***Assignment Day 1***

1. **Sum of Natural Numbers up to N Sol-**

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

int n, sum = 0;

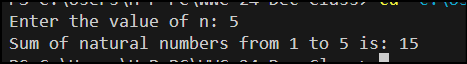
cout << "Enter the value of n: "; cin >> n;

sum = (n\*(n+1)/2);

cout << "Sum of natural numbers from 1 to " << n << " is: " << sum << endl; return 0;

}

**Output –**

****

1. **Check if a Number is Prime Sol-**

#include <iostream> #include<math.h> using namespace std; bool isPrime(int num) {

if (num <= 1) return false;

for (int i = 2; i <= sqrt(num); i++) { if (num % i == 0)

return false;

}

return true;

}

int main() { int num;

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

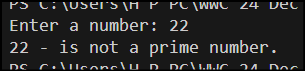
if (isPrime(num))

cout << num << " - is a prime number."; else

cout << num << " - is not a prime number."; return 0;

}

**Output-**

****

1. **Print Odd Numbers up to N Sol-**

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

int N, i;

cout << "Enter the value of N: "; cin >> N;

for(i=1; i<=N; i+=2) { cout << i << " ";

}

return 0;

}

**Output-**

****

1. **Sum of Odd Numbers up to N Sol-**

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

int N, i, sum = 0;

cout << "Enter the value of N: "; cin >> N;

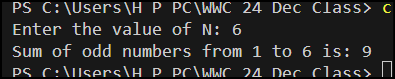
for(i=1; i<=N; i+=2) { sum += i;

}

cout << "Sum of odd numbers from 1 to " << N << " is: " << sum << endl; return 0;

}

**Output-**

****

1. **Print Multiplication Table of a Number Sol-**

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

int num, i, j;

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

for (i = 1; i <= 10; i++) { for (j = 1; j <= 10; j++) {

cout << num << " x " << j << " = " << num \* j << endl;

}

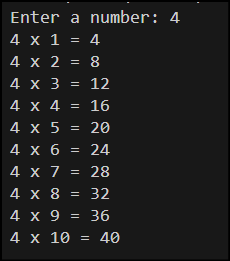
cout << endl;

}

return 0;

}

**Output –**



1. **Count Digits in a Number Sol –**

#include<iostream> using namespace std;

int countDigits(int n) { if (n == 0) return 1;

n = abs(n);

int count = 0; while (n != 0) {

n = n / 10; count++;

}

return count;

}

int main() { int num;

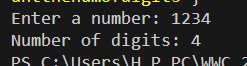
cout << "Enter a number: ";

cin >> num;

cout << "Number of digits: " << countDigits(num) << endl; return 0;

}

**Output –**

****

1. **Reverse a Number Sol –** #include<iostream> using namespace std;

int reverseNumber(int num) { int reverse = 0;

while(num != 0) {

reverse = reverse \* 10 + num % 10; num /= 10;

}

return reverse;

}

int main() { int num;

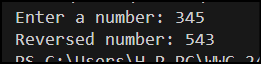
cout << "Enter a number: "; cin >> num;

cout << "Reversed number: " << reverseNumber(num) << endl;

return 0;

}

**Output –**

****

1. **Find the Largest Digit in a Number Sol –**

#include <iostream> using namespace std;

int main() {

int num, maxDigit = 0;

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

while (num != 0) {

int digit = num % 10; if (digit > maxDigit) { maxDigit = digit;

}

num /= 10;

}

cout << "The largest digit in the number is: " << maxDigit << endl;

return 0;

}

**Output –**

****

1. **Check if a Number is a Palindrome Sol-**

#include <iostream> using namespace std;

bool isPalindrome(int num) { int originalNum = num; int reverseNum = 0;

while (num != 0) {

int digit = num % 10;

reverseNum = reverseNum \* 10 + digit; num /= 10;

}

return originalNum == reverseNum;

}

int main() { int num;

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

if (isPalindrome(num)) {

cout << num << " is a palindrome." << endl;

} else {

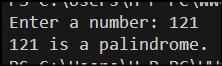
cout << num << " is not a palindrome." << endl;

