails();
    cout << "-------------------------------\n";
}

int main() {
    Car car1("Toyota", "Camry", 2020, 25000, 4, true);
    Bike bike1("Yamaha", "YZF-R3", 2021, 5000, "Sport", true);

    displayVehicleDetails(car1);
    displayVehicleDetails(bike1);

    return 0;
}
```
Output:

Result:

Design a university management system with base class Person and derived classes for Student and Faculty.

Exp.No:6.1. d	**University Management System using class**
Date:	

Aim:

To design a University Management System where the base class Person contains common attributes and behaviors (like name and age), and the derived classes Student and Faculty represent specific roles in a university. These derived classes will add specific attributes and methods based on the role (like studentID, degree for Student, and employeeID, department for Faculty).

Algorithm:

1. **Define the base class Person:**
   - The base class should contain common attributes like name, age, and address.
   - Include methods to input and display the details of the person.
2. **Define derived classes Student and Faculty:**
   - **Student class**: Contains additional attributes like studentID, degree, and methods for enrolling and displaying student-specific information.
   - **Faculty class**: Contains additional attributes like employeeID, department, and methods for assigning courses and displaying faculty-specific information.
3. **Use polymorphism** to handle different types of people in the university dynamically.
4. **Display the details** of the person using the overridden methods of the derived classes

Program

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

class Person {
protected:
   string name;
   int age;
   string address;

public:
   Person(string n, int a, string addr) : name(n), age(a), address(addr) {}

   virtual void displayDetails() const {
      cout << "Name: " << name << endl;
      cout << "Age: " << age << endl;
      cout << "Address: " << address << endl;
   }

   virtual ~Person() {}
```

```
};

class Student : public Person {
private:
    string studentID;
    string degree;

public:
    Student(string n, int a, string addr, string id, string deg)
        : Person(n, a, addr), studentID(id), degree(deg) {}

    void displayDetails() const override {
        Person::displayDetails(); // Calling base class display
        cout << "Student ID: " << studentID << endl;
        cout << "Degree: " << degree << endl;
    }

    void enrollCourse(string courseName) {
        cout << "Student " << name << " has enrolled in " << courseName << " course." << endl;
    }
};

class Faculty : public Person {
private:
    string employeeID;
    string department;

public:
    Faculty(string n, int a, string addr, string id, string dept)
        : Person(n, a, addr), employeeID(id), department(dept) {}

    void displayDetails() const override {
        Person::displayDetails(); // Calling base class display
        cout << "Employee ID: " << employeeID << endl;
        cout << "Department: " << department << endl;
    }

    void assignCourse(string courseName) {
        cout << "Faculty " << name << " has been assigned to teach " << courseName << " course."
<< endl;
    }
};

void displayPersonDetails(const Person& p) {
    p.displayDetails();
    cout << "------------------------------\n";
```

```
}

int main() {
    Student student1("Alice", 20, "1234 Maple St", "S12345", "Computer Science");
    Faculty faculty1("Dr. Smith", 40, "5678 Oak Rd", "F98765", "Mathematics");

    displayPersonDetails(student1);
    displayPersonDetails(faculty1);


    student1.enrollCourse("Data Structures");
    faculty1.assignCourse("Calculus");

    return 0;
}
```

Output:

Result:

Develop a transport management system with hierarchical classes

Exp.No: 6.1. e	**Transport Management System**
Date:	

Aim:
To create a **Transport Management System** using a class hierarchy where:

- The base class **Transport** will have common attributes and methods for all types of transport.
- The derived classes **Bus**, **Train**, and **Airplane** will have specific attributes and methods for each type of transport.

The system will allow users to view details of different types of transport, including the number of seats, transport type, and operating range.

Algorithm:

1. **Define the base class Transport:**
   - This class will have common attributes like transportType, capacity, and routeRange.
   - It will include general methods like displayDetails() to display transport information.
2. **Define derived classes Bus, Train, and Airplane:**
   - Each class will inherit from Transport and add specific attributes and methods.
   - **Bus**: Additional attributes like numDoors.
   - **Train**: Additional attributes like numCarriages.
   - **Airplane**: Additional attributes like numEngines and flightAltitude.
3. **Use polymorphism** to display different transport details dynamically.
4. **Display transport details** using the overridden methods of derived classes.

Program

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

class Transport {
protected:
   string transportType;
   int capacity;
   double routeRange;

public:
    Transport(string type, int cap, double range) : transportType(type), capacity(cap),
routeRange(range) {}

    virtual void displayDetails() const {
        cout << "Transport Type: " << transportType << endl;
        cout << "Capacity: " << capacity << " passengers" << endl;
        cout << "Route Range: " << routeRange << " km" << endl;
    }

    virtual ~Transport() {}
};

class Bus : public Transport {
private:
    int numDoors;

public:
    Bus(string type, int cap, double range, int doors) : Transport(type, cap, range),
numDoors(doors) {}

    void displayDetails() const override {
        Transport::displayDetails(); // Calling base class display
        cout << "Number of Doors: " << numDoors << endl;
    }
};

class Train : public Transport {
private:
    int numCarriages;

public:
    Train(string type, int cap, double range, int carriages) : Transport(type, cap, range),
numCarriages(carriages) {}

    void displayDetails() const override {
        Transport::displayDetails(); // Calling base class display
        cout << "Number of Carriages: " << numCarriages << endl;
    }
};
```

```
class Airplane : public Transport {
private:
    int numEngines;
    double flightAltitude;

public:
    Airplane(string type, int cap, double range, int engines, double altitude)
        : Transport(type, cap, range), numEngines(engines), flightAltitude(altitude) {}

    void displayDetails() const override {
        Transport::displayDetails(); // Calling base class display
        cout << "Number of Engines: " << numEngines << endl;
        cout << "Flight Altitude: " << flightAltitude << " meters" << endl;
    }
};
void displayTransportDetails(const Transport& t) {
    t.displayDetails();
    cout << "------------------------------\n";
}

int main() {
    Bus bus1("Bus", 50, 300, 2);
    Train train1("Train", 500, 1500, 10);
    Airplane airplane1("Airplane", 150, 12000, 4, 35000);

    displayTransportDetails(bus1);
    displayTransportDetails(train1);
    displayTransportDetails(airplane1);

