**Inheritance\_Lab**

/\*Animal Hierarchy:

Problem Statement: Create a hierarchy of animal classes. Start with a base class Animal and then create derived classes like Mammal, Bird, and Fish. Each of these derived classes should have specific properties and methods related to their respective categories of animals.\*/

#include <iostream>

#include <string>

using namespace std;

class Animal

{

private:

string name ;

int age ;

public:

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

virtual void sound() = 0;

virtual void move() = 0;

void info(){

cout<<"Animal name is "<<name<<" having age "<<age<<"."<<endl;

}

virtual ~Animal(){}

};

class Mammal : public Animal{

int numOfLegs;

public :

Mammal(string name, int age,int numOfLegs) : Animal (name, age), numOfLegs(numOfLegs){}

void sound(){

cout << "Mammal sound" << endl;

}

void move(){

cout << "Mammal moves by walking or running" << endl;

}

void info(){

info();

cout<< "Number of legs : "<<numOfLegs << endl;

}

void Mind(){

cout << "Mammals has sharp mind " << endl;

}

};

class Bird : public Animal {

string color;

public:

Bird(string name, int age,string color) : Animal(name, age), color(color) {}

void sound(){

cout << "Bird sound" << endl;

}

void move() {

cout << "Bird moves " << endl;

}

void info(){

info();

cout<< "Colour of bird : "<<color << endl;

}

void fly(){

cout << "Birds can fly " << endl;

}

};

class Fish : public Animal {

public:

Fish(string name, int age) : Animal(name, age) {}

void sound() {

cout << "Fish sound" << endl;

}

void move() {

cout << "Fish moves by swimming" << endl;

}

};

void menu(){

cout<< "\n1. Add Mammal " << endl;

cout<< "2. Add Bird " << endl;

cout<< "3. Add Fish " << endl;

cout<< "4. Display details " << endl;

cout<< "5. Special properties of mammal and bird " << endl;

cout<< "6. Exit " << endl;

cout<< "Enter your choice : ";

}

int main(){

int numOfAnimals = 10;

Animal\* Animals[numOfAnimals];

int numOfLegs;

string name , color;

int age ,ch, countAnimal = 0;

do

{

menu();

cin>>ch;

switch (ch)

{

case 1:

cout<<"Name Age NumerOfLegs"<<endl;

cin>>name>>age>>numOfLegs;

Animals [countAnimal ++] = new Mammal( name, age, numOfLegs);

cout<<"========================================="<<endl;

break;

case 2:

cout<<"Name Age color"<<endl;

cin>>name>>age>>color;

Animals [countAnimal ++] = new Bird( name, age, color);

cout<<"========================================="<<endl;

break;

case 3:

cout<<"Name Age "<<endl;

cin>>name>>age;

Animals [countAnimal ++] = new Fish( name, age);

cout<<"========================================="<<endl;

break;

case 4:

cout<<"Display details"<<endl;

for (int i = 0; i < countAnimal; i++)

{

/\* code \*/

Animals[i]->info();

Animals[i]->move();

Animals[i]->sound();

cout<<"===================================="<<endl;

}

break;

case 5:

cout<<"Specific details"<<endl;

for (int i = 0; i < countAnimal; i++)

{

if (typeid(\*Animals [i]) == typeid(Mammal))

{

cout<<"Type of Animal is "<<typeid(\*Animals[i]).name()<<"."<<endl;

cout<<"Mammal type at idex:"<<i<<endl;

Mammal \*m = dynamic\_cast<Mammal\*>(Animals[i]);

m->Mind();

}

if (typeid(\*Animals [i]) == typeid(Bird))

{

cout<<"Type of animal is "<<typeid(\*Animals[i]).name()<<"."<<endl;

cout<<"Bird type at idex:"<<i<<endl;

Bird \*b = dynamic\_cast <Bird\*> (Animals[i]);

b->fly();

/\* code \*/

}

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

}

break;

case 6:

cout<<"Exiting......"<<endl;

cout<<"===================================="<<endl;

break;

default:

break;