}

return 0;

}

**Output –**

****

1. **Find the Sum of Digits of a Number Sol –**

#include<iostream> using namespace std; int sumOfDigits(int n) { int sum = 0;

while (n != 0) { sum += n % 10; n /= 10;

}

return sum;

}

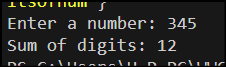
int main() { int num;

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

cout << "Sum of digits: " << sumOfDigits(num) << endl; return 0;

}

**Output-**

****

1. **Function Overloading for Calculating Area. Sol –**

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

double area(double radius) { return 3.14 \*radius \* radius;

}

double area(double base, double height) { return 0.5 \* base \* height;

}

int area(int length, int width) { return length \* width;

}

int main() {

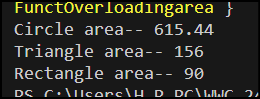
cout << "Circle area-- " << area(14) << endl;

cout << "Triangle area-- " << area(13,12) << endl; cout << "Rectangle area-- " << area(15, 6) << endl;

return 0;

}

**Output -**

****

1. **Function Overloading with Hierarchical Structure. Sol –**

#include <iostream> using namespace std; class Shape {

public:

virtual void print() = 0;

};

class Circle : public Shape { private:

int radius; public:

Circle(int r) : radius(r) {} void print() {

cout << "Circle: Radius = " << radius << endl;

}

};

class Rectangle : public Shape { private:

int length, width; public:

Rectangle(int l, int w) : length(l), width(w) {} void print() {

cout << "Rectangle: Length = " << length << ", Width = " << width <<

endl;

}

};

int main() {

Shape\* shapes[] = {new Circle(5), new Rectangle(4, 6)}; int nShapes = sizeof(shapes) / sizeof(shapes[0]);

for (int i = 0; i < nShapes; i++) { shapes[i]->print();

delete shapes[i];

}

return 0;

}

**Output -**

****

1. **Encapsulation with Employee Details Sol –**

#include <iostream>

#include <string> using namespace std;

class Employee { private:

string name; int id;

float salary;

public:

Employee(string n, int i, float s) { name = n;

id = i; setSalary(s);

}

void setSalary(float s) { if (s >= 0) {

salary = s;

} else {

cout << "Error: Salary cannot be negative." << endl;

}

}

float getSalary() { return salary;

}

void displayEmployeeDetails() {

cout << "Name: " << name << endl; cout << "ID: " << id << endl;

cout << "Salary: $" << salary << endl;

}

string getName() { return name;

}

};

int main() {

Employee emp1("John Doe", 12345, 50000.0); emp1.displayEmployeeDetails();

Employee emp2("Jane Smith", 67890, 65000.0); emp2.displayEmployeeDetails();

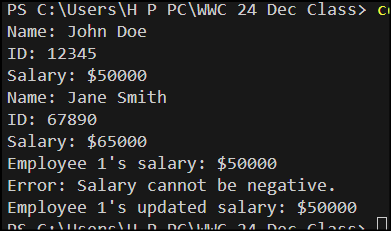
cout << "Employee 1's salary: $" << emp1.getSalary() << endl; emp1.setSalary(-10000.0);

cout << "Employee 1's updated salary: $" << emp1.getSalary() << endl;

return 0;

}

**Output –**

****

1. **Inheritance with Student and Result Classes. Sol –**

#include<iostream> using namespace std;

class Result { int marks;

public:

void setMarks(int m) { marks = m;

}

int getMarks() { return marks;

}

};

class Student { string name;

int rollNumber; Result result;

public:

void setName(string n) { name = n;

}

string getName() { return name;

}

void setRollNumber(int r) { rollNumber = r;

}

int getRollNumber() { return rollNumber;

}

void setResult(Result res) { result = res;

}

Result getResult() { return result;

}

};

int main() { Student s; Result r;

s.setName("John Doe");

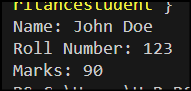
s.setRollNumber(123); r.setMarks(90); s.setResult(r);

cout << "Name: " << s.getName() << endl;

cout << "Roll Number: " << s.getRollNumber() << endl; cout << "Marks: " << s.getResult().getMarks() << endl; return 0;