    return 0;
}
```

Output:

Result:

Create a multi-level inheritance example with Animal -> Mammal -> Human.

Exp.No:6.1. f	Multilevel Inheritance
Date:	

Aim:

To demonstrate **multi-level inheritance** in C++, where:

- The base class Animal represents general animal attributes.
- The derived class Mammal inherits from Animal and adds attributes specific to mammals.
- The class Human further inherits from Mammal and represents human-specific attributes and behaviors.

Algorithm:

1. **Define the base class Animal:**
   - This class will have general attributes such as name and age.
   - It will include general methods like eat() and sleep().
2. **Define the derived class Mammal:**
   - This class inherits from Animal and adds attributes and methods specific to mammals, like hasFur() and giveBirth().
3. **Define the derived class Human:**
   - This class inherits from Mammal and adds human-specific attributes like name and occupation.
   - It will include a method speak().
4. **Demonstrate polymorphism** by creating objects of different types (Human, Mammal, Animal) and calling the methods.

Program

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

class Animal {
protected:
   string name;
   int age;

public:
   Animal(string n, int a) : name(n), age(a) {}

   void eat() {
      cout << name << " is eating." << endl;
   }
   void sleep() {
        cout << name << " is sleeping." << endl;
    }

    virtual void displayDetails() const {
        cout << "Animal Name: " << name << endl;
        cout << "Age: " << age << endl;
    }

    virtual ~Animal() {}
};

class Mammal : public Animal {
protected:
    bool hasFur;

public:
    Mammal(string n, int a, bool fur) : Animal(n, a), hasFur(fur) {}

    void giveBirth() {
        cout << name << " gives birth to live young." << endl;
    }

        void hasFurCheck() {
        if (hasFur)
            cout << name << " has fur." << endl;
        else
            cout << name << " does not have fur." << endl;
    }

    void displayDetails() const override {
        Animal::displayDetails(); // Calling base class display
        cout << "Has Fur: " << (hasFur ? "Yes" : "No") << endl;
    }
};

class Human : public Mammal {
private:
    string occupation;

public:
    Human(string n, int a, bool fur, string job) : Mammal(n, a, fur), occupation(job) {}

    void speak() {
        cout << name << " says: Hello, I am a " << occupation << "." << endl;
    }
    void displayDetails() const override {
        Mammal::displayDetails(); // Calling base class display
        cout << "Occupation: " << occupation << endl;
    }
};

int main() {
    Animal animal("Generic Animal", 5);
    Mammal mammal("Elephant", 10, true);
    Human human("John", 30, true, "Engineer");

        cout << "Animal Details:" << endl;
    animal.displayDetails();
    cout << endl;

    cout << "Mammal Details:" << endl;
    mammal.displayDetails();
    cout << endl;

    cout << "Human Details:" << endl;
    human.displayDetails();
    human.speak();
    cout << endl;

    mammal.giveBirth();
    human.giveBirth();
    human.eat();

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Create a polymorphic shape class with Circle, Rectangle, and Triangle

Exp.No:6.2. a	**Polymorphic Shape in class concepts**
Date:	

Aim:
To create a polymorphic shape class using C++ that demonstrates the concept of inheritance and polymorphism. In this program, the base class will be Shape, and the derived classes will be Circle, Rectangle, and Triangle. Each derived class will implement a method area() to calculate the area of the respective shape, and the base class will define a virtual function to allow polymorphic behavior.

Algorithm:

1. **Define the base class Shape:**
   - The class will have a virtual method area() to calculate the area of the shape.
   - It will be a pure virtual function, making the class abstract.
   - It will also include a constructor to initialize shape-related properties.
2. **Define the derived classes Circle, Rectangle, and Triangle:**
   - Each derived class will inherit from the base class Shape and implement the area() method to calculate the area of that specific shape.
   - Each shape will have its own specific attributes like radius (Circle), length and breadth (Rectangle), and base and height (Triangle).
3. **Use polymorphism** by creating pointers to the base class Shape and dynamically calling the area() function of the appropriate derived class.
4. **Demonstrate polymorphism** in the main() function by creating objects of different shapes and calculating their areas using base class pointers.

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

class Shape {
public:

   virtual double area() const = 0; // Pure virtual function, makes Shape an abstract class

   virtual ~Shape() {}
};

class Circle : public Shape {
private:
   double radius;
```

```
public:
        Circle(double r) : radius(r) {}

    double area() const override {
        return M_PI * radius * radius; // Area of Circle: π * r^2
    }
};

class Rectangle : public Shape {
private:
    double length, breadth;

public:
    Rectangle(double l, double b) : length(l), breadth(b) {}

    double area() const override {
        return length * breadth; // Area of Rectangle: length * breadth
    }
};

class Triangle : public Shape {
private:
    double base, height;

public:
    Triangle(double b, double h) : base(b), height(h) {}

    double area() const override {
        return 0.5 * base * height; // Area of Triangle: 0.5 * base * height
    }
};

void displayArea(Shape* shape) {
    cout << "Area: " << shape->area() << endl;
}

int main() {
    Circle circle(5);
    Rectangle rectangle(4, 6);
    Triangle triangle(3, 7);

    Shape* shape1 = &circle;
    Shape* shape2 = &rectangle;
    Shape* shape3 = &triangle;

    cout << "Circle: ";
    displayArea(shape1);

    cout << "Rectangle: ";
    displayArea(shape2);

    cout << "Triangle: ";
    displayArea(shape3);

    return 0;
}
```

Output:

Result:

Implement function overloading for different arithmetic operations.

Exp.No:6.2. b	**Function Overloading**
Date:	

Aim:

To demonstrate function overloading in C++ by implementing multiple functions with the same name, but with different signatures (i.e., different parameter types or numbers). The program will provide multiple arithmetic operations, including addition, subtraction, multiplication, and division, by overloading the operate() function.

Algorithm:

1. **Function Overloading Concept:**
    - In function overloading, multiple functions can have the same name but different parameters (either in number or type).
    - The correct version of the overloaded function is chosen by the compiler based on the number and types of arguments passed.
2. **Implementing the Arithmetic Operations:**
    - Create multiple versions of the operate() function to handle different arithmetic operations:
        - operate(int, int) for operations on integers.
        - operate(double, double) for operations on floating-point numbers.
        - operate(int, int, int) for operations on three integers.
3. **Perform arithmetic operations** such as addition, subtraction, multiplication, and division in each overloaded function.
4. **Main Function:**
    - In the main() function, call the overloaded operate() function with different types and numbers of arguments.
    - Display the results of the operations

Program

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

int operate(int a, int b) {
   return a + b;
}

int operate(int a, int b, int c) {
   return a - b - c;
}

int operate(int a, int b, int c, int d) {
   return a * b * c * d;
}
```

```
double operate(double a, double b) {
    if (b == 0) {
        cout << "Error! Division by zero." << endl;
        return 0;
    }
    return a / b;
}

double operate(double a, double b, double c) {
    return a + b + c;
}

int main() {
    int num1 = 10, num2 = 5, num3 = 3, num4 = 2;
    double d1 = 7.5, d2 = 2.5, d3 = 1.5;

    cout << "Addition of two integers: " << operate(num1, num2) << endl;

    cout << "Subtraction of three integers: " << operate(num1, num2, num3) << endl;

    cout << "Multiplication of four integers: " << operate(num1, num2, num3, num4) << endl;

    cout << "Division of two floating-point numbers: " << operate(d1, d2) << endl;

        cout << "Addition of three floating-point numbers: " << operate(d1, d2, d3) << endl;

    return 0;
}
```

Output:

Result:

Develop a dynamic dispatch mechanism for different employee types

Exp.No:6.2.c	**Dynamic dispatch mechanism**
Date:	

Aim:
To develop a dynamic dispatch mechanism in C++ using virtual functions to handle different types of employees (e.g., Full-Time, Part-Time, Intern) and calculate their respective salaries based on specific rules. This demonstrates the principle of runtime polymorphism.

Algorithm:

1. **Create a base class Employee:**
   - Include common properties such as name and employee ID.
   - Declare a virtual function calculateSalary() to be overridden by derived classes.
2. **Create derived classes** like:
   - FullTimeEmployee: with a fixed monthly salary.
   - PartTimeEmployee: salary calculated based on hours worked and hourly rate.
   - Intern: with a stipend.
3. **Override the calculateSalary() method** in each derived class.
4. In main(), **use base class pointers** to store different employee types and invoke their salary calculation via **dynamic dispatch** (i.e., calling the correct overridden method at runtime).

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

class Employee {
protected:
   string name;
   int empId;

public:
   Employee(string n, int id) : name(n), empId(id) {}
   virtual void calculateSalary() = 0; // Pure virtual function
   virtual ~Employee() {}
};

class FullTimeEmployee : public Employee {
private:
   double monthlySalary;

public:
   FullTimeEmployee(string n, int id, double salary)
      : Employee(n, id), monthlySalary(salary) {}
```

```
    void calculateSalary() override {
        cout << "Full-Time Employee: " << name << ", ID: " << empId
             << ", Monthly Salary: $" << monthlySalary << endl;
    }
};

class PartTimeEmployee : public Employee {
private:
    int hoursWorked;
    double hourlyRate;

public:
    PartTimeEmployee(string n, int id, int hours, double rate)
        : Employee(n, id), hoursWorked(hours), hourlyRate(rate) {}

    void calculateSalary() override {
        double salary = hoursWorked * hourlyRate;
        cout << "Part-Time Employee: " << name << ", ID: " << empId
             << ", Salary: $" << salary << " (" << hoursWorked << " hrs @ $" << hourlyRate <<
"/hr)" << endl;
    }
};

class Intern : public Employee {
private:
    double stipend;

public:
    Intern(string n, int id, double stip)
        : Employee(n, id), stipend(stip) {}

    void calculateSalary() override {
        cout << "Intern: " << name << ", ID: " << empId
             << ", Stipend: $" << stipend << endl;
    }
};

int main() {
    vector<Employee*> employees;

    employees.push_back(new FullTimeEmployee("Alice", 101, 5000));
  employees.push_back(new PartTimeEmployee("Bob", 102, 20, 15.5));
    employees.push_back(new Intern("Charlie", 103, 800));

    for (Employee* emp : employees) {
        emp->calculateSalary();
    }

    for (Employee* emp : employees) {
        delete emp;
    }

    return 0;
}
```

Output:

Result:

Implement runtime polymorphism for different transport modes.

Exp.No:6.2.d	**Runtime polymorphism**
Date:	

Aim:
To implement runtime polymorphism in C++ by defining a base class Transport with a virtual function move() and overriding it in derived classes such as Car, Bike, and Train. This demonstrates how different transport modes can exhibit distinct behaviors when accessed via base class pointers or references.

