PROGRAMS

DATE:18/03/24

**1-**Create a base class Shape with a member function getArea(). Derive classes -Rectangle and Circle from Shape. Implement the getArea() function in each derived class to calculate the area of a rectangle and a circle, respectively.

#include <iostream>

class Shape {

public:

virtual double getArea() const = 0;

};

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double \_length, double \_width) : length(\_length), width(\_width) {}

double getArea() const override {

return length \* width;

}

};

class Circle : public Shape {

private:

double radius;

public:

Circle(double \_radius) : radius(\_radius) {}

double getArea() const override {

return 3.14159265358979323846 \* radius \* radius;

}

};

int main() {

Rectangle rectangle(5, 4);

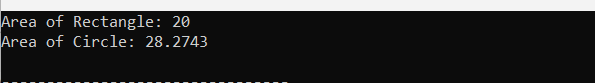
Circle circle(3);

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

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

return 0;

}



2-**.** Define a base class Animal with a virtual function makeSound(). Derive classes Dog and Cat from Animal. Implement the makeSound() function in each derived class to output appropriate sounds for a dog and a cat.

#include <iostream>

using namespace std;

class Animal {

public:

virtual void makeSound() const {

cout << "Some generic sound from an animal" <<endl;

}

};

class Dog : public Animal {

public:

void makeSound() const override {

cout << "Woof! Woof!" <<endl;

}

};

class Cat : public Animal {

public:

void makeSound() const override {

cout << "Meow! Meow!" <<endl;

}

};

int main() {

Animal \*animal1 = new Dog();

Animal \*animal2 = new Cat();

animal1->makeSound();

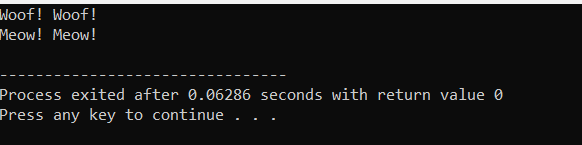
animal2->makeSound();

delete animal1;

delete animal2;

return 0;

}



3-Create a base class **Employee** with attributes **name** and **salary**. Derive classes **Manager** and **Worker** from **Employee**. Add an extra attribute **department** for the **Manager** class. Implement constructors in each class to initialize the attributes.

#include <iostream>

#include <string>

using namespace std;

class Employee {

protected:

string name;

double salary;

public:

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

void display() {

cout << "Name: " << name << ", Salary: $" << salary << endl;

}

};

class Manager : public Employee {

private:

string department;

public:

Manager(const string& name, double salary, const string& department) : Employee(name, salary), department(department) {}

void display() {

cout << "Name: " << name << ", Salary: $" << salary << ", Department: " << department << endl;

}

};

class Worker : public Employee {

public:

Worker(const string& name, double salary) : Employee(name, salary) {}

};

int main() {

Manager manager("John Doe", 50000.0, "Engineering");

Worker worker("Jane Smith", 30000.0);

cout << "Manager: ";

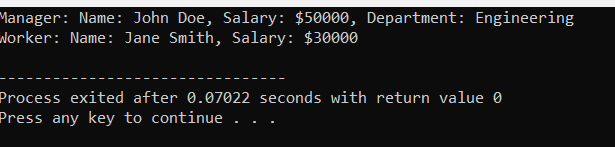
manager.display();

cout << "Worker: ";

worker.display();

return 0;

}



4-Define a base class **Shape** with a virtual function **draw()**. Derive classes **Circle**, **Square**, and **Triangle** from **Shape**. Implement the **draw()** function in each derived class to draw respective shapes using cout statements.

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() const = 0; // Pure virtual function

};

class Circle : public Shape {

public:

void draw() const override {

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

}

};

class Square : public Shape {

public:

void draw() const override {

cout << "Drawing a square." << endl;

}

};

class Triangle : public Shape {

public:

void draw() const override {

cout << "Drawing a triangle." << endl;

}

};

int main() {

Circle;

Square;

Triangle triangle;

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

cout << "--------------" << endl;

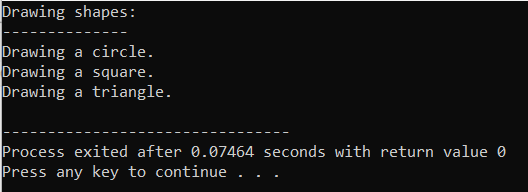
circle.draw();

square.draw();

triangle.draw();

return 0;

}



5-Create a base class **Vehicle** with attributes **speed** and **fuel**. Derive classes **Car** and **Bike** from **Vehicle**. Implement constructors in each class to initialize the attributes.

#include <iostream>

class Vehicle {

protected:

int speed;

int fuel;

public:

Vehicle(int \_speed = 0, int \_fuel = 100) : speed(\_speed), fuel(\_fuel) {}

void accelerate(int amount) {

speed += amount;

}

void decelerate(int amount) {

speed -= amount;

}

void refuel(int amount) {

fuel += amount;

}

int getSpeed() const {

return speed;

}

int getFuel() const {

return fuel;

}

};

class Car : public Vehicle {

private:

int numWheels;

public:

Car(int \_speed = 0, int \_fuel = 100, int \_numWheels = 4)

: Vehicle(\_speed, \_fuel), numWheels(\_numWheels) {}

int getNumWheels() const {

return numWheels;

}

};

class Bike : public Vehicle {

private:

int numWheels;

public:

Bike(int \_speed = 0, int \_fuel = 100, int \_numWheels = 2)

