**DSA0187: OBJECT ORIENTED PROGRAMMING WITH C++**

**18/03/2024**

**192210667**

1.

#include <iostream>

// Base class Shape

class Shape {

public:

virtual double getArea() const = 0; // Virtual function to get the area

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

double length;

double width;

public:

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

double getArea() const override { // Override base class function

return length \* width;

}

};

// Derived class Circle

class Circle : public Shape {

private:

double radius;

public:

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

double getArea() const override { // Override base class function

return 3.14159 \* radius \* radius; // Using pi as an approximation

}

};

int main() {

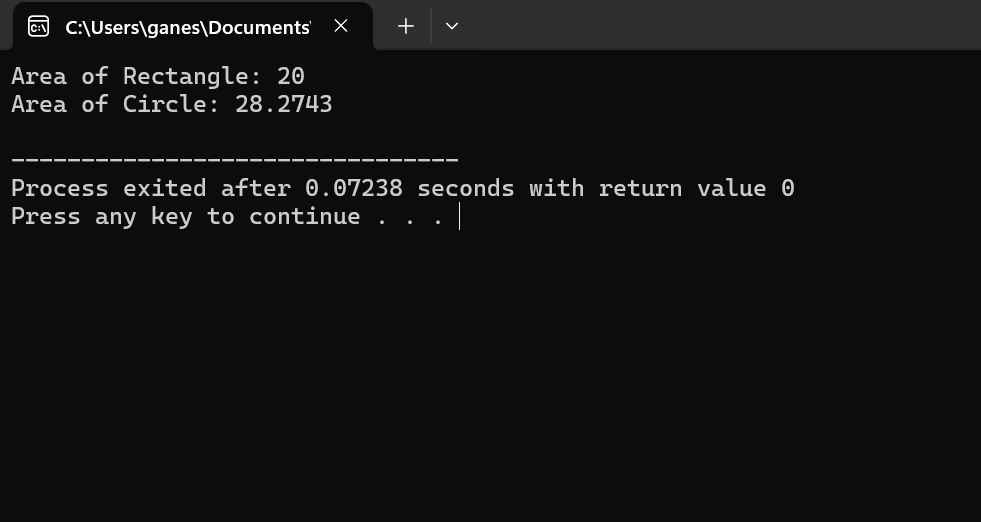
Rectangle rectangle(5, 4);

Circle circle(3);

std::cout << "Area of Rectangle: " << rectangle.getArea() << std::endl;

std::cout << "Area of Circle: " << circle.getArea() << std::endl;

Return



2.

#include <iostream>

// Base class Animal

class Animal {

public:

virtual void makeSound() const = 0; // Virtual function to make sound

};

// Derived class Dog

class Dog : public Animal {

public:

void makeSound() const override { // Override base class function

std::cout << "Woof! Woof!" << std::endl;

}

};

// Derived class Cat

class Cat : public Animal {

public:

void makeSound() const override { // Override base class function

std::cout << "Meow! Meow!" << std::endl;

}

};

int main() {

Dog dog;

Cat cat;

std::cout << "Dog says: ";

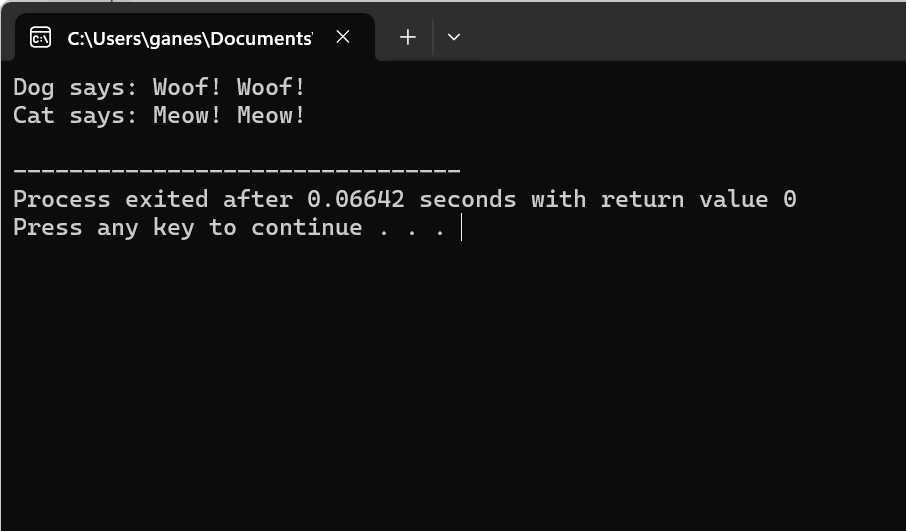
dog.makeSound();

std::cout << "Cat says: ";

cat.makeSound();

return 0;

}



3.

#include <iostream>

#include <string>

// Base class Employee

class Employee {

protected:

std::string name;

double salary;

public:

// Constructor to initialize name and salary

Employee(const std::string& empName, double empSalary) : name(empName), salary(empSalary) {}

// Virtual destructor (in case you have polymorphic behavior)

virtual ~Employee() {}

// Function to display employee information

virtual void display() const {

std::cout << "Name: " << name << std::endl;

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

}

};

// Derived class Manager from Employee

class Manager : public Employee {

private:

std::string department;

public:

// Constructor to initialize name, salary, and department

Manager(const std::string& empName, double empSalary, const std::string& dept)

: Employee(empName, empSalary), department(dept) {}

// Override base class display function to include department

void display() const override {

Employee::display();

std::cout << "Department: " << department << std::endl;

}

};

// Derived class Worker from Employee

class Worker : public Employee {

public:

// Constructor to initialize name and salary

Worker(const std::string& empName, double empSalary) : Employee(empName, empSalary) {}

};

int main() {

// Create instances of Manager and Worker

Manager manager("John Doe", 50000, "Sales");

Worker worker("Jane Smith", 30000);

// Display information of Manager and Worker

std::cout << "Manager's Information: " << std::endl;

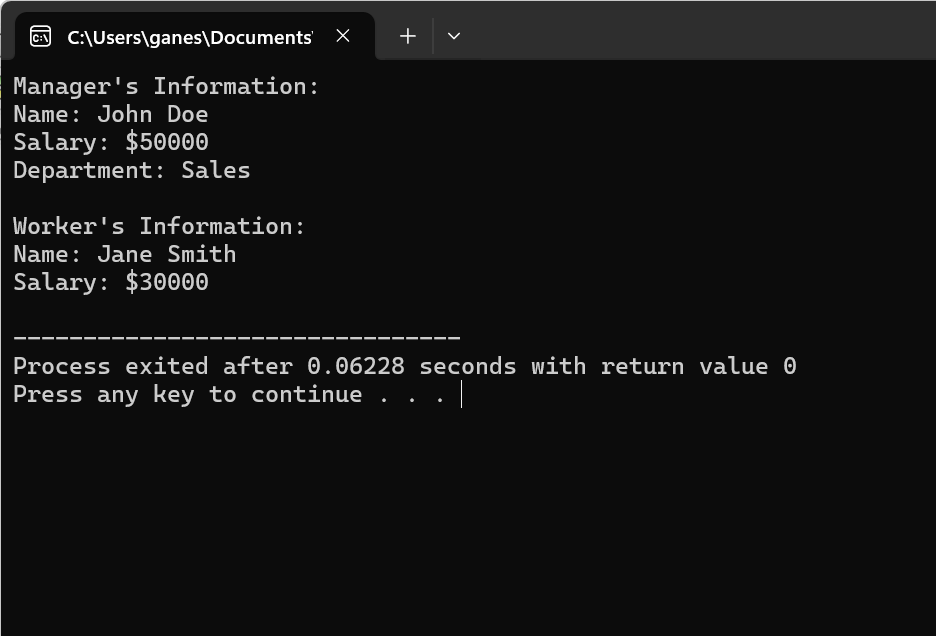
manager.display();

std::cout << "\nWorker's Information: " << std::endl;

worker.display();

return 0;

}



4.

#include <iostream>

// Base class Shape

class Shape {

public:

// Virtual function to draw shape

virtual void draw() const = 0;

};

// Derived class Circle from Shape

class Circle : public Shape {

public:

// Implementation of draw function for Circle

void draw() const override {

std::cout << "Drawing Circle..." << std::endl;

}

};

// Derived class Square from Shape

class Square : public Shape {

public:

// Implementation of draw function for Square

void draw() const override {

std::cout << "Drawing Square..." << std::endl;

}

};

// Derived class Triangle from Shape

class Triangle : public Shape {

public:

// Implementation of draw function for Triangle

void draw() const override {

std::cout << "Drawing Triangle..." << std::endl;

}

};

int main() {

// Create instances of Circle, Square, and Triangle

Circle circle;

Square square;

Triangle triangle;

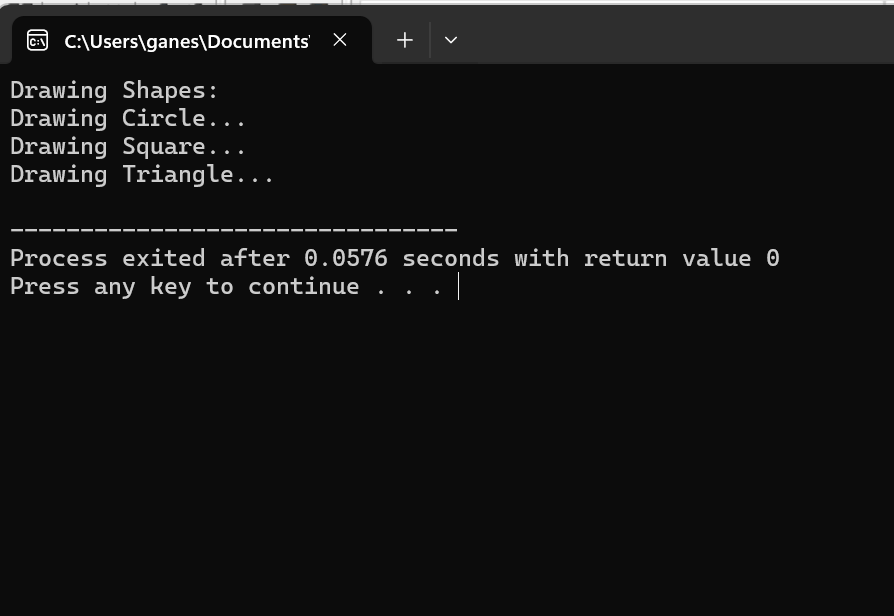
// Draw each shape

std::cout << "Drawing Shapes:" << std::endl;

circle.draw();

square.draw();

triangle.draw();

Return

5.