Algorithm:

1. **Create a base class Transport:**
   - Declare a virtual function move() to describe the mode of transport.
2. **Create derived classes** like:
   - Car
   - Bike
   - Train
   - Each of these overrides the move() function to provide specific behavior.
3. **In the main function:**
   - Create an array or vector of Transport* pointing to objects of different derived classes.
   - Call the move() function using the base class pointer.
   - The correct overridden version will be executed based on the object type (runtime polymorphism)

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

class Transport {
public:
    virtual void move() = 0; // Pure virtual function
    virtual ~Transport() {}
};

class Car : public Transport {
public:
    void move() override {
        cout << "Car is moving on the road." << endl;
    }
};

class Bike : public Transport {
public:
    void move() override {
        cout << "Bike is zipping through the traffic." << endl;
    }
};

class Train : public Transport {
public:
    void move() override {
        cout << "Train is running on the railway tracks." << endl;
    }
};

int main() {
    vector<Transport*> transports;

    transports.push_back(new Car());
    transports.push_back(new Bike());
    transports.push_back(new Train());

    for (Transport* t : transports) {
        t->move(); // Runtime polymorphism
    }

    for (Transport* t : transports) {
        delete t;
    }

    return 0;
}
```

Output:

Result:

Design a hospital appointment system using virtual functions

Exp.No:6.2.e	**Hospital Appointment System using Virtual Functions**
Date:	

Aim:
To design a hospital appointment system using virtual functions in C++, where different types of appointments (General, Specialist, Emergency) inherit from a base class and override a virtual method to display specific appointment details.

Algorithm:

1. **Create a base class Appointment:**
   - Contains virtual function bookAppointment() to be overridden.
   - May store common attributes like patient name and appointment ID.
2. **Create derived classes**:
   - GeneralAppointment
   - SpecialistAppointment
   - EmergencyAppointment
   - Each class overrides bookAppointment() to display its own booking details.
3. **In the main() function:**
   - Use base class pointers to point to objects of derived classes.
   - Call bookAppointment() via base class pointers.
   - Demonstrate **runtime polymorphism**.

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

class Appointment {
protected:
    string patientName;
    int appointmentID;

public:
    Appointment(string name, int id) : patientName(name), appointmentID(id) {}

    virtual void bookAppointment() = 0;

    virtual ~Appointment() {}
};

class GeneralAppointment : public Appointment {
public:
    GeneralAppointment(string name, int id) : Appointment(name, id) {}
    void bookAppointment() override {
        cout << "General Appointment booked for " << patientName
             << " [ID: " << appointmentID << "]" << endl;
    }
};

class SpecialistAppointment : public Appointment {
    string specialistType;

public:
    SpecialistAppointment(string name, int id, string spec)
        : Appointment(name, id), specialistType(spec) {}

    void bookAppointment() override {
        cout << "Specialist Appointment (" << specialistType << ") booked for "
             << patientName << " [ID: " << appointmentID << "]" << endl;
    }
};

class EmergencyAppointment : public Appointment {
public:
    EmergencyAppointment(string name, int id) : Appointment(name, id) {}

    void bookAppointment() override {
        cout << "EMERGENCY! Appointment booked immediately for "
             << patientName << " [ID: " << appointmentID << "]" << endl;
    }
};

int main() {
vector<Appointment*> appointments;

    appointments.push_back(new GeneralAppointment("Alice", 1001));
    appointments.push_back(new SpecialistAppointment("Bob", 1002, "Cardiologist"));
    appointments.push_back(new EmergencyAppointment("Charlie", 1003));

    cout << "Hospital Appointment System:\n------------------------------\n";

    for (Appointment* app : appointments) {
        app->bookAppointment();  // Runtime polymorphism
    }

    for (Appointment* app : appointments) {
        delete app;
    }
    return 0;
}
```
Output:

Result:

Create a virtual base class example for a hierarchy of electronic devices.

Exp.No:6.2. f	Concept on Virtual Base Class
Date:	

Aim:
To create a C++ program that uses a virtual base class to avoid the diamond problem in multiple inheritance, within a hierarchy of electronic devices such as Computers, Smartphones, and SmartWatches, all derived from a common base class.

Algorithm:

1. **Create a virtual base class ElectronicDevice**:
   - Contains general attributes (e.g., brand) and a virtual function showDeviceInfo().
2. **Create intermediate classes**:
   - Computer
   - Phone
   - Both inherit **virtually** from ElectronicDevice.
3. **Create a derived class SmartWatchPhone**:
   - Inherits from both Computer and Phone.
   - Overrides showDeviceInfo() and demonstrates that only **one instance** of ElectronicDevice is present due to virtual inheritance.
4. In main(), create an object of SmartWatchPhone and call the function to display details.

Program

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

class ElectronicDevice {
protected:
    string brand;

public:
    ElectronicDevice(string b) : brand(b) {}

    virtual void showDeviceInfo() {
        cout << "Brand: " << brand << endl;
    }
};

class Computer : virtual public ElectronicDevice {
public:
    Computer(string b) : ElectronicDevice(b) {}

    void computerFeature() {
        cout << "Feature: Can run high-performance applications." << endl;
    }
};

class Phone : virtual public ElectronicDevice {
public:
    Phone(string b) : ElectronicDevice(b) {}

    void phoneFeature() {
        cout << "Feature: Can make calls and access internet." << endl;
    }
};

class SmartWatchPhone : public Computer, public Phone {
public:
    SmartWatchPhone(string b)
        : ElectronicDevice(b), Computer(b), Phone(b) {}

    void showDeviceInfo() override {
        cout << "SmartWatchPhone Device Info:" << endl;
        ElectronicDevice::showDeviceInfo();
        computerFeature();
        phoneFeature();
        cout << "Feature: Wearable and fitness tracking enabled." << endl;
    }
};

int main() {
    SmartWatchPhone device("TechCorp");

    device.showDeviceInfo();

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Implement an Employee Database using a linked list with dynamic memory allocation

Exp.No:7 a	**Employee Database linked list with dynamic memory allocation**
Date:	

Aim:
To implement an Employee Database using a singly linked list in C++ with dynamic memory allocation, enabling insertion, display, and deletion of employee records.

Algorithm:

1. **Define a structure or class Employee**:
   - Contains data fields: ID, name, salary, and a pointer to the next node.
2. **Create functions**:
   - addEmployee(): Adds a new employee to the end of the list.
   - displayEmployees(): Displays all employee records.
   - deleteEmployee(): Deletes an employee by ID.
   - Use new for dynamic memory allocation and delete for deallocation.
3. **In main() function**:
   - Provide a menu for the user to perform operations like add, display, delete.
   - Loop until the user chooses to exit.

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

class Employee {
public:
   int id;
   string name;
   float salary;
   Employee* next;

   Employee(int id, string name, float salary) {
      this->id = id;
      this->name = name;
      this->salary = salary;
      this->next = nullptr;
   }
};

class EmployeeList {
private:
   Employee* head;

public:
```

```
EmployeeList() {
    head = nullptr;
}

void addEmployee(int id, string name, float salary) {
    Employee* newEmp = new Employee(id, name, salary);
    if (head == nullptr) {
        head = newEmp;
    } else {
        Employee* temp = head;
        while (temp->next != nullptr)
            temp = temp->next;
        temp->next = newEmp;
    }
    cout << "Employee added successfully.\n";
}

void displayEmployees() {
    if (head == nullptr) {
        cout << "No employees in the database.\n";
        return;
    }
    Employee* temp = head;
    cout << "\nEmployee List:\n";
    while (temp != nullptr) {
        cout << "ID: " << temp->id
             << ", Name: " << temp->name
             << ", Salary: " << temp->salary << "\n";
        temp = temp->next;
    }
}

void searchEmployee(int id) {
    Employee* temp = head;
    while (temp != nullptr) {
        if (temp->id == id) {
            cout << "Employee Found - ID: " << temp->id
                 << ", Name: " << temp->name
                 << ", Salary: " << temp->salary << "\n";
            return;
        }
        temp = temp->next;
    }
    cout << "Employee with ID " << id << " not found.\n";
}
```

```
    void deleteEmployee(int id) {
        Employee* temp = head;
        Employee* prev = nullptr;

        while (temp != nullptr && temp->id != id) {
            prev = temp;
            temp = temp->next;
        }

        if (temp == nullptr) {
            cout << "Employee with ID " << id << " not found.\n";
            return;
        }

        if (prev == nullptr) {
            head = head->next;
        } else {
            prev->next = temp->next;
        }

        delete temp;
        cout << "Employee with ID " << id << " deleted successfully.\n";
    }

    ~EmployeeList() {
        while (head != nullptr) {
            Employee* temp = head;
            head = head->next;
            delete temp;
        }
    }
};

int main() {
    EmployeeList empDB;
    int choice, id;
    string name;
    float salary;

    do {
        cout << "\n--- Employee Database Menu ---\n";
        cout << "1. Add Employee\n";
        cout << "2. Display All Employees\n";
        cout << "3. Search Employee by ID\n";
        cout << "4. Delete Employee by ID\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter ID: ";
                cin >> id;
                cin.ignore();
                cout << "Enter Name: ";
                getline(cin, name);
                cout << "Enter Salary: ";
                cin >> salary;
                empDB.addEmployee(id, name, salary);
                break;
            case 2:
                empDB.displayEmployees();
                break;
            case 3:
                cout << "Enter ID to search: ";
                cin >> id;
                empDB.searchEmployee(id);
                break;
            case 4:
                cout << "Enter ID to delete: ";
                cin >> id;
                empDB.deleteEmployee(id);
                break;
            case 5:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice. Try again.\n";
        }

