ASSIGNMENT 2,3,4

Assignment 2

#include <iostream>

#include <vector>

#include <iomanip>

using namespace std;

// ===== Base Class: Person =====

class Person {

protected:

string name;

int age;

string ID;

string contact;

public:

Person(string n, int a, string id, string c) {

setName(n);

setAge(a);

ID = id;

contact = c;

}

virtual ~Person() {}

// Encapsulation with validation

void setName(string n) {

if (n.empty()) name = "Unknown";

else name = n;

}

void setAge(int a) {

if (a > 0 && a < 120) age = a;

else age = 18; // Default

}

string getName() const { return name; }

int getAge() const { return age; }

string getID() const { return ID; }

string getContact() const { return contact; }

virtual void displayDetails() const {

cout << "Name: " << name << "\nAge: " << age << "\nID: " << ID << "\nContact: " << contact << endl;

}

// Polymorphic method

virtual double calculatePayment() const {

return 0.0;

}

};

// ===== Derived Class: Student =====

class Student : public Person {

private:

string enrollmentDate;

string program;

double GPA;

public:

Student(string n, int a, string id, string c, string eDate, string prog, double g)

: Person(n, a, id, c), enrollmentDate(eDate), program(prog) {

setGPA(g);

}

void setGPA(double g) {

if (g >= 0.0 && g <= 4.0) GPA = g;

else GPA = 0.0;

}

double getGPA() const { return GPA; }

void displayDetails() const override {

Person::displayDetails();

cout << "Program: " << program << "\nEnrollment Date: " << enrollmentDate

<< "\nGPA: " << fixed << setprecision(2) << GPA << endl;

}

double calculatePayment() const override {

return 1500.0; // Flat fee for demonstration

}

};

// ===== Derived Class: Professor =====

class Professor : public Person {

private:

string department;

string specialization;

string hireDate;

public:

Professor(string n, int a, string id, string c, string dept, string spec, string hDate)

: Person(n, a, id, c), department(dept), specialization(spec), hireDate(hDate) {}

void displayDetails() const override {

Person::displayDetails();

cout << "Department: " << department << "\nSpecialization: " << specialization

<< "\nHire Date: " << hireDate << endl;

}

double calculatePayment() const override {

return 5000.0; // Monthly salary example

}

};

// ===== Test function demonstrating polymorphism =====

void testPolymorphism(const Person\* p) {

p->displayDetails();

cout << "Payment: $" << fixed << setprecision(2) << p->calculatePayment() << "\n\n";

}

int main() {

// Create an array of Person pointers

vector<Person\*> people;

// Add Student and Professor objects

people.push\_back(new Student("Alice", 20, "S1001", "alice@univ.edu", "2023-08-15", "BSc CS", 3.8));

people.push\_back(new Professor("Dr. Bob", 45, "P2001", "bob@univ.edu", "Computer Science", "AI", "2010-05-10"));

people.push\_back(new Student("Charlie", 22, "S1002", "charlie@univ.edu", "2022-09-01", "BBA", 3.5));

people.push\_back(new Professor("Dr. Dana", 50, "P2002", "dana@univ.edu", "Business", "Marketing", "2008-02-20"));

// Demonstrate polymorphic behavior

for (const auto& person : people) {

testPolymorphism(person);

}

// Clean up

for (auto& person : people) {

delete person;

}

return 0;

}

Assignment 3

Part A:

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// ========== BASE CLASS ==========

class Person {

protected:

string name;

int age;

string ID;

string contact;

public:

Person(string n, int a, string i, string c) : name(n), age(a), ID(i), contact(c) {}

virtual void displayDetails() const {

cout << "Name: " << name << "\nAge: " << age

<< "\nID: " << ID << "\nContact: " << contact << endl;

}

virtual double calculatePayment() const {

return 0.0;

}

virtual ~Person() {}

};

// ========== STUDENT CLASS ==========

class Student : public Person {

protected:

string enrollmentDate;

string program;

double GPA;

public:

Student(string n, int a, string i, string c, string ed, string p, double g)

: Person(n, a, i, c), enrollmentDate(ed), program(p), GPA(g) {}

void displayDetails() const override {

Person::displayDetails();

cout << "Program: " << program << "\nEnrollment Date: " << enrollmentDate

<< "\nGPA: " << GPA << endl;