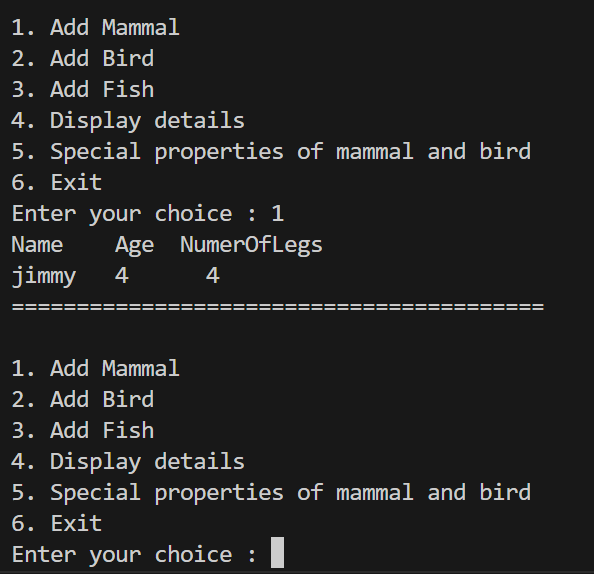
}

} while (ch != 6);

delete Animals[countAnimal];

return 0;

}



/\*Shape Hierarchy:

Problem Statement: Design a hierarchy of shape classes. Begin with a base class Shape and then create derived classes like Circle, Rectangle, and Triangle. Each shape should have methods for calculating area and perimeter specific to its geometry.\*/

#include <iostream>

#include <cmath>

using namespace std;

class Shape

{

public:

virtual double area() = 0 ;

virtual double perimeter() = 0;

~Shape(){}

};

class Circle : public Shape{

double r;

public :

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

double area() {

return M\_PI \* r \* r;

}

double perimeter() {

return 2 \* M\_PI \* r;

}

};

class Rectangle : public Shape{

double l,w;

public :

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

double area(){

return l \* w;

}

double perimeter() {

return 2 \* (l + w);

}

};

class Triangle : public Shape{

double s1,s2,s3;

public :

Triangle (double s1 , double s2, double s3) : s1(s1), s2(s2), s3(s3){};

double area() {

double s = (s1 + s2 + s3)/2;

return sqrt(s \* (s - s1) \* (s - s2) \*(s - s3) );

}

double perimeter () {

return s1 + s2 + s3;

}

};

int main() {

Circle c(5);

Rectangle rectangle(4, 6);

Triangle triangle(3, 4, 5);

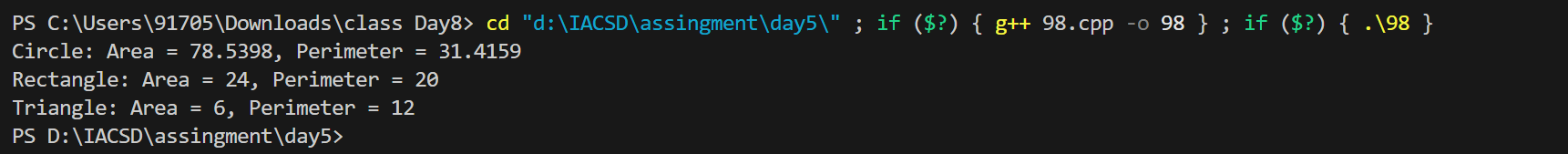
cout << "Circle: Area = " << c.area() << ", Perimeter = " << c.perimeter() << endl;

cout << "Rectangle: Area = " << rectangle.area() << ", Perimeter = " << rectangle.perimeter() << endl;

cout << "Triangle: Area = " << triangle.area() << ", Perimeter = " << triangle.perimeter() << endl;

return 0;

}



/\*

Employee Inheritance:

Problem Statement: Build a system for managing employees. Create a base class Employee with attributes such as name, employee ID, and salary. Then, derive classes like Manager and Developer, each with its own attributes and methods. Implement a common method, like calculate\_salary(), in the base class.\*/

#include <iostream>

#include <string>

using namespace std;

// Base class Employee

class Employee {

protected:

string name;

int employeeID;

double salary;

public:

Employee(string name, int employeeID, double salary) : name(name), employeeID(employeeID), salary(salary) {}

// Common method for calculating salary (virtual function)

virtual double calculate\_salary() const {

return salary;

}

virtual ~Employee() {}

};

// Derived class Manager

class Manager : public Employee {

private:

double bonusPercentage;

public:

Manager(string name, int employeeID, double salary, double bonusPercentage) : Employee(name, employeeID, salary), bonusPercentage(bonusPercentage) {}

// Overriding calculate\_salary method for managers

double calculate\_salary() const override {

return salary + (salary \* bonusPercentage / 100);