#include <iostream>

// Base class Vehicle

class Vehicle {

protected:

double speed;

double fuel;

public:

// Constructor to initialize speed and fuel

Vehicle(double initialSpeed, double initialFuel) : speed(initialSpeed), fuel(initialFuel) {}

};

// Derived class Car from Vehicle

class Car : public Vehicle {

public:

// Constructor to initialize speed and fuel for Car

Car(double initialSpeed, double initialFuel) : Vehicle(initialSpeed, initialFuel) {}

};

// Derived class Bike from Vehicle

class Bike : public Vehicle {

public:

// Constructor to initialize speed and fuel for Bike

Bike(double initialSpeed, double initialFuel) : Vehicle(initialSpeed, initialFuel) {}

};

int main() {

// Creating instances of Car and Bike

Car car(60.0, 40.0); // Car with initial speed 60 km/h and initial fuel 40 liters

Bike bike(30.0, 10.0); // Bike with initial speed 30 km/h and initial fuel 10 liters

// Outputting information about Car and Bike

std::cout << "Car - Speed: " << car.speed << " km/h, Fuel: " << car.fuel << " liters" << std::endl;

std::cout << "Bike - Speed: " << bike.speed << " km/h, Fuel: " << bike.fuel << " liters" << std::endl;

return 0;

}

6.

#include <iostream>

// Base class Bird

class Bird {

public:

// Virtual function to represent flying behavior

virtual void fly() const = 0;

};

// Derived class Eagle from Bird

class Eagle : public Bird {

public:

// Implementation of fly function for Eagle

void fly() const override {

std::cout << "Eagle is soaring high in the sky." << std::endl;

}

};

// Derived class Sparrow from Bird

class Sparrow : public Bird {

public:

// Implementation of fly function for Sparrow

void fly() const override {

std::cout << "Sparrow is flapping its wings and flying around." << std::endl;

}

};

int main() {

// Create instances of Eagle and Sparrow

Eagle eagle;

Sparrow sparrow;

// Perform flying behavior

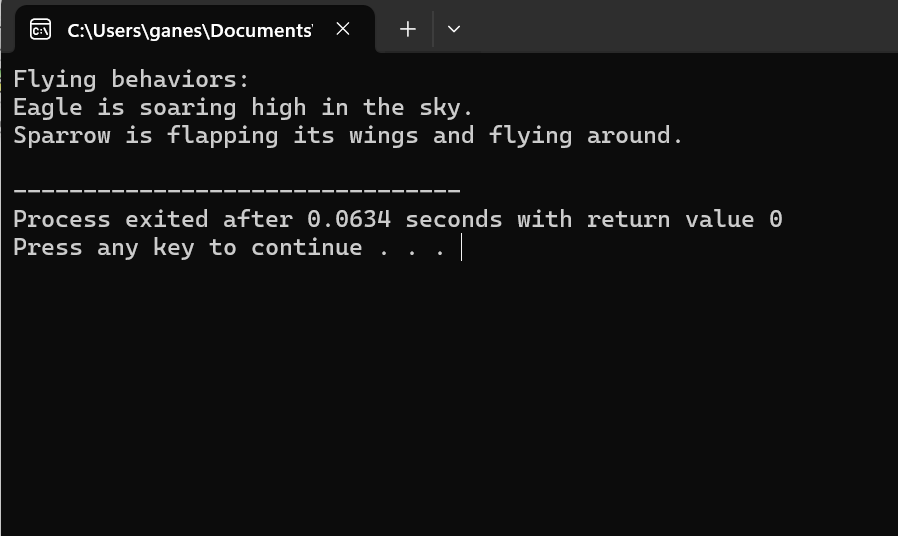
std::cout << "Flying behaviors:" << std::endl;

eagle.fly();

sparrow.fly();

return 0;

}



7.

#include <iostream>

// Abstract base class Shape

class Shape {

public:

// Pure virtual function to calculate area

virtual double calculateArea() const = 0;

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

private:

double length;

double width;

public:

// Constructor to initialize length and width

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

// Implementation of calculateArea function for Rectangle

double calculateArea() const override {

return length \* width;

}

};

// Derived class Circle from Shape

class Circle : public Shape {

private:

double radius;

public:

// Constructor to initialize radius

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

// Implementation of calculateArea function for Circle

double calculateArea() const override {

return 3.14159 \* radius \* radius; // Using pi as an approximation

}

};

int main() {

// Create instances of Rectangle and Circle

Rectangle rectangle(5, 4);

Circle circle(3);

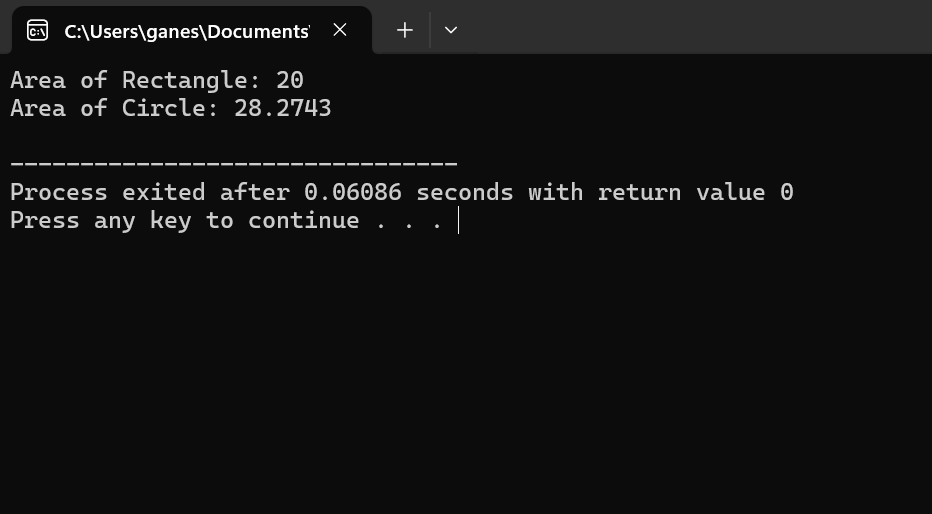
// Calculate and output areas

std::cout << "Area of Rectangle: " << rectangle.calculateArea() << std::endl;

std::cout << "Area of Circle: " << circle.calculateArea() << std::endl;

return 0;

}



8

#include <iostream>

#include <string>

// Base class Person

class Person {

protected:

std::string name;

int age;

public:

// Constructor to initialize name and age

Person(const std::string& n, int a) : name(n), age(a) {}

};

// Derived class Student from Person

class Student : public Person {

private:

char grade;

public:

// Constructor to initialize name, age, and grade

Student(const std::string& n, int a, char g) : Person(n, a), grade(g) {}

// Function to display student information

void display() const {

std::cout << "Name: " << name << std::endl;

std::cout << "Age: " << age << std::endl;

std::cout << "Grade: " << grade << std::endl;

}

};

int main() {

// Create instances of Student

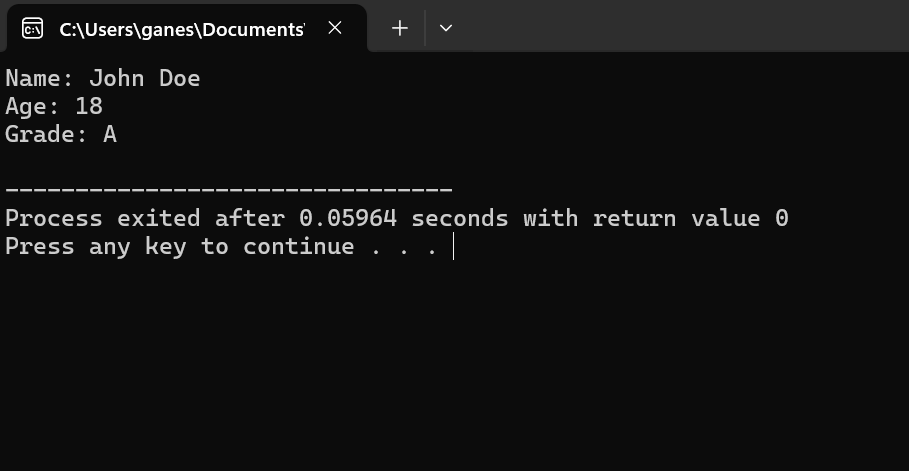
Student student("John Doe", 18, 'A');

// Display student information

student.display();

return 0;

}

.

9.