: Vehicle(\_speed, \_fuel), numWheels(\_numWheels) {}

int getNumWheels() const {

return numWheels;

}

};

int main() {

Car;

std::cout << "Car speed: " << car.getSpeed() << std::endl;

std::cout << "Car fuel: " << car.getFuel() << std::endl;

std::cout << "Car wheels: " << car.getNumWheels() << std::endl;

Bike bike;

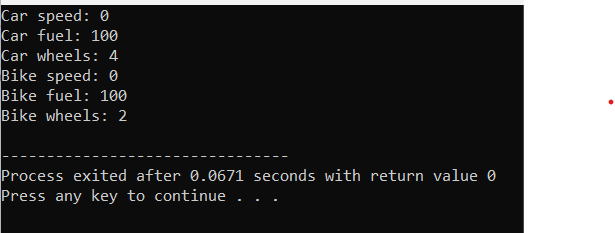
std::cout << "Bike speed: " << bike.getSpeed() << std::endl;

std::cout << "Bike fuel: " << bike.getFuel() << std::endl;

std::cout << "Bike wheels: " << bike.getNumWheels() << std::endl;

return 0;

}



1. 6-**.** Define a base class Bird with a virtual function fly(). Derive classes Eagle and Sparrow from Bird. Implement the fly() function in each derived class to output flying behavior.

#include <iostream>

using namespace std;

class Bird {

public:

virtual void fly() {

cout << "Bird flying\n";

}

};

class Eagle : public Bird {

public:

void fly() override {

cout << "Eagle soaring high\n";

}

};

class Sparrow : public Bird {

public:

void fly() override {

cout << "Sparrow fluttering around\n";

}

};

int main() {

Bird\* bird1 = new Eagle();

bird1->fly();

Bird\* bird2 = new Sparrow();

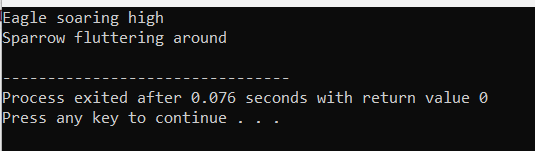
bird2->fly();

delete bird1;

delete bird2;

return 0;

}



7-Create an abstract base class **Shape** with a pure virtual function **calculateArea()**. Derive classes **Rectangle** and **Circle** from **Shape** and implement the **calculateArea()** function in each derived class.

#include <iostream>

class Shape {

public:

virtual float calculateArea() const = 0;

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

Rectangle(float \_length, float \_width) : length(\_length), width(\_width) {}

float calculateArea() const override {

return length \* width;

}

};

class Circle : public Shape {

private:

float radius;

public:

Circle(float \_radius) : radius(\_radius) {}

float calculateArea() const override {

return 3.14159 \* radius \* radius;

}

};

int main() {

Rectangle rectangle(5, 3);

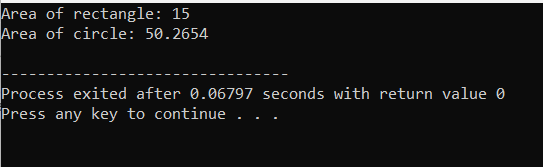
std::cout << "Area of rectangle: " << rectangle.calculateArea() << std::endl;

Circle circle(4);

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

return 0;

}



8-

#include <iostream>

#include <string>

class Person {

protected:

std::string name;

int age;

public:

Person(const std::string& \_name, int \_age) : name(\_name), age(\_age) {}

void display() const {

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

}

};

class Student : public Person {

private:

int grade;

public:

Student(const std::string& \_name, int \_age, int \_grade) : Person(\_name, \_age), grade(\_grade) {}

void display() const {

Person::display();

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

}

};

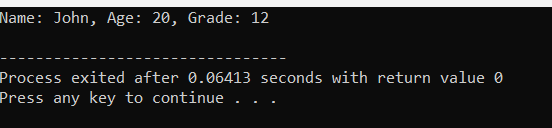
int main() {

Student student("John", 20, 12);

student.display();

return 0;

}



9-Write a C++ program to swap two integers using pointers.

#include <iostream>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int num1 = 5;

int num2 = 10;

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

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

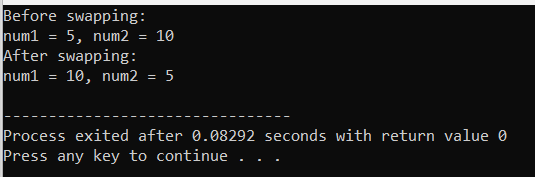
swap(&num1, &num2);

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

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

return 0;

}



10-Create a class **Circle** with a private member **radius**. Write a C++ program to find the area of a circle using a pointer to the object

#include <iostream>

using namespace std;

class Circle {

private:

double radius;

public:

Circle(double \_radius) : radius(\_radius) {}

double calculateArea() const {

return 3.14159 \* radius \* radius;

}

double getRadius() const {

return radius;

}

};

int main() {

Circle\* circlePtr = new Circle(5.0);

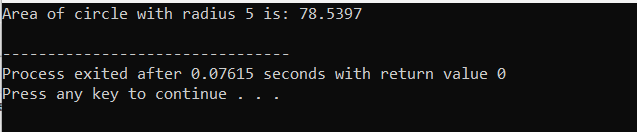
double area = circlePtr->calculateArea();

std::cout << "Area of circle with radius " << circlePtr->getRadius() << " is: " << area << std::endl;

delete circlePtr;

return 0;

}



11-Define a class **Rectangle** with private members **length** and **width**. Write a C++ program to calculate the perimeter of a rectangle using a pointer to an object.