}

double calculatePayment() const override {

return 1500.0;

}

};

// ========== UNDERGRADUATE STUDENT ==========

class UndergraduateStudent : public Student {

private:

string major, minor, expectedGradDate;

public:

UndergraduateStudent(string n, int a, string i, string c, string ed, string p, double g,

string maj, string min, string grad)

: Student(n, a, i, c, ed, p, g), major(maj), minor(min), expectedGradDate(grad) {}

void displayDetails() const override {

Student::displayDetails();

cout << "Major: " << major << "\nMinor: " << minor

<< "\nExpected Graduation: " << expectedGradDate << endl;

}

double calculatePayment() const override {

return 1400.0;

}

};

// ========== GRADUATE STUDENT ==========

class GraduateStudent : public Student {

private:

string researchTopic, advisor, thesisTitle;

public:

GraduateStudent(string n, int a, string i, string c, string ed, string p, double g,

string topic, string adv, string thesis)

: Student(n, a, i, c, ed, p, g), researchTopic(topic), advisor(adv), thesisTitle(thesis) {}

void displayDetails() const override {

Student::displayDetails();

cout << "Research Topic: " << researchTopic << "\nAdvisor: " << advisor

<< "\nThesis: " << thesisTitle << endl;

}

double calculatePayment() const override {

return 1000.0;

}

};

// ========== PROFESSOR BASE CLASS ==========

class Professor : public Person {

protected:

string department, specialization, hireDate;

public:

Professor(string n, int a, string i, string c, string d, string s, string h)

: Person(n, a, i, c), department(d), specialization(s), hireDate(h) {}

void displayDetails() const override {

Person::displayDetails();

cout << "Department: " << department << "\nSpecialization: " << specialization

<< "\nHire Date: " << hireDate << endl;

}

double calculatePayment() const override {

return 5000.0;

}

};

// ========== ASSISTANT PROFESSOR ==========

class AssistantProfessor : public Professor {

private:

bool pursuingTenure;

public:

AssistantProfessor(string n, int a, string i, string c, string d, string s, string h, bool tenure)

: Professor(n, a, i, c, d, s, h), pursuingTenure(tenure) {}

void displayDetails() const override {

Professor::displayDetails();

cout << "Pursuing Tenure: " << (pursuingTenure ? "Yes" : "No") << endl;

}

double calculatePayment() const override {

return 4500.0;

}

};

// ========== ASSOCIATE PROFESSOR ==========

class AssociateProfessor : public Professor {

private:

int publications;

public:

AssociateProfessor(string n, int a, string i, string c, string d, string s, string h, int pubs)

: Professor(n, a, i, c, d, s, h), publications(pubs) {}

void displayDetails() const override {

Professor::displayDetails();

cout << "Publications: " << publications << endl;

}

double calculatePayment() const override {

return 6000.0;

}

};

// ========== FULL PROFESSOR ==========

class FullProfessor : public Professor {

private:

double researchGrant;

public:

FullProfessor(string n, int a, string i, string c, string d, string s, string h, double grant)

: Professor(n, a, i, c, d, s, h), researchGrant(grant) {}

void displayDetails() const override {

Professor::displayDetails();

cout << "Research Grant: $" << researchGrant << endl;

}

double calculatePayment() const override {

return 7000.0 + researchGrant \* 0.05;

}

};

// ========== MAIN FUNCTION ==========

int main() {

vector<Person\*> universityPeople;

// Add a mix of students and professors

universityPeople.push\_back(new UndergraduateStudent("Alice", 20, "U1001", "alice@uni.edu", "2023-08-15", "BSc CS", 3.6, "CS", "Math", "2026-05"));

universityPeople.push\_back(new GraduateStudent("Bob", 25, "G2001", "bob@uni.edu", "2022-09-01", "MSc AI", 3.8, "AI Ethics", "Dr. Smith", "Bias in AI"));

universityPeople.push\_back(new AssistantProfessor("Dr. Carol", 35, "P3001", "carol@uni.edu", "CS", "Networks", "2018-06-10", true));

universityPeople.push\_back(new AssociateProfessor("Dr. Dave", 45, "P3002", "dave@uni.edu", "Business", "Finance", "2010-03-12", 25));

universityPeople.push\_back(new FullProfessor("Dr. Eve", 55, "P3003", "eve@uni.edu", "Physics", "Quantum", "2000-01-15", 20000.0));

for (Person\* p : universityPeople) {

p->displayDetails();

cout << "Calculated Payment: $" << p->calculatePayment() << "\n\n";