#include <iostream>

// Function to swap two integers using pointers

void swapIntegers(int\* a, int\* b) {

int temp = \*a; // Store the value of 'a' in temp

\*a = \*b; // Assign the value of 'b' to 'a'

\*b = temp; // Assign the stored value of 'a' (in temp) to 'b'

}

int main() {

int num1 = 5;

int num2 = 10;

std::cout << "Before swapping:" << std::endl;

std::cout << "num1 = " << num1 << ", num2 = " << num2 << std::endl;

// Pass the addresses of num1 and num2 to the swapIntegers function

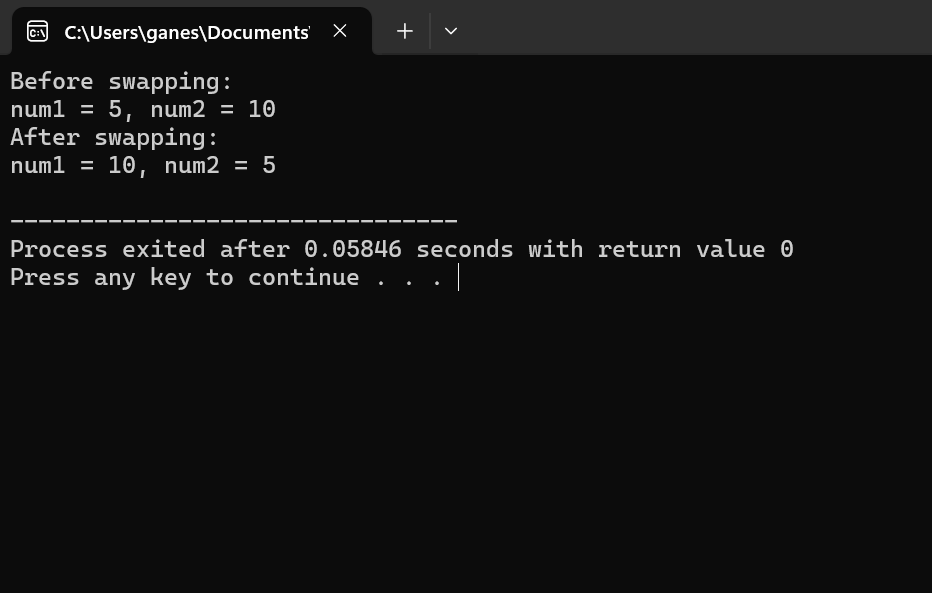
swapIntegers(&num1, &num2);

std::cout << "After swapping:" << std::endl;

std::cout << "num1 = " << num1 << ", num2 = " << num2 << std::endl;

return 0;

}



10.

#include <iostream>

// Class Circle

class Circle {

private:

double radius;

public:

// Constructor

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

// Function to calculate area

double calculateArea() const {

return 3.14159 \* radius \* radius; // Using pi as an approximation

}

};

int main() {

double radius;

std::cout << "Enter the radius of the circle: ";

std::cin >> radius;

// Creating an instance of Circle dynamically using a pointer

Circle\* circlePtr = new Circle(radius);

// Using the pointer to calculate the area of the circle

double area = circlePtr->calculateArea();

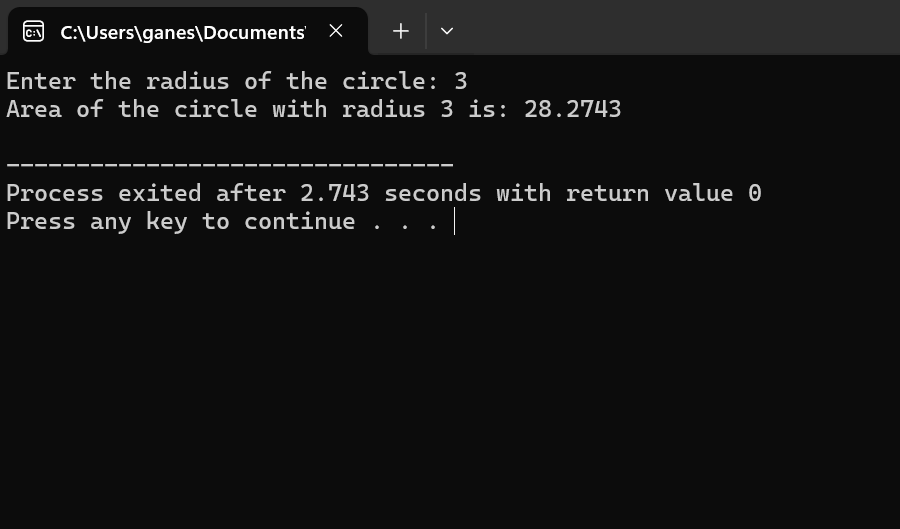
std::cout << "Area of the circle with radius " << radius << " is: " << area << std::endl;

// Don't forget to deallocate memory to avoid memory leaks

delete circlePtr;

return 0;

}



11.

#include <iostream>

// Class Rectangle

class Rectangle {

private:

double length;

double width;

public:

// Constructor

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

// Function to calculate perimeter

double calculatePerimeter() const {

return 2 \* (length + width);

}

};

int main() {

double length, width;

std::cout << "Enter the length of the rectangle: ";

std::cin >> length;

std::cout << "Enter the width of the rectangle: ";

std::cin >> width;

// Creating an instance of Rectangle dynamically using a pointer

Rectangle\* rectanglePtr = new Rectangle(length, width);

// Using the pointer to calculate the perimeter of the rectangle

double perimeter = rectanglePtr->calculatePerimeter();

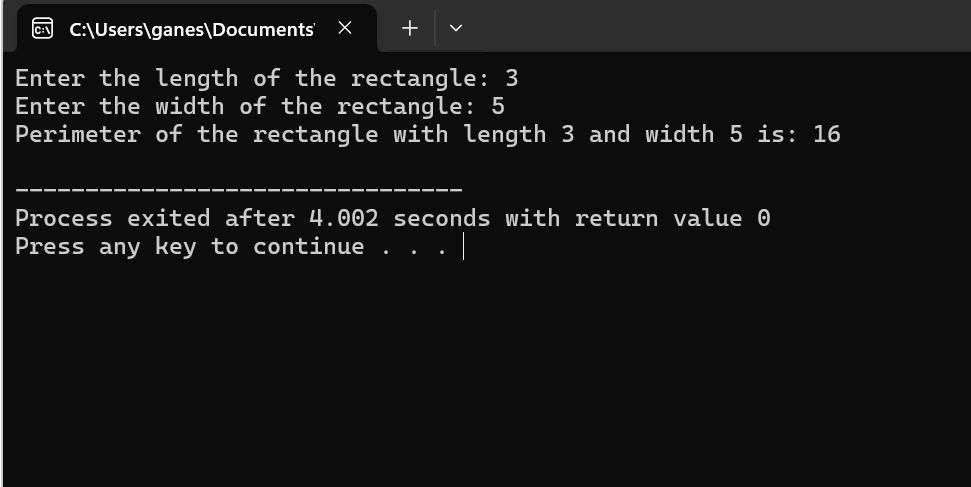
std::cout << "Perimeter of the rectangle with length " << length << " and width " << width << " is: " << perimeter << std::endl;

// Don't forget to deallocate memory to avoid memory leaks

delete rectanglePtr;

return 0;

}



12.

#include <iostream>

#include <string>

// Class Employee

class Employee {

private:

std::string name;

double salary;

public:

// Constructor

Employee(const std::string& empName, double empSalary) : name(empName), salary(empSalary) {}

// Function to display employee details

void displayDetails() const {

std::cout << "Employee Name: " << name << std::endl;

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

}

};

int main() {

std::string name;

double salary;

std::cout << "Enter the employee's name: ";

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

std::cout << "Enter the employee's salary: $";

std::cin >> salary;

// Creating an instance of Employee dynamically using a pointer

Employee\* employeePtr = new Employee(name, salary);

// Using the pointer to display the details of the employee

std::cout << "\nEmployee Details:" << std::endl;

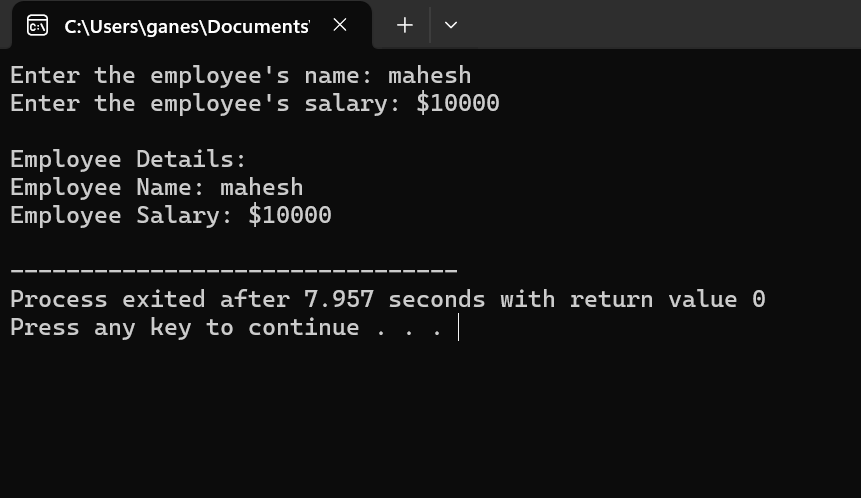
employeePtr->displayDetails();

// Don't forget to deallocate memory to avoid memory leaks

delete employeePtr;

return 0;

}



13.

#include <iostream>

#include <string>

// Class Student

class Student {

private:

std::string name;

char grade;

public:

// Constructor

Student(const std::string& studentName, char studentGrade) : name(studentName), grade(studentGrade) {}

// Function to display student details using 'this' pointer

void displayDetails() {

std::cout << "Student Name: " << this->name << std::endl;

std::cout << "Student Grade: " << this->grade << std::endl;

}

};

int main() {

std::string name;

char grade;

std::cout << "Enter the student's name: ";

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

std::cout << "Enter the student's grade: ";

std::cin >> grade;

// Creating an instance of Student

Student student(name, grade);

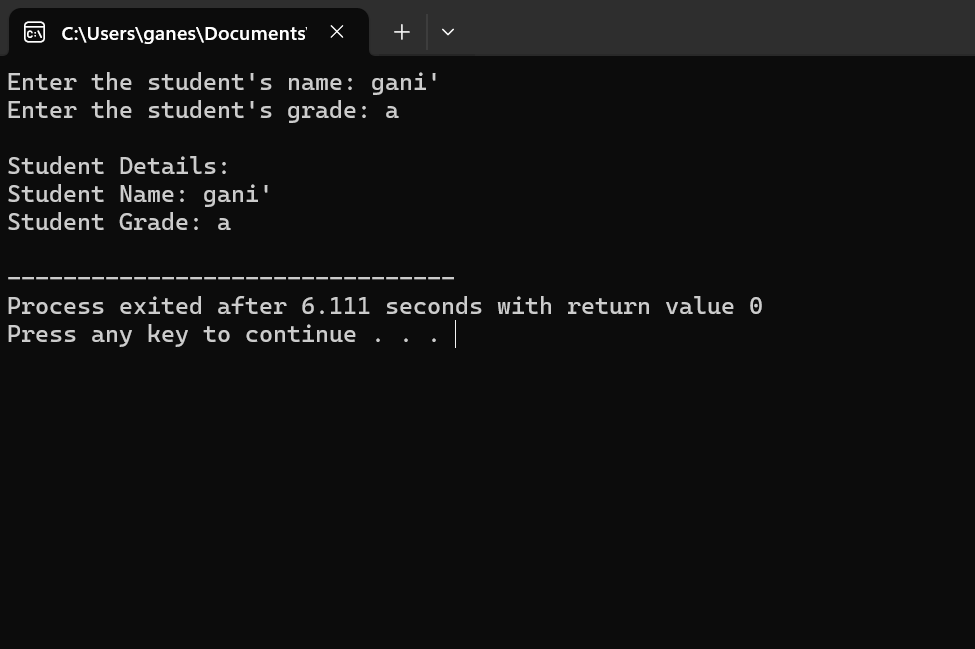
// Using the instance to display the details of the student using 'this' pointer

std::cout << "\nStudent Details:" << std::endl;

student.displayDetails();

return 0;

}



14.