}

**Output –**

****

1. **Polymorphism with Shape Area Calculation. Sol –**

#include <iostream> class Shape { public:

virtual double area() const = 0;

};

class Circle : public Shape { double radius;

public:

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

double area() const override { return 3.14\* radius \* radius;

}

void display() const {

std::cout << "Circle area: " << area() << std::endl;

}

};

class Rectangle : public Shape { double length, width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double area() const override { return length \* width;

}

void display() const {

std::cout << "Rectangle area: " << area() << std::endl;

}

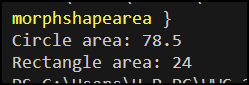
};

int main() { Circle circle(5); circle.display();

Rectangle rectangle(4, 6); rectangle.display(); return 0;

}

**Output –**

****

1. **Implementing Polymorphism for Shape Hierarchies. Sol –**

#include <iostream> #include <cmath>

using namespace std;

class Shape { public:

virtual double area() const = 0;

};

class Circle : public Shape { double radius;

public:

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

double area() const override { return 3.14159 \* radius \* radius;

}

double circumference() const { return 2 \* 3.14159 \* radius;

}

};

class Rectangle : public Shape { double length, breadth;

public:

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

double area() const override { return length \* breadth;

}

double perimeter() const { return 2 \* (length + breadth);

}

};

class Triangle : public Shape { double base, height;

public:

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

double area() const override { return 0.5 \* base \* height;

}

double perimeter() const {

return base + 2 \* sqrt(base \* base + height \* height);

}

};

int main() { Circle c(5);

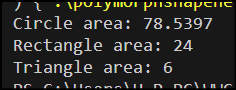
Rectangle r(4, 6);

Triangle t(3, 4);

cout << "Circle area: " << c.area() << std::endl; cout << "Rectangle area: " << r.area() << std::endl; cout << "Triangle area: " << t.area() << std::endl; return 0;

}

**Output –**

****

1. **Matrix Multiplication Using Function Overloading Sol –**

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

int m =0 , n=0 , p=0 ; int A[m][n];

cout << "Enter the dimensions of matrix A - "; cin >> m >> n;

int B[n][p];

cout << "Enter the dimensions of matrix B - "; cin >> n >> p;

int C[m][p];

cout << "Enter the elements of matrix A - "; for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) { cin >> A[i][j];

}

}

cout << "Enter the elements of matrix B - "; for (int i = 0; i < n; i++) {

for (int j = 0; j < p; j++) { cin >> B[i][j];

}

}

int operation;

cout << "Enter the operation type (1 for Matrix Addition, 2 for Matrix Multiplication) - ";

cin >> operation;

if (operation == 1) { if (n != m) {

cout << "Invalid dimensions for operation." << endl; return 0;

}

for (int i = 0; i < m; i++) { for (int j = 0; j < n; j++) {

C[i][j] = A[i][j] + B[i][j];

}

}

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

for (int j = 0; j < n; j++) { cout << C[i][j] << " ";

}

cout << endl;

}

}

else if (operation == 2) { if (m != n) {

cout << "Invalid dimensions for operation." << endl; return 0;

}

for (int i = 0; i < m; i++) { for (int j = 0; j < p; j++) {

C[i][j] = 0;

for (int k = 0; k < n; k++) { C[i][j] += A[i][k] \* B[k][j];

}

}

}

for (int i = 0; i < m; i++) { for (int j = 0; j < p; j++) {

cout << C[i][j] << " ";

}

cout << endl;

}

} else {

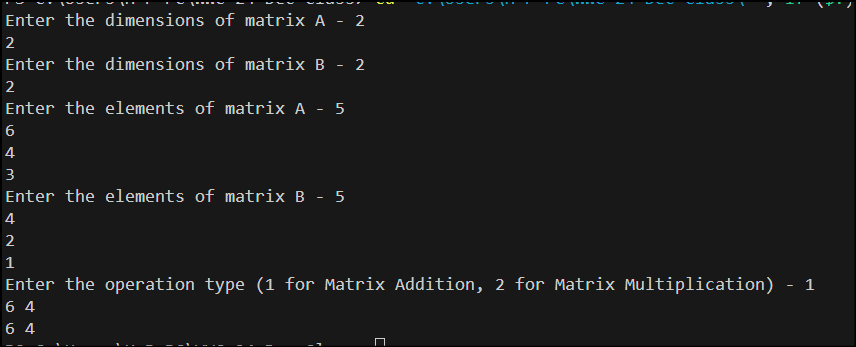
cout << "Invalid operation type." << endl;

}

return 0;

}

**Output –**

****

1. **Polymorphism in Shape Classes Sol –**

#include <iostream> using namespace std; class Shape {

public:

virtual double getArea() = 0;

};

class Rectangle : public Shape { int length, breadth;

public:

Rectangle(int l, int b) { length = l;

breadth = b;

}

double getArea() override { return length \* breadth;

}

};

class Circle : public Shape { int radius;

public: Circle(int r) {

radius = r;

}

double getArea() override {

return 3.14159 \* radius \* radius;

}

};

class Triangle : public Shape { int base, height;

public:

Triangle(int b, int h) { base = b;

height = h;

}

double getArea() override { return 0.5 \* base \* height;

}

};

int main() { Rectangle r(10, 20); Circle c(5); Triangle t(10, 15);

Shape\* shapes[] = {&r, &c, &t}; for (int i = 0; i < 3; i++) {

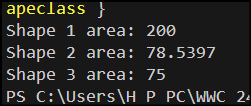
cout << "Shape " << i + 1 << " area: " << shapes [i]->getArea() << std::endl;

}

return 0;

}

**Output –**

****

1. **Implement Multiple Inheritance to Simulate a Library System Sol –**

#include <iostream> #include <string> using namespace std; class Book { protected:

string title; string author; int isbn;

public:

void setBookDetails(const string& t, const string& a, int i) {

title = t; author = a; isbn = i;

}

void displayBookDetails() const {

cout << "Book Title: " << title << endl; cout << "Author: " << author << endl; cout << "ISBN: " << isbn << endl;

}

};

class Borrower { protected:

string name; int id;

string borrowedBook; public:

void setBorrowerDetails(const string& n, int i) { name = n;

id = i; borrowedBook = "";

}

void displayBorrowerDetails() const {

cout << "Borrower Name: " << name << endl; cout << "Borrower ID: " << id << endl;

if (!borrowedBook.empty()) {

cout << "Currently Borrowed Book: " << borrowedBook << endl;

} else {

cout << "No books currently borrowed." << endl;

}

}

void borrowBook(const string& bookTitle) { borrowedBook = bookTitle;

}

void returnBook() { borrowedBook = "";

}

};

class Library : public Book, public Borrower { public:

void borrowBookFromLibrary() { if (borrowedBook.empty()) {

cout << "Enter the book title to borrow: "; string bookTitle;

cin.ignore(); getline(cin, bookTitle);

borrowBook(bookTitle);

cout << "You have borrowed: " << bookTitle << endl;

} else {

cout << "You already have a borrowed book: " << borrowedBook <<

endl;

}

}

void returnBookToLibrary() {

if (!borrowedBook.empty()) {

cout << "You have returned: " << borrowedBook << endl; returnBook();

} else {

cout << "You have no borrowed books to return." << endl;

}

}

};

int main() { Library library;

string title, author, name; int isbn, id, action;

cout << "Enter book title: "; getline(cin, title);

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

cout << "Enter ISBN (1000-9999): ";

cin >> isbn;

while (isbn < 1000 || isbn > 9999) {

cout << "Invalid ISBN. Please enter a valid ISBN (1000-9999): "; cin >> isbn;

}

library.setBookDetails(title, author, isbn); cout << "Enter borrower name: "; cin.ignore();

getline(cin, name);

cout << "Enter borrower ID (1-1000): "; cin >> id;

while (id < 1 || id > 1000) {

cout << "Invalid ID. Please enter a valid ID (1-1000): "; cin >> id;