#include <iostream>

class Rectangle {

private:

double length;

double width;

public:

Rectangle(double \_length, double \_width) : length(\_length), width(\_width) {}

double calculatePerimeter() const {

return 2 \* (length + width);

}

double getLength() const {

return length;

}

double getWidth() const {

return width;

}

};

int main() {

Rectangle\* rectanglePtr = new Rectangle(5.0, 3.0);

double perimeter = rectanglePtr->calculatePerimeter();

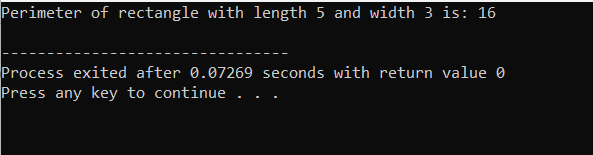
std::cout << "Perimeter of rectangle with length " << rectanglePtr->getLength()

<< " and width " << rectanglePtr->getWidth() << " is: " << perimeter << std::endl;

delete rectanglePtr;

return 0;

}



12-Implement a class **Employee** with private member variables **name** and **salary**. Write a C++ program to display the details of an employee using a pointer to an object.

#include <iostream>

#include <string>

using namespace std;

class Employee {

private:

string name;

double salary;

public:

Employee(string n, double s) : name(n), salary(s) {}

string getName() const {

return name;

}

double getSalary() const {

return salary;

}

};

int main() {

Employee\* empPtr = new Employee("John Doe", 50000.0);

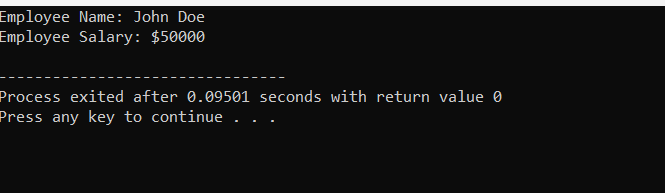
cout << "Employee Name: " << empPtr->getName() << endl;

cout << "Employee Salary: $" << empPtr->getSalary() << endl;

delete empPtr;

return 0;

}



13-Create a class **Student** with private member variables **name** and **grade**. Write a C++ program to display the name and grade of a student using the **this** pointer.

#include <iostream>

#include <string>

class Student {

private:

std::string name;

char grade;

public:

Student(std::string name, char grade) {

this->name = name;

this->grade = grade;

}

void display() {

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

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

}

};

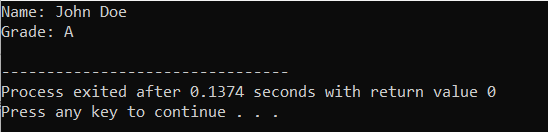
int main() {

Student student1("John Doe", 'A');

student1.display();

return 0;

}



14-Define a base class **Shape** with a virtual function **draw()**. Implement a derived class **Circle** with its own implementation of the **draw()** function. Write a C++ program to demonstrate polymorphism using a pointer to the base class.

#include <iostream>

class Shape {

public:

virtual void draw() {

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

}

};

class Circle : public Shape {

public:

void draw() override {

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

}

};

int main() {

Shape \*shapePtr;

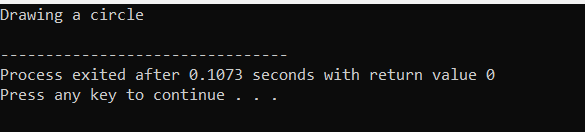
Circle circle;

shapePtr = &circle;

shapePtr->draw();

return 0;

}



15. Create a class **Person** with private member variables **name** and **age**. Implement a derived class **Student** with its own implementation of the **display()** function to display the name and age of a student. Write a C++ program to demonstrate function overriding using a pointer to the base class.

#include <iostream>

#include <string>

class Person {

private:

std::string name;

int age;

std::string country;

public:

void setName(const std::string & n) {

name = n;

}

void setAge(int a) {

age = a;

}

void setCountry(const std::string & c) {

country = c;

}

std::string getName() {

return name;

}

int getAge() {

return age;

}

std::string getCountry() {

return country;

}

};

int main() {

Person person;

person.setName("Saveli Sujatha");

person.setAge(25);

person.setCountry("USA");

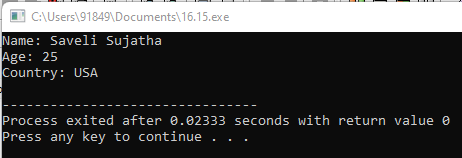
std::cout << "Name: " << person.getName() << std::endl;

std::cout << "Age: " << person.getAge() << std::endl;

std::cout << "Country: " << person.getCountry() << std::endl;

return 0;

}

OUTPUT:  


16. Define a base class Vehicle with attributes speed and fuel. Derive classes Car and Bike from Vehicle. Implement constructors in each class to initialize the attributes. Also, create a member function displayInfo() in each derived class to display the speed and fuel of the vehicle.