#include <iostream>

// Base class Shape

class Shape {

public:

// Virtual function draw

virtual void draw() const {

std::cout << "Drawing a generic shape." << std::endl;

}

};

// Derived class Circle from Shape

class Circle : public Shape {

public:

// Override draw function for Circle

void draw() const override {

std::cout << "Drawing a circle." << std::endl;

}

};

int main() {

// Create an instance of Circle

Circle circle;

// Create a pointer to the base class Shape

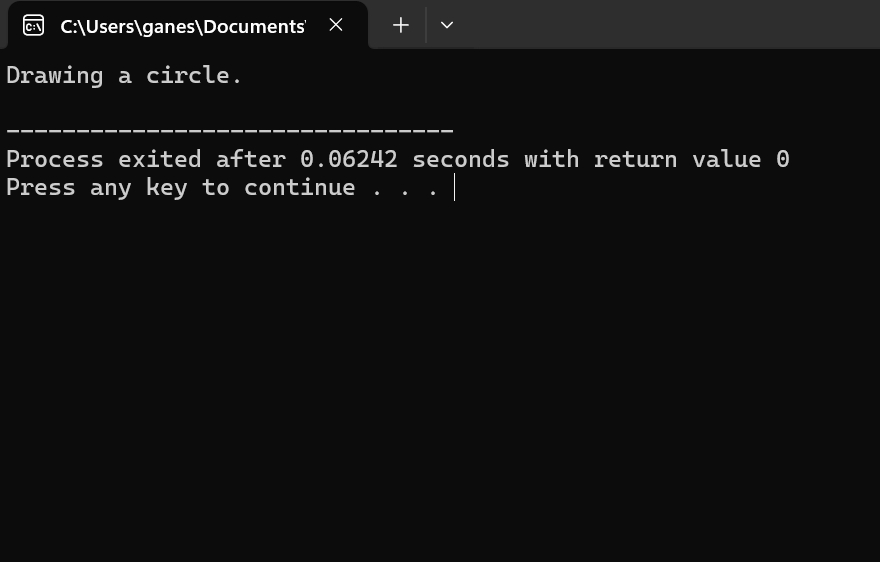
Shape\* shapePtr = &circle;

// Call the draw function using the pointer

shapePtr->draw();

return 0;

}



15.

#include <iostream>

#include <string>

// Base class Person

class Person {

protected:

std::string name;

int age;

public:

// Constructor

Person(const std::string& n, int a) : name(n), age(a) {}

// Virtual function to display person details

virtual void display() const {

std::cout << "Name: " << name << std::endl;

std::cout << "Age: " << age << std::endl;

}

};

// Derived class Student from Person

class Student : public Person {

public:

// Constructor

Student(const std::string& n, int a) : Person(n, a) {}

// Override display function for Student

void display() const override {

std::cout << "Student Name: " << name << std::endl;

std::cout << "Student Age: " << age << std::endl;

}

};

int main() {

// Create an instance of Student

Student student("John Doe", 20);

// Create a pointer to the base class Person

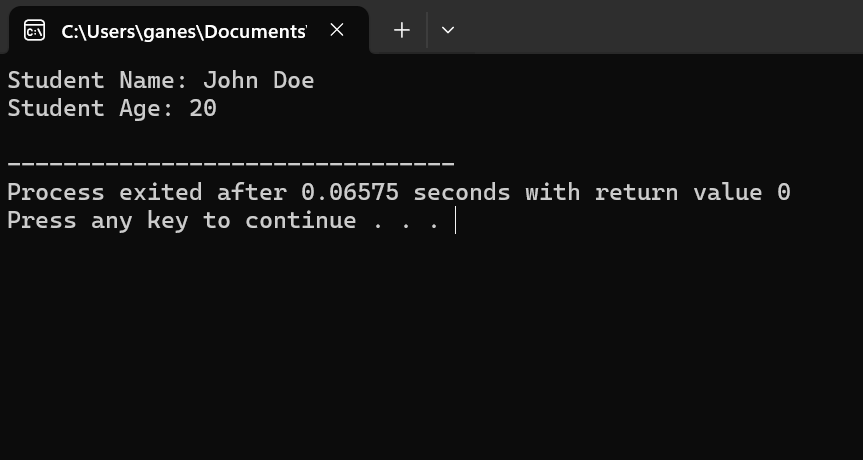
Person\* personPtr = &student;

// Call the display function using the pointer

personPtr->display();

return 0;

}



**Medium**

16.

#include <iostream>

// Base class Vehicle

class Vehicle {

protected:

double speed;

double fuel;

public:

// Constructor to initialize speed and fuel

Vehicle(double s, double f) : speed(s), fuel(f) {}

// Virtual destructor

virtual ~Vehicle() {}

// Virtual function to display vehicle information

virtual void displayInfo() const {

std::cout << "Speed: " << speed << " km/h" << std::endl;

std::cout << "Fuel: " << fuel << " liters" << std::endl;

}

};

// Derived class Car from Vehicle

class Car : public Vehicle {

public:

// Constructor to initialize speed and fuel for Car

Car(double s, double f) : Vehicle(s, f) {}

// Function to display car information

void displayInfo() const override {

std::cout << "Car Information:" << std::endl;

Vehicle::displayInfo();

}

};

// Derived class Bike from Vehicle

class Bike : public Vehicle {

public:

// Constructor to initialize speed and fuel for Bike

Bike(double s, double f) : Vehicle(s, f) {}

// Function to display bike information

void displayInfo() const override {

std::cout << "Bike Information:" << std::endl;

Vehicle::displayInfo();

}

};

int main() {

// Create instances of Car and Bike

Car car(100, 30);

Bike bike(50, 10);

// Display information of Car and Bike

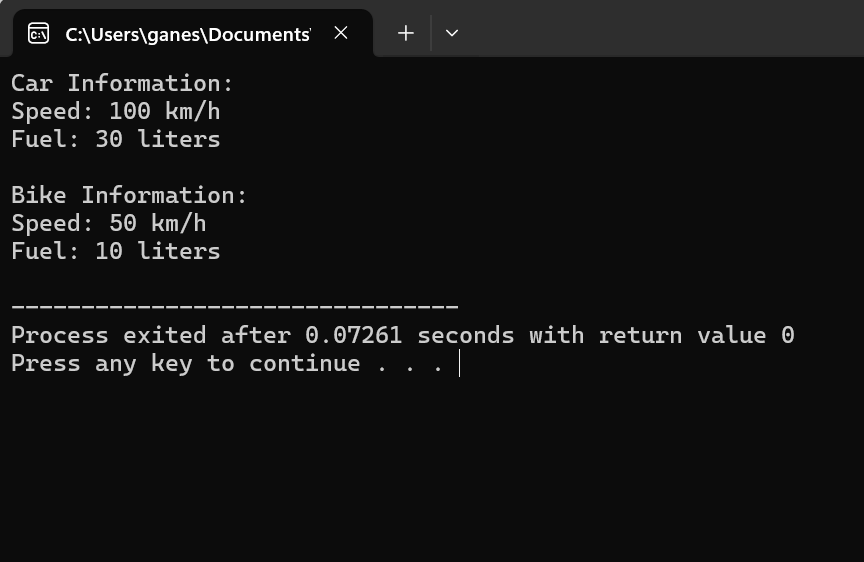
car.displayInfo();

std::cout << std::endl;

bike.displayInfo();

return 0;

}



17.

#include <iostream>

// Base class Shape

class Shape {

public:

// Virtual function to calculate area

virtual double area() const = 0;

// Virtual function to calculate perimeter

virtual double perimeter() const = 0;

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

private:

double length;

double width;

public:

// Constructor to initialize length and width

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

// Implementation of area function for Rectangle

double area() const override {

return length \* width;

}

// Implementation of perimeter function for Rectangle

double perimeter() const override {

return 2 \* (length + width);

}

};

// Derived class Circle from Shape

class Circle : public Shape {

private:

double radius;

public:

// Constructor to initialize radius

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

// Implementation of area function for Circle

double area() const override {

return 3.14159 \* radius \* radius; // Using pi as an approximation

}

// Implementation of perimeter function for Circle

double perimeter() const override {

return 2 \* 3.14159 \* radius; // Using pi as an approximation

}

};

int main() {

// Create instances of Rectangle and Circle

Rectangle rectangle(4, 6);

Circle circle(3);

// Calculate and output area and perimeter for Rectangle

std::cout << "Rectangle:" << std::endl;

std::cout << "Area: " << rectangle.area() << std::endl;

std::cout << "Perimeter: " << rectangle.perimeter() << std::endl;

// Calculate and output area and perimeter for Circle

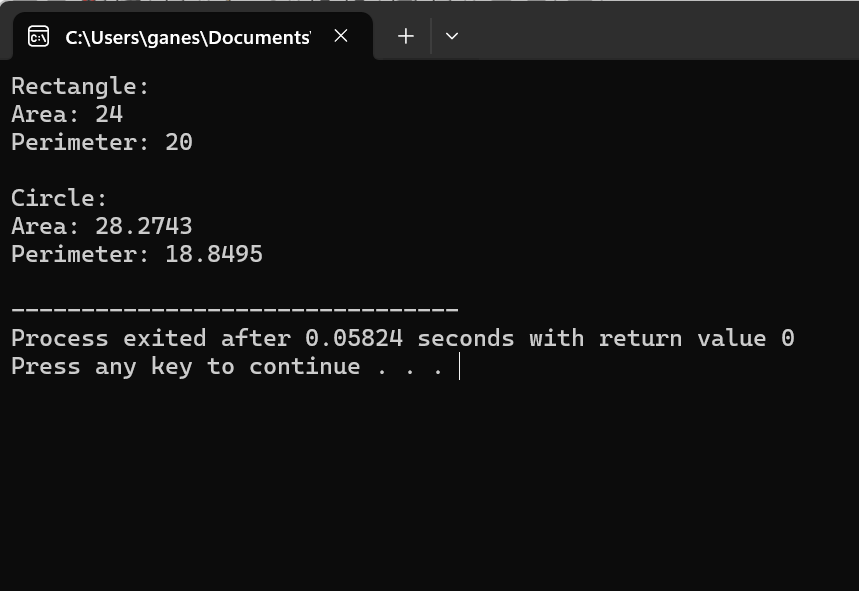
std::cout << "\nCircle:" << std::endl;

std::cout << "Area: " << circle.area() << std::endl;

std::cout << "Perimeter: " << circle.perimeter() << std::endl;

return 0;

}



18.