}

library.setBorrowerDetails(name, id); cout << "\nBook Details:\n"; library.displayBookDetails();

cout << "\nBorrower Details:\n"; library.displayBorrowerDetails(); do {

cout << "\nEnter action (1 to borrow a book, 2 to return a book, 0 to exit):

";

cin >> action;

switch (action) { case 1:

library.borrowBookFromLibrary(); break;

case 2:

library.returnBookToLibrary(); break;

case 0:

cout << "Exiting the library system." << endl; break;

default:

cout << "Invalid action. Please try again." << endl;

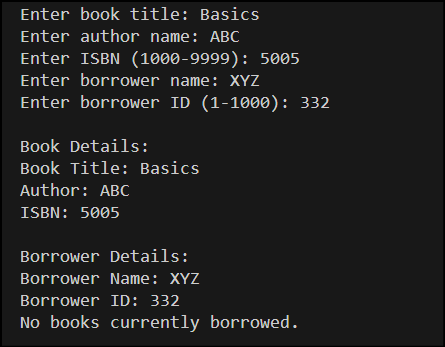
}

} while (action != 0);

return 0;

}

**Output –**

****

1. **Implement Polymorphism for Banking Transactions Sol –**

#include <iostream> using namespace std;

class Account { protected:

int balance;

public:

Account(int b) : balance(b) {}

virtual void display() const = 0;

virtual ~Account() {}

};

class SavingsAccount : public Account { double rate;

int time;

public:

SavingsAccount(int b, double r, int t) : Account(b), rate(r), time(t) {}

void display() const override {

double interest = balance \* rate \* time / 100.0;

cout << "Savings Account Interest: " << interest << endl;

}

};

class CurrentAccount : public Account { int fee;

public:

CurrentAccount(int b, int f) : Account(b), fee(f) {}

void display() const override {

cout << "Balance after fee deduction (Between 15 to 500): " << balance - fee << endl;

}

};

int main() {

int type, balance;

cout << "Enter Account Type (1 for Savings, 2 for Current): "; cin >> type;

if (type == 1) { double rate; int time;

cout << "Enter Balance (More than 1k to Less than 10L): "; cin >> balance;

cout << "Enter Interest Rate (percentage - 1 to 15): "; cin >> rate;

cout << "Enter Time (in years - 1 to 10 ): "; cin >> time;

if (balance >= 1000 && balance <= 1000000 && rate >= 1 && rate <= 15 && time >= 1 && time <= 10) {

SavingsAccount account(balance, rate, time); account.display();

} else {

cout << "Invalid input constraints." << endl;

}

} else if (type == 2) { int fee;

cout << "Enter Balance: ";

cin >> balance;

cout << "Enter Monthly Maintenance Fee: "; cin >> fee;

if (balance >= 1000 && balance <= 1000000 && fee >= 50 && fee <= 500) {

CurrentAccount account(balance, fee); account.display();

} else {

cout << "Invalid input constraints." << endl;

}

} else {

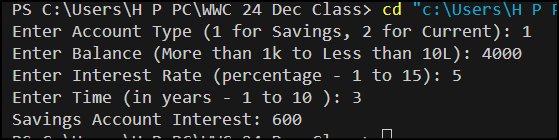
cout << "Invalid account type." << endl;

}

return 0;

}

**Output –**

****

1. **Hierarchical Inheritance for Employee Management System Sol –**

#include <iostream>

#include <string> using namespace std;

class Employee { protected:

string name; int id;

int salary;

public:

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

virtual void display() = 0;

};

class Manager : public Employee { private:

int rating;

public:

Manager(string n, int i, int s, int r) : Employee(n, i, s), rating(r) {}

void display() override {

double bonus = 0.1 \* salary \* rating; double totalEarnings = salary + bonus;

cout << "Employee: " << name << " (ID: " << id << ")" << endl;

cout << "Role: Manager" << endl;

cout << "Base Salary: " << salary << endl; cout << "Bonus: " << bonus << endl;

cout << "Total Earnings: " << totalEarnings << endl;

}

};

class Developer : public Employee { private:

int extraHours;

public:

Developer(string n, int i, int s, int hours) : Employee(n, i, s), extraHours(hours) {}

void display() override {

double overtimeCompensation = 500 \* extraHours; double totalEarnings = salary + overtimeCompensation;

cout << "Employee: " << name << " (ID: " << id << ")" << endl; cout << "Role: Developer" << endl;

cout << "Base Salary: " << salary << endl;

cout << "Overtime Compensation: " << overtimeCompensation << endl; cout << "Total Earnings: " << totalEarnings << endl;

}

};

int main() {

int employeeType;

cout << "Enter Employee Type (1 for Manager, 2 for Developer): "; cin >> employeeType;

if (employeeType < 1 || employeeType > 2) { cout << "Invalid employee type." << endl; return 0;

}

string name; int id, salary;

cout << "Enter Name: "; cin >> name;

cout << "Enter ID: "; cin >> id;

cout << "Enter Salary: "; cin >> salary;

if (salary < 10000 || salary > 1000000) { cout << "Invalid salary." << endl; return 0;

}

if (employeeType == 1) { // Manager int rating;

cout << "Enter Performance Rating (1-5): ";

cin >> rating;

if (rating < 1 || rating > 5) {

cout << "Invalid rating." << endl; return 0;

}

Manager manager(name, id, salary, rating); manager.display();

}

else if (employeeType == 2) { // Developer int extraHours;

cout << "Enter Extra Hours Worked: "; cin >> extraHours;

if (extraHours < 0 || extraHours > 100) { cout << "Invalid extra hours." << endl; return 0;

}

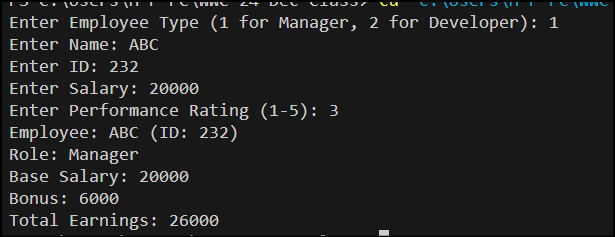
Developer developer(name, id, salary, extraHours); developer.display();

}

return 0;

}

**Output –**

****

1. **Multi-Level Inheritance for Vehicle Simulation Sol –**

#include <string> #include <iostream>

using namespace std;

class Vehicle { protected:

string brand; string model; double mileage;

public:

Vehicle(const string& b, const string& m, double mil) : brand(b), model(m), mileage(mil) {}

virtual void displayDetails() const { cout << "Brand: " << brand << endl; cout << "Model: " << model << endl;

cout << "Mileage: " << mileage << " miles" << endl;

}

};

class Car : public Vehicle { protected:

double fuel;

double distanceCovered;

public:

Car(const string& b, const string& m, double mil, double f, double d)

: Vehicle(b, m, mil), fuel(f), distanceCovered(d) {}

double calculateFuelEfficiency() const { return distanceCovered / fuel;

}

void displayDetails() const override { Vehicle::displayDetails();

cout << "Fuel Efficiency: " << calculateFuelEfficiency() << " miles per gallon" << endl;

}

};

class ElectricCar : public Car { private:

double batteryCapacity; double efficiency;

public:

ElectricCar(const string& b, const string& m, double mil, double cap, double eff)

: Car(b, m, mil, 0, 0), batteryCapacity(cap), efficiency(eff) {}

double calculateRange() const { return batteryCapacity \* efficiency;

}

void displayDetails() const override { Car::displayDetails();

cout<< "Battery Capacity: " << batteryCapacity << " kWh" <<endl; cout << "Range: " << calculateRange() << " miles" << endl;

}

};

int main() {

int vehicleType; string brand, model;

double mileage, fuel, distance, batteryCapacity, efficiency;

cout<< "Enter Vehicle Type (1 for Car, 2 for Electric Car): "; cin>> vehicleType;

cout << "Enter Brand: "; cin.ignore();

getline(cin, brand);

cout << "Enter Model: "; getline(cin, model);

cout << "Enter Mileage (0 - 500,000): "; cin >> mileage;

while (mileage < 0 || mileage > 500000) {

cout << "Invalid mileage. Please enter a mileage between 0 and 500,000: "; cin >> mileage;