    } while (choice != 5);

    return 0;
}
```

Output:

Result:

Develop an Undo/Redo system using a stack.

Exp.No:7 b	Develop an Undo/Redo system using a stack.
Date:	

Aim:
To develop an Undo/Redo system using stack in C++ that mimics the behavior of simple text editors, allowing the user to perform, undo, and redo operations.

Algorithm:

1. **Create two stacks**:
   - undoStack: Stores history of performed actions.
   - redoStack: Stores actions undone for potential redo.
2. **On a new action**:
   - Push the action onto undoStack.
   - Clear the redoStack (as new action invalidates redo history).
3. **On undo**:
   - Pop the top of undoStack and push it onto redoStack.
4. **On redo**:
   - Pop the top of redoStack and push it back onto undoStack.
5. Display current action list (top of undoStack is the current state)

Program

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

class TextEditor {
private:
    string currentText;
    stack<string> undoStack;
    stack<string> redoStack;

public:
    void addText(const string& newText) {
        undoStack.push(currentText); // Save current state for undo
        currentText += newText;
        while (!redoStack.empty()) redoStack.pop();
        cout << "Text added.\n";
    }

    void undo() {
        if (!undoStack.empty()) {
            redoStack.push(currentText);
            currentText = undoStack.top();
            undoStack.pop();
            cout << "Undo performed.\n";
        } else {
            cout << "Nothing to undo.\n";
        }
    }

    void redo() {
        if (!redoStack.empty()) {
            undoStack.push(currentText);
            currentText = redoStack.top();
            redoStack.pop();
            cout << "Redo performed.\n";
        } else {
            cout << "Nothing to redo.\n";
        }
    }

    void display() const {
        cout << "Current Text: " << currentText << "\n";
    }
};

int main() {
    TextEditor editor;
    int choice;
    string text;

    do {
        cout << "\n--- Text Editor Menu ---\n";
        cout << "1. Add Text\n";
        cout << "2. Undo\n";
        cout << "3. Redo\n";
        cout << "4. Display Text\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore();
        switch (choice) {
            case 1:
                cout << "Enter text to add: ";
                getline(cin, text);
                editor.addText(text);
                break;
            case 2:
                editor.undo();
                break;
            case 3:
                editor.redo();
                break;
            case 4:
                editor.display();
                break;
            case 5:
                cout << "Exiting editor...\n";
                break;
            default:
                cout << "Invalid choice.\n";
        }
    } while (choice != 5);

    return 0;
}
```

Output:

Result:

Create a print queue management system using a queue

Exp.No:7 c	Queue Management System
Date:	

Aim:
To develop a Print Queue Management System using queue data structure in C++ that manages print jobs in a First In First Out (FIFO) order.

Algorithm:

1. **Use std::queue** to manage print jobs.
2. Create a structure PrintJob with:
   - Job ID
   - Document Name
   - Number of Pages
3. Create operations:
   - addJob(): Add a new print job to the queue.
   - processJob(): Process and remove the front job from the queue.
   - displayQueue(): Show all pending jobs.
4. In main(), present a menu for users to choose operations.

Program

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

class PrintJob {
public:
    int jobId;
    string documentName;

    PrintJob(int id, string name) {
        jobId = id;
        documentName = name;
    }
};

class PrintQueue {
private:
    queue<PrintJob> jobQueue;
    int nextJobId = 1;

public:
    void addJob(const string& documentName) {
        PrintJob newJob(nextJobId++, documentName);
        jobQueue.push(newJob);
        cout << "Added print job: ID = " << newJob.jobId << ", Document = " <<
newJob.documentName << "\n";
    }

        void processJob() {
        if (jobQueue.empty()) {
            cout << "No jobs to process.\n";
            return;
        }
        PrintJob job = jobQueue.front();
        jobQueue.pop();
        cout << "Printing job ID " << job.jobId << ": " << job.documentName << "\n";
    }

    void viewNextJob() {
        if (jobQueue.empty()) {
            cout << "No jobs in the queue.\n";
            return;
        }
        PrintJob job = jobQueue.front();
        cout << "Next job - ID: " << job.jobId << ", Document: " << job.documentName << "\n";
    }

    void displayAllJobs() {
        if (jobQueue.empty()) {
            cout << "No jobs in the queue.\n";
            return;
        }

        queue<PrintJob> tempQueue = jobQueue;
        cout << "Jobs in the queue:\n";
        while (!tempQueue.empty()) {
            PrintJob job = tempQueue.front();
            cout << "ID: " << job.jobId << ", Document: " << job.documentName << "\n";
            tempQueue.pop();
        }
    }
};

int main() {
    PrintQueue printer;
    int choice;
    string docName;

    do {
        cout << "\n--- Print Queue Menu ---\n";
        cout << "1. Add Print Job\n";
        cout << "2. Process Next Job\n";
        cout << "3. View Next Job\n";
        cout << "4. Display All Jobs\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(); // flush newline

        switch (choice) {
            case 1:
                cout << "Enter document name: ";
                getline(cin, docName);
                printer.addJob(docName);
                break;
            case 2:
                printer.processJob();
                break;
            case 3:
                printer.viewNextJob();
                break;
            case 4:
                printer.displayAllJobs();
                break;
            case 5:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice. Try again.\n";
        }

    } while (choice != 5);

    return 0;
}
```
Output:

Result:

Manage product inventory using a vector and sort products by price.

Exp.No:7 d	**Product Inventory using a vector**
Date:	

Aim:
To develop a C++ program that manages product inventory using a vector and allows sorting of products by price, demonstrating the use of STL and sorting mechanisms.

Algorithm:

1. Define a Product structure with:
   - Product ID
   - Name
   - Price
2. Use a vector<Product> to store the inventory.
3. Implement functions to:
   - Add a product
   - Display all products
   - Sort products by price (ascending)
4. Use std::sort with a custom comparator to sort by price.

Program

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

class Product {
public:
   int id;
   string name;
   float price;

   Product(int id, string name, float price) {
      this->id = id;
      this->name = name;
      this->price = price;
   }

   void display() const {
      cout << left << setw(10) << id
           << setw(20) << name
           << fixed << setprecision(2) << price << "\n";
   }
};
```

```
class Inventory {
private:
    vector<Product> products;

public:
    void addProduct(int id, const string& name, float price) {
        products.emplace_back(id, name, price);
        cout << "Product added successfully.\n";
    }

    void displayAll() const {
        if (products.empty()) {
            cout << "No products in inventory.\n";
            return;
        }
        cout << left << setw(10) << "ID"
             << setw(20) << "Name"
             << "Price\n";
        cout << "----------------------------------------\n";
        for (const auto& p : products) {
            p.display();
        }
    }

    void sortByPrice() {
        sort(products.begin(), products.end(), [](const Product& a, const Product& b) {
            return a.price < b.price;
        });
        cout << "Products sorted by price (ascending).\n";
    }

    void searchById(int id) const {
        for (const auto& p : products) {
            if (p.id == id) {
                cout << "Product found:\n";
                cout << "ID: " << p.id << ", Name: " << p.name << ", Price: " << p.price << "\n";
                return;
            }
        }
        cout << "Product with ID " << id << " not found.\n";
    }
};

int main() {
    Inventory inventory;
```

```
int choice, id;
string name;
float price;

do {
    cout << "\n--- Inventory Management Menu ---\n";
    cout << "1. Add Product\n";
    cout << "2. Display All Products\n";
    cout << "3. Sort Products by Price\n";
    cout << "4. Search Product by ID\n";
    cout << "5. Exit\n";
    cout << "Enter your choice: ";
    cin >> choice;
    cin.ignore();  // flush newline

    switch (choice) {
        case 1:
            cout << "Enter Product ID: ";
            cin >> id;
            cin.ignore();
            cout << "Enter Product Name: ";
            getline(cin, name);
            cout << "Enter Product Price: ";
            cin >> price;
            inventory.addProduct(id, name, price);
            break;
        case 2:
            inventory.displayAll();
            break;
        case 3:
            inventory.sortByPrice();
            break;
        case 4:
            cout << "Enter Product ID to search: ";
            cin >> id;
            inventory.searchById(id);
            break;
        case 5:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice. Try again.\n";
    }
} while (choice != 5);

return 0;
```

```
}
```

Output:

Result:

Implement a contact list with a map and search functionality

Exp.No:7 e	**Contact list with a map**
Date:	

Aim:

To develop a C++ program that implements a contact list using std::map, where contacts can be stored, displayed, and searched by name efficiently.

Algorithm:

1. Use std::map<string, string> to store:
   - **Key:** Contact name
   - **Value:** Phone number
2. Implement functions to:
   - Add a new contact
   - Display all contacts
   - Search for a contact by name
3. Use the map’s efficient lookup capabilities (find()) for searching.

Program

```
#include <iostream>
#include <map>
#include <string>

class ContactList {
private:
    std::map<std::string, std::string> contacts;

public:
    void addContact(const std::string& name, const std::string& phone) {
        contacts[name] = phone;
        std::cout << "Contact added successfully.\n";
    }

    void searchContact(const std::string& name) const {
        auto it = contacts.find(name);
        if (it != contacts.end()) {
            std::cout << "Contact found:\n";
            std::cout << "Name: " << it->first << "\n";
            std::cout << "Phone: " << it->second << "\n";
        } else {
            std::cout << "Contact not found.\n";
        }
    }

    void displayContacts() const {
        if (contacts.empty()) {
            std::cout << "Contact list is empty.\n";
            return;
        }
        std::cout << "---- Contact List ----\n";
        for (const auto& entry : contacts) {
            std::cout << "Name: " << entry.first << " | Phone: " << entry.second << "\n";
        }
    }
};

int main() {
    ContactList cl;
    int choice;
    std::string name, phone;

    do {
        std::cout << "\n--- Contact List Menu ---\n";
        std::cout << "1. Add Contact\n";
        std::cout << "2. Search Contact\n";
        std::cout << "3. Display All Contacts\n";
        std::cout << "4. Exit\n";
        std::cout << "Enter your choice: ";
        std::cin >> choice;
        std::cin.ignore(); // To ignore leftover newline

        switch (choice) {
            case 1:
                std::cout << "Enter Name: ";
                std::getline(std::cin, name);
                std::cout << "Enter Phone Number: ";
                std::getline(std::cin, phone);
                cl.addContact(name, phone);
                break;
            case 2:
                std::cout << "Enter Name to Search: ";
                std::getline(std::cin, name);
                cl.searchContact(name);
                break;
            case 3:
                cl.displayContacts();
                break;
            case 4:
                std::cout << "Exiting program.\n";
                break;
            default:
                std::cout << "Invalid choice. Try again.\n";
        }

    } while (choice != 4);

    return 0;
}
```

Output:

Result:

Create a program that finds unique elements in a list using sets.

Exp.No:7 f	**Find a unique elements in a list**
Date:	

Aim:
To develop a C++ program that uses the std::set container to extract and display unique elements from a list of integers, eliminating duplicates automatically.

Algorithm:

1. Take input from the user for the number of elements.
2. Read all the elements into a vector or directly into a set.
3. Insert each element into a std::set (automatically removes duplicates).
4. Display the contents of the set — these are the **unique elements**.

Program

```
#include <iostream>
#include <vector>
#include <set>

class UniqueFinder {
private:
    std::vector<int> numbers;

public:

    void inputNumbers() {
        int n, value;
        std::cout << "Enter number of elements: ";
        std::cin >> n;

        std::cout << "Enter " << n << " elements:\n";
        for (int i = 0; i < n; ++i) {
            std::cin >> value;
            numbers.push_back(value);
        }
    }

    void displayUniqueElements() const {
        std::set<int> uniqueSet(numbers.begin(), numbers.end());

        std::cout << "Unique elements are:\n";
        for (int val : uniqueSet) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    UniqueFinder uf;
    uf.inputNumbers();
    uf.displayUniqueElements();

    return 0;
}
```

Output:

Result:

Implement a priority-based task scheduler using priority queues.

Exp.No:7 g	**Task Scheduler using Priority queue**
Date:	

Aim:
To develop a priority-based task scheduler where tasks with higher priority are executed first, using std::priority_queue. This program will simulate task scheduling where each task has a priority level.

Algorithm:

1. Define a Task structure with:
    - taskId (Unique ID for the task)
    - taskDescription (Description of the task)
    - priority (Priority of the task, higher number means higher priority)
2. Use std::priority_queue with a custom comparator to schedule tasks by priority.
3. Insert tasks into the priority queue.
4. The task with the highest priority (largest priority value) will be processed first.
5. Implement functions to:
    - Add a task
    - Display all tasks in priority order
    - Process and remove the highest-priority task.

Program
```
#include <iostream>
#include <queue>
#include <vector>
#include <string>

class Task {
public:
   std::string name;
   int priority;

   Task(std::string name, int priority) : name(name), priority(priority) {}

   bool operator<(const Task& other) const {
      return priority > other.priority; // Reverse for min-priority queue
   }
};

class TaskScheduler {
private:
   std::priority_queue<Task> taskQueue;

public:
    void addTask(const std::string& name, int priority) {
        taskQueue.push(Task(name, priority));
        std::cout << "Task \"" << name << "\" added with priority " << priority << ".\n";
    }

    void executeTask() {
        if (taskQueue.empty()) {
            std::cout << "No tasks to execute.\n";
            return;
        }
        Task top = taskQueue.top();
        std::cout << "Executing Task: " << top.name << " [Priority: " << top.priority << "]\n";
        taskQueue.pop();
    }

    void displayTasks() {
        if (taskQueue.empty()) {
            std::cout << "No tasks scheduled.\n";
            return;
        }

        std::priority_queue<Task> temp = taskQueue;
        std::cout << "--- Scheduled Tasks (by priority) ---\n";
        while (!temp.empty()) {
            Task t = temp.top();
            std::cout << "Task: " << t.name << " | Priority: " << t.priority << "\n";
            temp.pop();
        }
    }
};

int main() {
    TaskScheduler scheduler;
    int choice;
    std::string name;
    int priority;

    do {
        std::cout << "\n--- Priority Task Scheduler ---\n";
        std::cout << "1. Add Task\n";
        std::cout << "2. Execute Task\n";
        std::cout << "3. Display Tasks\n";
        std::cout << "4. Exit\n";
        std::cout << "Enter choice: ";
        std::cin >> choice;
        std::cin.ignore();
        switch (choice) {
            case 1:
                std::cout << "Enter task name: ";
                std::getline(std::cin, name);
                std::cout << "Enter priority (lower number = higher priority): ";
                std::cin >> priority;
                scheduler.addTask(name, priority);
                break;
            case 2:
                scheduler.executeTask();
                break;
            case 3:
                scheduler.displayTasks();
                break;
            case 4:
                std::cout << "Exiting scheduler.\n";
                break;
            default:
                std::cout << "Invalid choice. Try again.\n";
        }

    } while (choice != 4);

    return 0;
}
```

Output:

Result:

Develop a dictionary application using maps.

Exp.No:7 h	**Dictionary application using maps.**
Date:	

Aim:
To develop a dictionary application in C++ using std::map, where the user can perform operations such as adding words with definitions, searching for definitions, and displaying all words in the dictionary.

Algorithm:

1. Define a map<string, string> where:
   - **Key:** Word (string)
   - **Value:** Definition (string)
2. Implement functions to:
   - **Add a word with definition**
   - **Search for a word's definition**
   - **Display all words and their definitions**
3. Use the std::map container to manage the dictionary efficiently, as it provides sorted order and efficient lookups.