#include <iostream>

#include <string>

// Base class Employee

class Employee {

protected:

std::string name;

int id;

double salary;

public:

// Constructor to initialize name, id, and salary

Employee(const std::string& empName, int empId, double empSalary) : name(empName), id(empId), salary(empSalary) {}

// Virtual destructor

virtual ~Employee() {}

// Function to display employee details

virtual void displayDetails() const {

std::cout << "Name: " << name << std::endl;

std::cout << "ID: " << id << std::endl;

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

}

};

// Derived class Manager from Employee

class Manager : public Employee {

private:

std::string department;

public:

// Constructor to initialize name, id, salary, and department for Manager

Manager(const std::string& empName, int empId, double empSalary, const std::string& dept)

: Employee(empName, empId, empSalary), department(dept) {}

// Function to display manager details

void displayDetails() const override {

Employee::displayDetails();

std::cout << "Department: " << department << std::endl;

}

};

// Derived class Worker from Employee

class Worker : public Employee {

public:

// Constructor to initialize name, id, and salary for Worker

Worker(const std::string& empName, int empId, double empSalary)

: Employee(empName, empId, empSalary) {}

// Function to display worker details

void displayDetails() const override {

Employee::displayDetails();

}

};

int main() {

// Create instances of Manager and Worker

Manager manager("John Doe", 12345, 60000, "Engineering");

Worker worker("Jane Smith", 67890, 40000);

// Display details of Manager and Worker

std::cout << "Manager Details:" << std::endl;

manager.displayDetails();

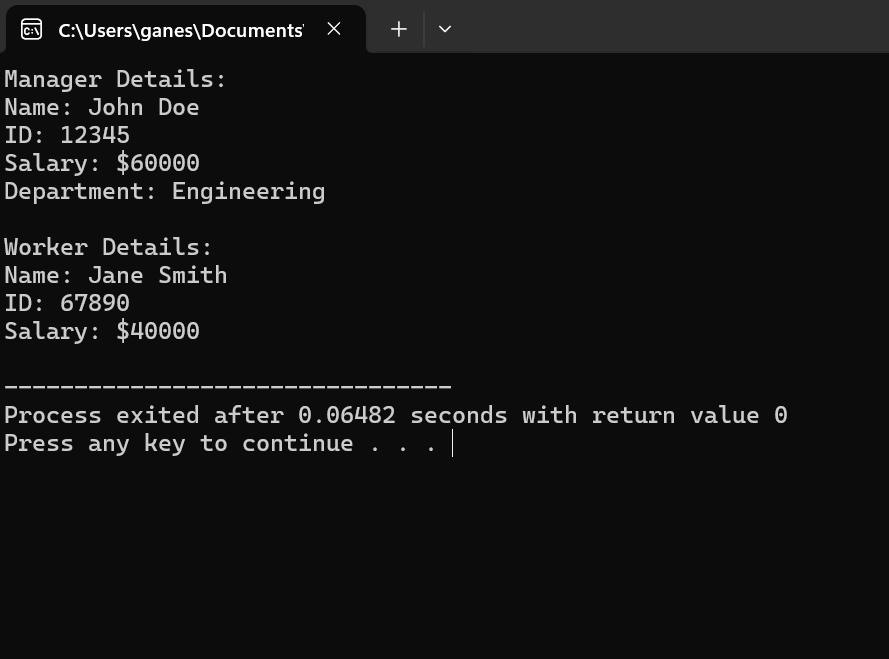
std::cout << std::endl;

std::cout << "Worker Details:" << std::endl;

worker.displayDetails();

return 0;

}



19.

#include <iostream>

// Base class Shape

class Shape {

public:

// Virtual function draw

virtual void draw() const = 0;

};

// Derived class Square from Shape

class Square : public Shape {

public:

// Implementation of draw function for Square

void draw() const override {

std::cout << "Drawing a square" << std::endl;

}

};

// Derived class Circle from Shape

class Circle : public Shape {

public:

// Implementation of draw function for Circle

void draw() const override {

std::cout << "Drawing a circle" << std::endl;

}

};

// Derived class Triangle from Shape

class Triangle : public Shape {

public:

// Implementation of draw function for Triangle

void draw() const override {

std::cout << "Drawing a triangle" << std::endl;

}

};

int main() {

// Create instances of Square, Circle, and Triangle

Square square;

Circle circle;

Triangle triangle;

// Draw each shape

std::cout << "Drawing shapes:" << std::endl;

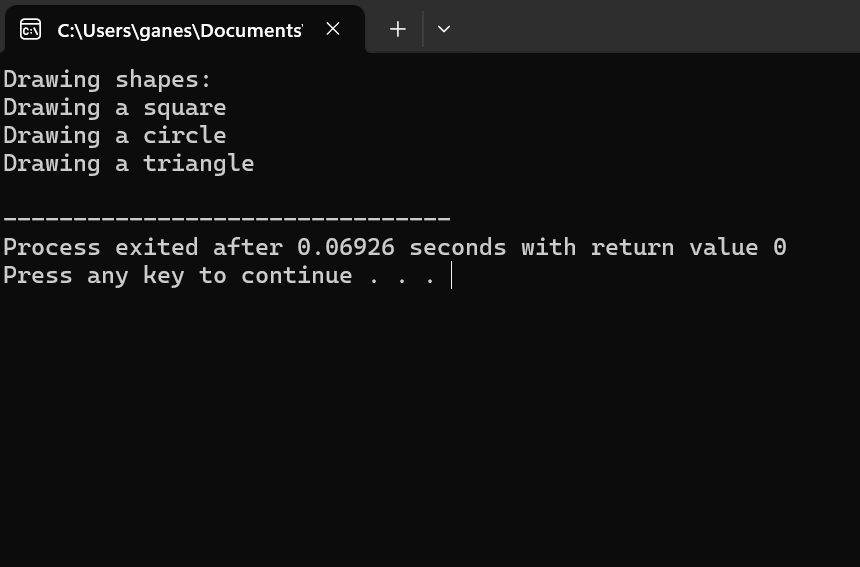
square.draw();

circle.draw();

triangle.draw();

return 0;

}



20.

#include <iostream>

class BankAccount {

protected:

int accountNumber;

double balance;

public:

BankAccount(int accNumber, double bal) : accountNumber(accNumber), balance(bal) {}

void deposit(double amount) {

balance += amount;

std::cout << "Deposited $" << amount << ". New balance: $" << balance << std::endl;

}

virtual void withdraw(double amount) {

if (amount <= balance) {

balance -= amount;

std::cout << "Withdrew $" << amount << ". New balance: $" << balance << std::endl;

} else {

std::cout << "Insufficient funds!" << std::endl;

}

}

};

class SavingsAccount : public BankAccount {

double interestRate;

public:

SavingsAccount(int accNumber, double bal, double rate) : BankAccount(accNumber, bal), interestRate(rate) {}

void calculateInterest() {

double interestAmount = balance \* (interestRate / 100);

deposit(interestAmount);

std::cout << "Interest added: $" << interestAmount << std::endl;

}

};

class CurrentAccount : public BankAccount {

double overdraftLimit;

public:

CurrentAccount(int accNumber, double bal, double limit) : BankAccount(accNumber, bal), overdraftLimit(limit) {}

void withdraw(double amount) override {

if (amount <= balance + overdraftLimit) {

balance -= amount;

std::cout << "Withdrew $" << amount << ". New balance: $" << balance << std::endl;

} else {

std::cout << "Transaction declined: Exceeds overdraft limit!" << std::endl;

}

}

};

int main() {

SavingsAccount savings(12345, 1000, 5.0);

CurrentAccount current(54321, 500, 200);

savings.deposit(500);

savings.calculateInterest();

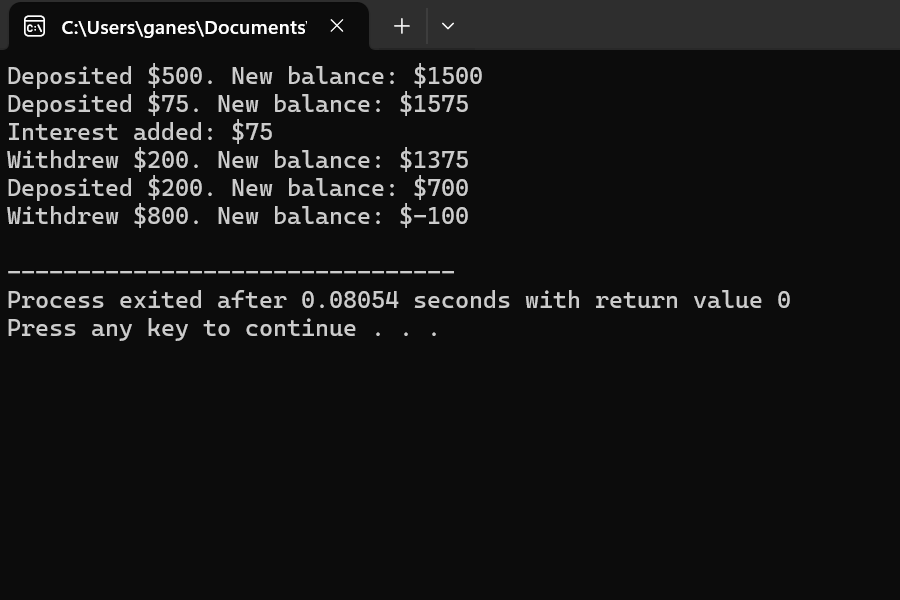
savings.withdraw(200);

current.deposit(200);

current.withdraw(800);

return 0;

}



21.

