

ASSIGNMENT 4

```
// University System with Exception Handling
#include <iostream>
#include <string>
#include <map>
#include <vector>
using namespace std;
class UniversitySystemException {
protected:
    string message;
public:
    UniversitySystemException(string msg) : message(msg) {}
    string what() { return message; }
};
class EnrollmentException : public UniversitySystemException {
public:
    EnrollmentException(string msg) : UniversitySystemException("Enrollment Error: " +
msg) {}
};
class GradeException : public UniversitySystemException {
public:
    GradeException(string msg) : UniversitySystemException("Grade Error: " + msg) {}
};
class PaymentException : public UniversitySystemException {
public:
    PaymentException(string msg) : UniversitySystemException("Payment Error: " + msg)
{}
};
class InvalidDataException : public UniversitySystemException {
public:
    InvalidDataException(string msg) : UniversitySystemException("Invalid Data: " + msg)
{}
};
class Person {
private:
    string name, id, contact;
    int age;
public:
    Person(string n = "", int a = 0, string i = "", string c = "") {
        setName(n);
        setAge(a);
        setID(i);
        setContact(c);
    }
}
```

```

virtual ~Person() {}
void setName(string n) { if (!n.empty()) name = n; }
void setAge(int a) { if (a > 0 && a < 120) age = a; }
void setID(string i) {
    if (i.empty()) throw InvalidDataException("ID cannot be empty.");
    id = i;
}
void setContact(string c) {
    if (c.empty()) throw InvalidDataException("Contact cannot be empty.");
    contact = c;
}
string getName() { return name; }
int getAge() { return age; }
string getID() { return id; }
string getContact() { return contact; }

virtual void displayDetails() {
    cout << "Name: " << name << ", Age: " << age << ", ID: " << id << ", Contact: " <<
contact << endl;
}
virtual float calculatePayment() {
    if (age <= 0) throw PaymentException("Invalid age for payment calculation.");
    return 0.0;
}
};

class Student : public Person {
protected:
    string enrollmentDate, program;
    float gpa;
public:
    Student(string n, int a, string i, string c, string e, string p, float g) : Person(n, a, i, c) {
        enrollmentDate = e;
        program = p;
        setGPA(g);
    }
    void setGPA(float g) {
        if (g < 0.0 || g > 4.0) throw GradeException("GPA must be between 0.0 and 4.0.");
        gpa = g;
    }
    float getGPA() { return gpa; }
    void displayDetails() override {
        Person::displayDetails();
        cout << "Program: " << program << ", GPA: " << gpa << endl;
    }
    float calculatePayment() override { return 15000.0; }
};

class UndergraduateStudent : public Student {

```

```

private:
    string major, minor, gradDate;
public:
    UndergraduateStudent(string n, int a, string i, string c, string e, string p, float g, string
mj, string mn, string gd)
        : Student(n, a, i, c, e, p, g), major(mj), minor(mn), gradDate(gd) {}
    void displayDetails() override {
        Student::displayDetails();
        cout << "Major: " << major << ", Minor: " << minor << ", Graduation: " << gradDate <<
endl;
    }
};

class GraduateStudent : public Student {
private:
    string researchTopic, advisor, thesisTitle;
public:
    GraduateStudent(string n, int a, string i, string c, string e, string p, float g, string rt,
string adv, string tt)
        : Student(n, a, i, c, e, p, g), researchTopic(rt), advisor(adv), thesisTitle(tt) {}
    void displayDetails() override {
        Student::displayDetails();
        cout << "Research: " << researchTopic << ", Advisor: " << advisor << ", Thesis: " <<
thesisTitle << endl;
    }
};

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() override {
        Person::displayDetails();
        cout << "Dept: " << department << ", Spec: " << specialization << endl;
    }
    float calculatePayment() override { return 40000.0; }
};

class AssistantProfessor : public Professor {
public:
    AssistantProfessor(string n, int a, string i, string c, string d, string s, string h) :
Professor(n, a, i, c, d, s, h) {}
    float calculatePayment() override { return 35000.0; }
};

class AssociateProfessor : public Professor {
public:
    AssociateProfessor(string n, int a, string i, string c, string d, string s, string h) :
Professor(n, a, i, c, d, s, h) {}