#include <iostream>

using namespace std;

class Vehicle {

protected:

int speed;

int fuel;

public:

Vehicle(int \_speed, int \_fuel) : speed(\_speed), fuel(\_fuel) {}

virtual void displayInfo() const {

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

cout << "Fuel Level: " << fuel << " %" <<endl;

}

};

class Car : public Vehicle {

public:

Car(int \_speed, int \_fuel) : Vehicle(\_speed, \_fuel) {}

void displayInfo() const override {

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

cout << "Car Fuel Level: " << fuel << " %" <<endl;

}

};

class Bike : public Vehicle {

public:

Bike(int \_speed, int \_fuel) : Vehicle(\_speed, \_fuel) {}

void displayInfo() const override {

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

cout << "Bike Fuel Level: " << fuel << " %" <<endl;

}

};

int main() {

Car car(120, 70);

Bike bike(80, 50);

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

car.displayInfo();

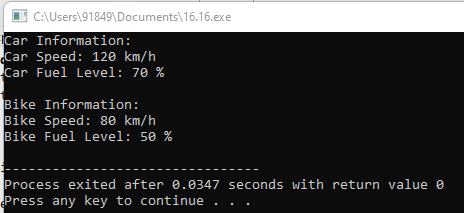
cout << "\nBike Information:" <<endl;

bike.displayInfo();

return 0;

}

OUTPUT:



17. Create a base class Shape with virtual functions area() and perimeter(). Derive classes Rectangle and Circle from Shape. Implement these functions in each derived class to calculate the area and perimeter of a rectangle and a circle, respectively.

#include <iostream>

#include <cmath>

const double PI = 3.14159;

class Shape {

public:

virtual double calculateArea() const = 0;

virtual double calculatePerimeter() const = 0;

};

class Circle: public Shape {

private:

double radius;

public:

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

double calculateArea() const override {

return PI \* pow(radius, 2);

}

double calculatePerimeter() const override {

return 2 \* PI \* radius;

}

};

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);

}

};

int main() {

Circle circle(7.0);

Rectangle rectangle(4.2, 8.0);

cout << "Circle: " ;

cout << "Area: " << circle.calculateArea();

cout << "Perimeter: " << circle.calculatePerimeter() ;

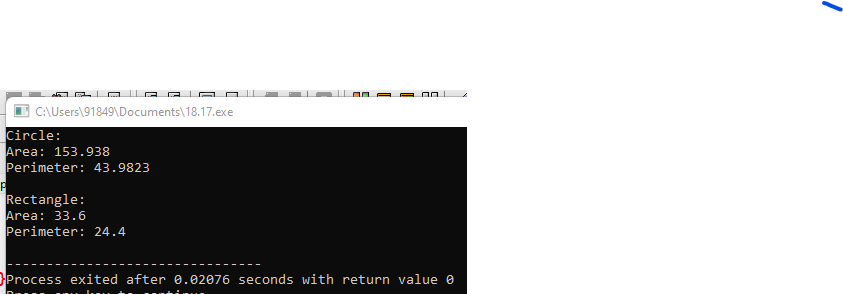
cout << "\nRectangle: " ;

cout << "Area: " << rectangle.calculateArea() ;

cout << "Perimeter: " << rectangle.calculatePerimeter() ;

return 0;

}

OUTPUT: 

18. Create a base class Employee with attributes name, id, and salary. Derive classes Manager and Worker from Employee. Implement constructors in each class to initialize the attributes. Also, implement member functions to display the details of each type of employee.

#include <iostream>

using namespace std;

class employee

{

int emp\_number;

char emp\_name[20];

float emp\_basic;

float emp\_da;

float emp\_it;

float emp\_net\_sal;

public:

void get\_emp\_details();

float find\_net\_salary(float basic, float da, float it);

void show\_emp\_details();

};

void employee :: get\_emp\_details()

{

cout<<"\nEnter employee number: ";

cin>>emp\_number;

cout<<"\nEnter employee name: ";

cin>>emp\_name;

cout<<"\nEnter employee basic: ";

cin>>emp\_basic;

cout<<"\nEnter employee DA: ";

cin>>emp\_da;

cout<<"\nEnter employee IT: ";

cin>>emp\_it;

}

float employee :: find\_net\_salary(float basic, float da, float it)

{

return (basic+da)-it;

}

void employee :: show\_emp\_details()

{

cout<<"\n\n\*\*\*\* Details of Employee \*\*\*\*";

cout<<"\nEmployee Name : "<<emp\_name;

cout<<"\nEmployee number : "<<emp\_number;

cout<<"\nBasic salary : "<<emp\_basic;

cout<<"\nEmployee DA : "<<emp\_da;

cout<<"\nIncome Tax : "<<emp\_it;

cout<<"\nNet Salary : "<<find\_net\_salary(emp\_basic, emp\_da, emp\_it);

cout<<"\n-------------------------------\n\n";

}

int main()

{

employee emp;

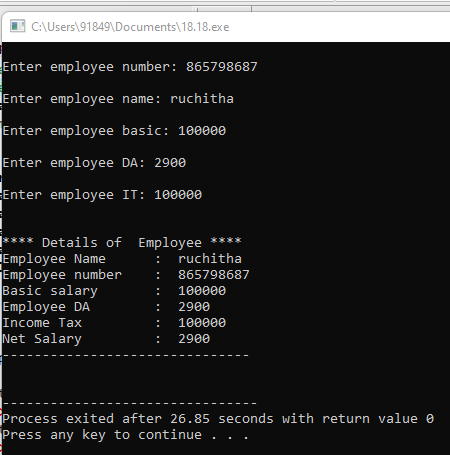
emp.get\_emp\_details();

emp.show\_emp\_details();

return 0;

}

OUTPUT:



19. Define a base class Shape with a virtual function draw(). Derive classes Square, Circle, and Triangle from Shape. Implement the draw() function in each derived class to draw respective shapes using cout statements.