#include <iostream>

class Shape {

public:

virtual double calculateArea() const = 0;

virtual double calculatePerimeter() const = 0;

};

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double len, double wid) : length(len), width(wid) {}

double calculateArea() const override {

return length \* width;

}

double calculatePerimeter() const override {

return 2 \* (length + width);

}

};

class Circle : public Shape {

private:

double radius;

public:

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

double calculateArea() const override {

return 3.14159 \* radius \* radius; // Assuming Pi value as 3.14159

}

double calculatePerimeter() const override {

return 2 \* 3.14159 \* radius; // Assuming Pi value as 3.14159

}

};

int main() {

Rectangle rectangle(5, 4);

Circle circle(3);

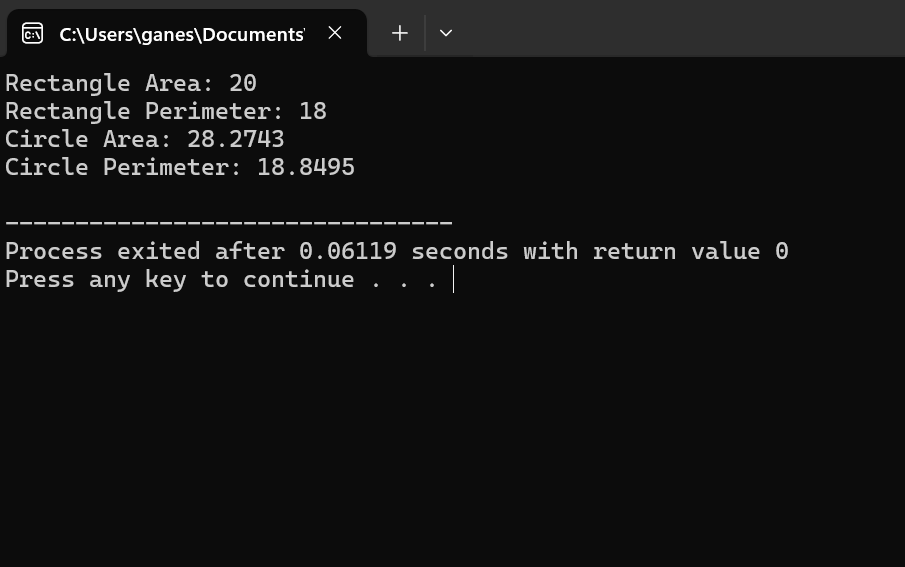
std::cout << "Rectangle Area: " << rectangle.calculateArea() << std::endl;

std::cout << "Rectangle Perimeter: " << rectangle.calculatePerimeter() << std::endl;

std::cout << "Circle Area: " << circle.calculateArea() << std::endl;

std::cout << "Circle Perimeter: " << circle.calculatePerimeter() << std::endl;

return 0;



22.

#include <iostream>

#include <string>

class Student {

private:

std::string name;

int age;

public:

Student(const std::string& n, int a) : name(n), age(a) {}

void displayDetails() {

std::cout << "Name: " << this->name << std::endl;

std::cout << "Age: " << this->age << std::endl;

}

};

int main() {

// Dynamically allocate memory for a Student object

Student\* ptrStudent = new Student("John Doe", 20);

// Use the pointer to call the member function to display details

ptrStudent->displayDetails();

// Don't forget to free the allocated memory

delete ptrStudent;

return 0;

}

A screenshot of a computer

Description automatically generated

23.

#include <iostream>

#include <cmath>

// Base class Shape

class Shape {

public:

// Pure virtual function draw

virtual void draw() const = 0;

};

// Derived class Circle from Shape

class Circle : public Shape {

private:

double radius;

public:

// Constructor

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

// Implementing draw() function for Circle

void draw() const override {

std::cout << "Drawing a circle with radius " << radius << std::endl;

}

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

private:

double length;

double width;

public:

// Constructor

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

// Implementing draw() function for Rectangle

void draw() const override {

std::cout << "Drawing a rectangle with length " << length << " and width " << width << std::endl;

}

};

// Derived class Triangle from Shape

class Triangle : public Shape {

private:

double side1;

double side2;

double side3;

public:

// Constructor

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

// Implementing draw() function for Triangle

void draw() const override {

std::cout << "Drawing a triangle with sides " << side1 << ", " << side2 << ", and " << side3 << std::endl;

}

};

int main() {

// Create an array of pointers to objects of type Shape

const int numShapes = 3;

Shape\* shapes[numShapes];

// Dynamically allocate objects of derived classes and assign pointers to the array

shapes[0] = new Circle(5);

shapes[1] = new Rectangle(4, 6);

shapes[2] = new Triangle(3, 4, 5);

// Call draw() function for each shape using the array of pointers

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

shapes[i]->draw();

}

// Cleanup: deallocate memory

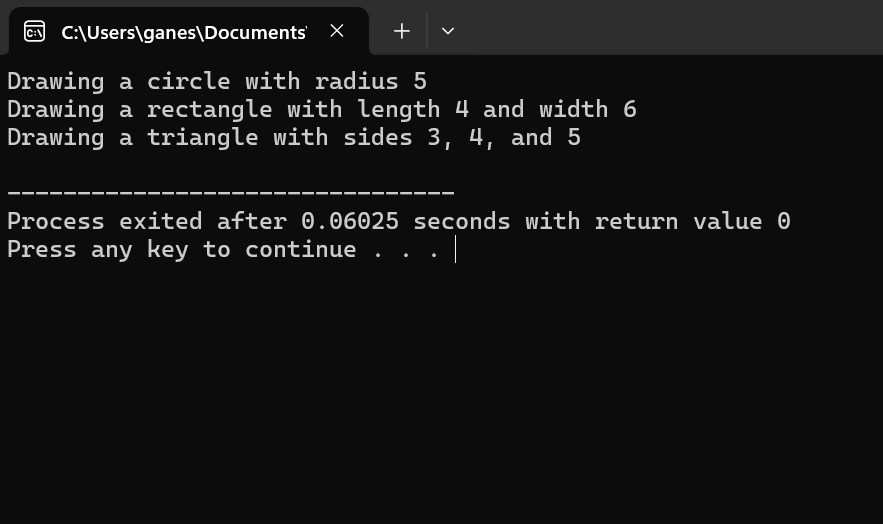
for (int i = 0; i < numShapes; ++i) {

delete shapes[i];

}

return 0;

}



24.

#include <iostream>

// Base class Vehicle

class Vehicle {

protected:

int speed;

int fuel;

public:

// Constructor

Vehicle(int s, int f) : speed(s), fuel(f) {}

// Virtual function to print details

virtual void printDetails() const = 0;

};

// Derived class Car from Vehicle

class Car : public Vehicle {

public:

// Constructor

Car(int s, int f) : Vehicle(s, f) {}

// Implementing printDetails() function for Car

void printDetails() const override {

std::cout << "Car Details: Speed: " << speed << " km/h, Fuel: " << fuel << " liters" << std::endl;

}

};

// Derived class Bike from Vehicle

class Bike : public Vehicle {

public:

// Constructor

Bike(int s, int f) : Vehicle(s, f) {}

// Implementing printDetails() function for Bike

void printDetails() const override {

std::cout << "Bike Details: Speed: " << speed << " km/h, Fuel: " << fuel << " liters" << std::endl;

}

};

int main() {

// Create an array of pointers to objects of type Vehicle

const int numVehicles = 2;

Vehicle\* vehicles[numVehicles];

// Dynamically allocate objects of derived classes and assign pointers to the array

vehicles[0] = new Car(100, 30);

vehicles[1] = new Bike(60, 10);

// Call printDetails() function for each vehicle using the array of pointers

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

vehicles[i]->printDetails();

}

// Cleanup: deallocate memory

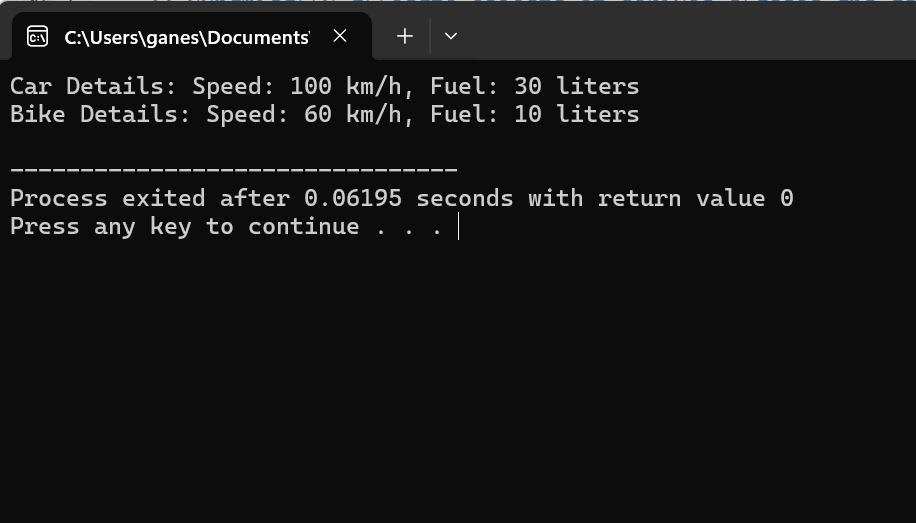
for (int i = 0; i < numVehicles; ++i) {

delete vehicles[i];

}

return 0;

}



25.