}

if (vehicleType == 1) {

cout << "Enter Fuel (1 - 100 gallons): "; cin >> fuel;

while (fuel < 1 || fuel > 100) {

cout << "Invalid fuel. Please enter a fuel amount between 1 and 100 gallons: ";

cin >> fuel;

}

cout << "Enter Distance Covered (1 - 1,000 miles): "; cin >> distance;

while (distance < 1 || distance > 1000) {

cout << "Invalid distance. Please enter a distance between 1 and 1,000 miles: ";

cin >> distance;

}

Car car(brand, model, mileage, fuel, distance); car.displayDetails();

} else if (vehicleType == 2) {

cout << "Enter Battery Capacity (10 - 150 kWh): "; cin >> batteryCapacity;

while (batteryCapacity < 10 || batteryCapacity > 150) {

cout << "Invalid battery capacity. Please enter a capacity between 10 and 150 kWh: ";

cin >> batteryCapacity;

}

cout << "Enter Efficiency (1 - 10 miles per kWh): "; cin >> efficiency;

while (efficiency < 1 || efficiency > 10) {

cout << "Invalid efficiency. Please enter an efficiency between 1 and 10 miles per kWh: ";

cin >> efficiency;

}

ElectricCar electricCar(brand, model, mileage, batteryCapacity, efficiency);

electricCar.displayDetails();

} else {

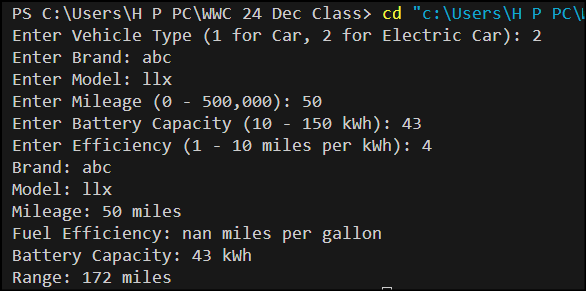
cout << "Invalid vehicle type." <<endl;

}

return 0;

}

**Output –**

****

1. **Function Overloading for Complex Number Operations. Sol –**

#include <iostream> #include <cmath> #include <iomanip>

using namespace std;

class Complex { private:

double real; double imaginary;

public:

Complex(double r = 0, double i = 0) : real(r), imaginary(i) {}

Complex operator+(const Complex& other) const {

return Complex(real + other.real, imaginary + other.imaginary);

}

Complex operator\*(const Complex& other) const {

return Complex(real \* other.real - imaginary \* other.imaginary, real \* other.imaginary + imaginary \* other.real);

}

double magnitude() const {

return sqrt(real \* real + imaginary \* imaginary);

}

void display() const { if (imaginary >= 0) {

cout << real << " + " << imaginary << "i" << endl;

} else {

cout << real << " - " << -imaginary << "i" << endl;

}

}

};

int main() {

int operationType;

cout << "Enter Operation Type (1 for Addition, 2 for Multiplication, 3 for Magnitude): ";

cin >> operationType;

if (operationType == 1 || operationType == 2) { double real1, imaginary1, real2, imaginary2;

cout << "Enter first complex number (real1 imaginary1): "; cin >> real1 >> imaginary1;

cout << "Enter second complex number (real2 imaginary2): "; cin >> real2 >> imaginary2;

Complex c1(real1, imaginary1); Complex c2(real2, imaginary2);

if (operationType == 1) { Complex result = c1 + c2; cout << "Result of Addition: "; result.display();

} else if (operationType == 2) { Complex result = c1 \* c2;

cout << "Result of Multiplication: "; result.display();

}

} else if (operationType == 3) { double real, imaginary;

cout << "Enter complex number (real imaginary): "; cin >> real >> imaginary;

Complex c(real, imaginary); double result = c.magnitude();

cout << "Magnitude: " << fixed << setprecision(2) << result << endl;

} else {

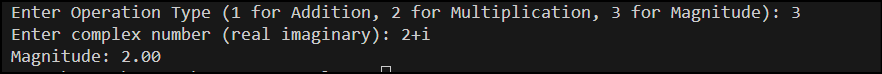
cout << "Invalid operation type." << endl;