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() {

cout << "Drawing a shape..." << endl;

}

};

class Square : public Shape {

public:

void draw() override {

cout << "Drawing a square..." << endl;

}

};

class Circle : public Shape {

public:

void draw() override {

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

}

};

class Triangle : public Shape {

public:

void draw() override {

cout << "Drawing a triangle..." << endl;

}

};

int main() {

Square;

Circle;

Triangle;

square.draw();

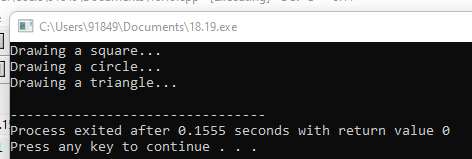
circle.draw();

triangle.draw();

return 0;

}

OUTPUT:



20-Create a base class Person with attributes name and age. Derive class Student from Person with an additional attribute grade. Implement constructors in each class to initialize the attributes. Also, include a member function displayInfo() in each class to display the details of the person or student.

#include <iostream>

#include <string>

using namespace std;

class Person {

protected:

string name;

int age;

public:

Person(string name, int age) : name(name), age(age) {}

virtual void displayInfo() {

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

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

}

};

class Student : public Person {

private:

int grade;

public:

Student(string name, int age, int grade) : Person(name, age), grade(grade) {}

void displayInfo() override {

Person::displayInfo();

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

}

};

int main() {

// Example usage

Person person("John", 25);

Student student("Jane", 20, 11);

cout << "Person's Info:" << endl;

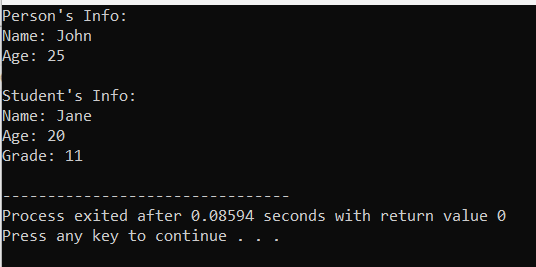
person.displayInfo();

cout << "\nStudent's Info:" << endl;

student.displayInfo();

return 0;

}



21-Define a base class BankAccount with attributes accountNumber and balance. Derive classes SavingsAccount and CurrentAccount from BankAccount. Implement constructors in each class to initialize the attributes. Also, include member functions to deposit and withdraw money.

#include <iostream>

class BankAccount {

protected:

std::string accountNumber;

double balance;

public:

BankAccount(const std::string& accNumber, double initialBalance) : accountNumber(accNumber), balance(initialBalance) {}

void deposit(double amount) {

balance += amount;

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

std::cout << "Current Balance: $" << balance << std::endl;

}

virtual void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

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

std::cout << "Current Balance: $" << balance << std::endl;

} else {

std::cout << "Insufficient balance." << std::endl;

}

}

};

class SavingsAccount : public BankAccount {

public:

SavingsAccount(const std::string& accNumber, double initialBalance) : BankAccount(accNumber, initialBalance) {}

void withdraw(double amount) override {

if (balance - amount >= 100.0) {

BankAccount::withdraw(amount);

} else {

std::cout << "Minimum balance requirement not met." << std::endl;

}

}

};

class CurrentAccount : public BankAccount {

public:

CurrentAccount(const std::string& accNumber, double initialBalance) : BankAccount(accNumber, initialBalance) {}

};

int main() {

SavingsAccount savings("SAV-12345", 1000.0);

CurrentAccount current("CUR-67890", 2000.0);

savings.deposit(500.0);

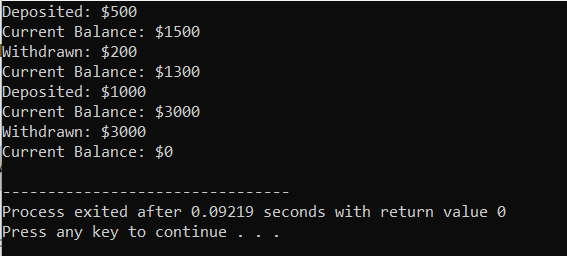
savings.withdraw(200.0);

current.deposit(1000.0);

current.withdraw(3000.0);

return 0;

}



22- Create an abstract base class Shape with pure virtual functions calculateArea() and calculatePerimeter(). Derive classes Rectangle and Circle from Shape and implement these functions in each derived class.

#include <iostream>

class Shape {

public:

virtual double calculateArea() const = 0;

virtual double calculatePerimeter() const = 0;

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

Rectangle(double w, double h) : width(w), height(h) {}

double calculateArea() const override {

return width \* height;

}

double calculatePerimeter() const override {

return 2 \* (width + height);

}

};

class Circle : public Shape {

private:

double radius;

const double PI = 3.14159; // Define PI

public:

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

double calculateArea() const override {

return PI \* radius \* radius;

}

double calculatePerimeter() const override {

return 2 \* PI \* radius;

}

};

int main() {

Rectangle rectangle(5.0, 3.0);

Circle circle(4.0);

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

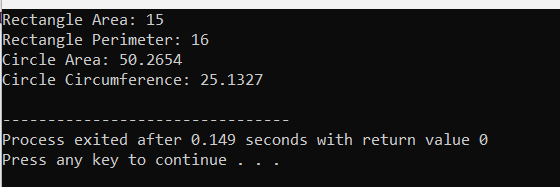
std::cout << "Rectangle Perimeter: " << rectangle.calculatePerimeter() << std::endl;

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

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

return 0;

}



23- Write a C++ program to create a class Student with attributes name and age. Create a pointer to an object of class Student and dynamically allocate memory to it. Use the this pointer to display the details of the student.

