Question 21:

#include <iostream>

class Complex {

private:

double real;

double imaginary;

public:

// Constructors

Complex() : real(0.0), imaginary(0.0) {}

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

// Getter functions

double getReal() const { return real; }

double getImaginary() const { return imaginary; }

// Friend functions for complex arithmetic operations

friend Complex operator+(const Complex& c1, const Complex& c2);

friend Complex operator-(const Complex& c1, const Complex& c2);

friend Complex operator\*(const Complex& c1, const Complex& c2);

friend Complex operator/(const Complex& c1, const Complex& c2);

// Friend functions for operations between Complex and primitive floating-point numbers

friend Complex operator+(const Complex& c, double d);

friend Complex operator+(double d, const Complex& c);

friend Complex operator-(const Complex& c, double d);

friend Complex operator-(double d, const Complex& c);

friend Complex operator\*(const Complex& c, double d);

friend Complex operator\*(double d, const Complex& c);

friend Complex operator/(const Complex& c, double d);

friend Complex operator/(double d, const Complex& c);

// Overload the insertion operator for easy output

friend std::ostream& operator<<(std::ostream& os, const Complex& c);

};

// Complex arithmetic operations

Complex operator+(const Complex& c1, const Complex& c2) {

return Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);

}

Complex operator-(const Complex& c1, const Complex& c2) {

return Complex(c1.real - c2.real, c1.imaginary - c2.imaginary);

}

Complex operator\*(const Complex& c1, const Complex& c2) {

double realPart = c1.real \* c2.real - c1.imaginary \* c2.imaginary;

double imagPart = c1.real \* c2.imaginary + c1.imaginary \* c2.real;

return Complex(realPart, imagPart);

}

Complex operator/(const Complex& c1, const Complex& c2) {

double denominator = c2.real \* c2.real + c2.imaginary \* c2.imaginary;

double realPart = (c1.real \* c2.real + c1.imaginary \* c2.imaginary) / denominator;

double imagPart = (c1.imaginary \* c2.real - c1.real \* c2.imaginary) / denominator;

return Complex(realPart, imagPart);

}

// Operations between Complex and primitive floating-point numbers

Complex operator+(const Complex& c, double d) {

return Complex(c.real + d, c.imaginary);

}

Complex operator+(double d, const Complex& c) {

return Complex(d + c.real, c.imaginary);

}

Complex operator-(const Complex& c, double d) {

return Complex(c.real - d, c.imaginary);

}

Complex operator-(double d, const Complex& c) {

return Complex(d - c.real, -c.imaginary);

}

Complex operator\*(const Complex& c, double d) {

return Complex(c.real \* d, c.imaginary \* d);

}

Complex operator\*(double d, const Complex& c) {

return Complex(d \* c.real, d \* c.imaginary);

}

Complex operator/(const Complex& c, double d) {

return Complex(c.real / d, c.imaginary / d);

}

Complex operator/(double d, const Complex& c) {

double denominator = c.real \* c.real + c.imaginary \* c.imaginary;

double realPart = (d \* c.real) / denominator;

double imagPart = (-d \* c.imaginary) / denominator;

return Complex(realPart, imagPart);

}

// Overload the insertion operator for easy output

std::ostream& operator<<(std::ostream& os, const Complex& c) {

os << c.real;

if (c.imaginary >= 0) {

os << " + " << c.imaginary << "i";

} else {

os << " - " << -c.imaginary << "i";

}

return os;

}

int main() {

// Testing the Complex class

Complex c1(2.0, 3.0);

Complex c2(1.0, -1.0);

std::cout << "c1: " << c1 << std::endl;

std::cout << "c2: " << c2 << std::endl;

// Complex arithmetic operations

Complex sum = c1 + c2;

Complex difference = c1 - c2;

Complex product = c1 \* c2;

Complex quotient = c1 / c2;

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

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

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

std::cout << "Quotient: " << quotient << std::endl;

// Operations between Complex and primitive floating-point numbers

Complex result1 = c1 + 5.0;

