INDEX

CBSE Data Management System

The project aims to create a system for CBSE schools to organize and manage information about students, teachers, and schools. The code implements classes for students, teachers, and schools, allowing users to input data and save details to files. Additionally, it identifies the top-performing student across multiple schools based on their academic marks.

This project demonstrates the implementation of Object-Oriented Programming concepts in C++ to manage educational data within a CBSE school system, facilitating data organization, storage, and retrieval.

CODE

```cpp
#include <bits/stdc++.h>
using namespace std;

// Abstract base class for CBSE entities
class CBSE {
public:
    virtual void display() const = 0; // Pure virtual function for display
};

// Class representing a Student
class Student : public CBSE {
public:
    string name;
    int rollNumber;
    float marks;
    int schoolCode;

    // Constructors
    Student() {}
    Student(string _name, int _rollNumber, float _marks, int _schoolCode){
         name=_name;
         rollNumber=_rollNumber;
         marks=_marks;
         schoolCode=_schoolCode;
    }

    void display() const override { // Override display function to show
student details
        cout << "Student - Name: " << name ;
        cout<< ", Roll Number: " << rollNumber ;
        cout<< ", Marks: " << marks ;
        cout<< ", School Code: " << schoolCode << endl;
    }

    // Function to save student details to a file
    void saveToFile(const string& filename) const {
        ofstream file(filename, ios::app);
        if (file.is_open()) {
            file << name << " " << rollNumber << " " << marks << " " <<
schoolCode << endl;
            file.close();
        } else {
            cout << "Unable to open file!" << endl;
        }
    }
    // Getter functions
    string getName() const {
        return name;
    }

    int getRollNumber() const {
        return rollNumber;
    }

    float getMarks() const {
        return marks;
    }

    int getSchoolCode() const {
        return schoolCode;
    }
};

// Class representing a Teacher
class Teacher : public CBSE {
public:
    string name;
    string subject;
    int schoolCode;

    // Constructors
    Teacher() {}
    Teacher(string _name, string _subject, int _schoolCode) {
        name=_name;
        subject=_subject;
        schoolCode=_schoolCode;
    }

    void display() const override { // Override display function to show
teacher details
        cout << "Teacher - Name: " << name ;
        cout<< ", Subject: " << subject ;
        cout<< ", School Code: " << schoolCode << endl;
    }

    // Function to save teacher details to a file
    void saveToFile(const string& filename) const {
        ofstream file(filename, ios::app);
        if (file.is_open()) {
            file << name << " " << subject << " " << schoolCode << endl;
            file.close();
        } else {
            cout << "Unable to open file!" << endl;
        }
    }
};

// Class representing a School
class School : public CBSE {
private:
    vector<Teacher> teachers;
    vector<Student> students;

public:
    string name;
    string location;
    int schoolCode;

    School() { // Constructor to input school details
        cout << "Enter School Name: ";
        cin >> name;
        cout << "Enter Location: ";
        cin >> location;
        cout << "Enter School Code: ";
        cin >> schoolCode;
    }

    // Function to add a teacher to the school
    void addTeacher(const Teacher& teacher) {
        teachers.push_back(teacher);
    }

    // Function to add a student to the school
    void addStudent(const Student& student) {
        students.push_back(student);
    }

    // Override display function to show school details
    void display() const override {
        cout << "School - Name: " << name ;
        cout<< ", Location: " << location ;
        cout<< ", School Code: " << schoolCode << endl;
    }

    // Function to save school details to a file
    void saveSchoolDetails() const {
        ofstream schoolFile("school.txt", ios::app);
        if (schoolFile.is_open()) {
            schoolFile << name << " " << location << " " << schoolCode <<
endl;
            schoolFile.close();
        } else {
            cout << "Unable to open school file!" << endl;
        }
    }

    // Function to save teachers' details to a file
    void saveTeachersToFile() const {
        ofstream teacherFile("teachers.txt", ios::app);
        if (teacherFile.is_open()) {
            for (const auto& teacher : teachers) {
                teacherFile << teacher.name << " " << teacher.subject << " "
<< teacher.schoolCode << endl;
            }
            teacherFile.close();
        }
        else {
            cout << "Unable to open teachers file!" << endl;
        }
    }

    // Function to save students' details to a file
    void saveStudentsToFile() const {
        ofstream studentFile("students.txt", ios::app);
        if (studentFile.is_open()) {
            for (const auto& student : students) {
                studentFile << student.name << " " << student.rollNumber << "
" << student.marks << " " << student.schoolCode << endl;
            }
            studentFile.close();
        }
        else {
            cout << "Unable to open students file!" << endl;
        }
    }

    // Getter functions
    string getName() const {
        return name;
    }

    string getLocation() const {
        return location;
    }

    int getSchoolCode() const {
        return schoolCode;
    }
};
```

```cpp
// Struct to hold topper student's information
struct TopperInfo {
    string name;
    int rollNumber;
    float marks;
    string schoolName;
    string location;
};

// Function to get the topper student among all schools
template<typename SchoolType>
TopperInfo getTopper(const vector<SchoolType>& schools) {
    ifstream studentFile("students.txt");
    if (!studentFile.is_open()) {
        cerr << "Unable to open students file!" << endl;
        exit(EXIT_FAILURE);
    }

    float maxMarks = 0;
    TopperInfo topper;

    string studentDetails;
    while (getline(studentFile, studentDetails)) {
        stringstream ss(studentDetails);
        string name;
        int roll;
        float marks;
        int code;

        ss >> name;
        ss >> roll;
        ss >> marks;
        ss >> code;

        if (marks > maxMarks) {
            maxMarks = marks;
            topper.name = name;
            topper.rollNumber = roll;
            topper.marks = marks;
            {
                ifstream schoolFile("school.txt");

                string schoolDetails;
                while (getline(schoolFile, schoolDetails)) {
                    stringstream ss2(schoolDetails);
                    string name1;
                    string loc;
                    int code1;

                    ss2 >> name1;
                    ss2 >> loc;
                    ss2 >> code1;
                    if(code1==code){
                        topper.schoolName = name1;
                        topper.location = loc;
                        break;
                    }
                }
            }
        }
    }
    studentFile.close();

    return topper;
}

// Main function
int main() {
    int numSchools;
    cout << "Enter number of schools under CBSE: ";
    cin >> numSchools;

    vector<School> schools;
    for (int i = 0; i < numSchools; ++i) {
        School school;
        schools.push_back(school);
    }

    int numTeachers, numStudents;
    for (int i = 0; i < schools.size(); ++i) {
        cout << "Enter number of teachers for " << schools[i].name << ": ";
        cin >> numTeachers;
        for (int j = 0; j < numTeachers; ++j) {
            string name, subject;
            cout << "Enter Teacher Name: ";
            cin >> name;
            cout << "Enter Teacher Subject: ";
            cin >> subject;
            Teacher teacher(name, subject, schools[i].schoolCode);
            schools[i].addTeacher(teacher);
        }

        cout << "Enter number of students for " << schools[i].name << ": ";
        cin >> numStudents;
        for (int k = 0; k < numStudents; ++k) {
            string name;
            int rollNumber;
            float marks;
            cout << "Enter Student Name: ";
            cin >> name;
            cout << "Enter Student Roll Number: ";
            cin >> rollNumber;
            cout << "Enter Student Marks: ";
            cin >> marks;
            Student student(name, rollNumber, marks, schools[i].schoolCode);
            schools[i].addStudent(student);
        }

        schools[i].saveSchoolDetails();
        schools[i].saveTeachersToFile();
        schools[i].saveStudentsToFile();
    }

    // Get information about the topper student
    TopperInfo topperStudent = getTopper(schools);
    cout << "\nTopper Student Details:" << endl;
    cout << "Name: " << topperStudent.name << ", Roll Number: " <<
topperStudent.rollNumber << ", Marks: " << topperStudent.marks << endl;
    cout << "School Name: " << topperStudent.schoolName << ", Location: " <<
topperStudent.location << endl;

    return 0;
}
```

Major Used Concepts:

Encapsulation:

Encapsulation refers to the bundling of data (attributes) and functions (methods) that manipulate that data within a single unit, which here are the classes Student, Teacher, and School. Each class encapsulates its specific attributes and methods, ensuring data integrity and providing a clear interface for interacting with that data.

- Student Class:

Attributes: name, rollNumber, marks, schoolCode.

Methods: display(), saveToFile(), getName(), getRollNumber(), getMarks(), getSchoolCode().

- Teacher Class:

Attributes: name, subject, schoolCode.

Methods: display(), saveToFile().

- School Class:

Attributes: name, location, schoolCode.

Methods: addTeacher(), addStudent(), display(), saveSchoolDetails(), saveTeachersToFile(), saveStudentsToFile(), getName(), getLocation(), getSchoolCode().

Inheritance:

The code employs inheritance to establish an "is-a" relationship between the CBSE class (an abstract base class) and the Student, Teacher, and School classes.

CBSE Class:

It serves as an abstract base class with a pure virtual function display(), making it an abstract class. This class doesn't have any objects instantiated directly, but it defines an interface for its derived classes.

Student, Teacher, and School Classes:

These classes inherit from the CBSE class and implement the pure virtual function display(), thereby forming an inheritance hierarchy.

Polymorphism:

Polymorphism, the ability of objects to be treated as instances of their parent class, is exemplified through:

Function Overriding:

The display() function is overridden in Student, Teacher, and School classes to exhibit specific information for each.

Virtual Functions:

The display() function in the CBSE class is declared as a pure virtual function, setting an interface for its derived classes to implement their own version.

Abstraction:

Abstraction is a fundamental principle that hides complex implementation details and only provides the necessary functionalities to interact with objects.

Abstract Class (CBSE):

It includes the pure virtual function display(), emphasizing an abstraction layer where concrete implementations are expected in derived classes.

Constructors:

The code employs both default and parameterized constructors in classes like Student, Teacher, and School to initialize object attributes during their creation.

Access Modifiers:

Access modifiers (public, private, protected) are used in classes to control the accessibility of class members:

Private:

Encapsulates certain attributes and methods within classes, limiting direct access from outside the class.

Public:

Allows access to methods required for interaction with objects.

Member Functions:

The classes contain various member functions to perform specific operations:

display() Function:

Outputs information about Student, Teacher, and School objects to the console.

saveToFile() Function:

Saves object details to files (students.txt, teachers.txt, etc.) for later retrieval or persistence.

Getter Functions:

Provide controlled access to private attributes of classes, ensuring encapsulation.

File Handling:

File handling using ofstream and ifstream to save and read data from separate files (students.txt, teachers.txt, etc.).

Template Function:

The getTopper() function is a template function, allowing it to work with a vector of any type (SchoolType in this context).

Struct:

The TopperInfo struct encapsulates information about the topper student, providing a simple way to aggregate related data.

Arrays/Vectors of Objects:

- Vectors in School Class:

The School class maintains vectors (teachers and students) to store objects of the Teacher and Student classes, respectively. These vectors dynamically store multiple instances of these objects.

- Utilization of Vectors:

In methods like addTeacher() and addStudent() within the School class, the vectors (teachers and students) use the push_back() function to add new instances of Teacher and Student objects into their respective collections.

Dynamic Object Creation:

- Loop for Multiple School Instances:

In the main() function, a loop creates multiple instances of the School class based on user input (numSchools). This demonstrates the dynamic creation of multiple school objects.

Object Interaction and Utilization:

- Adding Teachers and Students to Schools:

Inside the loop in the main() function, users input the number of teachers and students for each school. Subsequently, they input details for each teacher and student, which are then added to their respective school's vectors using addTeacher() and addStudent().

Object Retrieval and Manipulation:

- Topper Information Retrieval:

The getTopper() function reads information from the students.txt file to determine the student with the highest marks (topper). It utilizes data about schools from the school.txt file to associate the topper's details with their school information.

Interaction via File Handling:

- Saving Object Details to Files:

saveSchoolDetails(), saveTeachersToFile(), and saveStudentsToFile() methods within the School class write specific details of schools, teachers, and students to separate text files (school.txt, teachers.txt, students.txt, respectively) using file streams (ofstream).

Utilization of Class Methods:

- Use of Class Methods for Object Operations:

Methods like saveToFile(), display(), getName(), getLocation(), getSchoolCode(), etc., provide functionalities for handling object-specific operations and accessing object attributes.

Object-based Data Retrieval:

- TopperInfo Struct Usage:

The TopperInfo struct aggregates data related to the topper student, collating information like name, roll number, marks, school name, and location, which are then accessed and utilized for output in the main() function.

Summary

This C++ project simulates a comprehensive school system following the CBSE curriculum, employing fundamental principles of object-oriented programming. The system comprises three main classes: Student, Teacher, and School, each encapsulating specific attributes and functionalities. Through inheritance, the classes exhibit an inheritance hierarchy, inheriting from an abstract base class (CBSE). Polymorphism is demonstrated via function overriding, enabling distinct implementations of the display() method in each derived class. Users can input multiple schools, teachers, and students, with the system storing their details in separate files using file streams. Dynamic object handling using vectors allows flexible management of teacher and student collections within schools. Overall, this project showcases the practical application of OOP concepts like encapsulation, inheritance, polymorphism, abstraction, file handling, and dynamic object manipulation in creating a structured model of a school system adhering to CBSE standards.

Conclusion

In conclusion, this C++ project stands as an illustrative embodiment of object-oriented programming's core principles within a simulated school system aligned with CBSE guidelines. By leveraging classes for students, teachers, and schools, the project adeptly showcases the essence of encapsulation, inheritance, polymorphism, and abstraction. The program's ability to handle dynamic object creation, storage, and retrieval using vectors and file streams underscores its practicality. Through user interactions to input school data and manage teacher-student relationships, this project not only demonstrates OOP concepts but also offers a tangible representation of a structured educational environment. Ultimately, its seamless integration of OOP principles and functional implementations underscores its utility in modeling real-world scenarios while fostering a deeper understanding of OOP paradigms