Test Cases:

Input: name = "John", age = 20

Output: Student Name: John, Age: 20

#include <iostream>

#include <string>

class Student {

private:

std::string name;

int age;

public:

Student(const std::string& newName, int newAge) : name(newName), age(newAge) {}

void displayDetails() {

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

}

};

int main() {

std::string name = "John";

int age = 20;

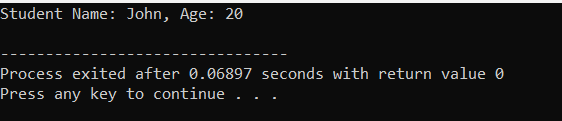
Student\* studentPtr = new Student(name, age);

studentPtr->displayDetails();

delete studentPtr;

return 0;

}



24- Define a base class Shape with a pure virtual function draw(). Derive classes Circle, Rectangle, and Triangle from Shape. Create an array of pointers to objects of type Shape and use them to call the draw() function for each shape.

Test Cases:

Verify if the draw() function is correctly called for each shape and displays the appropriate shape.

#include <iostream>

#include <cmath>

class Shape {

public:

virtual void draw() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

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

void draw() const override {

std::cout << "Drawing Circle with radius " << radius << std::endl;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

Rectangle(double w, double h) : width(w), height(h) {}

void draw() const override {

std::cout << "Drawing Rectangle with width " << width << " and height " << height << std::endl;

}

};

class Triangle : public Shape {

private:

double side1, side2, side3;

public:

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

void draw() const override {

std::cout << "Drawing Triangle with sides " << side1 << ", " << side2 << ", " << side3 << std::endl;

}

};

int main() {

const int numShapes = 3;

Shape\* shapes[numShapes];

shapes[0] = new Circle(5.0);

shapes[1] = new Rectangle(4.0, 6.0);

shapes[2] = new Triangle(3.0, 4.0, 5.0);

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

shapes[i]->draw();

}

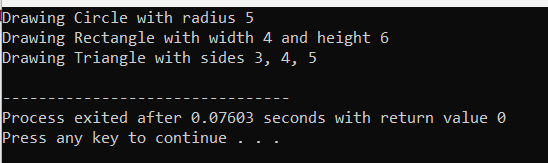
for (int i = 0; i < numShapes; ++i) {

delete shapes[i];

}

return 0;

}



25- Create a base class Vehicle with attributes speed and fuel. Derive classes Car and Bike from Vehicle. Write a function printDetails() in each class to display the details of the vehicle. Create an array of pointers to objects of type Vehicle and use them to call the printDetails() function for each vehicle.

Test Cases:

Input: Car speed = 100 km/h, fuel = 30 liters; Bike speed = 60 km/h, fuel = 10 liters

Output: Car Details: Speed: 100 km/h, Fuel: 30 liters; Bike Details: Speed: 60 km/h, Fuel: 10 liters

#include <iostream>

class Vehicle {

protected:

int speed;

int fuel;

public:

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

virtual void printDetails() const = 0;

};

class Car : public Vehicle {

public:

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

void printDetails() const override {

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

}

};

class Bike : public Vehicle {

public:

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

void printDetails() const override {

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

}

};

int main() {

const int numVehicles = 2;

Vehicle\* vehicles[numVehicles];

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

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

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

vehicles[i]->printDetails();

}

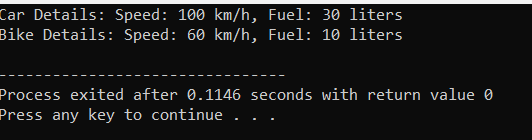
for (int i = 0; i < numVehicles; ++i) {

delete vehicles[i];

}

return 0;

}



26- Define a class Book with attributes title and author. Create a pointer to an object of class Book and dynamically allocate memory to it. Use the pointer to access the attributes of the book and display its details.

Test Cases:

Input: title = "Harry Potter", author = "J.K. Rowling"

Output: Book Details: Title: Harry Potter, Author: J.K. Rowling

**Code:**

#include <iostream>

#include <string>

class Book {

private:

std::string title;

std::string author;

public:

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

void displayDetails() const {

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

}

};

int main() {

std::string title = "Harry Potter";

std::string author = "J.K. Rowling";

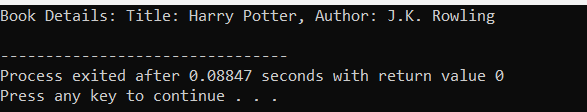
Book\* bookPtr = new Book(title, author);

bookPtr->displayDetails();

delete bookPtr;

return 0;

}



27- .Implement a base class BankAccount with attributes accountNumber and balance. Derive classes SavingsAccount and CurrentAccount from BankAccount. Implement constructors in each class to initialize the attributes. Additionally, include member functions in each derived class to deposit and withdraw money.

#include <iostream>

#include <string>

class BankAccount {

protected:

std::string accountNumber;

double balance;

public:

BankAccount(const std::string& accNumber, double initialBalance) : accountNumber(accNumber), balance(initialBalance) {}

void deposit(double amount) {

balance += amount;

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

std::cout << "Current Balance: $" << balance << std::endl;

}

virtual void withdraw(double amount) = 0;

};

class SavingsAccount : public BankAccount {

public:

SavingsAccount(const std::string& accNumber, double initialBalance) : BankAccount(accNumber, initialBalance) {}

void withdraw(double amount) override {

if (balance >= amount) {

balance -= amount;

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

std::cout << "Current Balance: $" << balance << std::endl;

} else {

std::cout << "Insufficient balance." << std::endl;

}

}

};

class CurrentAccount : public BankAccount {

public:

CurrentAccount(const std::string& accNumber, double initialBalance) : BankAccount(accNumber, initialBalance) {}

void withdraw(double amount) override {

if (balance >= amount) {

balance -= amount;

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

std::cout << "Current Balance: $" << balance << std::endl;

} else {

std::cout << "Insufficient balance." << std::endl;

}

}

};

int main() {

SavingsAccount savings("SAV-12345", 1000.0);

CurrentAccount current("CUR-67890", 2000.0);

savings.deposit(500.0);

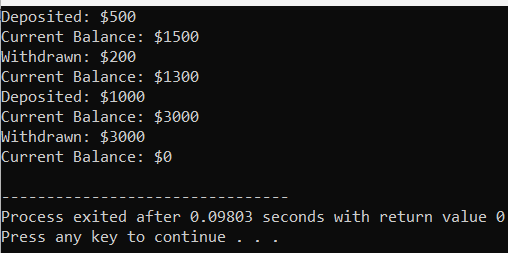
savings.withdraw(200.0);

current.deposit(1000.0);

current.withdraw(3000.0);

return 0;

}



28- Create a base class Person with attributes name, age, and gender. Derive class Student from Person with additional attributes rollNumber and marks. Implement constructors in each class to initialize the attributes. Also, include member functions to display the details of the person or student.

#include <iostream>

#include <string>

class Person {

protected:

std::string name;

int age;

char gender;

public:

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

void displayDetails() const {

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

}

};

class Student : public Person {

private:

std::string rollNumber;

double marks;

public:

Student(const std::string& n, int a, char g, const std::string& roll, double m) : Person(n, a, g), rollNumber(roll), marks(m) {}

void displayDetails() const {

Person::displayDetails();

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

}

};

int main() {

Person person("Alice", 25, 'F');

Student student("Bob", 20, 'M', "2022001", 85.5);

std::cout << "Person Details: ";

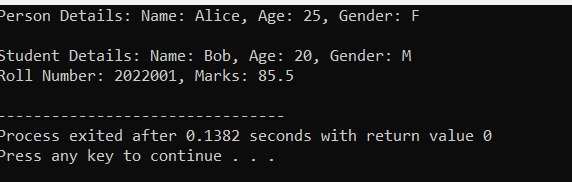
person.displayDetails();

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

student.displayDetails();

return 0;

}



29- Define a base class Shape with virtual functions draw() and calculateArea(). Derive classes Circle, Rectangle, and Triangle from Shape. Implement these functions in each derived class. Create an array of pointers to objects of type Shape, dynamically allocate memory to objects of each derived class, and use them to call the draw() function for each shape.

Test Cases:

Verify if the draw() function is correctly called for each shape and displays the appropriate shape.

Calculate the area of each shape using the calculateArea() function and validate the results.

#include <iostream>

#include <cmath>

class Shape {

public:

virtual void draw() const = 0;

virtual double calculateArea() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

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

void draw() const override {

std::cout << "Drawing Circle with radius " << radius << std::endl;

}

double calculateArea() const override {

return M\_PI \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

Rectangle(double w, double h) : width(w), height(h) {}

void draw() const override {

std::cout << "Drawing Rectangle with width " << width << " and height " << height << std::endl;

}

double calculateArea() const override {

return width \* height;

}

};

class Triangle : public Shape {

private:

double base;

double height;

public:

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

void draw() const override {

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

}

double calculateArea() const override {

return 0.5 \* base \* height;

}

};

int main() {

const int numShapes = 3;

Shape\* shapes[numShapes];

shapes[0] = new Circle(5.0);

shapes[1] = new Rectangle(4.0, 6.0);

shapes[2] = new Triangle(3.0, 4.0);

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

shapes[i]->draw();

}

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

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

}

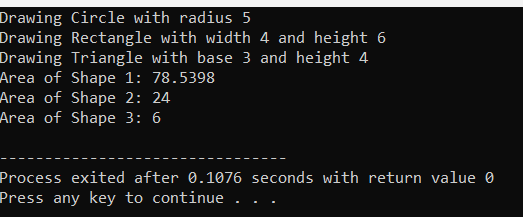
for (int i = 0; i < numShapes; ++i) {

delete shapes[i];

}

return 0;

}



30. Create a class Employee with attributes id and name. Implement a member function printDetails() to display the details of the employee. Define a derived class Manager from Employee with an additional attribute department. Use a pointer to Employee to dynamically allocate memory to an object of type Manager and call the printDetails() function.

Test Cases:

Input: id = 101, name = "John", department = "HR"

Output: Employee Details: ID: 101, Name: John, Department: HR

#include <iostream>

#include <string>

using namespace std;

class Employee {

protected:

int id;

string name;

public:

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

virtual void printDetails() const {

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

}

};

class Manager : public Employee {

private:

string department;

public:

Manager(int id, const string& name, const string& department)

: Employee(id, name), department(department) {}

void printDetails() const override {

cout << "Employee Details: ID: " << id << ", Name: " << name << ", Department: " << department << endl;

}

};

int main() {

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

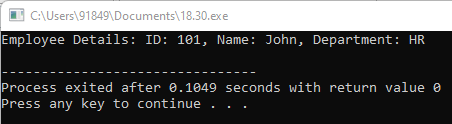
empPtr->printDetails();

delete empPtr;

return 0;

}

OUTPUT:



31. .Define a class Vector2D to represent 2-dimensional vectors. Implement overloaded operators for addition, subtraction, and scalar multiplication. Use pointers to objects of type Vector2D to perform vector operations and verify the results.