Complex result2 = 2.0 - c2;

Complex result3 = c1 \* 1.5;

Complex result4 = 10.0 / c2;

std::cout << "Result 1: " << result1 << std::endl;

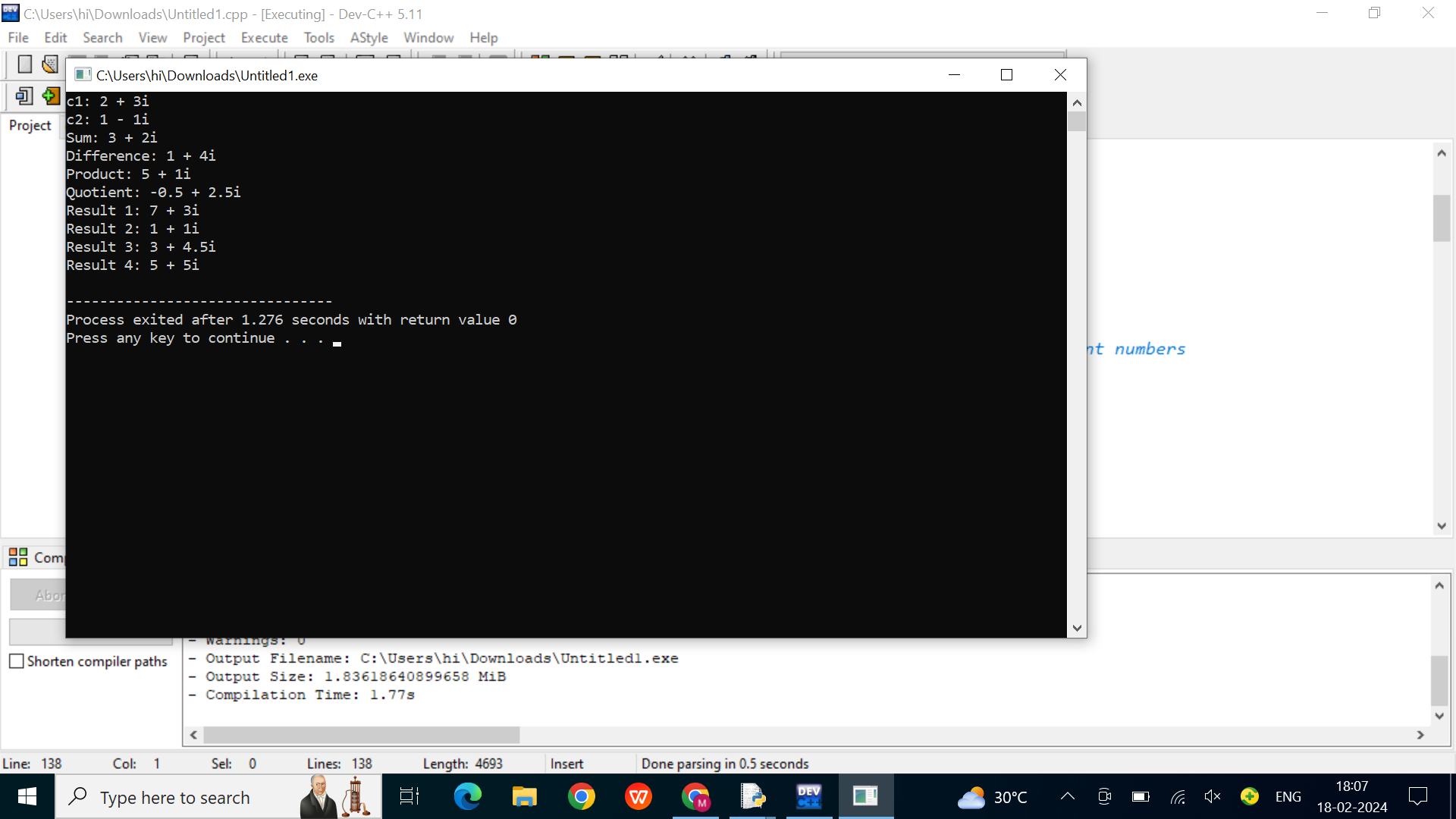
std::cout << "Result 2: " << result2 << std::endl;

std::cout << "Result 3: " << result3 << std::endl;

std::cout << "Result 4: " << result4 << std::endl;

return 0;

}



Question 22:

#include <iostream>

class Point {

private:

double x;

double y;

public:

// Constructors

Point() : x(0.0), y(0.0) {}

Point(double xValue, double yValue) : x(xValue), y(yValue) {}

// Getter functions

double getX() const { return x; }

double getY() const { return y; }

// Overload the + operator for Point addition

Point operator+(const Point& other) const {

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

}

// Overload the - operator for Point subtraction

Point operator-(const Point& other) const {

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

}

// Overload the \* operator for scalar multiplication

Point operator\*(double scalar) const {

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

}

// Overload the \* operator to support scalar \* Point multiplication

friend Point operator\*(double scalar, const Point& point);

// Overload the == operator for equality comparison

bool operator==(const Point& other) const {

return (x == other.x) && (y == other.y);

}

// Overload the != operator for inequality comparison

bool operator!=(const Point& other) const {

return !(\*this == other);

}

// Overload the insertion operator for easy output

friend std::ostream& operator<<(std::ostream& os, const Point& point);

};

// Overload the \* operator to support scalar \* Point multiplication

Point operator\*(double scalar, const Point& point) {

return Point(scalar \* point.x, scalar \* point.y);

}

// Overload the insertion operator for easy output

std::ostream& operator<<(std::ostream& os, const Point& point) {

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

return os;

}

int main() {

// Testing the Point class

Point p1(1.0, 2.0);

Point p2(3.0, 4.0);

std::cout << "p1: " << p1 << std::endl;

std::cout << "p2: " << p2 << std::endl;

// Point addition

Point sum = p1 + p2;

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

// Point subtraction

Point difference = p1 - p2;

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

// Scalar multiplication

Point scaled = p1 \* 1.5;

std::cout << "Scaled: " << scaled << std::endl;

// Scalar \* Point multiplication

Point scaledByScalar = 2.0 \* p2;

std::cout << "Scaled by scalar: " << scaledByScalar << std::endl;