}

};

// Derived class Developer

class Developer : public Employee {

private:

double overtimeHours;

double hourlyRate;

public:

Developer(string name, int employeeID, double salary, double overtimeHours, double hourlyRate) : Employee(name, employeeID, salary), overtimeHours(overtimeHours), hourlyRate(hourlyRate) {}

// Overriding calculate\_salary method for developers

double calculate\_salary() const override {

return salary + (overtimeHours \* hourlyRate);

}

};

int main() {

// Creating instances of Manager and Developer

Manager manager("John", 101, 5000, 10); // 10% bonus

Developer developer("Alice", 102, 4000, 5, 20); // 5 overtime hours, $20 hourly rate

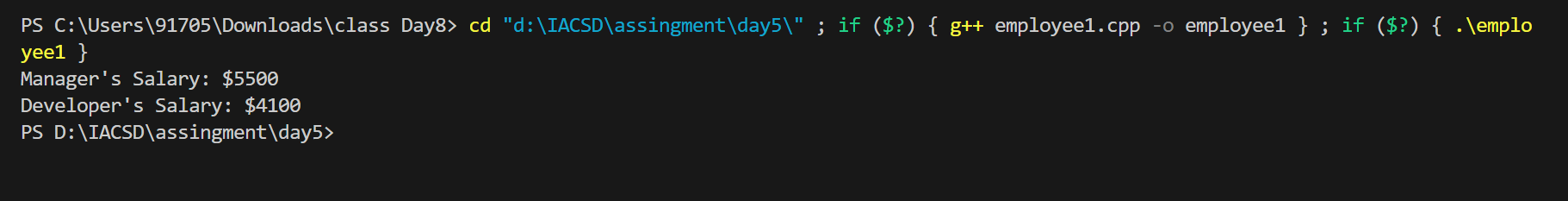
// Calculating and displaying salaries

cout << "Manager's Salary: $" << manager.calculate\_salary() << endl;

cout << "Developer's Salary: $" << developer.calculate\_salary() << endl;

return 0;

}



/\*Vehicle Inheritance:

Problem Statement: Develop a class hierarchy for vehicles. Start with a base class Vehicle and create derived classes like Car, Motorcycle, and Truck. Each derived class should have unique properties like the number of wheels and specific methods like start\_engine().

\*/

#include <iostream>

#include <string>

// Base class Vehicle

class Vehicle {

protected:

std::string name;

public:

Vehicle(const std::string& name) : name(name) {}

virtual ~Vehicle() {}

virtual void start\_engine() const = 0; // Abstract method

virtual void display\_info() const {

std::cout << "Vehicle Type: " << name << std::endl;

}

};

// Derived class Car

class Car : public Vehicle {

private:

int num\_wheels;

public:

Car(const std::string& name, int num\_wheels) : Vehicle(name), num\_wheels(num\_wheels) {}

void start\_engine() const override {

std::cout << "Starting car engine." << std::endl;

}

void display\_info() const override {

std::cout << "Car Type: " << name << std::endl;

std::cout << "Number of Wheels: " << num\_wheels << std::endl;

}

};

// Derived class Motorcycle

class Motorcycle : public Vehicle {

private:

int num\_wheels;

public:

Motorcycle(const std::string& name, int num\_wheels) : Vehicle(name), num\_wheels(num\_wheels) {}

void start\_engine() const override {

std::cout << "Starting motorcycle engine." << std::endl;

}

void display\_info() const override {

std::cout << "Motorcycle Type: " << name << std::endl;

std::cout << "Number of Wheels: " << num\_wheels << std::endl;

}

};

// Derived class Truck

class Truck : public Vehicle {

private:

int num\_wheels;

public:

Truck(const std::string& name, int num\_wheels) : Vehicle(name), num\_wheels(num\_wheels) {}

void start\_engine() const override {

std::cout << "Starting truck engine." << std::endl;

}

void display\_info() const override {

std::cout << "Truck Type: " << name << std::endl;

std::cout << "Number of Wheels: " << num\_wheels << std::endl;

}

};

int main() {

Car car("Sedan", 4);

Motorcycle motorcycle("Sport Bike", 2);

Truck truck("Pickup", 6);

car.display\_info();

car.start\_engine();

motorcycle.display\_info();