}

// Clean up

for (Person\* p : universityPeople) delete p;

return 0;

}

Part B:

class Department {

private:

string name;

string location;

double budget;

vector<Professor\*> professors; // Composition

public:

Department(string n, string l, double b) : name(n), location(l), budget(b) {}

void addProfessor(Professor\* prof) {

professors.push\_back(prof);

}

void showProfessors() const {

cout << "Department: " << name << "\nProfessors:\n";

for (const auto& p : professors) {

p->displayDetails();

}

}

};

class Course {

private:

string code, title, description;

int credits;

Professor\* instructor; // Aggregation

public:

Course(string c, string t, int cr, string d, Professor\* instr)

: code(c), title(t), credits(cr), description(d), instructor(instr) {}

void displayCourse() const {

cout << "\nCourse: " << title << " (" << code << "), Credits: " << credits << endl;

if (instructor) {

cout << "Instructor: ";

instructor->displayDetails();

}

}

};

class StudentEnrollment {

private:

Student\* student;

vector<Course\*> enrolledCourses;

public:

StudentEnrollment(Student\* s) : student(s) {}

void enrollInCourse(Course\* course) {

enrolledCourses.push\_back(course);

}

void showEnrollments() const {

student->displayDetails();

cout << "Enrolled Courses:\n";

for (auto c : enrolledCourses) {

c->displayCourse();

}

}

};

class University {

private:

vector<Department\*> departments;

public:

void addDepartment(Department\* dept) {

departments.push\_back(dept);

}

void showAllDepartments() const {

for (const auto& d : departments) {

d->showProfessors();

cout << endl;

}

}

};

class Classroom {

public:

string roomNumber;

int capacity;

Classroom(string rn, int cap) : roomNumber(rn), capacity(cap) {}

};

class Schedule {

private:

Course\* course;

Classroom\* room;

string timeSlot;

public:

Schedule(Course\* c, Classroom\* r, string t)

: course(c), room(r), timeSlot(t) {}

void showSchedule() const {

cout << "Scheduled: " << course->getTitle()

<< " at " << timeSlot

<< " in Room " << room->roomNumber << endl;

}

};

int main() {

University myUni;

// Create Professors

auto\* prof1 = new AssistantProfessor("Dr. Alice", 40, "P123", "alice@uni.edu", "CS", "AI", "2020", true);

auto\* prof2 = new FullProfessor("Dr. Bob", 55, "P124", "bob@uni.edu", "Math", "ML", "2010", 15000);

// Create Department

auto\* csDept = new Department("Computer Science", "Building A", 100000);

csDept->addProfessor(prof1);

csDept->addProfessor(prof2);

myUni.addDepartment(csDept);

// Create Course

auto\* course1 = new Course("CS101", "Intro to CS", 3, "Basic concepts", prof1);

// Create Student

auto\* student = new UndergraduateStudent("Tom", 19, "S1001", "tom@uni.edu", "2023-08", "BSc CS", 3.7, "CS", "Math", "2026");

// Enroll Student

StudentEnrollment enrollment(student);

enrollment.enrollInCourse(course1);

// Display

myUni.showAllDepartments();

enrollment.showEnrollments();

// Clean up

delete prof1;

delete prof2;

delete csDept;

delete course1;

delete student;

return 0;

}

Part C:

#include <iostream>

#include <string>

#include <vector>

class Person {

protected:

std::string name;

std::string contactInfo;

public:

Person(const std::string& name, const std::string& contactInfo)

: name(name), contactInfo(contactInfo) {}

virtual void displayDetails() const {

std::cout << "Name: " << name << ", Contact Info: " << contactInfo << std::endl;

}

};

class Student : public Person {

protected:

std::string major;

public:

Student(const std::string& name, const std::string& contactInfo, const std::string& major)