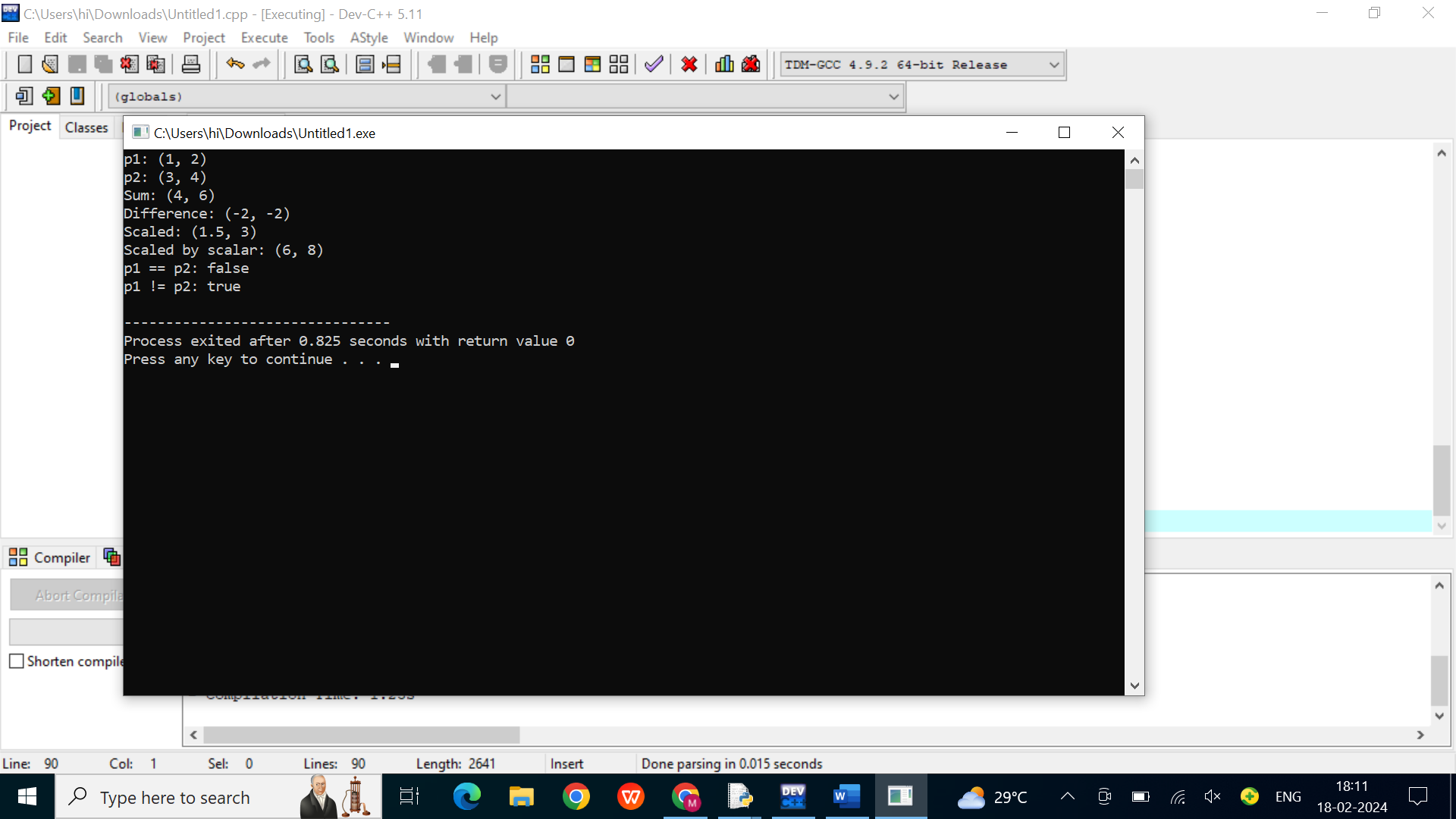
// Equality comparison

std::cout << "p1 == p2: " << std::boolalpha << (p1 == p2) << std::endl;

std::cout << "p1 != p2: " << std::boolalpha << (p1 != p2) << std::endl;

return 0;

}



Question 23:

#include <iostream>

#include <string>

// Base class representing an animal

class Animal {

public:

// Virtual function for producing the animal sound

virtual void makeSound() const {

std::cout << "Generic animal sound" << std::endl;

}

// Virtual destructor to ensure proper cleanup

virtual ~Animal() {}

};

// Derived classes for specific types of animals

class Dog : public Animal {

public:

void makeSound() const override {

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

}

};

class Cat : public Animal {

public:

void makeSound() const override {

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

}

};

class Duck : public Animal {

public:

void makeSound() const override {

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

}

};

// Simulation function that interacts with animals

void interactWithAnimal(const Animal& animal) {

std::cout << "Interacting with an animal: ";

animal.makeSound();

}

int main() {

// Create instances of different animals

Dog myDog;

Cat myCat;

Duck myDuck;

// Interact with each animal

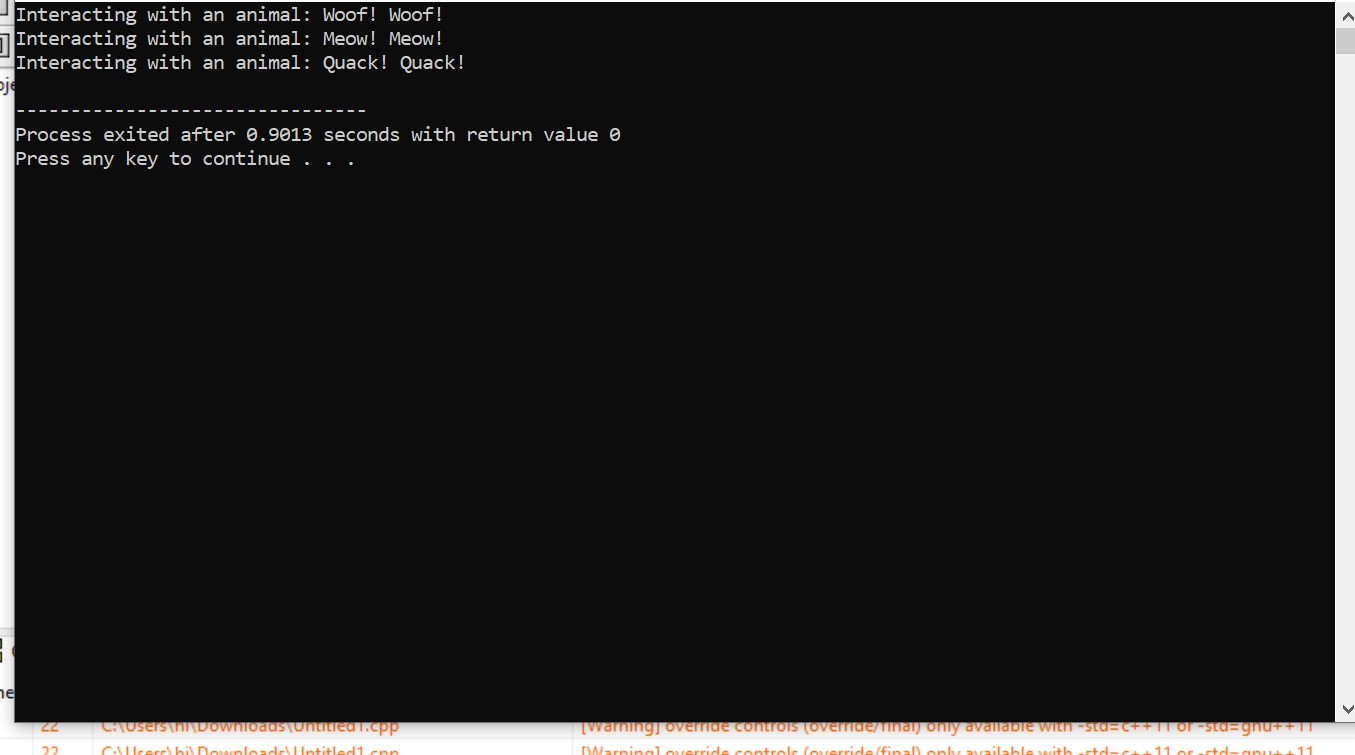
interactWithAnimal(myDog);

interactWithAnimal(myCat);

interactWithAnimal(myDuck);

return 0;

}



Question 24:

#include <iostream>

class Employee {

private:

static int lastAssignedID;

int employeeID;

std::string name;

public:

// Constructor

Employee(const std::string& employeeName) : name(employeeName) {

employeeID = ++lastAssignedID;

}

// Static member function to get the total number of employees

static int getTotalEmployees() {

return lastAssignedID;

}

// Getter function for employee ID

int getEmployeeID() const {

return employeeID;

}

// Getter function for employee name

const std::string& getName() const {

return name;

}

};

// Initialize the static member variable

int Employee::lastAssignedID = 0;

int main() {

// Create instances of Employee

Employee emp1("John Doe");

Employee emp2("Jane Smith");

// Display information about each employee

std::cout << "Employee ID: " << emp1.getEmployeeID() << ", Name: " << emp1.getName() << std::endl;

std::cout << "Employee ID: " << emp2.getEmployeeID() << ", Name: " << emp2.getName() << std::endl;

// Display the total number of employees

std::cout << "Total number of employees: " << Employee::getTotalEmployees() << std::endl;

return 0;

}