Program

```
#include <iostream>
#include <map>
#include <string>

class Dictionary {
private:
    std::map<std::string, std::string> wordMap;

public:
    void addWord(const std::string& word, const std::string& meaning) {
        if (wordMap.find(word) != wordMap.end()) {
            std::cout << "Word already exists. Use update option to change meaning.\n";
            return;
        }
        wordMap[word] = meaning;
        std::cout << "Word added successfully.\n";
    }

    void searchWord(const std::string& word) const {
        auto it = wordMap.find(word);
        if (it != wordMap.end()) {
            std::cout << "Meaning of '" << word << "': " << it->second << "\n";
        } else {
            std::cout << "Word not found in the dictionary.\n";
        }
    }


    void updateMeaning(const std::string& word, const std::string& newMeaning) {
        auto it = wordMap.find(word);
        if (it != wordMap.end()) {
            wordMap[word] = newMeaning;
            std::cout << "Meaning updated successfully.\n";
        } else {
            std::cout << "Word not found. Add it first.\n";
        }
    }

    void displayAllWords() const {
        if (wordMap.empty()) {
            std::cout << "Dictionary is empty.\n";
            return;
        }

        std::cout << "\n--- Dictionary Entries ---\n";
        for (const auto& entry : wordMap) {
            std::cout << entry.first << ": " << entry.second << "\n";
        }
    }
};

int main() {
    Dictionary dict;
    int choice;
    std::string word, meaning;

    do {
        std::cout << "\n--- Dictionary Menu ---\n";
        std::cout << "1. Add Word\n";
        std::cout << "2. Search Word\n";
        std::cout << "3. Update Meaning\n";
        std::cout << "4. Display All Words\n";
        std::cout << "5. Exit\n";
        std::cout << "Enter your choice: ";
        std::cin >> choice;
        std::cin.ignore(); // Flush newline

        switch (choice) {
            case 1:
                std::cout << "Enter word: ";
                std::getline(std::cin, word);
                std::cout << "Enter meaning: ";
                std::getline(std::cin, meaning);
                dict.addWord(word, meaning);
                break;

            case 2:
                std::cout << "Enter word to search: ";
                std::getline(std::cin, word);
                dict.searchWord(word);
                break;

            case 3:
                std::cout << "Enter word to update: ";
                std::getline(std::cin, word);
                std::cout << "Enter new meaning: ";
                std::getline(std::cin, meaning);
                dict.updateMeaning(word, meaning);
                break;

            case 4:
                dict.displayAllWords();
                break;

            case 5:
                std::cout << "Exiting dictionary.\n";
                break;

            default:
                std::cout << "Invalid choice. Try again.\n";
        }

    } while (choice != 5);

    return 0;
}
```

Output:

Result:

Implement a program that merges two sorted lists using STL algorithms.

Exp.No:7 i	**Merge the lists using STL algorithms**
Date:	

Aim:
To implement a program that merges two sorted lists into a single sorted list using the STL algorithms in C++. Specifically, we will use std::merge to merge two sorted containers and output the result.

Algorithm:

1. **Input two sorted lists**:
   - Take two sorted lists of integers from the user.
2. **Merge the lists**:
   - Use the std::merge algorithm from the <algorithm> library to merge the two sorted lists into a single sorted list.
3. **Output the merged list**:
   - Display the resulting merged list.
4. **Assumptions**:
   - Both input lists are already sorted.

Program

```
#include <iostream>
#include <vector>
#include <algorithm>

class SortedListMerger {
private:
    std::vector<int> list1, list2, mergedList;

public:
    void inputList(std::vector<int>& list, const std::string& listName) {
        int n, value;
        std::cout << "Enter number of elements for " << listName << ": ";
        std::cin >> n;

        std::cout << "Enter " << n << " sorted elements:\n";
        for (int i = 0; i < n; ++i) {
            std::cin >> value;
            list.push_back(value);
        }

        std::sort(list.begin(), list.end());
    }

    void inputLists() {
        inputList(list1, "List 1");
        inputList(list2, "List 2");
    }

    void mergeLists() {
        mergedList.clear(); // Clear in case of reuse
        std::merge(list1.begin(), list1.end(), list2.begin(), list2.end(),
std::back_inserter(mergedList));
    }

    void displayMergedList() const {
        std::cout << "Merged Sorted List:\n";
        for (int val : mergedList) {
            std::cout << val << " ";
        }
        std::cout << "\n";
    }
};

int main() {
    SortedListMerger merger;
    merger.inputLists();
    merger.mergeLists();
    merger.displayMergedList();

    return 0;
}
```

Output:

Result:

Develop a movie recommendation system using multimap.

Exp.No:7 j	**Movie recommendation system using multimap.**
Date:	

Aim:

To develop a movie recommendation system using the std::multimap in C++ where users can search for movies based on ratings, genres, or any other criteria stored in a multimap. The system will recommend movies by storing movie titles and their ratings or genres and will allow users to search for movies by these attributes.

Algorithm:

1. **Input movie data**:
   - Use a multimap to store movie data where the key is the rating (or genre) and the value is the movie title.
   - Allow the user to add movies to the system with their ratings or genres.
2. **Movie Recommendation**:
   - Based on user input, suggest movies either by rating or by genre.
   - The program will allow the user to search for movies with a specific rating or genre.
3. **Display the recommendations**:
   - List all movies that match the rating or genre criteria input by the user.

Program

```
#include <iostream>
#include <map>
#include <string>

class MovieRecommender {
private:
    std::multimap<std::string, std::string> movieMap; // genre → movie title

public:
    void addMovie(const std::string& genre, const std::string& movie) {
        movieMap.insert({genre, movie});
        std::cout << "Movie \"" << movie << "\" added under genre \"" << genre << "\".\n";
    }

    void recommendByGenre(const std::string& genre) const {
        auto range = movieMap.equal_range(genre);
        if (range.first == range.second) {
            std::cout << "No movies found for genre: " << genre << "\n";
            return;
        }

        std::cout << "Movies under genre \"" << genre << "\":\n";
        for (auto it = range.first; it != range.second; ++it) {
            std::cout << "- " << it->second << "\n";
        }
    }

    void displayAll() const {
        if (movieMap.empty()) {
            std::cout << "No movies available.\n";
            return;
        }

        std::cout << "\n--- Movie List by Genre ---\n";
        for (const auto& entry : movieMap) {
            std::cout << "Genre: " << entry.first << " | Movie: " << entry.second << "\n";
        }
    }
};

int main() {
    MovieRecommender recommender;
    int choice;
    std::string genre, movie;

    do {
        std::cout << "\n--- Movie Recommendation System ---\n";
        std::cout << "1. Add Movie\n";
        std::cout << "2. Recommend by Genre\n";
        std::cout << "3. Display All Movies\n";
        std::cout << "4. Exit\n";
        std::cout << "Enter choice: ";
        std::cin >> choice;
        std::cin.ignore(); // clear newline

        switch (choice) {
            case 1:
                std::cout << "Enter genre: ";
                std::getline(std::cin, genre);
                std::cout << "Enter movie title: ";
                std::getline(std::cin, movie);
                recommender.addMovie(genre, movie);
                break;

            case 2:
                std::cout << "Enter genre to recommend: ";
                std::getline(std::cin, genre);
                recommender.recommendByGenre(genre);
                break;

            case 3:
                recommender.displayAll();
                break;

            case 4:
                std::cout << "Exiting system.\n";
                break;

            default:
                std::cout << "Invalid choice. Try again.\n";
        }

    } while (choice != 4);

    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Implement a program to store and retrieve student details from a file.

Exp.No:8 a	**File Handling – R/W Operation**
Date:	

Aim:
The goal of this program is to store and retrieve student details (such as name, roll number, and grade) from a file. The program will use file handling in C++ to save student information to a file and read the stored information back into the program when required.

Algorithm:

1. **Define a Student class**:
   - This class will contain member variables for storing student details like name, rollNumber, and grade.
2. **Input student details**:
   - Prompt the user to enter student details such as name, rollNumber, and grade.
3. **Store details in a file**:
   - Use file handling (ofstream) to write the student details to a file.
4. **Retrieve student details from file**:
   - Use ifstream to read the student details from the file and display them.
5. **Menu for actions**:
   - Provide a menu to allow the user to either add new student details or retrieve and display existing student details from the file.

Program

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

class Student {
public:
    string name;
    int rollNumber;
    char grade;

    void inputDetails() {
        cout << "Enter student name: ";
        cin.ignore();  // Ignore any remaining characters in the input buffer
        getline(cin, name);
        cout << "Enter roll number: ";
        cin >> rollNumber;
        cout << "Enter grade: ";
        cin >> grade;
    }
    void saveToFile(ofstream &outFile) {
        outFile << name << endl;
        outFile << rollNumber << endl;
        outFile << grade << endl;
    }

    static void readFromFile(ifstream &inFile) {
        string name;
        int rollNumber;
        char grade;

        while (getline(inFile, name)) {
            inFile >> rollNumber;
            inFile >> grade;
            inFile.ignore();  // Ignore newline character after reading grade

            cout << "\nStudent Name: " << name << endl;
            cout << "Roll Number: " << rollNumber << endl;
            cout << "Grade: " << grade << endl;
            cout << "--------------------------------" << endl;
        }
    }
};

int main() {
    int choice;
    Student student;
    ofstream outFile("student_details.txt", ios::app);  // Open file in append mode
    ifstream inFile("student_details.txt");

    if (!outFile) {
        cout << "Error opening file for writing!" << endl;
        return 1;
    }

    do {
        cout << "\n--- Student Details Management ---\n";
        cout << "1. Add Student Details\n";
        cout << "2. View Student Details\n";
        cout << "3. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                student.inputDetails();  // Get student details from user
                student.saveToFile(outFile);  // Save student details to file
                cout << "Student details saved successfully!" << endl;
                break;

            case 2:
                if (inFile) {
                    Student::readFromFile(inFile);  // Display student details from file
                } else {
                    cout << "No student details found in file!" << endl;
                }
                break;

            case 3:
                cout << "Exiting the program." << endl;
                break;

            default:
                cout << "Invalid choice. Please try again." << endl;
        }
    } while (choice != 3);

    outFile.close();
    inFile.close();
    return 0;
}
```

Output:

Result:

Implement a program to read a text file and count the number of words

Exp.No:8 b	**File handling – Count Words**
Date:	

Aim:
The goal of this program is to read a text file and count the number of words in it. The program will use file handling and string manipulation techniques to read the contents of the file and count the words. The definition of a word in this context is a sequence of characters separated by spaces or punctuation.

