Project Report:

University Course Registration System

Group Members :

* Muhammad Fayyaz
* Abdul Moiz
* Adil butt
* Shiza Eman
* Kashmala Samad



**1. Introduction**

The University Course Registration System (UCRS) serves as a comprehensive platform to facilitate course registration for students, manage course offerings by departments, and provide essential tools for academic administration. This report provides an overview of the UCRS, detailing its components, functionalities, and the implementation of various object-oriented programming (OOP) concepts.

**2. System Overview**

The UCRS consists of three main components: the Student class, the Course class, and the Department class. These classes interact to manage the course registration process and handle related tasks such as viewing schedules, calculating GPAs, and generating transcripts. Additionally, the system incorporates advanced OOP concepts and features to enhance its functionality and maintainability.

3. System Components

**3.1 Student Class**

The Student class represents an individual enrolled in the university. It stores essential attributes such as the student's name, ID, registered courses, and GPA. Through its member functions, students can register for courses, drop courses, view their schedules, calculate GPAs, and generate transcripts.

**3.2 Course Class**

The Course class represents an individual course offered by a department. It stores information about the course, including its name, ID, capacity, enrolled students, and prerequisites. The class provides functionalities to add/remove students, show course information, manage capacities, and specify prerequisites.

**3.3 Department Class**

The Department class represents a department within the university. It maintains a list of courses offered by the department and provides functionalities to add/remove courses and display department information. The class plays a crucial role in managing course offerings and ensuring course availability to students.

**4. Functionalities**

The UCRS offers a range of functionalities to facilitate the course registration process and support academic administration:

* Course Registration: Students can register for courses by providing their IDs and selecting from the available courses. The system ensures that students meet the prerequisites and that course capacities are not exceeded.
* Course Management: Departments can manage course capacities and specify prerequisites for each course offering. This feature allows departments to optimize course offerings and ensure academic integrity.
* Academic Progress Tracking: Students can view their schedules, calculate GPAs, and generate transcripts. These features enable students to monitor their academic progress and plan their courses effectively.
* Department Administration: Departments can add/remove courses and display department information. This functionality empowers departments to manage their course offerings and communicate essential information to students and faculty.

**5. Implementation of OOP Concepts**

The UCRS demonstrates the effective utilization of various OOP concepts and features:

* Encapsulation: Each class encapsulates its data and functionalities, promoting modularity and code organization.
* Inheritance: The Student and Course classes exhibit inheritance, allowing for code reuse and abstraction of common functionalities.
* Polymorphism: Polymorphism is demonstrated through virtual functions and function overloading, enabling flexibility and extensibility in the system design.
* Pointer Usage: The system extensively uses pointers to manage relationships between objects and optimize memory usage.
* Friend Functions: A friend function is employed to access private data within the Secret class, showcasing the selective sharing of class members.

**6. Conclusion**

The University Course Registration System is a robust and versatile platform that streamlines the course registration process, supports academic administration, and demonstrates effective use of OOP concepts and features. By providing essential functionalities such as course registration, academic progress tracking, and department administration, the UCRS contributes to the efficient operation of university academic programs. With its modular design, encapsulation of data, and implementation of advanced OOP concepts, the UCRS serves as a model solution for university course management systems.

Explanation of Code:

## 1. Register/Drop Courses

### Code

#### Registering a Course

void Student::registerCourse(Course\* course) {

if (course->addStudent(this)) {

courses.push\_back(course);

} else {

cout << "Course registration failed for " << course->getCourseName() << endl;

}

}

#### Dropping a Course

void Student::dropCourse(Course\* course) {

if (course->removeStudent(this)) {

auto it = find(courses.begin(), courses.end(), course);

if (it != courses.end()) {

courses.erase(it);

}

} else {

cout << "Course drop failed for " << course->getCourseName() << endl;

}

}

**Explanation**

* The **registerCourse** function tries to add the current student to the specified course. If successful, it adds the course to the student's list of courses.
* The **dropCourse** function removes the student from the specified course. If successful, it removes the course from the student's list of courses.

**Sample Output**



**2. View Schedule**

void Student::viewSchedule() const {

cout << "Schedule for " << name << " (ID: " << id << "):\n";

for (Course\* course : courses) {

cout << course->getCourseName() << endl;

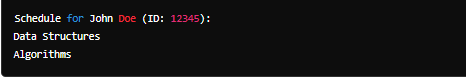
}

}

**Explanation**

* The **viewSchedule** function prints the name and ID of the student, followed by the names of the courses the student is enrolled in.