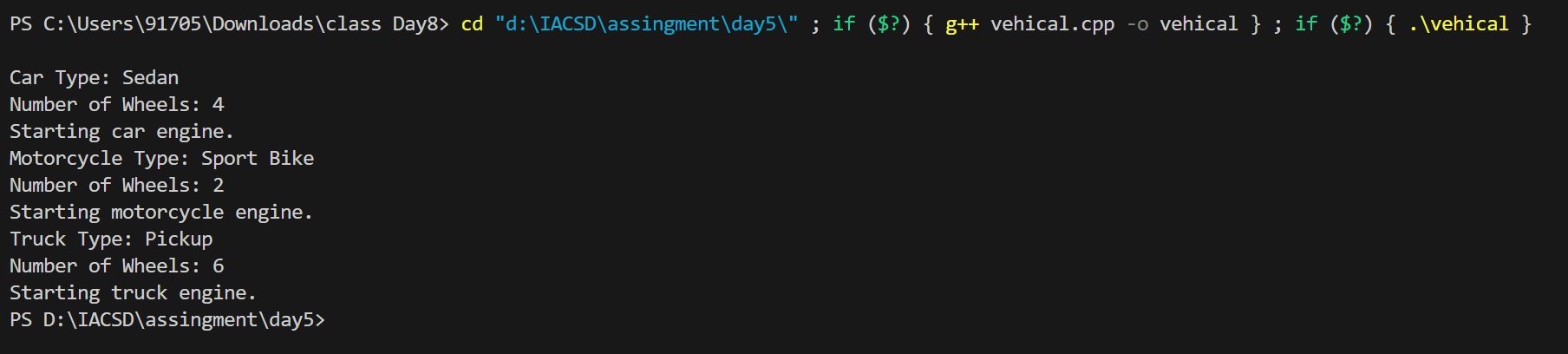
motorcycle.start\_engine();

truck.display\_info();

truck.start\_engine();

return 0;

}



/\*Bank Account Inheritance:

Problem Statement: Design a system for managing bank accounts. Create a base class BankAccount with attributes like account number and balance. Derive classes like SavingsAccount and CheckingAccount, each with specialized methods like withdraw() and calculate\_interest().\*/

#include <iostream>

using namespace std;

// Base class BankAccount

class BankAccount {

protected:

string accountNumber;

double balance;

public:

BankAccount(string accountNumber, double balance) : accountNumber(accountNumber), balance(balance) {}

// Method to deposit money into the account

void deposit(double amount) {

balance += amount;

cout << "Deposit of $" << amount << " successful." << endl;

}

// Virtual method to withdraw money from the account

virtual void withdraw(double amount) = 0;

// Method to display account information

virtual void display() const {

cout << "Account Number: " << accountNumber << ", Balance: $" << balance << endl;

}

virtual ~BankAccount() {}

};

// Derived class SavingsAccount inheriting from BankAccount

class SavingsAccount : public BankAccount {

private:

double interestRate;

public:

SavingsAccount(string accountNumber, double balance, double interestRate) : BankAccount(accountNumber, balance), interestRate(interestRate) {}

// Method to calculate and add interest to the account

void calculate\_interest() {

double interest = balance \* interestRate / 100;

balance += interest;

cout << "Interest of $" << interest << " added to the account." << endl;

}

// Method to withdraw money from the savings account

void withdraw(double amount) override {

if (amount <= balance) {

balance -= amount;

cout << "Withdrawal of $" << amount << " successful." << endl;

} else {

cout << "Insufficient balance for withdrawal." << endl;

}

}

// Method to display account information, overriding the base class method

void display() const override {

cout << "Savings Account ";

BankAccount::display();

}

};

// Derived class CheckingAccount inheriting from BankAccount

class CheckingAccount : public BankAccount {

private:

double overdraftLimit;

public:

CheckingAccount(string accountNumber, double balance, double overdraftLimit) : BankAccount(accountNumber, balance), overdraftLimit(overdraftLimit) {}

// Method to withdraw money from the checking account

void withdraw(double amount) override {

if (amount <= balance + overdraftLimit) {

balance -= amount;

cout << "Withdrawal of $" << amount << " successful." << endl;

} else {

cout << "Exceeded overdraft limit. Withdrawal not allowed." << endl;

}

}

// Method to display account information, overriding the base class method

void display() const override {

cout << "Checking Account ";

BankAccount::display();

}

};

int main() {

// Creating instances of SavingsAccount and CheckingAccount

SavingsAccount savings("SA123", 1000, 5); // Account number, initial balance, interest rate

CheckingAccount checking("CA456", 2000, 500); // Account number, initial balance, overdraft limit

// Displaying initial account information

savings.display();

checking.display();

cout << endl;

// Performing operations on accounts

savings.deposit(200);

savings.calculate\_interest();

savings.withdraw(300);

checking.deposit(500);

checking.withdraw(2500);

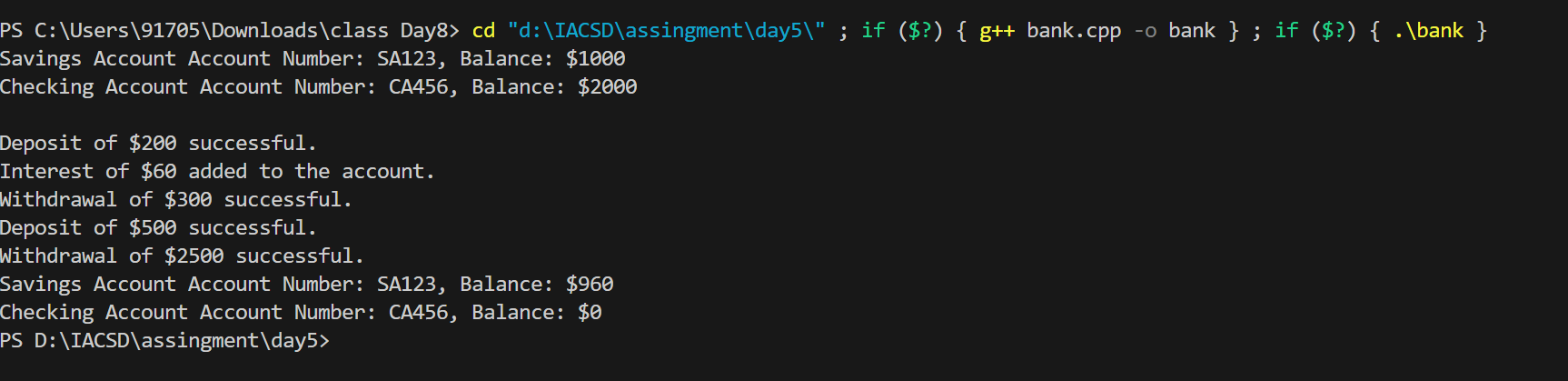
// Displaying updated account information

savings.display();

checking.display();

return 0;

}



/\*Geometric Shapes with Polymorphism:

Problem Statement: Extend the shape hierarchy example by implementing polymorphism. Define a base class Shape with methods to calculate area and perimeter. Then, create derived classes like Circle, Rectangle, and Triangle, each with its own implementation of these methods.

\*/

#include <iostream>

#include <cmath>

using namespace std;

class Shape

{

public:

virtual double area() = 0 ;

virtual double perimeter() = 0;

~Shape(){}

};

class Circle : public Shape{

double r;

public :

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

double area() {

return M\_PI \* r \* r;

}

double perimeter() {

return 2 \* M\_PI \* r;

}

};

class Rectangle : public Shape{

double l,w;

public :

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

double area(){

return l \* w;

}

double perimeter() {

return 2 \* (l + w);

}

};

class Triangle : public Shape{

double s1,s2,s3;

public :

Triangle (double s1 , double s2, double s3) : s1(s1), s2(s2), s3(s3){};

double area() {

double s = (s1 + s2 + s3)/2;

return sqrt(s \* (s - s1) \* (s - s2) \*(s - s3) );

}

double perimeter () {

return s1 + s2 + s3;

}

};

int main() {

Shape\* shapes[3];

shapes[0] = new Circle(5);

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

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

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

cout << "Shape " << i + 1 << ": Area = " << shapes[i]->area() << ", Perimeter = " << shapes[i]->perimeter() << endl;