Algorithm:
1. **Open the file**:
   - Use ifstream to open and read the text file.
2. **Read the file word by word**:
   - Read each word from the file using a loop.
   - Count each word encountered in the text.
3. **Word counting logic**:
   - A word is defined as a sequence of characters separated by spaces, newlines, or punctuation.
4. **Display the word count**:
   - After reading the file, output the total number of words counted

Program
```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;

int countWordsInFile(const string& fileName) {
    ifstream file(fileName); // Open the file for reading
    if (!file) {
        cerr << "Error opening file!" << endl;
        return -1; // Return -1 if file can't be opened
    }

    string line;
    int wordCount = 0;

    while (getline(file, line)) {
        stringstream ss(line);
        string word;

        while (ss >> word) {
            wordCount++;
        }
    }

    file.close();
    return wordCount;
}

int main() {
    string fileName;

    cout << "Enter the name of the text file to read: ";
    cin >> fileName;

    int wordCount = countWordsInFile(fileName);

    if (wordCount != -1) {
        cout << "The total number of words in the file '" << fileName << "' is: " << wordCount <<
endl;
    }

    return 0;
}
```

Output:

Result:

Write a program to copy the contents of one file to another

Exp.No:8 c	**File handling – Copy Content**
Date:	

Aim:
The goal of this program is to copy the contents of one file to another. It will use file handling in C++ to read data from a source file and write it to a destination file.

Algorithm:

1. **Open the source file for reading**:
   - Use ifstream to open the source file in read mode.
2. **Open the destination file for writing**:
   - Use ofstream to open the destination file in write mode (creating the file if it doesn't exist).
3. **Read and write data**:
   - Read the content of the source file in chunks (either line by line or character by character) and write it to the destination file.
4. **Close the files**:
   - After the copying operation is complete, close both the source and destination files.
5. **Error handling**:
   - If the source or destination file cannot be opened, print an error message and exit.

Program

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

void copyFileContents(const string &sourceFileName, const string &destFileName) {
    ifstream sourceFile(sourceFileName);
    if (!sourceFile) {
        cerr << "Error opening source file!" << endl;
        return;
    }

    ofstream destFile(destFileName);
    if (!destFile) {
        cerr << "Error opening destination file!" << endl;
        return;
    }

    string line;

    while (getline(sourceFile, line)) {
        destFile << line << endl;  // Write each line to the destination file
    }

    if (sourceFile.eof()) {
        cout << "File contents copied successfully!" << endl;
    }

    sourceFile.close();
    destFile.close();
}

int main() {
    string sourceFileName, destFileName;

    cout << "Enter the source file name: ";
    cin >> sourceFileName;
    cout << "Enter the destination file name: ";
    cin >> destFileName;

    copyFileContents(sourceFileName, destFileName);

    return 0;
}
```

Output:

Result:

Develop a program to append data to an existing file

Exp.No:8 d	File Handling – Data Appending
Date:	

Aim:
The goal of this program is to append data to an existing file. If the file doesn't exist, the program should create the file and then append the data to it. The program will use file handling to open the file in append mode and write the data at the end of the file without overwriting the existing content.

Algorithm:

1. **Open the file in append mode**:
    - Use ofstream to open the file in **append mode** (ios::app).
2. **Input data from the user**:
    - Prompt the user to enter the data that needs to be appended.
3. **Write data to the file**:
    - Use the << operator to append the entered data at the end of the file.
4. **Close the file**:
    - After writing, close the file to save the changes.
5. **Error handling**:
    - If the file cannot be opened, display an error message.

Program

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

void appendToFile(const string &fileName) {
    // Open the file in append mode, if it doesn't exist, it will be created
    ofstream file(fileName, ios::app);

    if (!file) {
        cerr << "Error opening file!" << endl;
        return;
    }

    string data;
    cout << "Enter the data to append to the file: ";
    cin.ignore();  // Ignore the leftover newline character from previous input
    getline(cin, data);  // Read the data from user input

    file << data << endl;

    file.close();
    cout << "Data appended successfully!" << endl;
}

int main() {
    string fileName;

    cout << "Enter the file name to append data to: ";
    cin >> fileName;

    appendToFile(fileName);

    return 0;
}
```

Output:

Result:

Create a simple log system that records user activities in a text file.

Exp.No:8 e	File Handling – Simple Log System
Date:	

Aim:
The aim of this program is to create a simple log system that records user activities in a text file. Each time a user performs an action, such as entering a command, the action will be logged with a timestamp to a file. This allows tracking and reviewing of user activities in an application.

Algorithm:

1. **Open the log file**:
    - Use ofstream to open a log file in append mode (ios::app). This ensures that each new log entry is added at the end of the file.
2. **Record user activity**:
    - Ask the user to input their activity or action.
    - Get the current time and date using the system's clock to include a timestamp with each log entry.
3. **Write the log entry**:
    - Format the log entry with the timestamp and the user activity description, and write it to the log file.
4. **Close the log file**:
    - After writing the log entry, close the file to ensure the changes are saved.
5. **Repeat logging**:
    - Allow the user to log multiple activities, or exit the program to stop logging.
6. **Error handling**:
    - Handle errors in case the file cannot be opened.

Program

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

void logActivity(const string &activity, const string &logFileName) {
   ofstream logFile(logFileName, ios::app);

   if (!logFile) {
      cerr << "Error opening log file!" << endl;
      return;
   }

   time_t now = time(0);
   char* dt = ctime(&now); // Get current date and time as a string
   dt[strlen(dt) - 1] = '\0'; // Remove the newline character at the end
```

```
    logFile << "[" << dt << "] " << activity << endl;

    logFile.close();
    cout << "Activity logged successfully!" << endl;
}

int main() {
    string logFileName = "user_activity_log.txt";  // Log file name
    string activity;

    while (true) {
        cout << "Enter an activity to log (or type 'exit' to quit): ";
        getline(cin, activity);  // Get the user activity input

        if (activity == "exit") {
            break;
        }

        logActivity(activity, logFileName);
    }

    cout << "Exiting the log system." << endl;
    return 0;
}
```

Output:

Result:

Develop a program to merge the contents of two files into a third file.

Exp.No:8 f	**File handling – Merge file contents**
Date:	

Aim:

To develop a C++ program that merges the contents of two files into a third file. This program reads data from two source files and appends their content into a third destination file.

Algorithm:

1. **Prompt the user** to enter the names of the two source files and the destination file.
2. **Open the first source file** in read mode using ifstream.
3. **Open the second source file** in read mode using ifstream.
4. **Open the destination file** in write mode using ofstream.
5. **Read content from the first source file** line by line and write it into the destination file.
6. **Read content from the second source file** line by line and write it into the destination file.
7. **Close all the files** after the operation is completed.
8. Handle file **opening errors** appropriately.

Program

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

void mergeFiles(const string &file1, const string &file2, const string &destFile) {
    ifstream input1(file1);
    ifstream input2(file2);
    ofstream output(destFile);

    if (!input1.is_open() || !input2.is_open() || !output.is_open()) {
        cerr << "Error: Unable to open one or more files!" << endl;
        return;
    }

    string line;


    while (getline(input1, line)) {
        output << line << endl;
    }

    while (getline(input2, line)) {
        output << line << endl;
    }

    cout << "Files merged successfully into '" << destFile << "'." << endl;

    input1.close();
    input2.close();
    output.close();
}

int main() {
    string file1, file2, destFile;

    cout << "Enter the first source file name: ";
    cin >> file1;
    cout << "Enter the second source file name: ";
    cin >> file2;
    cout << "Enter the destination file name: ";
    cin >> destFile;

    mergeFiles(file1, file2, destFile);

    return 0;
}
```

Output:

Result:

Write a program to store employee records in a binary file.

Exp.No:8 g	**File Handling – Data Storage**
Date:	

Aim:
To write a program in C++ that stores employee records in a binary file using structures and file handling. This helps in efficient storage and retrieval of data in a non-textual (binary) format.