: Person(name, contactInfo), major(major) {}

virtual void displayDetails() const override {

std::cout << "Student: " << name << ", Major: " << major << ", Contact Info: " << contactInfo << std::endl;

}

};

class UndergraduateStudent : public Student {

private:

std::string minor;

std::string expectedGraduationDate;

public:

UndergraduateStudent(const std::string& name, const std::string& contactInfo, const std::string& major,

const std::string& minor, const std::string& expectedGraduationDate)

: Student(name, contactInfo, major), minor(minor), expectedGraduationDate(expectedGraduationDate) {}

void displayDetails() const override {

std::cout << "Undergraduate Student: " << name << ", Major: " << major << ", Minor: " << minor

<< ", Expected Graduation Date: " << expectedGraduationDate << ", Contact Info: " << contactInfo << std::endl;

}

};

class GraduateStudent : public Student {

private:

std::string researchTopic;

std::string thesisTitle;

std::string advisor;

bool hasTeachingAssistantship;

bool hasResearchAssistantship;

public:

GraduateStudent(const std::string& name, const std::string& contactInfo, const std::string& major,

const std::string& researchTopic, const std::string& thesisTitle, const std::string& advisor)

: Student(name, contactInfo, major), researchTopic(researchTopic), thesisTitle(thesisTitle), advisor(advisor),

hasTeachingAssistantship(false), hasResearchAssistantship(false) {}

void assignTeachingAssistantship() {

hasTeachingAssistantship = true;

}

void assignResearchAssistantship() {

hasResearchAssistantship = true;

}

void displayDetails() const override {

std::cout << "Graduate Student: " << name << ", Major: " << major << ", Research Topic: " << researchTopic

<< ", Thesis Title: " << thesisTitle << ", Advisor: " << advisor << ", Teaching Assistantship: "

<< (hasTeachingAssistantship ? "Yes" : "No") << ", Research Assistantship: "

<< (hasResearchAssistantship ? "Yes" : "No") << ", Contact Info: " << contactInfo << std::endl;

}

};

class Professor : public Person {

protected:

int yearsOfService;

double baseSalary;

std::string academicRank;

double researchGrants;

public:

Professor(const std::string& name, const std::string& contactInfo, int yearsOfService, double baseSalary,

const std::string& academicRank, double researchGrants)

: Person(name, contactInfo), yearsOfService(yearsOfService), baseSalary(baseSalary), academicRank(academicRank),

researchGrants(researchGrants) {}

virtual double calculatePayment() const {

return baseSalary + researchGrants;

}

virtual void displayDetails() const override {

std::cout << "Professor: " << name << ", Rank: " << academicRank << ", Years of Service: " << yearsOfService

<< ", Contact Info: " << contactInfo << std::endl;

}

};

class AssistantProfessor : public Professor {

public:

AssistantProfessor(const std::string& name, const std::string& contactInfo, int yearsOfService, double baseSalary,

const std::string& academicRank, double researchGrants)

: Professor(name, contactInfo, yearsOfService, baseSalary, academicRank, researchGrants) {}

double calculatePayment() const override {

double additionalSalary = yearsOfService \* 1000; // Add salary based on years of service

return baseSalary + additionalSalary + researchGrants;

}

void displayDetails() const override {

std::cout << "Assistant Professor: " << name << ", Rank: " << academicRank << ", Years of Service: " << yearsOfService

<< ", Contact Info: " << contactInfo << std::endl;

}

};

class AssociateProfessor : public Professor {

public:

AssociateProfessor(const std::string& name, const std::string& contactInfo, int yearsOfService, double baseSalary,

const std::string& academicRank, double researchGrants)

: Professor(name, contactInfo, yearsOfService, baseSalary, academicRank, researchGrants) {}

double calculatePayment() const override {

double additionalSalary = yearsOfService \* 1500; // Add salary based on years of service

return baseSalary + additionalSalary + researchGrants;

}

void displayDetails() const override {

std::cout << "Associate Professor: " << name << ", Rank: " << academicRank << ", Years of Service: " << yearsOfService

<< ", Contact Info: " << contactInfo << std::endl;

}

};

class FullProfessor : public Professor {

public:

FullProfessor(const std::string& name, const std::string& contactInfo, int yearsOfService, double baseSalary,

const std::string& academicRank, double researchGrants)

: Professor(name, contactInfo, yearsOfService, baseSalary, academicRank, researchGrants) {}

double calculatePayment() const override {

double additionalSalary = yearsOfService \* 2000; // Add salary based on years of service

return baseSalary + additionalSalary + researchGrants;

}

void displayDetails() const override {

std::cout << "Full Professor: " << name << ", Rank: " << academicRank << ", Years of Service: " << yearsOfService

<< ", Contact Info: " << contactInfo << std::endl;

}

};

class Department {

private:

std::string name;

std::vector<Professor\*> professors; // Composition: Department contains professors

public:

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

void addProfessor(Professor\* professor) {

professors.push\_back(professor);

}

void displayProfessors() const {

std::cout << "Professors in " << name << " Department:" << std::endl;

for (const auto& professor : professors) {

professor->displayDetails();

}

}

};

class Course {

private:

std::string courseName;

Professor\* instructor; // Aggregation: Course has one professor as instructor

public:

Course(const std::string& courseName, Professor\* instructor)

: courseName(courseName), instructor(instructor) {}

void displayCourseDetails() const {

std::cout << "Course: " << courseName << ", Instructor: ";

instructor->displayDetails();

}

};

class University {

private:

std::vector<Department\*> departments;

public:

void addDepartment(Department\* department) {

departments.push\_back(department);

}

void displayUniversityDetails() const {

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

for (const auto& department : departments) {

department->displayProfessors();

}

}

};

int main() {

// Creating Professors

AssistantProfessor asstProf("Dr. Alice", "alice@uni.edu", 5, 50000, "Assistant Professor", 3000);

AssociateProfessor assocProf("Dr. Bob", "bob@uni.edu", 10, 60000, "Associate Professor", 5000);

FullProfessor fullProf("Dr. Charlie", "charlie@uni.edu", 15, 70000, "Full Professor", 7000);

// Creating Students

UndergraduateStudent undergrad("John Doe", "johndoe@gmail.com", "Computer Science", "Mathematics", "2025");

GraduateStudent grad("Jane Smith", "janesmith@gmail.com", "Physics", "Quantum Mechanics", "Quantum Computing", "Dr. Johnson");

// Assigning Assistantships

grad.assignTeachingAssistantship();

grad.assignResearchAssistantship();

// Creating Department

Department csDepartment("Computer Science");

csDepartment.addProfessor(&asstProf);

csDepartment.addProfessor(&assocProf);

csDepartment.addProfessor(&fullProf);

// Creating Course

Course dataStructures("Data Structures", &assocProf);

// Creating University and Adding Department

University university;

university.addDepartment(&csDepartment);

// Displaying University Details

university.displayUniversityDetails();

// Display Course Details

dataStructures.displayCourseDetails();

return 0;

}

Assignment 4

#include <iostream>

#include <stdexcept>

#include <string>

// Base Exception class

class UniversitySystemException : public std::exception {

protected:

std::string message;

public:

UniversitySystemException(const std::string& msg) : message(msg) {}

virtual const char\* what() const noexcept override {

return message.c\_str();

}

};

// Specific Exception for Enrollment-related errors

class EnrollmentException : public UniversitySystemException {

public:

EnrollmentException(const std::string& msg) : UniversitySystemException(msg) {}

};

// Specific Exception for Grade-related errors

class GradeException : public UniversitySystemException {

public:

GradeException(const std::string& msg) : UniversitySystemException(msg) {}

};

// Specific Exception for Payment-related errors

class PaymentException : public UniversitySystemException {

public:

PaymentException(const std::string& msg) : UniversitySystemException(msg) {}

};

// Any other additional exception types can be added as needed

class EnrollmentManager {

public:

void enrollStudent(int courseCapacity, bool prerequisitesMet, bool enrollmentDeadlinePassed) {

if (courseCapacity <= 0) {

throw EnrollmentException("Course is full, cannot enroll.");

}

if (!prerequisitesMet) {

throw EnrollmentException("Prerequisites not met for the course.");

}

if (enrollmentDeadlinePassed) {

throw EnrollmentException("Enrollment deadline has passed.");

}

// Otherwise, enroll the student

std::cout << "Student successfully enrolled.\n";

}

};

class GradeBook {

public:

void addGrade(int studentId, double grade) {

if (grade < 0 || grade > 100) {

throw GradeException("Invalid grade entered. Grades must be between 0 and 100.");

}

// Else, add the grade

std::cout << "Grade added for student ID: " << studentId << "\n";

}

void finalizeGrades(bool allRequirementsMet) {

if (!allRequirementsMet) {

throw GradeException("Cannot finalize grades as some students haven't met the requirements.");

}

// Finalize grades

std::cout << "Grades finalized.\n";

}

};

class Person {

protected:

std::string id;

std::string contactInfo;

public:

void setId(const std::string& newId) {

if (newId.empty()) {

throw UniversitySystemException("Invalid ID assignment.");

}

id = newId;

}

void setContactInfo(const std::string& newContactInfo) {

if (newContactInfo.empty()) {

throw UniversitySystemException("Invalid contact information.");

}

contactInfo = newContactInfo;

}

};

class Payment {

public:

double calculatePayment(double salary) {

if (salary < 0) {

throw PaymentException("Salary cannot be negative.");

}

return salary;

}

};

#include <fstream>

void logError(const std::string& errorMessage) {

std::ofstream logFile("error\_log.txt", std::ios\_base::app);

logFile << errorMessage << "\n";

}

int main() {

try {

EnrollmentManager enrollmentManager;

enrollmentManager.enrollStudent(0, true, false); // This will throw an exception

} catch (const EnrollmentException& e) {

std::cout << "Enrollment Error: " << e.what() << std::endl;

logError("Enrollment Exception: " + std::string(e.what()));

}

try {

GradeBook gradeBook;

gradeBook.addGrade(123, 105); // Invalid grade, throws exception

} catch (const GradeException& e) {

std::cout << "Grade Error: " << e.what() << std::endl;

logError("Grade Exception: " + std::string(e.what()));

}

try {

Person student;

student.setId(""); // Invalid ID, throws exception

} catch (const UniversitySystemException& e) {

std::cout << "Person Error: " << e.what() << std::endl;

logError("Person Exception: " + std::string(e.what()));

}

try {

Payment payment;

payment.calculatePayment(-100); // Invalid payment, throws exception

} catch (const PaymentException& e) {

std::cout << "Payment Error: " << e.what() << std::endl;

logError("Payment Exception: " + std::string(e.what()));

}

return 0;

}

class UniversitySystem {

private:

std::vector<std::string> departments;

std::vector<std::string> courses;

std::vector<std::string> students;

std::vector<std::string> professors;

public:

void addDepartment(const std::string& departmentName) {

departments.push\_back(departmentName);

}

void addCourse(const std::string& courseName) {

courses.push\_back(courseName);

}

void addStudent(const std::string& studentName) {

students.push\_back(studentName);

}

void addProfessor(const std::string& professorName) {

professors.push\_back(professorName);

}

void generateReports() {

std::cout << "Departments: ";

for (const auto& dept : departments) {

std::cout << dept << " ";

}

std::cout << "\nCourses: ";

for (const auto& course : courses) {

std::cout << course << " ";

}

std::cout << "\nStudents: ";

for (const auto& student : students) {

std::cout << student << " ";

}

std::cout << "\nProfessors: ";

for (const auto& professor : professors) {

std::cout << professor << " ";

}

std::cout << "\n";

}

};

int main() {

UniversitySystem university;

university.addDepartment("Computer Science");

university.addCourse("Data Structures");

university.addStudent("John Doe");

university.addProfessor("Dr. Smith");

bool running = true;

while (running) {

int choice;

std::cout << "1. Create new semester\n";

std::cout << "2. Enroll students\n";

std::cout << "3. Assign professors\n";

std::cout << "4. Enter grades\n";

std::cout << "5. Generate Reports\n";

std::cout << "6. Exit\n";

std::cin >> choice;

switch (choice) {

case 1:

// Create new semester logic

break;

case 2:

// Enroll students logic

break;

case 3:

// Assign professors logic

break;

case 4:

// Enter grades logic

break;

case 5:

university.generateReports();

break;

case 6:

running = false;

break;

}

}

return 0;

}