#include <iostream>

#include <string>

// Class Book

class Book {

private:

std::string title;

std::string author;

public:

// Constructor

Book(const std::string& t, const std::string& a) : title(t), author(a) {}

// Function to display details

void displayDetails() const {

std::cout << "Book Details: Title: " << title << ", Author: " << author << std::endl;

}

};

int main() {

// Dynamically allocate memory for a Book object

Book\* ptrBook = new Book("Harry Potter", "J.K. Rowling");

// Access attributes of the book and display details using the pointer

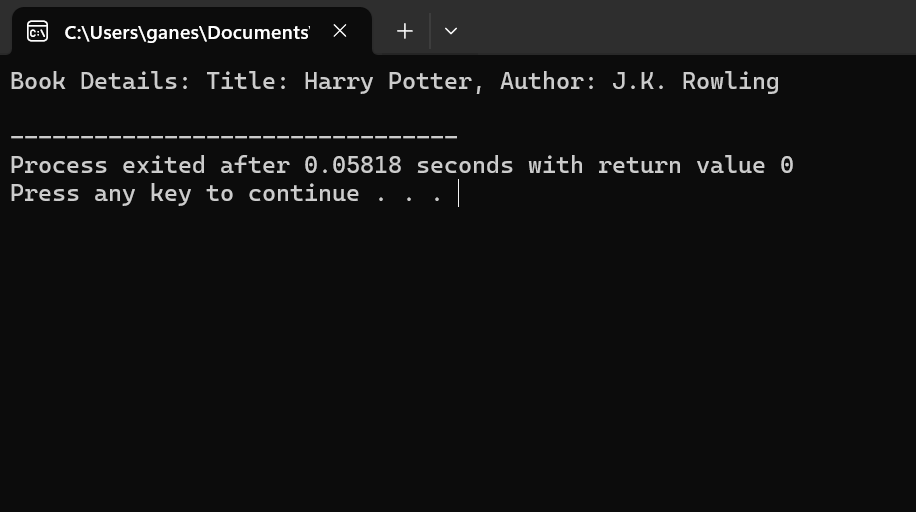
ptrBook->displayDetails();

// Cleanup: deallocate memory

delete ptrBook;

return 0;

}



**Hard**

**26.**

#include <iostream>

#include <string>

// Base class BankAccount

class BankAccount {

protected:

std::string accountNumber;

double balance;

public:

// Constructor

BankAccount(const std::string& accNum, double bal) : accountNumber(accNum), balance(bal) {}

// Member function to deposit money

void deposit(double amount) {

balance += amount;

std::cout << "Deposited $" << amount << " into account " << accountNumber << std::endl;

}

// Member function to withdraw money

virtual void withdraw(double amount) = 0;

};

// Derived class SavingsAccount from BankAccount

class SavingsAccount : public BankAccount {

public:

// Constructor

SavingsAccount(const std::string& accNum, double bal) : BankAccount(accNum, bal) {}

// Implementing withdraw() function for SavingsAccount

void withdraw(double amount) override {

if (balance >= amount) {

balance -= amount;

std::cout << "Withdrawn $" << amount << " from savings account " << accountNumber << std::endl;

} else {

std::cout << "Insufficient funds to withdraw $" << amount << " from savings account " << accountNumber << std::endl;

}

}

};

// Derived class CurrentAccount from BankAccount

class CurrentAccount : public BankAccount {

public:

// Constructor

CurrentAccount(const std::string& accNum, double bal) : BankAccount(accNum, bal) {}

// Implementing withdraw() function for CurrentAccount

void withdraw(double amount) override {

if (balance >= amount) {

balance -= amount;

std::cout << "Withdrawn $" << amount << " from current account " << accountNumber << std::endl;

} else {

std::cout << "Insufficient funds to withdraw $" << amount << " from current account " << accountNumber << std::endl;

}

}

};

int main() {

// Create a SavingsAccount object

SavingsAccount savings("SA123456", 5000);

savings.deposit(1000);

savings.withdraw(2000);

// Create a CurrentAccount object

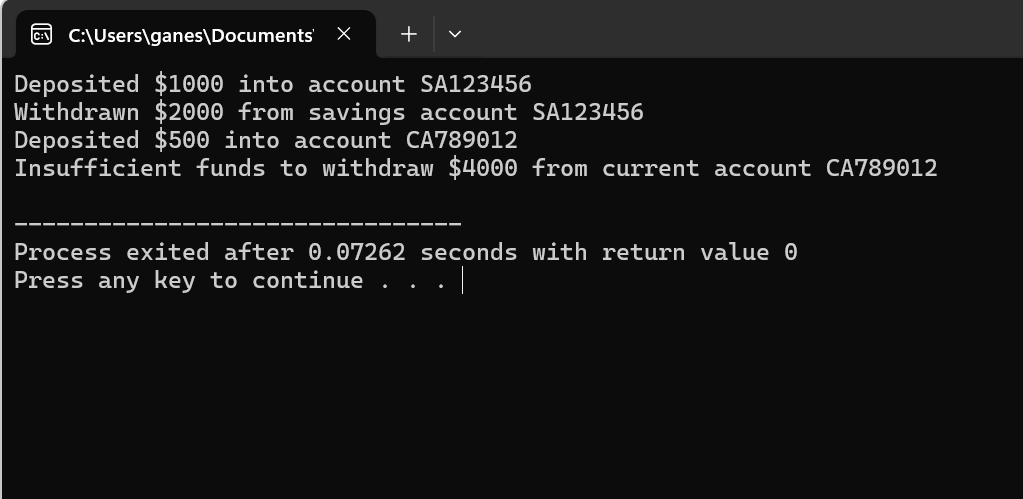
CurrentAccount current("CA789012", 3000);

current.deposit(500);

current.withdraw(4000);

return 0;

}

****

27.

#include <iostream>

#include <string>

// Base class Person

class Person {

protected:

std::string name;

int age;

std::string gender;

public:

// Constructor to initialize attributes

Person(const std::string& n, int a, const std::string& g) : name(n), age(a), gender(g) {}

// Member function to display details

void displayDetails() const {

std::cout << "Name: " << name << ", Age: " << age << ", Gender: " << gender;

}

};

// Derived class Student from Person

class Student : public Person {

private:

int rollNumber;

double marks;

public:

// Constructor to initialize attributes

Student(const std::string& n, int a, const std::string& g, int roll, double m)

: Person(n, a, g), rollNumber(roll), marks(m) {}

// Member function to display details

void displayDetails() const {

// Call displayDetails() of the base class to display name, age, and gender

Person::displayDetails();

std::cout << ", Roll Number: " << rollNumber << ", Marks: " << marks << std::endl;

}

};

int main() {

// Create a Person object

Person person("John Doe", 30, "Male");

// Display information about the person

std::cout << "Person Info: ";

person.displayDetails();

// Create a Student object

Student student("Alice Smith", 20, "Female", 12345, 95.5);

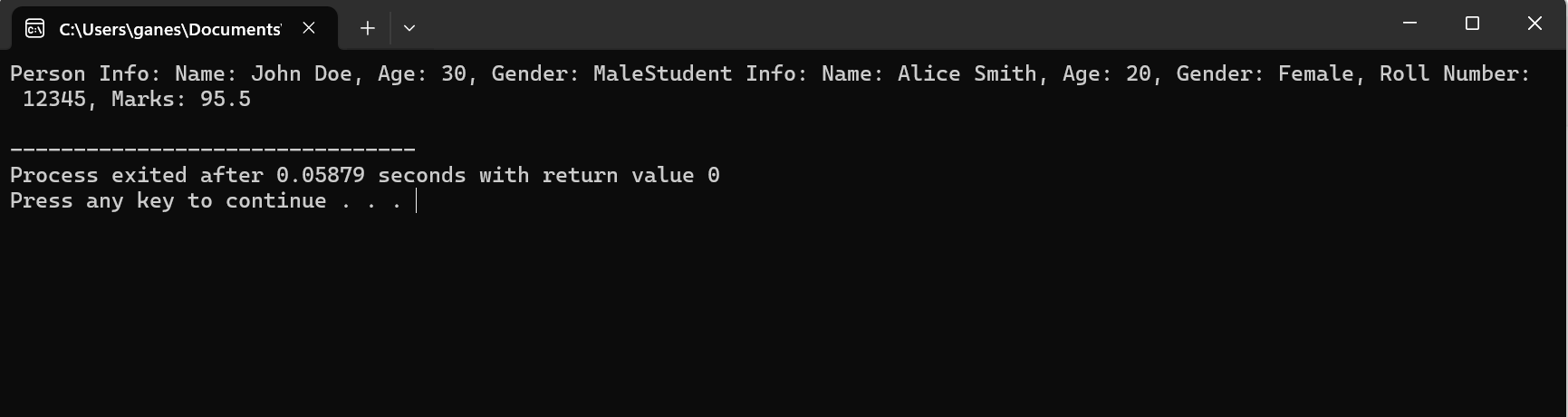
// Display information about the student

std::cout << "Student Info: ";

student.displayDetails();

return 0;

}



28.

#include <iostream>

#include <cmath>

// Base class Shape

class Shape {

public:

// Virtual function to draw the shape

virtual void draw() const = 0;

// Virtual function to calculate the area of the shape

virtual double calculateArea() const = 0;

};

// Derived class Circle from Shape

class Circle : public Shape {

private:

double radius;

public:

// Constructor

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

// Implementing draw() function for Circle

void draw() const override {

std::cout << "Drawing a circle with radius " << radius << std::endl;

}

// Implementing calculateArea() function for Circle

double calculateArea() const override {

return M\_PI \* radius \* radius;

}

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

private:

double length;

double width;

public:

// Constructor

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

// Implementing draw() function for Rectangle

void draw() const override {

std::cout << "Drawing a rectangle with length " << length << " and width " << width << std::endl;

}

// Implementing calculateArea() function for Rectangle

double calculateArea() const override {

return length \* width;

}

};

// Derived class Triangle from Shape

class Triangle : public Shape {

private:

double base;

double height;

public:

// Constructor

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

// Implementing draw() function for Triangle

void draw() const override {

std::cout << "Drawing a triangle with base " << base << " and height " << height << std::endl;

}

// Implementing calculateArea() function for Triangle

double calculateArea() const override {

return 0.5 \* base \* height;

}

};

int main() {

const int numShapes = 3;

// Create an array of pointers to objects of type Shape

Shape\* shapes[numShapes];

// Dynamically allocate objects of each derived class and assign pointers to the array

shapes[0] = new Circle(5);

shapes[1] = new Rectangle(4, 6);

shapes[2] = new Triangle(3, 4);

// Call draw() function for each shape using the array of pointers

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

shapes[i]->draw();

}

// Calculate and display the area of each shape

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

std::cout << "Area of shape " << i + 1 << ": " << shapes[i]->calculateArea() << std::endl;

}

// Cleanup: deallocate memory

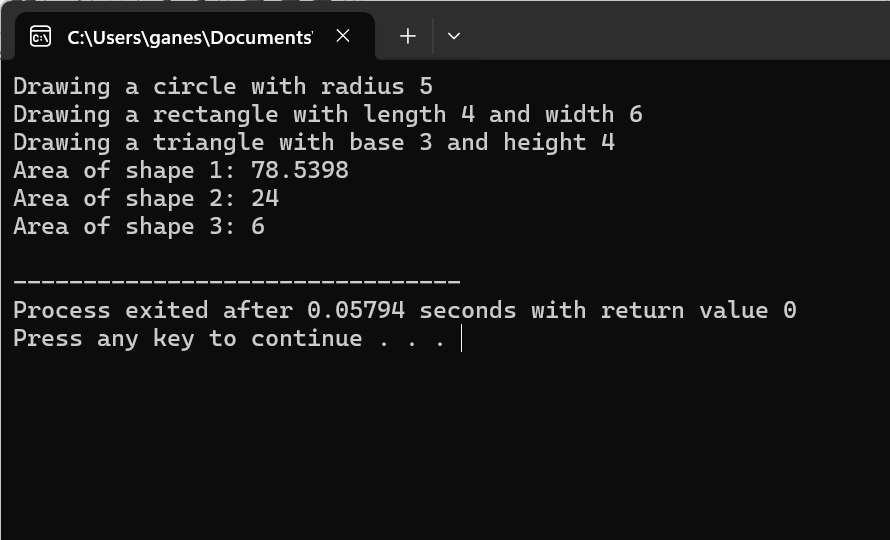
for (int i = 0; i < numShapes; ++i) {

delete shapes[i];

}

return 0;

}



29.

#include <iostream>

#include <string>

// Base class Employee

class Employee {

protected:

int id;

std::string name;

public:

// Constructor

Employee(int i, const std::string& n) : id(i), name(n) {}

// Member function to print details

virtual void printDetails() const {

std::cout << "Employee Details: ID: " << id << ", Name: " << name;

}

};

// Derived class Manager from Employee

class Manager : public Employee {

private:

std::string department;

public:

// Constructor

Manager(int i, const std::string& n, const std::string& d) : Employee(i, n), department(d) {}

// Override printDetails() function to include department

void printDetails() const override {

Employee::printDetails();

std::cout << ", Department: " << department << std::endl;

}

};

int main() {

// Create a pointer to Employee and dynamically allocate memory for a Manager object

Employee\* empPtr = new Manager(101, "John", "HR");

// Call printDetails() function using the pointer

empPtr->printDetails();

// Cleanup: deallocate memory

delete empPtr;

return 0;

}

A screenshot of a computer

Description automatically generated

30.

#include <iostream>

class Vector2D {

private:

double x;

double y;

public:

// Constructor

Vector2D(double x\_val = 0.0, double y\_val = 0.0) : x(x\_val), y(y\_val) {}

// Overloaded addition operator

Vector2D operator+(const Vector2D& vec) const {

return Vector2D(x + vec.x, y + vec.y);

}

// Overloaded subtraction operator

Vector2D operator-(const Vector2D& vec) const {

return Vector2D(x - vec.x, y - vec.y);

}

// Overloaded scalar multiplication operator

Vector2D operator\*(double scalar) const {

return Vector2D(x \* scalar, y \* scalar);

}

// Display function

void display() const {

std::cout << "(" << x << ", " << y << ")";

}

};

int main() {

// Create two Vector2D objects

Vector2D vec1(3.0, 4.0);

Vector2D vec2(1.0, 2.0);

// Perform vector operations using pointers

Vector2D\* sum = new Vector2D(vec1 + vec2);

Vector2D\* diff = new Vector2D(vec1 - vec2);

Vector2D\* scaled = new Vector2D(vec1 \* 2.0);

// Display results

std::cout << "Vector1: ";

vec1.display();

std::cout << std::endl;

std::cout << "Vector2: ";

vec2.display();

std::cout << std::endl;

std::cout << "Vector1 + Vector2: ";

sum->display();

std::cout << std::endl;

std::cout << "Vector1 - Vector2: ";

diff->display();

std::cout << std::endl;

std::cout << "Vector1 \* 2.0: ";

scaled->display();

std::cout << std::endl;

// Cleanup: deallocate memory

delete sum;

delete diff;

delete scaled;

return 0;

}

A screenshot of a computer program

Description automatically generated

31.

#include <iostream>

class Node {

public:

int data;

Node\* left;

Node\* right;

// Constructor

Node(int value) : data(value), left(nullptr), right(nullptr) {}

};

class BST {

private:

Node\* root;

// Helper function to insert a node

Node\* insertNode(Node\* root, int value) {

if (root == nullptr) {

return new Node(value);

}

if (value < root->data) {

root->left = insertNode(root->left, value);

} else if (value > root->data) {

root->right = insertNode(root->right, value);

}

return root;

}

// Helper function to search for a node

bool searchNode(Node\* root, int value) {

if (root == nullptr) {

return false;

}

if (root->data == value) {

return true;

} else if (value < root->data) {

return searchNode(root->left, value);

} else {

return searchNode(root->right, value);

}

}

// Helper function to find the minimum value node in a subtree

Node\* findMin(Node\* root) {

while (root->left != nullptr) {

root = root->left;

}

return root;

}

// Helper function to delete a node

Node\* deleteNode(Node\* root, int value) {

if (root == nullptr) {

return root;

}

if (value < root->data) {

root->left = deleteNode(root->left, value);

} else if (value > root->data) {

root->right = deleteNode(root->right, value);

} else {

// Node with only one child or no child

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

Node\* temp = findMin(root->right);

// Copy the inorder successor's content to this node

root->data = temp->data;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->data);

}

return root;

}

public:

// Constructor

BST() : root(nullptr) {}

// Function to insert a node

void insert(int value) {

root = insertNode(root, value);

}

// Function to search for a node

bool search(int value) {

return searchNode(root, value);

}

// Function to delete a node

void remove(int value) {

root = deleteNode(root, value);

}

// Function to print inorder traversal of the tree

void inorderTraversal(Node\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

std::cout << root->data << " ";

inorderTraversal(root->right);

}

}

// Function to print inorder traversal of the tree

void printInorder() {

inorderTraversal(root);

std::cout << std::endl;

}

};

int main() {

BST tree;

// Insert elements into the tree

tree.insert(50);

tree.insert(30);

tree.insert(70);

tree.insert(20);

tree.insert(40);

tree.insert(60);

tree.insert(80);

// Print inorder traversal of the tree

std::cout << "Inorder traversal of the tree: ";

tree.printInorder();

// Search for elements

std::cout << "Search for 70: " << (tree.search(70) ? "Found" : "Not found") << std::endl;

std::cout << "Search for 90: " << (tree.search(90) ? "Found" : "Not found") << std::endl;

// Remove elements

tree.remove(30);

tree.remove(80);

// Print inorder traversal of the modified tree

std::cout << "Inorder traversal of the modified tree: ";

tree.printInorder();

return 0;

}

32.

#include <iostream>

#include <vector>

class Matrix {

private:

int rows;

int cols;

std::vector<std::vector<int>> data;

public:

// Constructor

Matrix(int m, int n) : rows(m), cols(n), data(std::vector<std::vector<int>>(m, std::vector<int>(n, 0))) {}

// Overloaded addition operator

Matrix operator+(const Matrix& other) const {

if (rows != other.rows || cols != other.cols) {

std::cerr << "Error: Matrix dimensions do not match for addition." << std::endl;

return \*this;

}

Matrix result(rows, cols);

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

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

result.data[i][j] = data[i][j] + other.data[i][j];

}

}

return result;

}

// Overloaded subtraction operator

Matrix operator-(const Matrix& other) const {

if (rows != other.rows || cols != other.cols) {

std::cerr << "Error: Matrix dimensions do not match for subtraction." << std::endl;

return \*this;

}

Matrix result(rows, cols);

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

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

result.data[i][j] = data[i][j] - other.data[i][j];

}

}

return result;

}

// Overloaded multiplication operator

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

if (cols != other.rows) {

std::cerr << "Error: Matrix dimensions do not match for multiplication." << std::endl;

return \*this;

}

Matrix result(rows, other.cols);

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

for (int j = 0; j < other.cols; ++j) {

for (int k = 0; k < cols; ++k) {

result.data[i][j] += data[i][k] \* other.data[k][j];

}

}

}

return result;

}

// Function to display matrix

void display() const {

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

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

std::cout << data[i][j] << " ";

}

std::cout << std::endl;

}

}

};

int main() {

// Create two matrices

Matrix mat1(2, 3);

Matrix mat2(3, 2);

// Initialize matrices

int value = 1;

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

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

mat1.data[i][j] = value++;

mat2.data[j][i] = value++;

}

}

// Perform matrix operations

Matrix sum = mat1 + mat2;

Matrix diff = mat1 - mat2;

Matrix product = mat1 \* mat2;

// Display results

std::cout << "Matrix 1:" << std::endl;

mat1.display();

std::cout << "Matrix 2:" << std::endl;

mat2.display();

std::cout << "Sum:" << std::endl;

sum.display();

std::cout << "Difference:" << std::endl;

diff.display();

std::cout << "Product:" << std::endl;

product.display();

return 0;

}