Algorithm:

1. **Define a structure** Employee with fields like ID, name, age, and salary.
2. **Open a binary file** in output mode (ios::binary | ios::app) using ofstream.
3. **Take employee details as input** from the user.
4. **Write the data to the binary file** using the write() function.
5. Ask if the user wants to **add more records**.
6. **Close the file** after storing all records.
7. Optionally, display the data by **reading it back** from the file.

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

struct Employee {
   int id;
   char name[50];
   int age;
   float salary;
};

void writeEmployeeData(const char* filename) {
   ofstream outFile(filename, ios::binary | ios::app);

   if (!outFile) {
      cerr << "Error opening file for writing!" << endl;
      return;
   }

   Employee emp;
   char choice;

   do {
      cout << "\nEnter Employee ID: ";
      cin >> emp.id;
      cin.ignore();
      cout << "Enter Employee Name: ";
```

```
        cin.getline(emp.name, 50);
        cout << "Enter Age: ";
        cin >> emp.age;
        cout << "Enter Salary: ";
        cin >> emp.salary;

        outFile.write(reinterpret_cast<char*>(&emp), sizeof(emp));
        cout << "Record added successfully.\n";

       t
        cout << "Do you want to add another record? (y/n): ";
        cin >> choice;
    } while (choice == 'y' || choice == 'Y');

    outFile.close();
}

void displayEmployeeData(const char* filename) {
    ifstream inFile(filename, ios::binary);

    if (!inFile) {
        cerr << "Error opening file for reading!" << endl;
        return;    }

    Employee emp;
    cout << "\nStored Employee Records:\n";
    cout << "------------------------------------------\n";
    while (inFile.read(reinterpret_cast<char*>(&emp), sizeof(emp))) {
        cout << "ID: " << emp.id << "\n";
        cout << "Name: " << emp.name << "\n";
        cout << "Age: " << emp.age << "\n";
        cout << "Salary: $" << emp.salary << "\n";
        cout << "------------------------------------------\n";
    }

    inFile.close();
}

int main() {
    const char* filename = "employees.dat";

    writeEmployeeData(filename);

    displayEmployeeData(filename);

    return 0;
```

```
}
```
Output

Result

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		

Basic Multithreading: Create a simple program that runs two threads concurrently, one printing numbers and another printing alphabets

Exp.No:9 a	**Multithreading**
Date:	

Aim:

To create a C++ program that demonstrates **basic multithreading** by running **two threads concurrently**:

- One thread prints numbers (1-10)
- Another thread prints alphabets (A-J)

Algorithm:

1. Include the necessary headers: <iostream>, <thread>.
2. Define two separate functions:
    - printNumbers() – prints numbers 1 to 10.
    - printAlphabets() – prints letters A to J.
3. In the main() function:
    - Create two threads using std::thread.
    - Start them with the two defined functions.
    - Use join() to wait for both threads to complete.
4. Compile with a compiler that supports C++11 or later.

Program

```
#include <iostream>
#include <thread>
#include <chrono>

using namespace std;

void printNumbers() {
   for (int i = 1; i <= 10; ++i) {
      cout << "Number: " << i << endl;
      this_thread::sleep_for(chrono::milliseconds(100)); // Optional delay
   }
}

void printAlphabets() {
   for (char c = 'A'; c <= 'J'; ++c) {
      cout << "Alphabet: " << c << endl;
      this_thread::sleep_for(chrono::milliseconds(100)); // Optional delay
   }
}
```

```
int main() {
   thread t1(printNumbers);
   thread t2(printAlphabets);

   t1.join();
   t2.join();

   cout << "Both threads have completed execution." << endl;

   return 0;
}
```

Output:

Result:

Thread Synchronization with Mutex: Implement a program where multiple threads update a shared counter, ensuring synchronization using a mutex.

Exp.No:9 b	**Thread Synchronization in mutex**
Date:	

Aim:

To implement a C++ program where multiple threads update a shared counter, ensuring synchronized access using a mutex to prevent race conditions.

Algorithm:

1. Include the necessary headers: <iostream>, <thread>, <mutex>.
2. Declare a shared counter and a std::mutex object.
3. Define a function incrementCounter() that:
    - Locks the mutex.
    - Increments the shared counter.
    - Unlocks the mutex.
4. Create multiple threads that run incrementCounter() multiple times.
5. Join all threads and display the final value of the counter.

Program

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

int counter = 0;
 mutex mtx;

void incrementCounter(int times) {
    for (int i = 0; i < times; ++i) {
        mtx.lock();
        ++counter;
        mtx.unlock();
    }
}

int main() {
    int numThreads = 5;
    int incrementsPerThread = 1000;

    thread threads[numThreads];

    for (int i = 0; i < numThreads; ++i) {
        threads[i] = thread(incrementCounter, incrementsPerThread);
    }

    for (int i = 0; i < numThreads; ++i) {
        threads[i].join();
    }

    cout << "Final value of counter: " << counter << endl;
    return 0;
}
```

Output:

Result:

Asynchronous Function Execution: Write a program that demonstrates the use of std::async to execute a function asynchronously and retrieve the result.

Exp.No:9 c	**Asynchronous Function Execution**
Date:	

Aim:
To demonstrate asynchronous function execution in C++ using std::async, where a function runs in the background and returns a result that can be accessed later

Algorithm:
1. Include necessary headers: <iostream>, <future>, <chrono>.
2. Define a function (e.g., longComputation()) that performs some operation and returns a result.
3. In main():
   - Call std::async() with the function to start execution asynchronously.
   - Perform other tasks while the function executes.
   - Retrieve the result using .get() on the std::future object.
4. Display the final result.

Program
```
#include <iostream>
#include <future>
#include <chrono>

using namespace std;

int longComputation() {
    cout << "Long computation started..." << endl;
    this_thread::sleep_for(chrono::seconds(3));  // Simulate delay
    cout << "Long computation finished!" << endl;
    return 42;
}

int main() {

    future<int> result = async(launch::async, longComputation);

    cout << "Main thread is doing some work..." << endl;
    this_thread::sleep_for(chrono::seconds(1));
    cout << "Still waiting for result..." << endl;

    int value = result.get();
    cout << "Result of longComputation: " << value << endl;
    return 0;
}
```

Output:

Result:

Thread-based Factorial Calculation: Compute the factorial of a number using a separate thread.

Exp.No:9 d	**Thread-based Factorial Calculation**
Date:	

Aim:
To implement a C++ program that calculates the factorial of a number using a separate thread, showcasing how computational tasks can be offloaded to another thread for concurrent execution.

Algorithm:

1. Include necessary headers: <iostream>, <thread>.
2. Define a function calculateFactorial(int n):
   - Use a loop to compute factorial.
   - Store the result in a reference variable passed from main.
3. In main():
   - Take user input for the number.
   - Create a std::thread to execute the factorial function.
   - Wait for the thread to complete using join().
   - Display the result.

Program

```
#include <iostream>
#include <thread>

using namespace std;

void calculateFactorial(int n, unsigned long long &result) {
   result = 1;
   for (int i = 2; i <= n; ++i) {
      result *= i;
   }
}

int main() {
   int number;
   unsigned long long factorial = 1;
   cout << "Enter a number to calculate its factorial: ";
   cin >> number;
   thread t(calculateFactorial, number, ref(factorial));
   t.join();
   cout << "Factorial of " << number << " is: " << factorial << endl;
   return 0;
}
```

Output:

Result:

Simple Timer Using Threads: Create a program where a separate thread functions as a timer, printing elapsed time every second.

Exp.No:9 e	**Simple Timer Using Threads**
Date:	

Aim:
To create a C++ program using multithreading where a separate thread acts as a timer, printing the elapsed time every second.

Algorithm:

1. Include necessary headers: <iostream>, <thread>, <chrono>.
2. Define a timer function:
   - Loop with a sleep of 1 second.
   - Print the elapsed seconds.
   - Stop after a defined limit (optional).
3. In main():
   - Start the timer thread.
   - Let the main thread perform other tasks or just wait.
   - Join the timer thread if needed.

Program

```
#include <iostream>
#include <thread>
#include <chrono>

using namespace std;

void timer(int duration) {
    for (int i = 1; i <= duration; ++i) {
        this_thread::sleep_for(chrono::seconds(1));
        cout << "Elapsed time: " << i << " second(s)" << endl;
    }
}

int main() {
    int seconds;
    cout << "Enter timer duration in seconds: ";
    cin >> seconds;
    thread timerThread(timer, seconds);
    timerThread.join();
    cout << "Timer ended." << endl;
    return 0;
}
```

Output:

Result:

Object Oriented Programming using C++ Laboratory Evaluation		
Parameters	**Max marks**	**Marks obtained**
Pre Lab Quiz	20	
Program	25	
Output	20	
Viva	10	
Total	75	
Signature of the Faculty		