```

```

    float calculatePayment() override { return 50000.0; }
};
class FullProfessor : public Professor {
public:
    FullProfessor(string n, int a, string i, string c, string d, string s, string h) : Professor(n, a,
i, c, d, s, h) {}
    float calculatePayment() override { return 65000.0; }
};
class Course {
private:
    string code, title, description;
    int credits;
    int capacity;
public:
    Course(string c, string t, string d, int cr, int cap = 30) : code(c), title(t), description(d),
capacity(cap) {
        setCredits(cr);
    }
    void setCredits(int cr) { if (cr > 0) credits = cr; }
    int getCapacity() { return capacity; }
    string getCode() { return code; }
};
class Department {
private:
    string name, location;
    double budget;
    vector<Professor*> professors;
public:
    Department(string n, string l, double b) : name(n), location(l), budget(b) {}
    void addProfessor(Professor* p) { professors.push_back(p); }
};
class University {
private:
    vector<Department> departments;
public:
    void addDepartment(Department d) { departments.push_back(d); }
};
class GradeBook {
private:
    map<string, float> grades;
public:
    void addGrade(string studentID, float grade) {
        if (grade < 0 || grade > 100) throw GradeException("Grade must be between 0 and
100.");
        grades[studentID] = grade;
    }
    float calculateAverageGrade() {

```

```

        float total = 0;
        for (auto g : grades) total += g.second;
        return grades.empty() ? 0 : total
    }
    vector<string> getFailingStudents() {
        vector<string> fail;
        for (auto g : grades) if (g.second < 40) fail.push_back(g.first);
        return fail;
    }
};

class EnrollmentManager {
private:
    map<string, vector<string>> enrollments;
    map<string, Course> courseInfo;
public:
    void addCourse(Course c) { courseInfo[c.getCode()] = c; }
    void enrollStudent(string course, string studentID) {
        if (courseInfo.find(course) == courseInfo.end()) throw EnrollmentException("Course
not found.");
        if (enrollments[course].size() >= courseInfo[course].getCapacity())
            throw EnrollmentException("Course is full.");
        enrollments[course].push_back(studentID);
    }
    void dropStudent(string course, string studentID) {
        auto &list = enrollments[course];
        list.erase(remove(list.begin(), list.end(), studentID), list.end());
    }
    int getEnrollmentCount(string course) { return enrollments[course].size(); }
};

void showDetails(Person* p) {
    p->displayDetails();
    cout << "Payment: Rs. " << p->calculatePayment() << endl;
}

int main() {
    try {
        UndergraduateStudent s1("Kanishk ", 18, "UG101", "12345", "2023-08-01", "CSE",
3.6, "CSE", "Maths", "2027");
        GraduateStudent s2(" Ekagra", 22, "PG202", "54321", "2022-08-01", "CSE", 3.9, "AI",
"Dr. Rao", "AI in Healthcare");
        AssistantProfessor p1("Meena", 35, "AP301", "77889", "CSE", "ML", "2019-07-01");
        FullProfessor p2("Arjun", 50, "FP401", "88990", "CSE", "Quantum", "2005-02-20");

        showDetails(&s1);
        showDetails(&s2);
        showDetails(&p1);
        showDetails(&p2);
        EnrollmentManager em;
    }
}

```

```
    Course c1("CS101", "Intro to CS", "Basics", 3, 1);
    em.addCourse(c1);
    em.enrollStudent("CS101", "UG101");
    em.enrollStudent("CS101", "UG102");
} catch (UniversitySystemException& e) {
    cout << "[ERROR] " << e.what() << endl;
}
return 0;
```