}

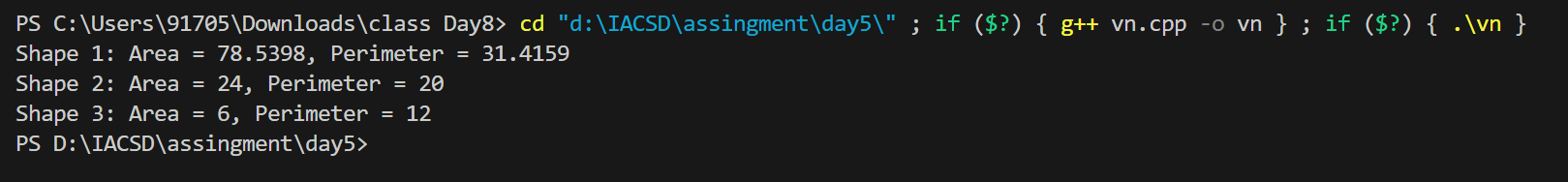
for (int i = 0; i < 3; ++i) {

delete shapes[i];

}

return 0;

}



/\*Person and Student Inheritance:

Problem Statement: Model a system for handling individuals and students within an educational institution. Create a base class Person with attributes like name and age. Derive a Student class with additional attributes like student ID and GPA, inheriting the common attributes from the Person class.\*/

#include <iostream>

#include <string>

using namespace std;

// Base class Person

class Person {

protected:

string name;

int age;

public:

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

// Method to display information about the person

void display() const {

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

}

};

// Derived class Student inheriting from Person

class Student : public Person {

private:

int studentID;

double GPA;

public:

// Constructor for Student class

Student(string name, int age, int studentID, double GPA) : Person(name, age), studentID(studentID), GPA(GPA) {}

// Method to display information about the student

void display\_student() const {

display(); // Call the base class method to display common attributes

cout << "Student ID: " << studentID << ", GPA: " << GPA << endl;

}

};

int main() {

// Creating instances of Person and Student

Person person("John", 25);

Student student("Alice", 20, 12345, 3.8);

// Displaying information about person and student

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

person.display();

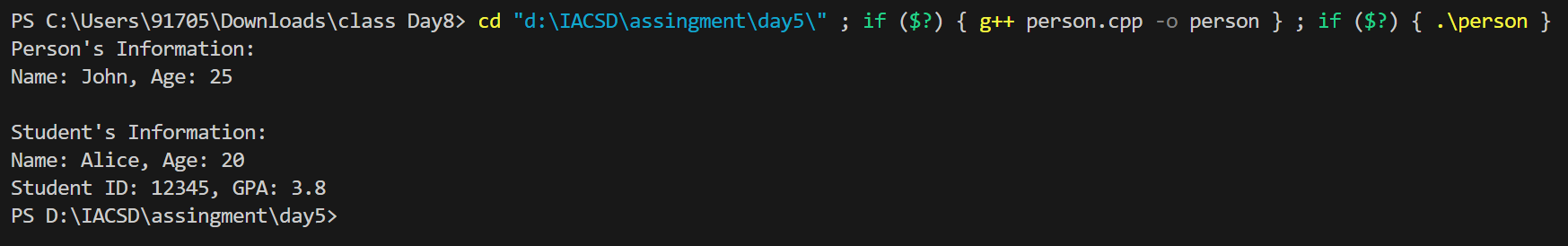
cout << endl;

cout << "Student's Information:" << endl;

student.display\_student();

return 0;

}



/\*Library Catalog with Books and Journals:

Problem Statement: Build a library catalog system. Create a base class LibraryItem with properties like title and author. Then, derive classes like Book and Journal, each with their unique properties. Implement methods to check out and return items in the derived classes.\*/

#include <iostream>

#include <string>

#include <vector>

using namespace std;

// Base class LibraryItem

class LibraryItem {

protected:

string title;

string author;

bool checkedOut;

public:

LibraryItem(string title, string author) : title(title), author(author), checkedOut(false) {}

// Method to check out the item

void checkOut() {

if (!checkedOut) {

checkedOut = true;

cout << title << " by " << author << " has been checked out." << endl;

} else {

cout << "Sorry, " << title << " by " << author << " is already checked out." << endl;

}

}

// Method to return the item

void returnItem() {

if (checkedOut) {

checkedOut = false;

cout << title << " by " << author << " has been returned." << endl;

} else {

cout << "There is no record of " << title << " by " << author << " being checked out." << endl;

}

}

// Virtual destructor

virtual ~LibraryItem() {}

};

// Derived class Book

class Book : public LibraryItem {

private:

int numPages;

public:

Book(string title, string author, int numPages) : LibraryItem(title, author), numPages(numPages) {}

// Method to display book information

void displayInfo() const {

cout << "Book: " << title << " by " << author << ", Pages: " << numPages;

if (checkedOut) {

cout << " (Checked out)";

}

cout << endl;

}

};

// Derived class Journal

class Journal : public LibraryItem {

private:

int issueNumber;

public:

Journal(string title, string author, int issueNumber) : LibraryItem(title, author), issueNumber(issueNumber) {}

// Method to display journal information

void displayInfo() const {

cout << "Journal: " << title << " by " << author << ", Issue Number: " << issueNumber;

if (checkedOut) {

cout << " (Checked out)";

}

cout << endl;

}

};

int main() {

// Creating instances of Book and Journal

Book book1("The Great Gatsby", "F. Scott Fitzgerald", 180);

Book book2("To Kill a Mockingbird", "Harper Lee", 281);

Journal journal1("Nature", "Nature Publishing Group", 123);

Journal journal2("Science", "American Association for the Advancement of Science", 456);

// Checking out and returning items

book1.checkOut();

journal1.checkOut();

book1.returnItem();

book2.checkOut();

journal1.returnItem();

journal2.checkOut();

// Displaying information about items

cout << "Library Catalog:" << endl;

book1.displayInfo();

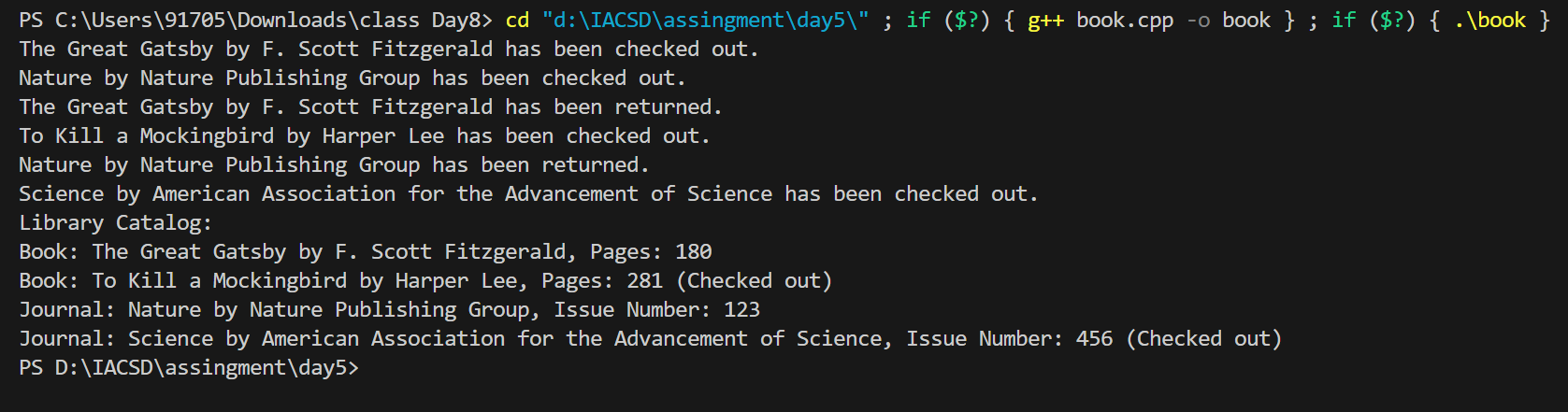
book2.displayInfo();

journal1.displayInfo();

journal2.displayInfo();

return 0;

}



/\*Employee Payroll System with Abstract Classes:

Problem Statement: Design an employee payroll system. Create an abstract class Employee with attributes like name and employee ID. Derive concrete classes like HourlyEmployee and SalariedEmployee. Define abstract methods for calculating pay in the base class and implement them in the derived classes.\*/

#include <iostream>

#include <string>

using namespace std;

// Abstract base class Employee

class Employee {

protected:

string name;

int employeeID;

public:

Employee(string name, int employeeID) : name(name), employeeID(employeeID) {}

// Pure virtual function to calculate pay

virtual double calculatePay() const = 0;

// Method to display employee information

void display() const {

cout << "Name: " << name << ", Employee ID: " << employeeID << ", Pay: $" << calculatePay() << endl;

}

virtual ~Employee() {}

};

// Derived class for hourly employees

class HourlyEmployee : public Employee {

private:

double hourlyRate;

double hoursWorked;

public:

HourlyEmployee(string name, int employeeID, double hourlyRate, double hoursWorked) : Employee(name, employeeID), hourlyRate(hourlyRate), hoursWorked(hoursWorked) {}

// Override calculatePay method for hourly employees

double calculatePay() const override {

return hourlyRate \* hoursWorked;

}

};

// Derived class for salaried employees

class SalariedEmployee : public Employee {

private:

double salary;

public:

SalariedEmployee(string name, int employeeID, double salary) : Employee(name, employeeID), salary(salary) {}

// Override calculatePay method for salaried employees

double calculatePay() const override {

return salary;

}

};

int main() {

// Creating instances of HourlyEmployee and SalariedEmployee

HourlyEmployee hourlyEmp("John", 101, 15.50, 40); // $15.50 hourly rate, 40 hours worked

SalariedEmployee salariedEmp("Alice", 102, 5000); // $5000 salary

// Displaying employee information

cout << "Hourly Employee:" << endl;

hourlyEmp.display();

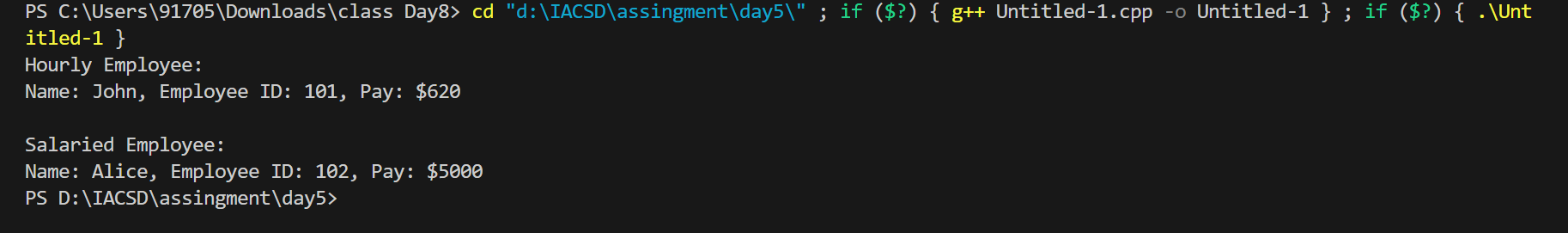
cout << endl;

cout << "Salaried Employee:" << endl;

salariedEmp.display();

return 0;

}



/\*Shape Sorting with Interfaces:

Problem Statement: Implement a shape sorting program. Define a base class Shape with properties like area and perimeter. Create derived classes like Circle, Rectangle, and Triangle. Implement an interface Sortable with a method to compare shapes by area. Use this interface to sort a list of shapes.\*/

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Interface for sortable objects

class Sortable {

public:

virtual double getArea() const = 0;

virtual bool compare(const Sortable\* other) const = 0;

virtual ~Sortable() {}

};

// Base class for shapes

class Shape : public Sortable {

protected:

double area;

public:

Shape(double area) : area(area) {}

// Get area of the shape

double getArea() const override {

return area;

}

// Compare two shapes based on their areas

bool compare(const Sortable\* other) const override {

return area < other->getArea();

}

};

// Derived class Circle

class Circle : public Shape {

private:

double radius;

public:

Circle(double radius) : Shape(3.14159 \* radius \* radius), radius(radius) {}

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

double width, height;

public:

Rectangle(double width, double height) : Shape(width \* height), width(width), height(height) {}

};

// Derived class Triangle

class Triangle : public Shape {

private:

double base, height;

public:

Triangle(double base, double height) : Shape(0.5 \* base \* height), base(base), height(height) {}

};

// Function to sort shapes based on their areas

void sortShapes(vector<Sortable\*>& shapes) {

sort(shapes.begin(), shapes.end(), [](const Sortable\* a, const Sortable\* b) {

return a->compare(b);

});

}

int main() {

// Create some shapes

Circle circle(5);

Rectangle rectangle(4, 6);

Triangle triangle(3, 4);

// Store shapes in a vector of Sortable pointers

vector<Sortable\*> shapes = {&circle, &rectangle, &triangle};

// Sort shapes based on their areas

sortShapes(shapes);

// Display sorted shapes

for (const auto& shape : shapes) {

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

}

// Clean up memory

for (auto& shape : shapes) {

delete shape;

}

return 0;

}