}

return 0;

}

**Output –**

****

1. **Polymorphism for Shape Area Calculations Sol –**

#include <iostream> #include <memory>

using namespace std;

class Shape { public:

virtual float calculateArea() const = 0; virtual ~Shape() {}

};

class Rectangle : public Shape { private:

float length; float width;

public:

Rectangle(float l, float w) : length(l), width(w) {}

float calculateArea() const override { return length \* width;

}

};

class Circle : public Shape { private:

float radius;

public:

Circle(float r) : radius(r) {}

float calculateArea() const override { return 3.14159f \* radius \* radius;

}

};

class Triangle : public Shape { private:

float base; float height;

public:

Triangle(float b, float h) : base(b), height(h) {}

float calculateArea() const override { return 0.5f \* base \* height;

}

};

int main() {

int shapeType;

cout << "Enter Shape Type (1 for Rectangle, 2 for Circle, 3 for Triangle): "; cin >> shapeType;

unique\_ptr<Shape> shape;

if (shapeType == 1) { float length, width;

cout << "Enter Length and Width: "; cin >> length >> width;

shape = make\_unique<Rectangle>(length, width);

} else if (shapeType == 2) { float radius;

cout << "Enter Radius: "; cin >> radius;

shape = make\_unique<Circle>(radius);

} else if (shapeType == 3) { float base, height;

cout << "Enter Base and Height: "; cin >> base >> height;

shape = make\_unique<Triangle>(base, height);

} else {

cout << "Invalid shape type." << endl; return 1;

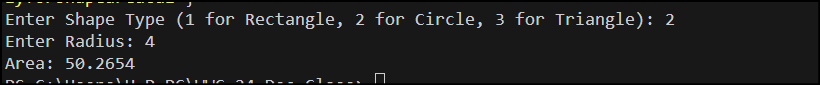
}

cout << "Area: " << shape->calculateArea() << endl;

return 0;

}

**Output –**

****

1. **Advanced Function Overloading for Geometric Shapes Sol –**

#include <iostream> #include <iomanip>

using namespace std;

float calculateArea(float radius) {

return 3.14159f \* radius \* radius;

}

float calculateArea(float length, float breadth) { return length \* breadth;

}

float calculateArea(float base, float height, bool isTriangle) { return 0.5f \* base \* height;

}

int main() { int choice;

cout << "Choose a shape to calculate the area:\n"; cout << "1. Circle\n";

cout << "2. Rectangle\n"; cout << "3. Triangle\n";

cout << "Enter your choice (1-3): "; cin >> choice;

if (choice < 1 || choice > 3) {

cout << "Invalid choice. Please enter a number between 1 and 3." << endl; return 1;

}

if (choice == 1) { float radius;

cout << "Enter the radius of the circle: "; cin >> radius;

if (radius <= 0) {

cout << "Invalid input. Radius must be a positive number." << endl; return 1;

}

cout << "Area of the circle: " << fixed << setprecision(2) << calculateArea(radius) << endl;

} else if (choice == 2) { float length, breadth;

cout << "Enter the length and breadth of the rectangle: "; cin >> length >> breadth;

if (length <= 0 || breadth <= 0) {

cout << "Invalid input. Length and breadth must be positive numbers."

<< endl;

return 1;

}

cout << "Area of the rectangle: " << fixed << setprecision(2) << calculateArea(length, breadth) << endl;

} else if (choice == 3) { float base, height;

cout << "Enter the base and height of the triangle: "; cin >> base >> height;

if (base <= 0 || height <= 0) {

cout << "Invalid input. Base and height must be positive numbers." <<

endl;

return 1;

}

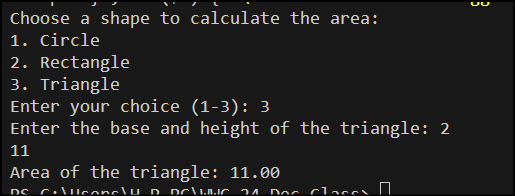
cout << "Area of the triangle: " << fixed << setprecision(2) << calculateArea(base, height, true) << endl;

}

return 0;

}

**Output –**

****