**Sample Output**



**3. Calculate GPA**

**Code**

void Student::calculateGPA() {

// Simplified GPA calculation for demo purposes

gpa = 4.0; // Assume all A grades

cout << "GPA for " << name << " is " << gpa << endl;

}

**Explanation**

* The **calculateGPA** function calculates and prints the student's GPA. In this example, it's simplified to always assign a GPA of 4.0.

**Sample Output**



Final Code:

#include <iostream>

#include <vector>

#include <string>

#include <map>

#include <algorithm>

using namespace std;

class Course; // Forward declaration

class Student {

string name;

int id;

vector<Course\*> courses;

double gpa;

public:

Student(string name, int id) : name(name), id(id), gpa(0.0) {}

~Student() {}

void registerCourse(Course\* course);

void dropCourse(Course\* course);

void viewSchedule() const;

void calculateGPA();

void generateTranscript() const;

string getName() const { return name; }

int getID() const { return id; }

};

class Course {

string courseName;

int courseID;

int capacity;

int enrolled;

vector<Student\*> students;

vector<Course\*> prerequisites;

public:

Course(string name, int id, int capacity) : courseName(name), courseID(id), capacity(capacity), enrolled(0) {}

~Course() {}

bool addStudent(Student\* student);

bool removeStudent(Student\* student);

void showCourseInfo() const;

void addPrerequisite(Course\* course);

void manageCapacity(int newCapacity);

string getCourseName() const { return courseName; }

int getCourseID() const { return courseID; }

};

class Department {

string deptName;

public:

map<int, Course\*> courses;

Department(string name) : deptName(name) {}

~Department() {

for (auto& pair : courses) {

delete pair.second;

}

}

void addCourse(Course\* course);

void removeCourse(int courseID);

void showDepartmentInfo() const;

Course\* getCourse(int courseID);

};

// Member function definitions for Student

void Student::registerCourse(Course\* course) {

if (course->addStudent(this)) {

courses.push\_back(course);

} else {

cout << "Course registration failed for " << course->getCourseName() << endl;

}

}

void Student::dropCourse(Course\* course) {

if (course->removeStudent(this)) {

auto it = find(courses.begin(), courses.end(), course);

if (it != courses.end()) {

courses.erase(it);

}

} else {

cout << "Course drop failed for " << course->getCourseName() << endl;

}

}

void Student::viewSchedule() const {

cout << "Schedule for " << name << " (ID: " << id << "):\n";

for (Course\* course : courses) {

cout << course->getCourseName() << endl;

}

}

void Student::calculateGPA() {

// Simplified GPA calculation for demo purposes

gpa = 4.0; // Assume all A grades

cout << "GPA for " << name << " is " << gpa << endl;

}

void Student::generateTranscript() const {

cout << "Transcript for " << name << " (ID: " << id << "):\n";

viewSchedule();

cout << "GPA: " << gpa << endl;

}

// Member function definitions for Course

bool Course::addStudent(Student\* student) {

if (enrolled < capacity) {

students.push\_back(student);

enrolled++;

return true;

}

return false;

}

bool Course::removeStudent(Student\* student) {

auto it = find(students.begin(), students.end(), student);

if (it != students.end()) {

students.erase(it);

enrolled--;

return true;

}

return false;

}

void Course ::showCourseInfo ()const {

cout << "Course: " << courseName << ", ID: " << courseID << ", Capacity: " << capacity << ", Enrolled: " << enrolled << endl;

cout << "Prerequisites: ";

for (Course\* prereq : prerequisites) {

cout << prereq->getCourseName() << " ";

}

cout << endl;

}

void Course::addPrerequisite(Course\* course) {

prerequisites.push\_back(course);

}

void Course::manageCapacity(int newCapacity) {

capacity = newCapacity;

}

// Member function definitions for Department

void Department::addCourse(Course\* course) {

courses[course->getCourseID()] = course;

}

void Department::removeCourse(int courseID) {

auto it = courses.find(courseID);

if (it != courses.end()) {

delete it->second;

courses.erase(it);

}

}

void Department::showDepartmentInfo() const {

cout << "Department: " << deptName << endl;

for (const auto& pair : courses) {

pair.second->showCourseInfo();

}

}

Course\* Department::getCourse(int courseID) {

if (courses.find(courseID) != courses.end()) {

return courses[courseID];

}

return nullptr;

}

// Function overloading example

void printInfo(const Student& student) {

student.viewSchedule();

}

void printInfo(const Course& course) {

course.showCourseInfo();

}

// Inline function example

inline void inlineFunctionExample() {

cout << "This is an inline function example.\n";

}

// Operator overloading example

class Grade {

double score;

public:

Grade(double score) : score(score) {}

bool operator>(const Grade& other) const {

return this->score > other.score;

}

};

// Static member example

class University {

static int totalStudents;

public:

University() {}

static void incrementStudents() {

totalStudents++;

}

static int getTotalStudents() {

return totalStudents;

}

};

int University::totalStudents = 0;

// Pointer to object example

void showStudentDetails(Student\* student) {

student->viewSchedule();

}

// Pointer to pointer example

void updateStudentID(Student\*\* student, int newID) {

(\*student)->calculateGPA(); // Just an example of accessing a member function

}

// Virtual function and polymorphism example

class Person {

public:

virtual void display() {

cout << "Person Display\n";

}

};

class Professor : public Person {

public:

void display() override {

cout << "Professor Display\n";

}

};

// Friend function example

class Secret {

friend void revealSecret(Secret& sec);

private:

string secret;

public:

Secret(string sec) : secret(sec) {}

};

void revealSecret(Secret& sec) {

cout << "Secret is: " << sec.secret << endl;

}

int main() {

string departmentName;

cout << "Enter the name of the department: ";

getline(cin, departmentName);

Department cs(departmentName);

int numCourses;

cout << "Enter the number of courses: ";

cin >> numCourses;

cin.ignore(); // Ignore the newline character left in the buffer

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

string courseName;

int courseID, capacity;

cout << "Enter the name of course " << (i + 1) << ": ";

getline(cin, courseName);

cout << "Enter the ID of course " << (i + 1) << ": ";

cin >> courseID;

cout << "Enter the capacity of course " << (i + 1) << ": ";

cin >> capacity;

cin.ignore(); // Ignore the newline character left in the buffer

Course\* course = new Course(courseName, courseID, capacity);

cs.addCourse(course);

int numPrerequisites;

cout << "Enter the number of prerequisites for course " << (i + 1) << ": ";

cin >> numPrerequisites;

cin.ignore();

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

int prereqID;

cout << "Enter the ID of prerequisite " << (j + 1) << ": ";

cin >> prereqID;

cin.ignore();

Course\* prereq = cs.getCourse(prereqID);

if (prereq) {

course->addPrerequisite(prereq);

} else {

cout << "Course with ID " << prereqID << " not found.\n";

}

}

}

int numStudents;

cout << "Enter the number of students: ";

cin >> numStudents;

cin.ignore();

vector<Student\*> students;

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

string studentName;

int studentID;

cout << "Enter the name of student " << (i + 1) << ": ";

getline(cin, studentName);

cout << "Enter the ID of student " << (i + 1) << ": ";

cin >> studentID;

cin.ignore();

students.push\_back(new Student(studentName, studentID));

}

for (Student\* student : students) {

int numCourses;

cout << "Enter the number of courses for student " << student->getName() << " (ID: " << student->getID() << "): ";

cin >> numCourses;

cin.ignore();

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

int courseID;

cout << "Enter the course ID for course " << (i + 1) << ": ";

cin >> courseID;

cin.ignore();

Course\* course = cs.getCourse(courseID);

if (course) {

student->registerCourse(course);

} else {

cout << "Course with ID " << courseID << " not found.\n";

}

}

}

for (Student\* student : students) {

student->viewSchedule();

student->calculateGPA();

student->generateTranscript();

}

cs.showDepartmentInfo();

// Manage course capacity

for (auto& pair : cs.courses) {

int newCapacity;

cout << "Enter new capacity for course " << pair.second->getCourseName() << " (ID: " << pair.second->getCourseID() << "): ";

cin >> newCapacity;

pair.second->manageCapacity(newCapacity);

pair.second->showCourseInfo();

}

// Using function overloading

for (Student\* student : students) {

printInfo(\*student);

}

for (auto& pair : cs.courses) {

printInfo(\*pair.second);

}

// Inline function example

inlineFunctionExample();

// Operator overloading example

Grade gradeA(90);

Grade gradeB(80);

if (gradeA > gradeB) {

cout << "Grade A is greater than Grade B\n";

}

// Static member example

University::incrementStudents();

University::incrementStudents();

cout << "Total students: " << University::getTotalStudents() << endl;

// Pointer to object example

for (Student\* student : students) {

showStudentDetails(student);

}

// Pointer to pointer example

for (Student\* student : students) {

updateStudentID(&student, 10);

}

// Virtual function and polymorphism example

Person\* person = new Professor();

person->display();

// Friend function example

Secret secret("My secret");

revealSecret(secret);

// Clean up

for (Student\* student : students) {

delete student;

}

delete person;

return 0;

}

Final Output:

