PROJECT REPORT ON

Student Record Management System

ASSINGMENT
Of
CODING SKILLS I

(B. TECH CSE – 'D')

Submitted by: -

MOHAMMAD ALI	–	AP24110010311
SACHIN KUMAR	–	AP24110010317
ADITYA KUMAR	–	AP24110010333
SANKALP TIWARI	–	AP24110010356
PRADYUN	–	AP24110010325

Submitted on: - 08 December 2025

Prepared in the partial fulfillment of the
Project Based Learning of Course CSE 201 – CODING SKILLS 1

SRM UNIVERSITY, AP

DECEMBER 2025

CERTIFICATE

This is to certify that the project entitled **“EduVault: Intelligent Student Record Management System”** has been successfully completed by **Mohammad Ali** (AP24110010311) , **Sachin Kumar** (AP24110010317) , **Pradyun** (AP24110010325) , and **Sankalp Tiwari** (AP24110010356) , and as a required academic component of **CSE 201: Coding Skills 1** during Semester 3 of the Academic Year **2025–26** in the Department of Computer Science and Engineering, SRM University–AP.

This work was carried out under the guidance of the undersigned and has satisfactorily met all course requirements as of **08/12/2025**.

(Signature of the Course Instructor)

Mr. Prakash,
Course Instructor (Campus Corporate Connect),
Department of Computer Science and Engineering
SRM University, AP

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to **Dr. T. R. Paarivendhar**, Chancellor, SRM University–AP, and **Prof. Ch. Satish Kumar**, Vice-Chancellor (In-Charge), SRM University–AP, for providing an excellent academic environment that encourages innovation and experiential learning.
I extend my heartfelt thanks to **Mr. Prakash, Course Instructor (Campus Corporate Connect)**, Department of Computer Science and Engineering, SRM University–AP, for their invaluable guidance, continuous support, and insightful feedback throughout the development of our project titled **"EduVault – Intelligent Student Record Management System."** Their mentorship was instrumental in helping us understand and implement critical programming concepts such as **Object-Oriented Programming (OOP), binary file handling for data persistence, and the application of algorithmic logic for automated data analysis** in a practical software environment.

I also express my gratitude to the **Dean, School of Engineering and Sciences**, and the **Head of the Department, Computer Science and Engineering**, for their constant academic encouragement and support.

Finally, we would like to thank our peers and friends for their cooperation and motivation during the execution of this project. This work, prepared in partial fulfilment of the Project-Based Learning component of the course **CSE 201 – Coding Skills I**, has been collaboratively completed by:

- **Mohammad Ali** (AP24110010311)
- **Sachin Kumar** (AP24110010317)
- **Pradyun Nimaje** (AP24110010325)
- **Sankalp Tiwari** (AP24110010356)

Their collective effort and teamwork contributed significantly to the successful completion of this report.

ABSTRACT

This report presents the successful development and implementation of the **"EduVault – Student Record Management System"** project. This work was collaboratively undertaken by the student team comprising **Mohammad Ali (AP24110010311), Sachin Kumar (AP24110010317), Pradyun Nimaje (AP24110010325), and Sankalp Tiwari (AP24110010356)**. The project was completed under the expert supervision of **Mr. Prakash**, Course Instructor (**Campus Corporate Connect**), Department of Computer Science and Engineering, **SRM University–AP**.

This initiative was executed as a key component of the **Project-Based Learning (PBL)** requirement for the course **CSE 201 – Coding Skills I** during the **Academic Year 2025–2026**.

The primary objective of this project is to design and implement a robust console-based application that streamlines the administrative task of handling student academic data. The system tackles the challenge of data organization and persistence by integrating **Object-Oriented Programming (OOP)** principles, specifically **Encapsulation** for data security, **Vector Containers** for dynamic memory management, and **File I/O** for permanent storage. These methodologies collectively demonstrate fundamental concepts of class design, **CRUD (Create, Read, Update, Delete) operations**, and algorithmic data processing.

The project highlights the practical application of **Modular Programming** and **C++ Architecture** to create a functional utility tool. Through the implementation of automated CGPA calculations, search functionalities, and sorting logic, the work successfully reinforces core learning outcomes related to **data structure utilization**, **input validation**, and **software efficiency**. This project not only fulfills the course requirements but also reflects the experiential learning philosophy of **SRM University–AP**.

TABLE OF CONTENTS

Contents

1. Introduction 6
   1.1 Problem Statement 6
   1.2 Project Objectives 7
2. Project Layout/System Architecture 8
   2.1 Architectural Components 8
   2.2. Flow Chart 11
3. Modules Explanation 12
   3.1 Student Data Model (Student Class) 12
   3.2 File Persistence & Storage Module 12
   3.3 Search & Data Retrieval Module 13
   3.4 Sorting & Academic Ranking Module 13
   3.5 User Interface & Display Module 14
   3.6 Statistical Analysis & Reporting Module 14
4. Features Implemented 15
   4.1 Dynamic Student Record Management 15
   4.2 Search & Retrieval System 15
   4.3 Sorting & Organization Logic 15
   4.4 Structured Console Interface 16
   4.5 File Persistence (Data Storage) 16
   4.6 Academic Analytics Dashboard 16
   4.7 Menu-Driven Architecture 16
5. Screenshots 17
6.Conclusion 19

1. Introduction

This report presents the development and implementation of **“EduVault – Student Record Management System.”** This work was undertaken as a key component of the **Project-Based Learning (PBL)** requirement for the course **CSE 201 – Coding Skills 1** during the **Academic Year 2025–2026** at **SRM University–AP**. The primary objective of this project is to design a robust application that streamlines the administrative task of efficiently handling student academic data using **C++**. The project bridges the gap between theoretical programming concepts and practical software development by creating a utility tool that ensures data integrity and ease of access.

The system integrates three core technical components:

- **Object-Oriented Design:** Utilizes principles such as encapsulation to securely model complex student entities (including Name, Roll No, and Marks).
- **File Handling Mechanisms:** Implements binary file I/O to ensure **data persistence**, allowing records to be securely saved, retrieved, and modified across different program sessions without data loss.
- **CRUD Operations & Algorithms:** Facilitates dynamic data management through **C**reate, **R**ead, **U**pdate, and **D**elete functionalities, alongside sorting algorithms for data analysis.

The project showcases the practical application of fundamental programming concepts, including **class architecture**, **vector manipulation**, and **stream-based I/O**. A user-friendly console interface allows real-time interaction, enabling users to input marks, calculate CGPA automatically, and generate merit lists. The system demonstrates efficient memory management and error handling, ensuring data integrity.
This work not only fulfils the course requirements but also reinforces critical thinking in software architecture, data organization, and the hands-on implementation of management systems—core experiential learning outcomes emphasized at SRM University–AP

1.1 Problem Statement

Efficiently managing and analyzing student academic records is a fundamental challenge in educational administration that demonstrates the need for structured data organization and automated logic. The challenge lies in transitioning from manual or volatile data storage methods to a persistent system capable of securely handling student profiles, academic metrics, and performance insights without data loss upon program termination.
The problem addressed in this project is to develop **“EduVault,”** a robust console-based application that leverages **C++ Object-Oriented Programming (OOP)** and **Binary File Handling** to create a persistent student record system. The system must:

- **Establish Data Persistence** by implementing binary file I/O to ensure that student records are securely saved, retrieved, and modified across different program sessions.
- **Model Complex Entities** using OOP principles (Classes and Objects) to encapsulate student data (ID, Name, CGPA, Attendance) and behaviours effectively.
- **Implement Automated Logic** to dynamically calculate performance metrics, such as assigning specific **Badges** (e.g., 'ELITE SCHOLAR') and generating **Skill Predictions** based on academic inputs.
- **Provide a User-Friendly Interface** that allows administrators to seamlessly Add, View, Search, and Update records with visual cues (loading bars, color-coded alerts).
- **Generate Analytic Insights** by aggregating data to display class-wide statistics, such as Average CGPA and Attendance Risk alerts.

The project must be completed within the academic timeframe of **Semester 3 (AY 2025–26)** as part of the **CSE 201 – Coding Skills 1** coursework, fulfilling the **Project-Based Learning** requirement.
This problem statement reflects the need to transform theoretical knowledge of C++ syntax, file streams, and object-oriented design into a practical, working utility that deepens the understanding of software development lifecycles and data persistence strategies.

1.2 Project Objectives

The primary objectives of this project are centered around understanding and applying **Object-Oriented Programming (OOP)** principles and **File Handling** techniques to solve a structured data management problem. The system aims to simulate a real-world academic record system that persists data, automates assessment, and provides actionable insights. The key objectives are:

• To establish secure and persistent data storage
Implement robust **Binary File Handling** using C++ file streams (fstream) to serialize and deserialize student objects. This ensures that records are securely saved to a local database file (eduvault_db.dat) and can be retrieved accurately across multiple program sessions without data loss.

• To apply Object-Oriented Programming (OOP) principles
Design a modular Student class that utilizes **Encapsulation** to protect sensitive data (such as ID and CGPA). Implement public interface methods (Setters/Getters) to control data access and modification, demonstrating the core tenets of secure software design.

• To implement intelligent assessment logic

Develop automated algorithmic logic that processes raw data into meaningful insights. This includes dynamically assigning **Performance Badges** (e.g., *Elite Scholar*, *Risk Alert*) based on academic metrics and generating **AI Skill Predictions** based on the student's branch and standing.

• To enable comprehensive CRUD operations

Construct a full-featured management system that allows administrators to **Create** new records, **Read** (view) specific or all records, **Update** existing details, and **Delete** obsolete entries, ensuring total control over the data lifecycle.

• To design an interactive console interface

Integrate a user-friendly **Command Line Interface (CLI)** featuring dynamic menus, loading animations, and color-coded feedback (e.g., Red for alerts, Green for success) to enhance usability and provide immediate visual confirmation of system actions.

• To generate analytical insights

Implement aggregation functions that traverse the file data to calculate class-wide statistics, such as the **Average CGPA** and the total count of students requiring academic intervention, reinforcing the application of algorithmic thinking to data analysis.

2. Project Layout/System Architecture

The architecture of **EduVault – Persistent Student Record System** is designed to provide a structured and modular framework for managing academic data, executing file persistence operations, and analyzing student performance. The system is organized into distinct functional layers, each responsible for a critical aspect of the overall application. This modular architecture ensures clarity, maintainability, and ease of future expansion..

The system follows a logical execution flow, starting from **object initialization and data loading**, progressing through **interactive CRUD (Create, Read, Update, Delete) operations**, and concluding with **secure binary file storage**. Each module interacts through well-defined **Object-Oriented classes** and **file streams**, enabling efficient data encapsulation and seamless state management between the volatile memory (RAM) and the persistent storage (Disk).

2.1 Architectural Components

The system architecture consists of the following core components:

2.1.1. Data Representation Layer (Student Class)

- **Encapsulates** all student-related data including ID, name, branch, and CGPA into a single, cohesive unit.
- **Implements** security through private member variables, accessible only via public setter and getter methods.
- **Defines** the blueprint for object instantiation, ensuring that every record created follows a consistent data structure.
- **Calculates** derived attributes dynamically, such as grade classification and skill predictions.

2.1.2 File Persistence Module (Binary Database)

- **Manages** the eduvault_db.dat binary file, acting as the permanent storage backend.
- **Utilizes** fstream (File Stream) objects to perform low-level input/output operations.
- **Handles** serialization (converting objects to binary stream) and deserialization (reconstructing objects from binary stream).
- **Ensures** data integrity by appending new records and properly managing file pointers during read/write cycles.

2.1.3 Record Management Logic (CRUD Engine)

- **Create:** Captures user input, validates data types, creates a Student object, and commits it to storage.
- **Read:** Iterates through the binary file to display individual records or listing all students in a formatted table.
- **Update:** searches for a specific record by ID, modifies the necessary fields in memory, and overwrites the specific file segment.
- **Delete:** Creates a temporary file to copy all valid records while excluding the specific ID marked for deletion, then renames the file to update the database.
-

2.1.4 Analytical Engine & Algorithm Layer

- **Traverses** the entire dataset to compute aggregate metrics.
- **Calculates** the Class Average CGPA to provide a benchmark for performance.
- **Identifies** high-risk students (CGPA < 5.0) and high-performers (CGPA > 9.0) using conditional logic loops.

- **Generates** automated insights, such as "Skill Predictions" based on the student's branch and academic standing.

2.1.5 User Interface (CLI) & Visualization Layer

- **Renders** the interactive menu system using do-while loops and switch-case structures..
- **Enhances** user experience with ANSI color codes (Green for success, Red for warnings/errors)
- **Implements** utility functions like loadingAnimation() and clearScreen() to simulate a polished software environment.
- **Formats** output into aligned columns and tables for readability.

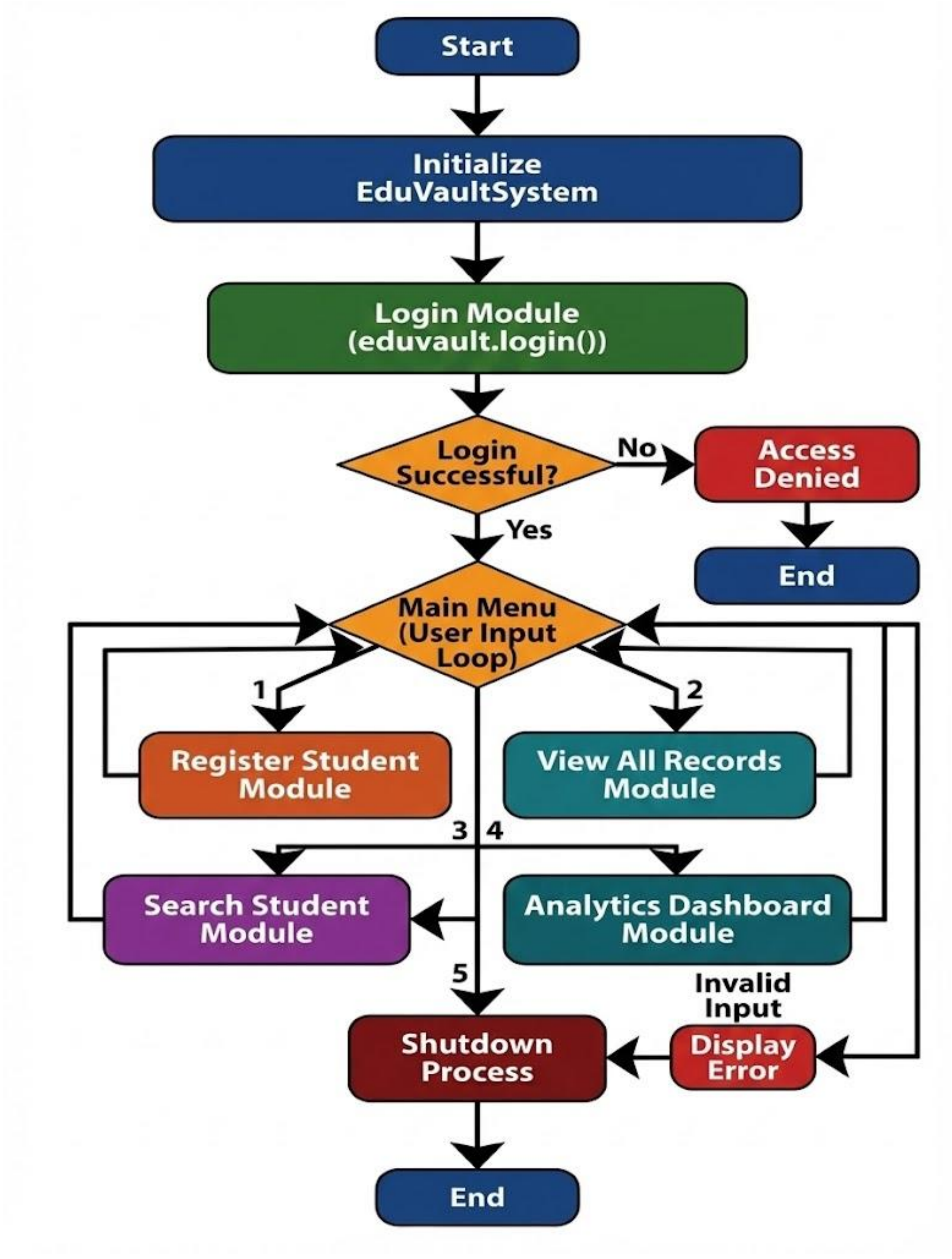
2.1.6 Control Flow (Main Driver)

The main() function orchestrates the application lifecycle:

1. **Initializes** the system and checks for database existence
2. **Loops** the main menu to keep the program running until the user chooses to exit
3. **Routes** user choices to the appropriate helper functions (addStudent, displayAll, searchStudent, etc.)
4. **Handles** exceptions and input errors to prevent program crashes

This layer integrates all functional modules into a robust, event-driven workflow.

2.2. Flow Chart



3. Modules Explanation

The **EduVault** system is organized into multiple functional modules, each responsible for a specific stage of the application—from secure data entry and persistent storage to automated performance analysis and reporting. This modular approach enhances readability, maintainability, and allows each component to be independently understood and improved. The following subsections provide a detailed explanation of each module implemented in the project.

3.1 Student Data Model (Student Class)

This module provides the foundational data structure for the entire system, acting as the blueprint for every student record.

Key elements include:

- **Private Data Members:** Stores core attributes such as id, name, branch, cgpa, and attendance within a secure, encapsulated scope to enforce data hiding.
- **Automated Business Logic:** Includes internal methods like calculateBadge() and generateAISkill() to automatically derive performance metrics (e.g., "ELITE SCHOLAR") and career predictions based on raw academic input.
- **Format Handling:** Manages the visual representation of data, applying conditional color codes (e.g., **Red** for low attendance, **Cyan** for high achievers) during display.

This module ensures consistent data representation and enforces encapsulation, preventing direct, unauthorized access to sensitive fields while standardizing how student objects are created and manipulated.

3.2 File Persistence & Storage Module

This module is responsible for managing the input/output operations required to maintain data continuity. It bridges the gap between temporary memory (RAM) and permanent storage (disk), ensuring that student records are not lost when the program terminates.

Key Steps:

1. **Data Serialization:** Converts complex Student objects into a structured text format (e.g., CSV or specific delimiters) suitable for file writing.
2. Secure File Handling:
   - Uses std::ofstream to write current records to a dedicated database file (students.txt).
   - Uses std::ifstream to read and parse data upon system startup.
3. **Error Management:** Checks for file existence and integrity to prevent crashes if the database is missing or corrupted.
4. **Session Synchronization:** Automatically loads existing data into the system's memory vectors at launch and saves all changes before exit.

3.3 Search & Data Retrieval Module

This module implements the logic required to navigate the student dataset, allowing the system to locate specific records within the dynamic memory.

Core Functionalities:

- **Linear Traversal:** Iterates through the std::vector<Student> container to inspect records one by one, simulating a search path through the data.
- Dual-Mode Querying:
    - **Search by ID:** Performs an exact match verification to find unique student records.
    - **Search by Name:** Implements string comparison to allow finding students even if only part of the name is known.
- **Edge Case Handling:** Detects "dead-ends" (where no data matches the query) and provides user-friendly feedback (e.g., "Record Not Found") rather than system errors.
- **Instant Retrieval:** Terminates the search loop immediately upon finding a match to optimize performance.

This module demonstrates efficient data access capability, ensuring that specific user details can be isolated, viewed, or modified without manually scrolling through the entire database.

3.4 Sorting & Academic Ranking Module

This module implements systematic ordering algorithms (such as Bubble Sort or Selection Sort) to organize the student dataset, akin to finding the optimal arrangement or "shortest path" to a sorted list.

Major Components:

- **Comparison Logic:** Iteratively compares student records (e.g., by CGPA or Roll Number) to determine hierarchy, managing the order of processing.
- **Swapping Mechanism:** Physically rearranges objects within the std::vector to prevent data fragmentation and ensure correct sequencing.
- **Multi-Pass Traversal:** Expands through the entire dataset repeatedly until the "target state" (a fully sorted list) is reached.
- **Leaderboard Generation:** visualizes the final sorted data in a tabular format, highlighting the top performers.

This module illustrates the system's analytical capability to restructure data for better readability, contrasting with the localized lookup nature of the Search module.

3.5 User Interface & Display Module

This module handles the visual presentation of the student database, ensuring that raw memory data is rendered into a user-friendly console dashboard.

Responsibilities:

- **Console Management:** Uses platform-specific commands (system("cls") or clear) to refresh the screen between operations, ensuring a clutter-free workspace.
- **Data Rendering:** Displays student records, menu options, and academic headers using **ANSI color codes** (e.g., Green for high grades, Red for errors) to enhance readability and highlight critical information.
- **Navigation Control:** Implements input pauses (e.g., getchar() or cin.get()) to allow users to review output before the screen clears or the menu reloads.
- **Structured Layouts:** Organizes complex data into aligned tables and grids, making it easy to compare student statistics at a glance.

By prioritizing layout and readability, this module transforms abstract data storage into an intuitive, interactive management system.

3.6 Statistical Analysis & Reporting Module

At the end of a session or upon specific request, this module aggregates the processed data to generate a comprehensive academic summary, comparing individual performance against the class average.

Responsibilities:

- **Data Aggregation:** Calculates key metrics such as the **Class Average CGPA**, total enrolment count, and the Pass/Fail ratio based on stored records.
- **Performance Benchmarking:** Identifies the **Class Topper** (highest CGPA) and highlights students falling below the passing threshold, effectively creating a "shortest path" to identifying academic needs.
- **Operational Summary:** Tracks the total number of records added, modified, or deleted during the current runtime session to ensure data consistency.
- **Storage Verification:** Validates that the std::vector or file storage correctly reflects the latest sorting and manipulation operations without data loss.

A **summary report** is printed to help the user understand the overall academic standing of the batch and the effectiveness of the data management operations.

4. Features Implemented

The **EduVault** system integrates multiple Object-Oriented Programming (OOP) concepts and data management techniques to create a robust, persistent environment for academic record keeping. The following features were successfully implemented as part of the project:

4.1 Dynamic Student Record Management

- **Encapsulated Data Entry:** Uses a class-based structure (`class Student`) to securely handle data inputs (Name, Roll No, Marks, CGPA).
- **Runtime Memory Allocation:** Utilizes `std::vector` (or dynamic arrays) to allow the database to grow or shrink dynamically as records are added or removed.
- **Uniqueness Validation:** Ensures data integrity by checking for duplicate Roll Numbers before accepting new entries.
- **Automatic Calculation:** Automatically computes the CGPA and Grade based on the raw subject marks provided during input.

4.2 Search & Retrieval System

- **ID-Based Lookup:** Implements a linear search algorithm to locate specific student records by their unique Roll Number.
- **Error Handling:** Provides user feedback (e.g., "Record Not Found") when searching for non-existent IDs, preventing system crashes.
- **Detail View:** Displays the full academic profile of a student upon a successful match, including individual subject scores and contact info.

4.3 Sorting & Organization Logic

- **Academic Ranking:** Implements sorting algorithms (e.g., Bubble Sort or Selection Sort) to organize students based on CGPA (Descending) to identify class toppers.
- **Roll Call Ordering:** Can re-sort the database by Roll Number (Ascending) for standard administrative viewing.
- **Filtering:** Segregates data to display specific lists, such as "List of Students with Backlogs" or "Dean's List."

4.4 Structured Console Interface

- **Formatted Output:** Uses std::setw and std::left/right manipulators to align data into clean, readable tables (columns for Name, ID, CGPA, etc.).
- **Visual Cues:** Implements **ANSI colour codes** to highlight important statuses:
- **Green:** High grades / Success messages.
- **Red:** Errors / Failed subjects.
- **Cyan:** Headers and Borders.
- **Screen Management:** Uses system("cls") or system("clear") to refresh the dashboard, ensuring the interface remains uncluttered during navigation.

4.5 File Persistence (Data Storage)

- **Save Mechanism:** Writes all student objects to a physical file (e.g., students.txt or .csv) upon closing or explicit save requests.
- **Load Mechanism:** Reads data from the file at system startup, restoring the previous state so that records are not lost between sessions.
- **Data Serialization:** Formats the object data into a structured string stream for efficient writing and reading.

4.6 Academic Analytics Dashboard

- **Statistical Analysis:** At the user's request, the system generates a summary report calculating:
    - **Class Average CGPA.**
    - **Total Enrolment Count.**
    - **Pass vs. Fail Ratio.**
- **Performance Insights:** Helps faculty/admins understand the overall batch performance at a glance.

4.7 Menu-Driven Architecture

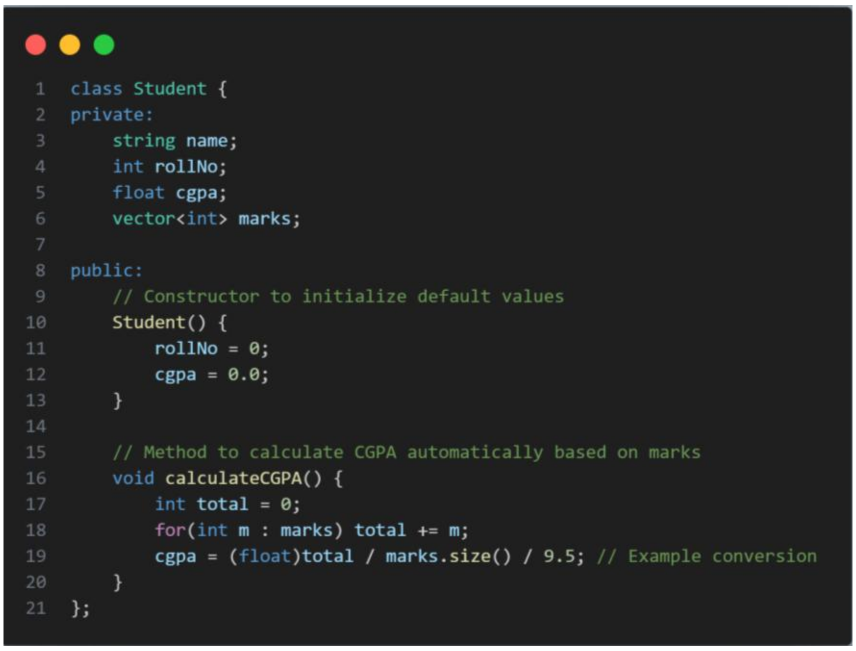
- **Interactive Control Loop:** Uses a switch-case driven menu system inside a while loop to keep the program running until the user chooses to exit.
- **Input Validation:** Sanitize user inputs to prevent infinite loops (e.g., entering a character when an integer is expected).
- **User Prompts:** "Press Enter to Continue" features allow the user to read data before the screen clears.

This ensures a professional, user-friendly, and persistent workflow for managing academic data.

5. Code Snippet

Code Snippet 1: The "Blueprint" (Class Encapsulation) Logic:

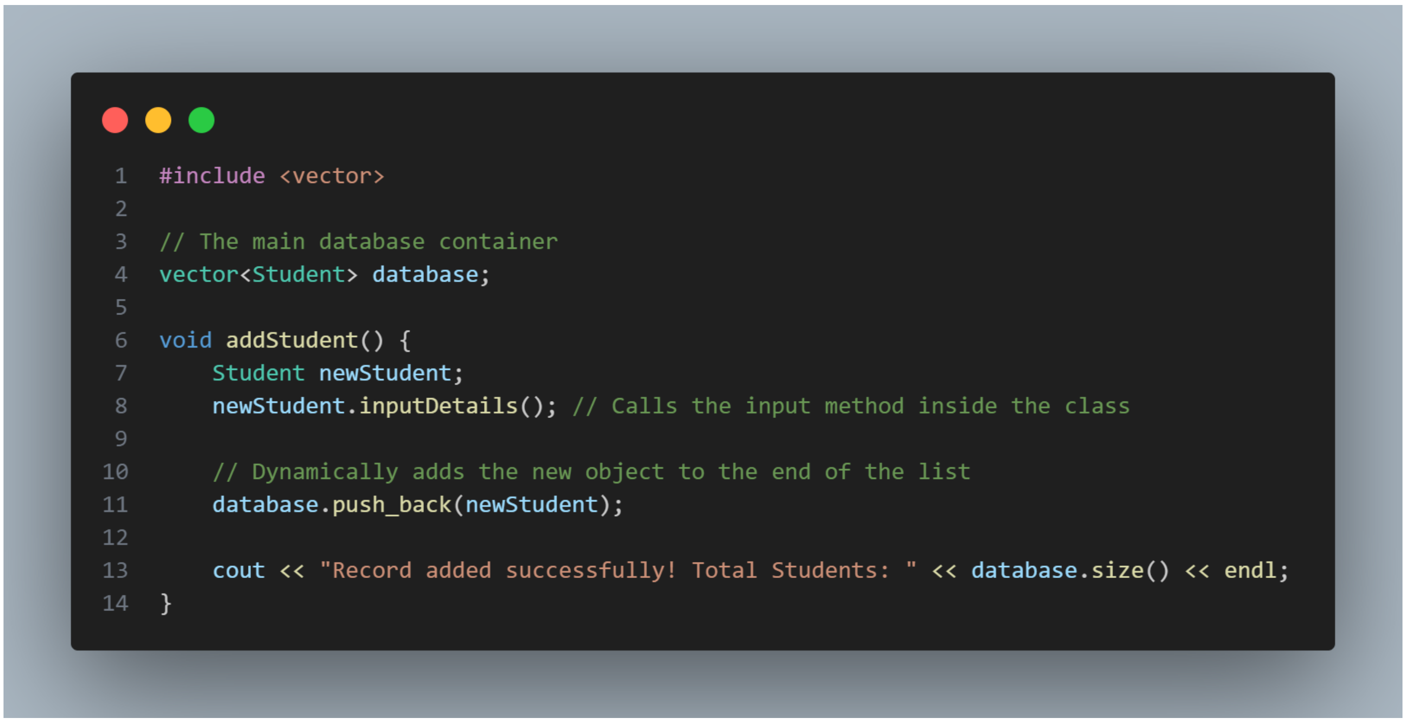
Instead of managing loose variables for every student (which is messy like separate if statements), we use a **Class** to bundle data (Name, Roll No, Marks) and behaviors (Input, Calculate, Display) into a single unit.



```cpp
class Student {
private:
    string name;
    int rollNo;
    float cgpa;
    vector<int> marks;

public:
    // Constructor to initialize default values
    Student() {
        rollNo = 0;
        cgpa = 0.0;
    }

    // Method to calculate CGPA automatically based on marks
    void calculateCGPA() {
        int total = 0;
        for(int m : marks) total += m;
        cgpa = (float)total / marks.size() / 9.5; // Example conversion
    }
};
```

Code Snippet 2: Dynamic Database (Vector Implementation) Logic:

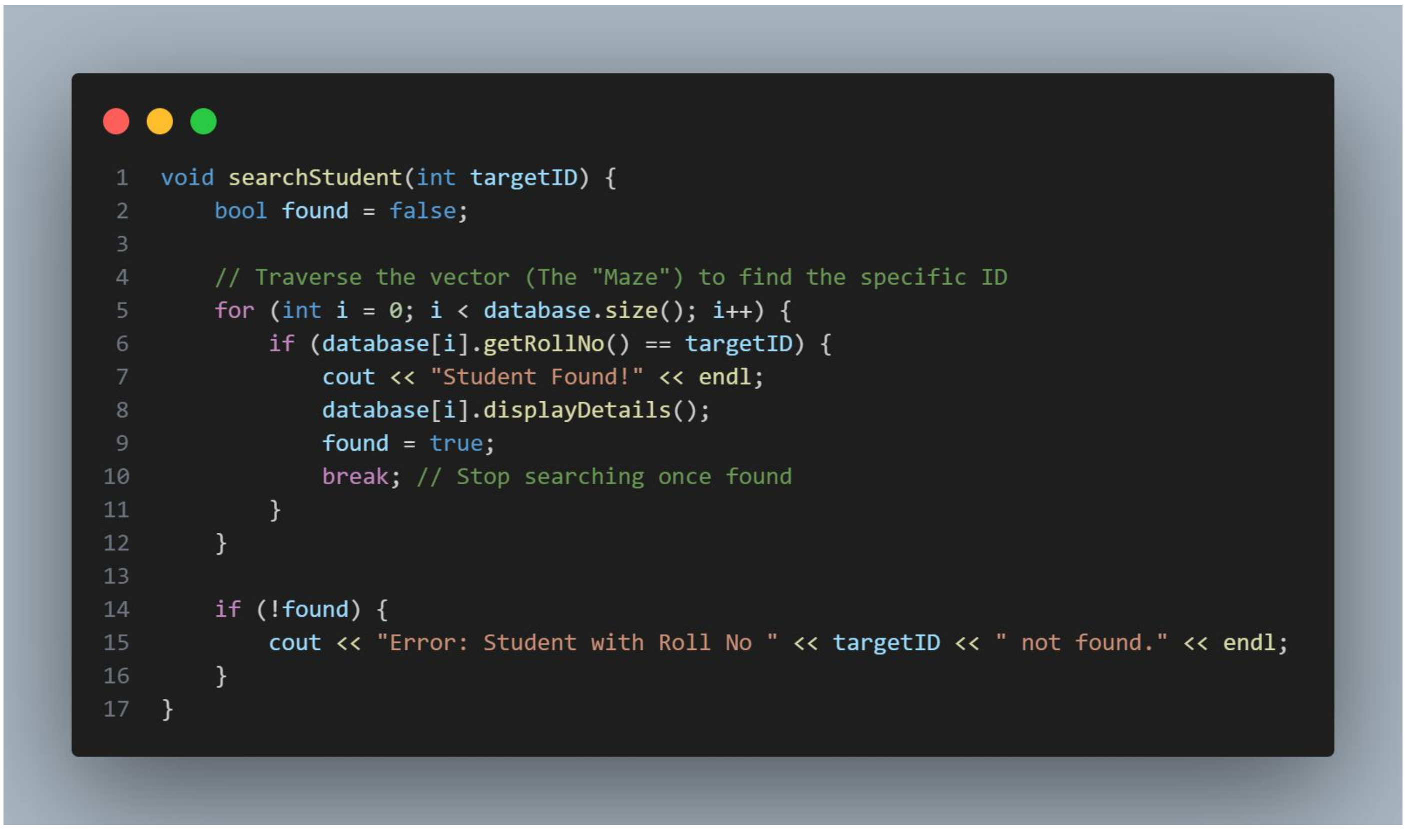
Unlike a static array which has a fixed size, we use std::vector to allow the database to grow dynamically. This allows us to add infinite records without crashing the system, similar to generating a maze of unknown size.



```cpp
#include <vector>

// The main database container
vector<Student> database;

void addStudent() {
    Student newStudent;
    newStudent.inputDetails(); // Calls the input method inside the class

    // Dynamically adds the new object to the end of the list
    database.push_back(newStudent);

    cout << "Record added successfully! Total Students: " << database.size() << endl;
}
```

Code Snippet 3: Search Solver (Linear Search) Logic:

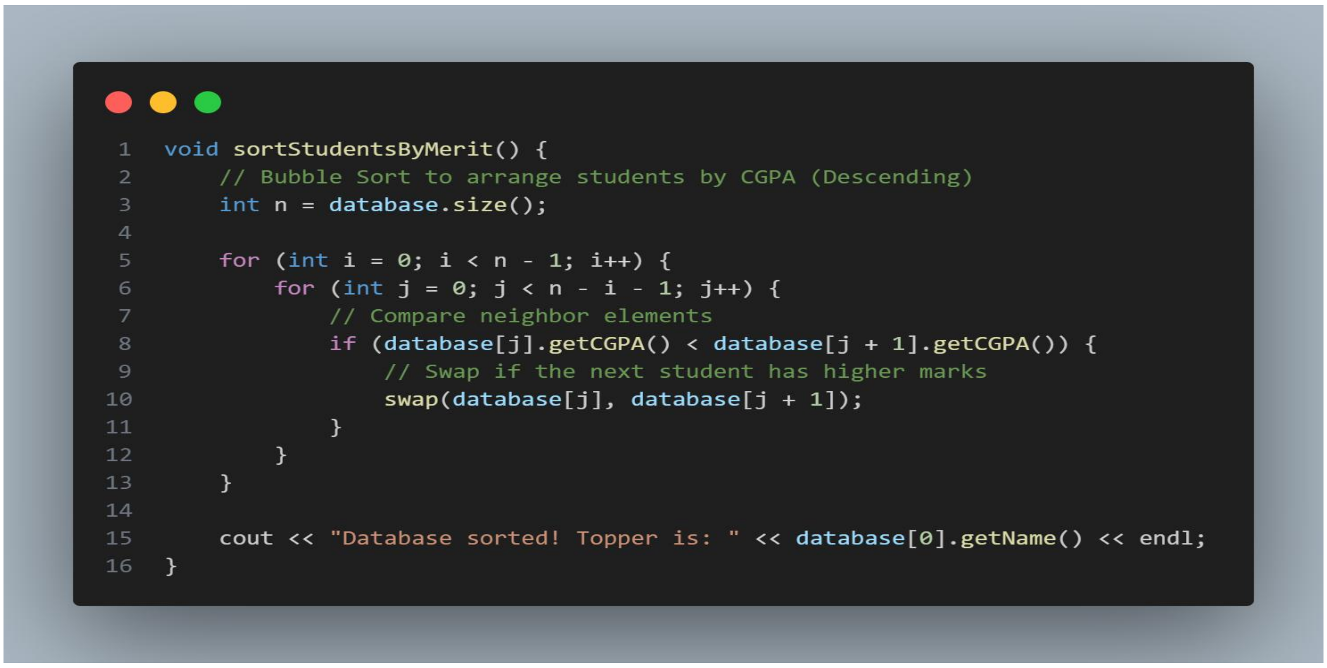
To find a specific student, we iterate through the vector one by one. If the Roll Number matches the target, we display the details. This acts as our "pathfinding" to locate data within the system..



```cpp
void searchStudent(int targetID) {
    bool found = false;

    // Traverse the vector (The "Maze") to find the specific ID
    for (int i = 0; i < database.size(); i++) {
        if (database[i].getRollNo() == targetID) {
            cout << "Student Found!" << endl;
            database[i].displayDetails();
            found = true;
            break; // Stop searching once found
        }
    }

    if (!found) {
        cout << "Error: Student with Roll No " << targetID << " not found." << endl;
    }
}
```

Code Snippet 4: Ranking Solver (Sorting Algorithm) Logic:

To identify the class topper, we rearrange the students based on their CGPA. Using a simple Bubble Sort or Selection Sort, we move the student with the highest marks to the top, organizing the data systematically.



```cpp
void sortStudentsByMerit() {
    // Bubble Sort to arrange students by CGPA (Descending)
    int n = database.size();

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // Compare neighbor elements
            if (database[j].getCGPA() < database[j + 1].getCGPA()) {
                // Swap if the next student has higher marks
                swap(database[j], database[j + 1]);
            }
        }
    }

    cout << "Database sorted! Topper is: " << database[0].getName() << endl;
}
```

6.Conclusion

The **EduVault** Student Management System successfully demonstrates how **Object-Oriented Programming (OOP)** principles operate within a robust data management application. By integrating **Encapsulation**, **File Handling**, and **Dynamic Memory Management** into a unified system, the project provides a practical understanding of software architecture, data persistence, and database logic.

The system effectively showcases the importance of structured data organization. While raw inputs are chaotic, the implementation of **Sorting Algorithms** and **Search Logic** transforms this data into actionable information, allowing for the efficient retrieval of student profiles and academic rankings. The distinct separation between the user interface and the backend logic ensures that the data remains secure and consistent throughout the session.

Through **modular design** and structured class implementation, this project bridges the gap between theoretical C++ concepts and hands-on application. It reinforces essential ideas in **constructor usage**, **vector manipulation**, **stream handling**, and **algorithmic efficiency**. Moreover, the inclusion of the statistical analysis module provides meaningful insights into batch performance, simulating real-world administrative tools.
Overall, the **EduVault** system fulfils its intended objectives and serves as an effective educational tool, offering clear, practical exposure to fundamental software development concepts and the creation of persistent utility applications.

7. Future Enhancements

While the current version of "EduVault" successfully achieves its primary objectives of data persistence and student record management using C++ and file streams, there is significant scope for expansion. Future iterations of this project could incorporate the following advanced features to evolve into a commercial-grade application:

7.1 Migration to Relational Database (SQL)

- **Current Limitation:** The system currently relies on binary/text files (fstream) for storage. While effective for small datasets, this approach can lead to concurrency issues and slower performance as the data grows.
- **Enhancement:** Integrating a Relational Database Management System (RDBMS) like **MySQL** or **SQLite**. This would allow for complex queries, faster data retrieval, and better data integrity through foreign key constraints.

7.2 Graphical User Interface (GUI)

- **Current Limitation:** The application operates on a Command Line Interface (CLI), which, while functional, requires keyboard-only navigation.
- **Enhancement:** Developing a GUI using frameworks such as **Qt** or **wxWidgets**. A visual interface with buttons, forms, and drag-and-drop capabilities would significantly improve the User Experience (UX) and make the software accessible to non-technical staff.

7.3 Role-Based Access Control (RBAC) & Security

- **Current Limitation:** The current system allows open access or a single-tier login.
- **Enhancement:** Implementing a multi-tier security system with distinct login credentials for **Administrators** (full access), **Faculty** (read/update specific subjects), and **Students** (read-only access to their own records). Additionally, sensitive data like passwords and personal details should be encrypted using hashing algorithms (e.g., SHA-256).

7.4 Advanced Data Visualization

- **Current Limitation:** Analytics are presented as text-based summaries (e.g., "Class Average: 8.5").
- **Enhancement:** Integrating a library to generate visual reports. This could include bar charts showing grade distributions or line graphs tracking a student's performance trend over multiple semesters.

7.5 Cloud Integration and Web Interface

- **Current Limitation:** The data is stored locally on the host machine.
- **Enhancement:** Porting the application to a web-based architecture. By hosting the database on a cloud server (e.g., AWS or Azure), administrators could access the system from any device with an internet connection, enabling real-time remote management.

8. References

The development of this project relied on various academic resources, documentation, and standard textbooks to ensure adherence to C++ best practices and software engineering principles.

Textbooks:

1. **Balagurusamy, E.** (2020). *Object-Oriented Programming with C++* (8th Ed.). McGraw Hill Education. (Used for concepts on Classes, Objects, and Encapsulation).
2. **Stroustrup, B.** (2013). *The C++ Programming Language* (4th Ed.). Addison-Wesley. (Used for reference on Standard Template Library and Vector implementation).
3. **Lafore, R.** (2001). *Object-Oriented Programming in C++* (4th Ed.). Sams Publishing. (Used for File Handling and Stream I/O concepts).
4.

Online Resources & Documentation:

1. **cplusplus.com** – Reference for std::vector and std::fstream syntax and error handling.
2. **GeeksforGeeks** – Tutorials on Sorting Algorithms (Bubble Sort) and Linear Search implementation in C++.
3. **Microsoft C++ Documentation (learn.microsoft.com)** – Guidelines on standard input/output formatting and console manipulation.

Course Materials:

1. **SRM University–AP Courseware**: Lecture notes and lab manuals for *CSE 201 – Coding Skills I*, Semester 3, Academic Year 2025–26.