Test Cases:

Verify the correctness of addition, subtraction, and scalar multiplication operations for different vectors.

PROGRAM

#include <iostream>

using namespace std;

class Vector2D {

private:

double x;

double y;

public:

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

Vector2D operator+(const Vector2D& other) const {

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

}

Vector2D operator-(const Vector2D& other) const {

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

}

Vector2D operator\*(double scalar) const {

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

}

friend ostream& operator<<(ostream& os, const Vector2D& vec) {

os << "(" << vec.x << ", " << vec.y << ")";

return os;

}

};

int main() {

Vector2D v1(2.0, 3.0);

Vector2D v2(1.0, -1.0);

Vector2D\* result\_add = new Vector2D(v1 + v2);

cout << "Addition: " << \*result\_add << endl;

delete result\_add;

Vector2D\* result\_sub = new Vector2D(v1 - v2);

cout << "Subtraction: " << \*result\_sub << endl;

delete result\_sub;

double scalar = 2.5;

Vector2D\* result\_scalar\_mult = new Vector2D(v1 \* scalar);

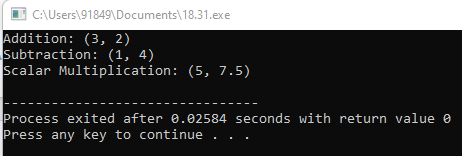
cout << "Scalar Multiplication: " << \*result\_scalar\_mult << endl;

delete result\_scalar\_mult;

return 0;

}

OUTPUT:



32. Write a C++ program to implement a binary search tree. Define a class Node to represent tree nodes with attributes data, left, and right. Implement member functions to insert, search, and delete nodes in the binary search tree using pointers.

Test Cases:

Insert various elements into the binary search tree and verify their correct placement.

Search for existing and non-existing elements in the tree and validate the search results.

Delete nodes from the tree and verify the correctness of the resulting tree structure.

#include <iostream>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

struct node\* newNode(int item)

{

struct node\* temp

= new struct node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

struct node\* insert(struct node\* node, int key)

{

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

return node;

}

struct node\* search(struct node\* root, int key)

{

if (root == NULL || root->key == key)

return root;

if (root->key < key)

return search(root->right, key);

return search(root->left, key);

}

int main()

{

struct node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

int key = 6;

if (search(root, key) == NULL)

cout << key << " not found" << endl;

else

cout << key << " found" << endl;

key = 60;

if (search(root, key) == NULL)

cout << key << " not found" << endl;

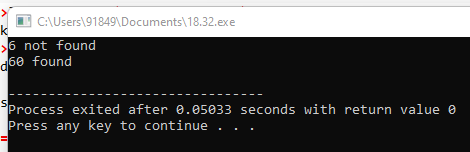
else

cout << key << " found" << endl;

return 0;

}

OUTPUT:



33. Create a class Matrix to represent a 2-dimensional matrix. Implement overloaded operators for addition, subtraction, and multiplication. Use pointers to objects of type Matrix to perform matrix operations and validate the results.

Test Cases:

Verify the correctness of addition, subtraction, and multiplication operations for different matrices.

Ensure proper handling of matrices with different dimensions and edge cases.

#include <iostream>

#include <vector>

using namespace std;

class Matrix {

private:

vector<vector<int>> data;

size\_t rows;

size\_t cols;

public:

Matrix(size\_t rows, size\_t cols) : rows(rows), cols(cols) {

data.resize(rows, vector<int>(cols, 0));

}

Matrix(vector<vector<int>> matrix) : data(matrix) {

rows = matrix.size();

cols = matrix[0].size();

}

Matrix operator+(const Matrix& other) const {

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

cerr << "Matrix dimensions must match for addition." << endl;

exit(1);

}

Matrix result(rows, cols);

for (size\_t i = 0; i < rows; ++i) {

for (size\_t j = 0; j < cols; ++j) {

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

}

}

return result;

}

Matrix operator-(const Matrix& other) const {

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

cerr << "Matrix dimensions must match for subtraction." << endl;

exit(1);

}

Matrix result(rows, cols);

for (size\_t i = 0; i < rows; ++i) {

for (size\_t j = 0; j < cols; ++j) {

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

}

}

return result;

}

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

if (cols != other.rows) {

cerr << "Number of columns in the first matrix must match the number of rows in the second matrix for multiplication." << endl;

exit(1);

}

Matrix result(rows, other.cols);

for (size\_t i = 0; i < rows; ++i) {

for (size\_t j = 0; j < other.cols; ++j) {

for (size\_t k = 0; k < cols; ++k) {

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

}

}

}

return result;

}

friend ostream& operator<<(ostream& os, const Matrix& matrix) {

for (size\_t i = 0; i < matrix.rows; ++i) {

for (size\_t j = 0; j < matrix.cols; ++j) {

os << matrix.data[i][j] << " ";

}

os << endl;

}

return os;

}

};

int main() {

Matrix A({{1, 2, 3}, {4, 5, 6}});

Matrix B({{7, 8}, {9, 10}, {11, 12}});

Matrix C({{1, 2, 3}, {4, 5, 6}});

Matrix D({{7, 8}, {9, 10}, {11, 12}});

cout << "Matrix A:" << endl << A << endl;

cout << "Matrix B:" << endl << B << endl;

cout << "Addition (A + C):" << endl << (A + C) << endl;

cout << "Subtraction (A - C):" << endl << (A - C) << endl;

cout << "Multiplication (A \* B):" << endl << (A \* B) << endl;

return 0;

}

OUTPUT:

