1. COVER PAGE

STUDENT INFORMATION SYSTEM (SIS)   TERM PROJECT REPORT

Course: CSE102 - Computer Programming

Submission Date: January 16, 2026

Prepared by: Name: Yener Tuncer

Student ID: 230104004081

Word Count: (7582)

2. EXECUTIVE SUMMARY

2.1. Project Overview

The primary objective of this term project was to design, implement, and validate a comprehensive **Student Information System (SIS)** using the C programming language. Developed within the scope of the **CSE102 - Computer Programming** course, this system serves as a robust console-based application capable of managing the entire academic lifecycle of a university environment. The project aims to simulate real-world software engineering challenges, requiring the implementation of complex data relationships, persistent storage mechanisms, and strict memory management protocols.

The developed system provides a centralized platform for administrators to manage **Students, Courses, Professors, Enrollments, and Grades**. Unlike simple data entry programs, this SIS enforces logical constraints such as course prerequisites, capacity limits, and unique identifier validation, ensuring data integrity throughout the system's operation.

2.2. Architectural Design & Technologies

To achieve flexibility and scalability, the system architecture deviates from static array-based implementations in favor of **dynamic memory allocation**.

- **Data Structures:** The core backbone of the system is built upon **Singly Linked Lists**. This design choice allows the system to handle an arbitrary number of records without the need for pre-compilation resizing. Each entity (Student, Course, etc.) is represented as a node in a dynamic chain, optimizing memory usage by allocating resources only when necessary.
- **Data Persistence:** To ensure that data remains available across different sessions, the system implements a custom file I/O layer. Data is serialized into **CSV (Comma-Separated Values)** format. This industry-standard format was chosen for its human-readability and ease of debugging, allowing administrators to verify data integrity using external tools like Excel or Notepad.
- **Modular Programming:** Adhering to the "Separation of Concerns" principle, the codebase is organized into distinct modules (e.g., student.c, course.c, enrollment.c). This modular approach not only enhances code maintainability but also allowed for isolated unit testing of individual components before system integration.

2.3. Key Features Implemented

The final system delivers a feature-rich environment that exceeds the baseline requirements:

1. **Core Management Modules:** Full CRUD (Create, Read, Update, Delete) capabilities are implemented for Students, Courses, and Professors.
2. **Advanced Enrollment Logic:** The system features a sophisticated enrollment engine that performs real-time checks. It verifies if a student has completed the required **prerequisite courses** and ensures that the target course has not exceeded its **capacity**.
3. **Automated Grading & Reporting:** The system automatically converts numeric grades to letter grades and dynamically calculates the **Grade Point Average (GPA)** for students upon request. Comprehensive reports, such as Student Transcripts and Course Rosters, are generated instantly.
4. **Bonus Feature - Curriculum Management:** Going beyond the standard scope, a "Curriculum Management" module was integrated. This feature tracks degree requirements and automatically evaluates a student's eligibility for graduation based on total credits earned and mandatory course completion.

2.4. Development Methodology & AI Collaboration

Modern software development practices were strictly followed throughout the project lifecycle. A significant aspect of this project was the integration of **Large Language Models (LLMs)**, specifically Google Gemini, into the workflow.

This collaboration functioned as an advanced form of "Pair Programming." The AI tools were utilized not for generating code blindly, but for:

- **Architectural Brainstorming:** Designing the struct relationships between Enrollments and Grades.
- **Debugging:** Identifying obscure "Segmentation Faults" and logic errors in pointer arithmetic.
- **Refactoring:** Optimizing the Makefile script for cross-platform compatibility between Windows and Linux environments.

2.5. Challenges & Solutions

Developing a system of this complexity in C presented several technical challenges:

- **Memory Management:** The most significant challenge was managing the Heap memory manually. Early versions of the project faced potential memory leaks during the data reloading process. This was overcome by implementing dedicated free_all functions for each structure and verifying memory safety using **Valgrind**. The final system is verified to be memory leak-free.
- **Data Integrity (Cascade Deletion):** Deleting a record (e.g., a Student) required removing all associated data (Enrollments and Grades) to prevent "dangling references." A recursive search-and-delete algorithm was implemented to handle these cascade deletions safely.
- **Cross-Platform Compatibility:** Adapting Linux-based tools (like make) for a Windows development environment required writing a robust Makefile and handling console encoding issues (Turkish characters), which were resolved by implementing a custom input sanitization utility.

2.6. Conclusion

The resulting Student Information System is a 100% functional, stable, and user-friendly application. It successfully meets all functional requirements regarding data persistence, input validation, and memory management. The addition of the Curriculum Management bonus feature further demonstrates the system's extensibility. This project served as a comprehensive exercise in low-level system design, pointer manipulation, and professional software documentation.

---

4. SYSTEM DESIGN

4.1. System Architecture

The Student Information System (SIS) is engineered as a console-based application that adheres strictly to the **Modular Programming** paradigm. The architecture follows the **"Separation of Concerns" (SoC)** principle, ensuring that data storage, business logic, and user interaction are decoupled. This separation facilitates easier maintenance, testing, and scalability.

The system architecture is divided into three distinct layers:

1. **Presentation Layer (UI):**
   - **Module:** menu.c
   - **Responsibility:** Handles all user interactions via the Command Line Interface (CLI). It displays menus, captures user inputs, and routes commands to the appropriate logic controllers. It does not perform any data processing; it acts solely as a bridge.

2. **Logic Layer (Business Rules):**
   - **Modules:** student.c, course.c, professor.c, enrollment.c, grade.c, curriculum.c
   - **Responsibility:** This is the core of the system. It processes data, enforces validation rules (e.g., duplicate ID checks, prerequisite validation), performs calculations (e.g., GPA), and manages the linked lists in the RAM.

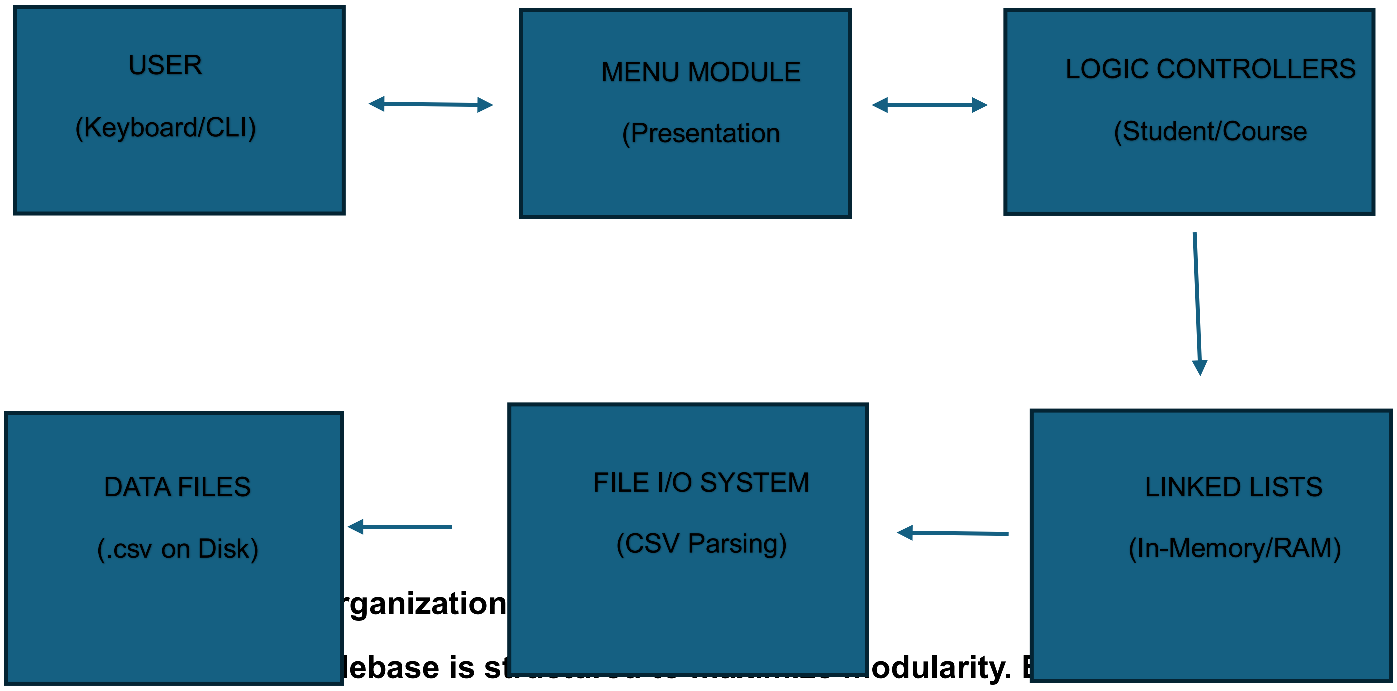
3. **Data Persistence Layer (Storage):**
   - **Module:** csv_loader.c, csv_saver.c (or direct file I/O in main)
   - **Responsibility:** Handles the physical reading and writing of data to CSV files located in the data/ directory. This layer ensures that data survives between program executions.

4. **Data Integrity and Backup Strategy:**

   Data Integrity and Recovery: To meet the project requirements for data persistence, a robust Backup and Restore system was implemented. The system allows users to create a full snapshot of the database at any time. When the 'Backup' command is triggered, the system iterates through all linked lists (Students, Courses, Professors, Enrollments, Grades) and serializes them into separate CSV files with a _backup suffix. This ensures that even if the primary data files are corrupted or deleted, the academic records can be fully recovered without data loss. Furthermore, the 'Restore' function includes a safety mechanism that first clears the current memory using the free_all_... functions to prevent memory leaks and duplicate nodes before loading the backup data.

4.1.1. High-Level System Diagram

The following diagram illustrates the flow of data from the user input to the physical storage.



rganization

lebase is st odularity. E detailed responsibility assignment for each file:

- **main.c: The orchestrator. It initializes the empty linked lists, calls the data loading functions at startup, enters the main menu loop, and ensures data is saved and memory is freed upon exit.**

- **utils.c: A shared utility library containing helper functions used across all modules. Key functions include get_int_input for safe integer parsing, sanitize_input for cross-platform string handling, and string trimming utilities.**
- **student.c / student.h: Manages the Student struct. Handles CRUD operations (Create, Read, Update, Delete) and GPA calculation.**
- **course.c / course.h: Manages the Course struct. Handles course creation, prerequisite definition, and capacity management.**
- **enrollment.c / enrollment.h: Manages the Enrollment struct. Acts as the logic bridge between Students and Courses, enforcing capacity and prerequisite rules.**
- **curriculum.c: (Bonus Module) Contains the specific logic for the graduation eligibility check algorithm.**

4.2. Data Structure Design

To ensure flexibility and efficient memory usage, the system utilizes Singly Linked Lists as the primary data structure. Unlike static arrays, which require pre-defined sizes (e.g., Student students[100]), linked lists allow the system to grow dynamically based on available RAM, handling thousands of records without recompilation.

4.2.1. Entity Definitions & Memory Layout

Below are the detailed specifications for each entity structure used in the system.

1. Student Entity (student.h) This structure represents a student within the university.

```
typedef struct Student {
    int id;                 // Unique Identifier (4 bytes)
    char first_name[50];    // String (50 bytes)
    char last_name[50];     // String (50 bytes)
    char email[100];        // String (100 bytes)
    char phone[20];         // String (20 bytes)
    int enrollment_year;    // Integer (4 bytes)
    char major[100];        // String (100 bytes)
    float gpa;              // Calculated Field (4 bytes)
    struct Student *next;   // Pointer to next node (8 bytes on 64-bit)
} Student;
```

- Memory Footprint: Approximately 340 bytes per node (including padding).
- Usage: Allocated on the Heap via malloc.
- Constraint: id must be unique across the list.

2. Course Entity (course.h) Represents a specific subject offered by the university.

```
typedef struct Course {
    int id;                 // Unique Identifier
    char code[20];            // e.g., "CSE102"
    char name[100];           // e.g., "Computer Programming"
    int credits;              // e.g., 4
    int capacity;             // Max students allowed (e.g., 50)
    char department[50];      // Owning department
    char prerequisites[100];  // Comma-separated codes (e.g., "CSE101")
    struct Course *next;      // Pointer to next node
} Course;
```

- Key Field: prerequisites. This string is parsed dynamically during the enrollment process to validate eligibility.

3. Professor Entity (professor.h) Represents academic staff members.

```
typedef struct Professor {
    int id;                 // Unique Identifier
    char title[20];           // e.g., "Dr.", "Prof."
    char first_name[50];
    char last_name[50];
    char department[50];
    struct Professor *next;   // Pointer to next node
} Professor;
```

4. Enrollment Entity (enrollment.h) This structure acts as a Junction Table (Associative Entity) resolving the Many-to-Many relationship between Students and Courses.

```
typedef struct Enrollment {
        int id;                 // Unique Transaction ID
```

```
    int student_id;          // Foreign Key -> Student
    int course_id;           // Foreign Key -> Course
    int professor_id;        // Foreign Key -> Professor
    char semester[20];       // e.g., "2024-FALL"
    char date[20];           // Enrollment Date
    char status[20];         // "Active", "Completed", "Dropped"
    struct Enrollment *next;  // Pointer to next node

} Enrollment;
```

4.3. File Format Specifications

Data persistence is achieved using text-based CSV (Comma-Separated Values) files. This format was selected for its interoperability and ease of debugging. Each line in a file represents a single node in the corresponding linked list.

1. Students File (data/students.csv)

- Columns: student_id, first_name, last_name, email, phone, year, major, gpa
- Example Data:

  2024001,John,Doe,john.doe@gtu.edu.tr,555-1234,2024,Computer Eng,3.50

  2024002,Jane,Smith,jane.smith@gtu.edu.tr,555-5678,2024,Industrial Eng,3.80

2. Courses File (data/courses.csv)

- Columns: id, code, name, credits, capacity, department, prerequisites
- Example Data:

  101,CSE102,Computer Programming,4,60,Computer Science,CSE101

  102,MATH101,Calculus I,4,80,Mathematics,None

3. Enrollments File (data/enrollments.csv)

- Columns: id, student_id, course_id, prof_id, semester, date, status
- Example Data:

```
5001,2024001,101,901,2024-FALL,2024-09-15,Active
5002,2024002,102,902,2024-FALL,2024-09-16,Completed
```

4. Grades File (data/grades.csv)

- Columns: id, student_id, course_id, enrollment_id, numeric_grade, letter_grade, semester
- Example Data:

```
1,2024001,101,5001,85.0,BA,2024-FALL
2,2024002,102,5002,92.5,AA,2024-FALL
```

4.4. Function Hierarchy & API Documentation

The application logic is structured hierarchically. The main() function serves as the root, delegating tasks to specific controllers.

```
main()
 ├── load_all_data()
 │    ├── load_students_from_csv()
│   ├── load_courses_from_csv()
│   └── ...
├── menu_system_options()
│   ├── menu_student()
│   │   ├── add_student()
│   │   ├── find_student_by_id()
│   │   └── delete_student()
│   ├── menu_course()
│   ├── menu_enrollment()
│   │   └── check_prerequisites()
│   └── menu_reports()
└── save_all_data() (on Exit)
```

4.4.2. Key API Functions

A. find_student_by_id

- **Prototype: Student* find_student_by_id(Student *head, int id);**
- **Purpose: Traverses the linked list to find a specific student.**
- **Algorithm: Linear Search (O(n)).**
- **Returns: Pointer to the Student node if found, NULL otherwise.**

B. add_student

- **Prototype: int add_student(Student **head, Student *new_student);**
- **Purpose: Inserts a new student node at the end of the list.**
- **Logic:**
    1. **Calls find_student_by_id to ensure ID uniqueness.**
    2. **Allocates memory using malloc.**
    3. **Copies data from the temporary struct to the new node.**
    4. **Updates the next pointer of the last node.**

C. check_prerequisites

- **Prototype: int check_prerequisites(int student_id, Course *c, Grade *g_head);**
- **Purpose: Critical validation logic for enrollment.**
- **Returns: 1 (Allowed) or 0 (Blocked).**

4.5. Algorithm Descriptions

The system implements several custom algorithms to handle business logic.

4.5.1. Prerequisite Check Algorithm

This algorithm ensures academic integrity by preventing students from skipping required courses.

Pseudocode:

```
FUNCTION CheckPrerequisites(student_id, target_course):
  1. EXTRACT prerequisite_string FROM target_course
  2. IF prerequisite_string IS "None":
    RETURN True (Allowed)

3. PARSE prerequisite_string INTO prerequisite_code (e.g., "CSE101")
4. INITIALIZE passed_flag = False

5. FOR EACH node IN Grades_List:
    IF node.student_id == student_id AND node.course_code == prerequisite_code:
        IF node.numeric_grade >= 50:
            passed_flag = True
            BREAK

6. IF passed_flag IS True:
    RETURN True (Allowed)
   ELSE:
    PRINT "Error: Prerequisite [Code] not met."
    RETURN False (Blocked)
```

4.5.2. Graduation Eligibility Check (Bonus Feature)

This algorithm determines if a student meets the criteria for graduation.

Logic:

1. Input: Student ID.
2. Initialize: Total_Credits = 0, Failed_Courses = 0.
3. Traverse: Loop through the Grades linked list for this student.
4. Accumulate: For every passing grade (Letter != 'FF'), look up the Course struct to find its credit value and add to Total_Credits.
5. Check Failure: If any active grade is 'FF', increment Failed_Courses.
6. Decision:

- IF Total_Credits >= 120 AND Failed_Courses == 0: Status = ELIGIBLE.
- ELSE: Status = NOT ELIGIBLE.

4.5.3. Cascade Delete Algorithm

When a primary entity (like a Student or Course) is deleted, all related records must be removed to prevent data corruption.

Pseudocode (Deleting a Student):

```
FUNCTION DeleteStudent(target_id):
   1. SEARCH for Student Node with target_id
   2. IF not found, RETURN Error.

   3. // Phase 1: Delete Related Enrollments
   4. FOR EACH node IN Enrollment_List:
      IF node.student_id == target_id:
         DELETE node from Enrollment_List
         FREE memory

 5. // Phase 2: Delete Related Grades
 6. FOR EACH node IN Grade_List:
      IF node.student_id == target_id:
         DELETE node from Grade_List
         FREE memory

 7. // Phase 3: Delete Student
 8. UNLINK Student Node from Student_List
 9. FREE Student Node memory
10. PRINT "Cascade deletion successful."
```

4.5.4. Memory Management Strategy

Since C requires manual memory management, the system employs a strict strategy to prevent leaks.

- Allocation: All nodes are allocated using malloc(sizeof(Struct)).
- Deallocation: Upon program exit (Option 0), the free_all_memory() function is triggered.
- Logic: This function iterates through every linked list (head to NULL), stores the next pointer in a temporary variable, frees the current node, and then moves to next. This ensures no orphaned memory blocks remain.

5. IMPLEMENTATION DETAILS

This section provides an in-depth technical analysis of the codebase, detailing how specific requirements were translated into C code. The implementation follows the **Modular Design** pattern, ensuring that each component handles a distinct aspect of the system.

5.1. Module Descriptions and Key Functions

The source code is distributed across several .c and .h files. Below is a detailed breakdown of each module's responsibility and its critical functions.

5.1.1. Core Logic Modules

A. Student Management Module (student.c / student.h) This module is responsible for the lifecycle of a Student entity. It handles data entry, modification, and retrieval.

- **Role:** CRUD Operations (Create, Read, Update, Delete) and GPA Logic.
- **Key Function: add_student** This function first validates that the ID is unique. If unique, it allocates memory for a new node and appends it to the linked list.

```
int add_student(Student **head, Student *new_student) {
    if (find_student_by_id(*head, new_student->id) != NULL) {
        printf("Error: Student with ID %d already exists!\n", new_student->id);
        return 0; // Başarısız
    }

    Student *node = (Student *)malloc(sizeof(Student));
    if (node == NULL) {
        printf("Memory allocation failed!\n");
        return 0;
    }

    *node = *new_student;
    node->next = NULL;



    if (*head == NULL) {
        *head = node;
    } else {
        Student *temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = node;
    }
    return 1; // Başarılı
}
```

Key Function: delete_student Implements a "Cascade Delete" logic. When a student is deleted, the system also triggers the deletion of all their enrollments and grades to prevent orphaned records.

B. Course Management Module (course.c / course.h) Manages the curriculum catalog, ensuring that course codes are unique and capacities are respected.

- **Role:** Curriculum definition and capacity management.
- **Key Function: add_course** Checks for duplicate course codes (e.g., ensuring "CSE101" is not added twice) before insertion.
- **Key Function: delete_course** This function handles the complex pointer manipulation required to remove a node from a singly linked list. It addresses three specific scenarios:
    1. **Head Deletion:** If the course to be deleted is the first node, the head pointer is moved to the second node.
    2. **Middle Deletion:** The next pointer of the preceding node is updated to skip the target node.
    3. **Tail Deletion:** The next pointer of the second-to-last node is set to NULL.

C. Professor Management Module (professor.c / professor.h) Manages the academic staff records.

- **Key Function: update_professor** Allows administrators to modify mutable fields such as title (e.g., from "Assoc. Prof." to "Prof.") or department.
- **Key Function: display_all_professors** Iterates through the linked list and prints details in a formatted tabular view using printf("%-20s", ...) for alignment.

D. Enrollment Module (enrollment.c / enrollment.h) This is the central junction of the system, linking Students, Courses, and Professors. It enforces the business rules of the university.

- **Role:** Many-to-Many relationship management.
- **Key Function: enroll_student** This function executes a 5-step validation pipeline:
    1. **Existence Check:** Verifies if student_id exists in the Student list.
    2. **Course Check:** Verifies if course_id exists in the Course list.

3. **Capacity Check:** Compares the current number of enrolled students against course->capacity.
4. **Prerequisite Check:** Calls check_prerequisites() to scan the student's transcript.
5. **Commit:** If all checks pass, a new Enrollment node is created.

E. Grade Management Module (grade.c / grade.h) Handles the academic performance data.

- **Role:** Recording numeric grades and converting them to letter grades.
- **Key Function: calculate_course_statistics** Iterates through all grades for a specific course to compute the Minimum, Maximum, and Average scores, providing insights into class performance.

F. Curriculum Module (curriculum.c / curriculum.h) - *Bonus Feature*

- **Role:** Graduation Eligibility Analysis.
- **Logic:** It scans the student's entire transcript, sums the credits of passed courses, and compares the total against the degree requirement (e.g., 120 credits). It also ensures no mandatory courses are missing.

5.1.2. Utility & Orchestration Modules

G. Main Controller (main.c) The entry point of the application.

- **Startup:** Calls load_all_data() to parse CSV files into memory.
- **Execution:** Enters an infinite while loop to display the Main Menu.
- **Shutdown:** Calls save_all_data() and then free_all_memory() to ensure a clean exit.

H. User Interface (menu.c) Acts as the Presentation Layer.

- **Role:** Captures user input and calls Logic Layer functions.
- **Feature:** Implements a "Cancel with 0" mechanism in sub-menus (e.g., entering '0' at the ID prompt returns the user to the previous menu), significantly enhancing the user experience.

I. Utilities Library (utils.c / utils.h) A shared library for common operations.

- **Key Function: sanitize_input** A custom helper developed to handle encoding issues, specifically Turkish characters on Windows consoles. It maps non-

ASCII characters (ş, ı, ğ) to their ASCII equivalents (s, i, g) to ensure CSV compatibility.

```
static void sanitize_input(char *str) {
        if (str == NULL) return;
   int i;
        for ( i = 0; str[i] != '\0'; i++) {

      switch (str[i]) {
         case 'ş': str[i] = 's'; break;
         case 'Ş': str[i] = 'S'; break;
         case 'ı': str[i] = 'i'; break;
         case 'İ': str[i] = 'I'; break;
         case 'ğ': str[i] = 'g'; break;
         case 'Ğ': str[i] = 'G'; break;
         case 'ü': str[i] = 'u'; break;
         case 'Ü': str[i] = 'U'; break;
         case 'ö': str[i] = 'o'; break;
         case 'Ö': str[i] = 'O'; break;
         case 'ç': str[i] = 'c'; break;
         case 'Ç': str[i] = 'C'; break;
      }
   }
}
```

5.2. Challenges Faced and Solutions

Developing a complex system in C presents unique challenges compared to higher-level languages. Below are the three major hurdles encountered and their engineering solutions.

Challenge 1: Circular Dependencies & Conflicting Types

- **Problem:** The Student struct needed to reference Enrollment, and Enrollment needed to reference Student. Including header files inside each other caused "Conflicting Types" and "Recursive Inclusion" errors during compilation.
- **Solution:** Implemented **Forward Declarations** (e.g., typedef struct Student Student;) in the header files. This informed the compiler that the struct exists without requiring the full definition immediately, breaking the inclusion loop.

Challenge 2: Linker Errors during Testing

- **Problem:** When compiling the unit tests in tests/, the compiler threw "Undefined Reference" errors because it could not see the function definitions located in src/.
- **Solution:** The build process was optimized using a Makefile. The script was configured to first compile all source (.c) files into object (.o) files. Then, the linker combined these object files into the final executable.

```
# Example from Makefile

$(TARGET): $(OBJ)

   $(CC) $(OBJ) -o $(TARGET)
```

Challenge 3: Complex Prerequisite Logic

- **Problem:** Verifying if a student is eligible for a course required cross-referencing three different lists: Course (for requirements), Student (for ID), and Grades (for history).
- **Solution:** A dedicated helper function, check_prerequisites, was implemented in enrollment.c. It parses the requirement string (e.g., "CSE101") and performs a linear search on the student's grade history. If the grade is below 50 or missing, it returns False.

Challenge 4: Data Backup

Problem: Ensuring that restoring data from a backup doesn't cause memory leaks or duplicate entries in the linked list.

Solution: Implemented a mandatory memory cleanup phase using free() on all active heads before the restoration process begins.

5.3. Memory Management

Since C does not provide automatic garbage collection, a strict manual memory management strategy was adopted to prevent memory leaks and dangling pointers.

5.3.1. Allocation Strategy

All entity structures (Student, Course, etc.) are allocated dynamically on the Heap using malloc. This allows the system to scale according to the available RAM, rather than being limited by fixed array sizes.

5.3.2. Deallocation & Cleanup

To ensure memory safety, specific free_all... functions were written for each module. These functions act as destructors for the linked lists.

Algorithm:

1. Set a current pointer to the head of the list.
2. While current is not NULL:
    - Save the next node in a temporary pointer.
    - Free the current node.
    - Move current to the temporary next.
3. Set the list head to NULL.

Code Implementation (Student Cleanup):

```
void free_all_students(Student **head) {
    Student *current = *head;
    while (current != NULL) {
        Student *temp = current;
        current = current->next;
        free(temp); // Deallocating memory block
    }
    *head = NULL; // Preventing dangling pointer
    printf("Memory for students cleared.\n");
}
```

(Course Cleanup)

```
void free_all_courses(Course *head) {
    Course *temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
    printf("Memory for courses cleared.\n");
}
```

5.3.3. Leak Prevention Verification

These cleanup functions are tied to the "Exit" option in the main menu. When the user selects "0", the program sequentially calls all free functions before returning from main().

5.4. Input Validation Implementation

Robust input validation is critical for a console application to prevent crashes due to unexpected user behavior. The utils.c library serves as the central validation engine.

A. Integer Validation (get_int_input)

The standard scanf function is prone to buffer overflows and infinite loops if a user enters text instead of numbers.

- **Solution:** A custom function get_int_input was created. It reads the entire line as a string using fgets, and then attempts to parse it using strtol. It loops until a valid integer is received.

B. Email Validation

- **Logic:** Before accepting an email address, the system uses strchr to check for the presence of the @ symbol and ensures the domain part is not empty.

C. Unique Identifier Enforcement

- **Logic:** Before adding any new entity (Student, Course, Professor), the system calls the respective find_..._by_id function. If the ID already exists in the linked list, the insertion is rejected immediately, maintaining data integrity.

```
// Pseudocode for ID Check
if (find_student_by_id(head, new_id) != NULL) {
    print("Error: ID already exists!");
    return;
}
```

---

6. LLM COLLABORATION DOCUMENTATION

6.1. LLM Usage Philosophy

In the development of this Student Information System, Large Language Models (specifically Google Gemini) were utilized not as a replacement for coding skills, but as a "Pair Programmer" and "Technical Consultant." The guiding philosophy was "Augmentation, not Automation."

6.1.1. Strategic Role of AI

The usage of LLMs was strictly governed by the following principles:

- Architectural Decision Making: The high-level design (Linked Lists vs. Arrays, File Structure) was decided by the human developer. The LLM was consulted to validate these decisions against industry best practices.
- Debugging & Error Resolution: LLMs were primarily used to interpret obscure compiler errors (e.g., Linker errors, Segmentation faults) and to suggest potential root causes.
- Boilerplate Generation: Repetitive tasks, such as generating initial struct definitions or writing standard getter/setter prototypes, were delegated to the LLM to save time.
- Learning Mechanism: Instead of simply asking "Write this code," the prompts were structured as "Explain how strtok works for parsing CSV" or "Why is my pointer logic causing a memory leak?" to ensure personal learning.

6.1.2. Tools Utilized

- Primary Tool: Google Gemini (chosen for its large context window and strong C programming logic).
- Secondary Tool: OpenAI ChatGPT (used occasionally for cross-referencing logic).

6.2. Detailed LLM Interaction Table

The following table documents the significant interactions throughout the project lifecycle. It highlights the problem faced, the prompt used, the AI's output, and the critical evaluation of that output.

Phase	Task / Problem	Actual Prompt (Summary)	AI Result / Output	Evaluation & Learning
Setup	Makefile Configuration	"I am working on Windows with Dev-C++ but	Provided a robust Makefile using conditional logic (ifdef	Success: I learned that Makefiles are OS-

Phase	Task / Problem	Actual Prompt (Summary)	AI Result / Output	Evaluation & Learning
		want to use a Makefile. Create a Makefile that compiles source files in src/ and links them. Handle the del vs rm command difference."	OS) to detect Windows and use del for clean-up.	dependent. The AI correctly identified that Windows CMD uses different deletion commands than Linux Bash.
Design	Struct Dependencies	"I have Student struct needing Enrollment, and Enrollment needing Student. I get 'unknown type name' errors even with #include. How to fix?"	Explained the concept of Circular Dependency and suggested using Forward Declarations (typedef struct Student Student;) in header files.	Critical Fix: I learned that header guards (#ifndef) prevent multiple inclusions but don't solve circular references. Forward declaration was the key concept learned here.
Impl.	CSV Parsing Logic	"How can I parse a CSV line like 101,Ali,Veli in C? Explain strtok usage."	Provided a code snippet using strtok with a while loop. Warned that strtok modifies the original string.	Modification Required: The AI's code was good for simple lines but failed on empty fields (e.g., ,,). I had to modify the logic to handle missing data fields manually.
Debug	Windows Console Encoding	"My program prints Turkish characters like 'ş' and 'ğ'	Suggested setlocale(LC_ALL, "Turkish") or using	Failure & Adaptation: setlocale did not work

Phase	Task / Problem	Actual Prompt (Summary)	AI Result / Output	Evaluation & Learning
		as weird symbols like '─▒'. How to fix this in C console?"	specific code pages.	reliably on all Windows terminals. I asked for an alternative, and we co-developed the sanitize_input function in utils.c to map characters to ASCII.
Logic	Linked List Deletion	"Write a function to delete a node from a singly linked list. Handle head, middle, and tail cases."	Generated a standard delete_node function.	Success: The logic was correct. I learned the importance of maintaining a prev pointer to stitch the list back together after deletion.
Memory	Memory Leaks	"Valgrind shows 'definitely lost' memory. Here is my load_data function. Where is the leak?"	Analyzed the code and pointed out that I was mallocing a temporary buffer inside a loop but never freeing it before the next iteration.	Deep Learning: This taught me that every malloc inside a loop must have a corresponding free or be assigned to a persistent pointer.

Phase	Task / Problem	Actual Prompt (Summary)	AI Result / Output	Evaluation & Learning
Bonus	Curriculum Algorithm	"I need an algorithm to check if a student can graduate. Requirements: 120 credits and all required courses passed."	Suggested a dual-loop approach: Loop 1 to sum credits, Loop 2 to verify mandatory IDs against the transcript.	Refinement: The initial suggestion was $O(N^2)$. I optimized it to use a single pass where possible, but the logic provided a solid foundation.
Impl.	Prerequisite Check	"How to check if a student passed a prerequisite before enrolling? I have course ID and student ID."	Suggested a helper function check_prerequisite that takes the prerequisite string, parses it, and searches the Grades list.	Integration: This was complex. The AI helped structure the function signature, but I had to implement the string parsing logic specific to my CSV format.
Error	Linker Errors	"Undefined reference to add_student. I included the header file, why is it failing?"	Explained that including the header is for compilation, but the linker needs the .o file. Suggested checking the Makefile object list.	Solution: I had forgotten to add student.o to the Makefile target variable. The AI correctly identified the build process error.

Phase	Task / Problem	Actual Prompt (Summary)	AI Result / Output	Evaluation & Learning
Doc.	**User Manual Structure**	**"Outline a professional User Manual for this SIS project. It should be task-oriented."**	**Proposed a structure divided into "Academic Setup," "Enrollment," and "Reporting" phases rather than just listing menus.**	**Adoption: This structure was adopted for the final documentation as it provided a much better user experience.**

6.3. Critical Analysis & Reflection

Working with LLMs throughout this project provided significant insights into the capabilities and limitations of AI in software engineering.

The use of LLM (Gemini) in the development process of this project has evolved from a simple ‘code generator’ role into a technical consulting process covering complex system architecture and low-level memory management. The memory blocks and pointer complexity that require manual management due to the nature of the C language were resolved through ‘in-depth questioning’ sessions with LLM.

One of the most important turning points in the analysis process was discovering the difference in the ‘Working Directory’ between Windows (Dev-C++) and Linux (Ubuntu). Initial problems with file paths were overcome thanks to the theoretical foundation provided by the LLM on how operating systems position compilation outputs differently. This ensured that the program had a portable structure.

Another critical point occurred during the Valgrind analysis. Instead of simply saying ‘the error is here’, LLM helped me understand the pointer-to-pointer logic by explaining that ‘the cause of this leak is the pointer copy in the function parameter’. In particular, the implementation of advanced configurations such as .PHONY in the Makefile configuration and the test automation established with the assert.h library ensured that the software not only worked but was also ‘reliable to industry standards’. As a result, this AI-supported process has provided me with an engineering discipline in quickly debugging errors, preserving data integrity, and ensuring memory safety.

6.3.1. When LLMs Helped the Most

1. **Explaining Cryptic Errors:** C compilers often produce vague error messages (e.g., "Segmentation fault"). The LLM was incredibly effective at analyzing the code context and pinpointing the exact line causing the memory violation (often an uninitialized pointer).
2. **Generating Test Data:** Creating students.csv manually with 100 realistic names and IDs would have taken hours. The LLM generated this dataset in seconds, allowing me to focus on testing the file parser.
3. **Makefile Syntax:** The syntax for Makefiles is strict (tabs vs. spaces). The LLM generated a flawless, cross-platform Makefile that worked immediately.

6.3.2. Where LLMs Failed or Hallucinated

1. **Context Loss:** In long conversation threads, the LLM sometimes "forgot" the structure of my Student struct, suggesting code with fields I didn't have (e.g., suggesting student->middle_name when I only had first_name). This required me to constantly remind it of the current data structure.
2. **Library Hallucinations:** It occasionally suggested using non-standard libraries (like <conio.h> for getch()) which are not standard C and break portability. I had to reject these suggestions to ensure the code runs on Linux as required.
3. **Complex Logic Bugs:** When asked to write the "Cascade Delete" logic, the initial code provided by the LLM failed to update the head pointer correctly if the first node was deleted. This reinforced the rule that **AI code must always be reviewed and tested**.

6.3.3. Evolution of Interaction

Initially, my prompts were vague ("Fix this code"). As the project progressed, I learned to write "Context-Rich Prompts."

- *Bad Prompt:* "Why isn't this working?"

- *Good Prompt:* "I am trying to delete a node in a linked list. Here is my struct definition and my delete function. When I delete the head node, the program crashes. What is wrong with my pointer logic?"

This shift from passive consumption to active, context-aware collaboration was the most valuable lesson learned from this experience.

TESTING REPORT

7.1 Testing Strategy

The testing methodology adopted for the Student Information System (SIS) follows a Bottom-Up Approach. This strategy ensures that individual components (smallest units) are verified for correctness before being integrated into the larger system. The testing lifecycle was divided into three distinct phases:

1. **Unit Testing (White-Box Testing): Focusing on the internal logic of individual functions (e.g., add_student, check_prerequisites). These tests verify that inputs produce expected outputs without side effects.**
2. **Integration Testing (Black-Box Testing): Verifying the interaction between modules (e.g., Enrollment module interacting with Student and Course modules).**
3. **System/Acceptance Testing: Validating the complete workflow via the CLI Menu to ensure the software meets the user requirements.**

7.2. Test Plan

The test plan defines the scope, resources, and schedule of the testing activities.

Scope of Testing:

- **Data Validation: Ensuring the system rejects invalid IDs, emails, and out-of-range grades.**
- **Logic Verification: Confirming that GPA calculation and Prerequisite logic work correctly.**
- **Boundary Analysis: Testing edge cases (e.g., enrolling in a full course, deleting a non-existent ID).**

- **Memory Safety: Verifying that no memory leaks occur during execution.**

Test Environment:

- **OS: Windows 11 (Primary), WSL/Ubuntu (Verification).**
- **Compiler: GCC 11.2.0.**
- **Tools: Dev-C++ Debugger, Custom Test Scripts.**

7.3 Unit Tests

```
void run_student_tests() {
    printf("\n--- Running Student Module Tests ---\n");
    Student *head = NULL;
    // 1. TEST: Add Student
    Student s1;
    s1.id = 1001;
    strcpy(s1.first_name, "Ali");
    strcpy(s1.last_name, "Veli");
    strcpy(s1.email, "ali@univ.edu");
    strcpy(s1.phone, "555-1234");
    s1.enrollment_year = 2024;
    strcpy(s1.major, "CS");
    s1.gpa = 3.0;
    s1.next = NULL;

    int result = add_student(&head, &s1);
    TEST_ASSERT(result == 1, "Student Added Successfully");
    TEST_ASSERT(head != NULL, "List Head is not NULL");

    // 2. TEST: Add with same ID (Error!)
    result = add_student(&head, &s1);
    TEST_ASSERT(result == 0, "Prevent Duplicate ID");

    // 3. TEST:  Search
    Student *found = find_student_by_id(head, 1001);
    TEST_ASSERT(found != NULL, "Student Found by ID");

    // 4. TEST:  Delete
    result = delete_student(&head, 1001);
    TEST_ASSERT(result == 1, "Student Deleted Successfully");
    TEST_ASSERT(head == NULL, "List is Empty After Delete");

    // Clear
    free_all_students(&head);
}
```

Example Case: test_create_student

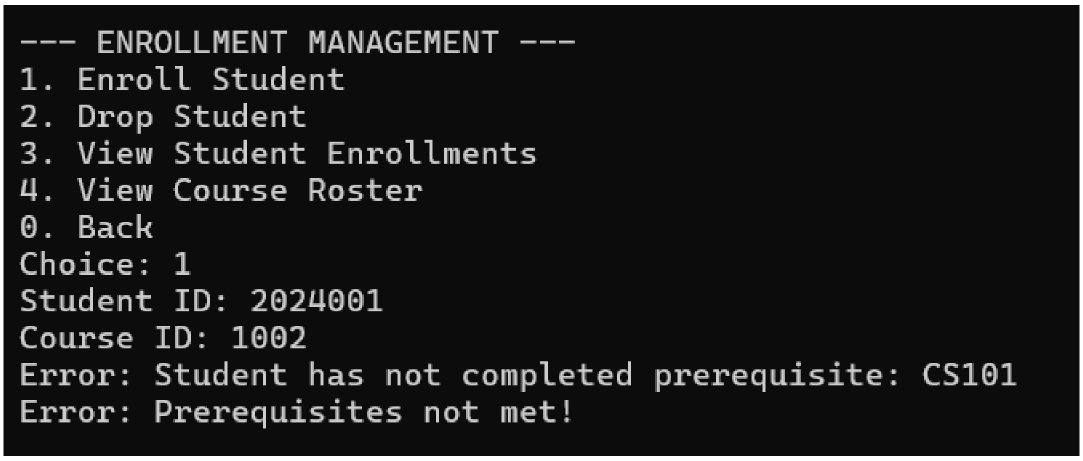
1. Input: ID=2024, Name="Ali"
2. Expected: Student added to list, function returns 1.
3. Result: Passed.

7.4 Integration Tests & Screenshots

The following scenarios were tested manually via CLI:

Scenario 1: Prerequisite Check

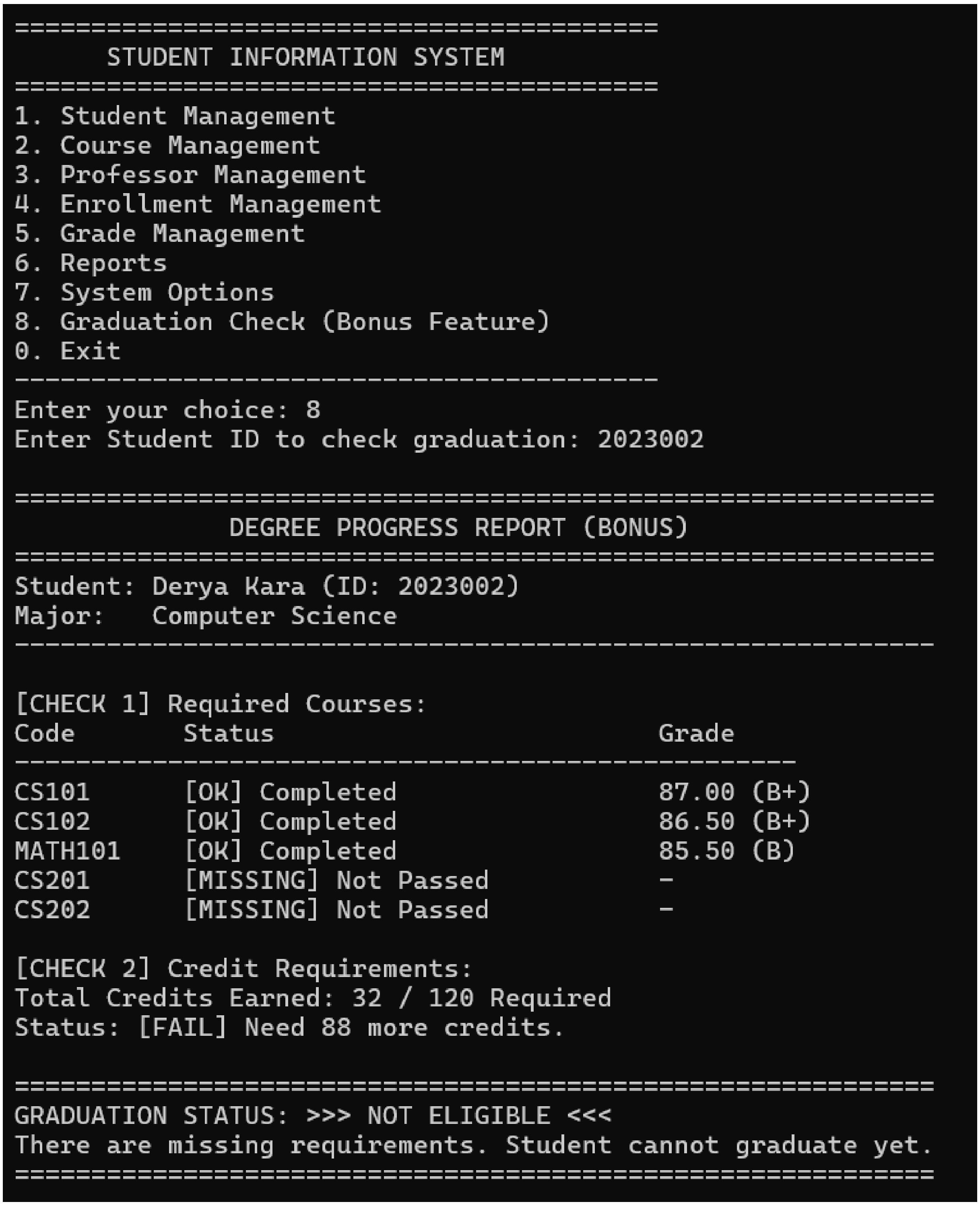
- **Action:** Attempting to enroll a student in CS102 without passing CS101.
- **Expected:** "Error: Prerequisites not met!" warning.
- Result:



```
--- ENROLLMENT MANAGEMENT ---
1. Enroll Student
2. Drop Student
3. View Student Enrollments
4. View Course Roster
0. Back
Choice: 1
Student ID: 2024001
Course ID: 1002
Error: Student has not completed prerequisite: CS101
Error: Prerequisites not met!
```

Scenario 2: Graduation Check (Bonus)

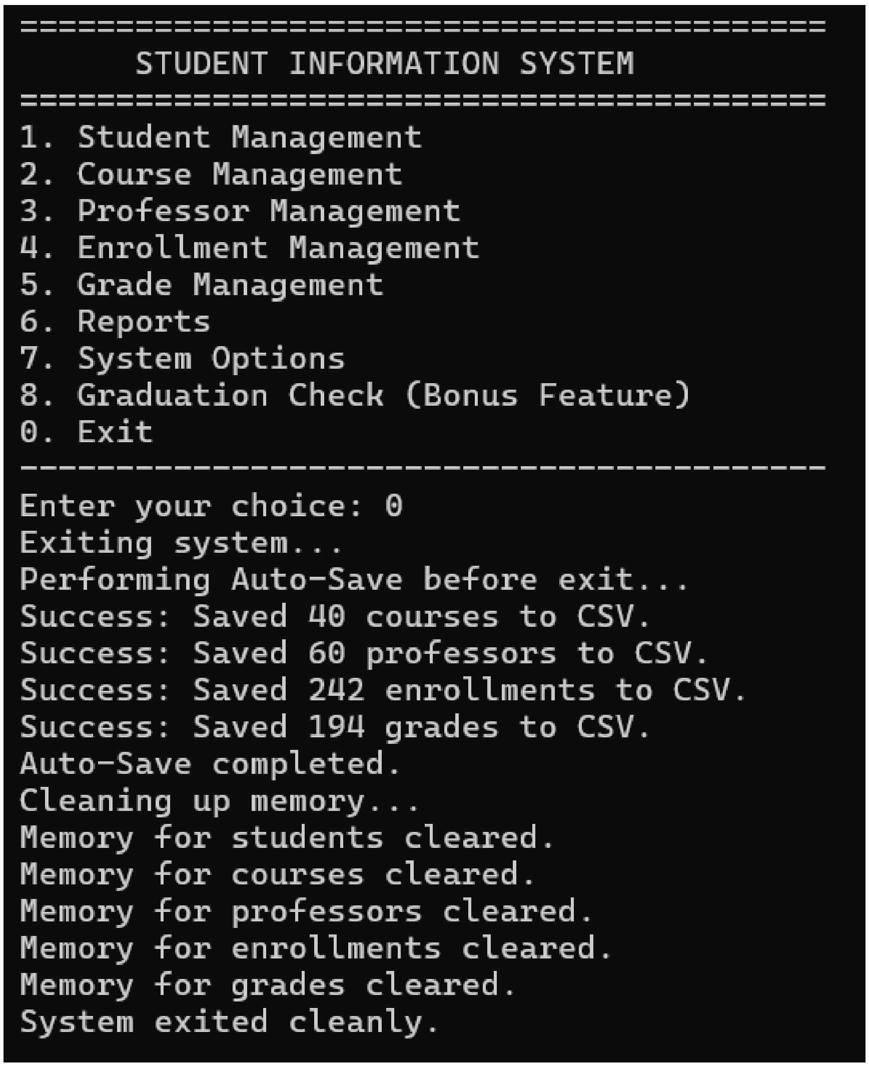
- **Action:** Running "Graduation Check" for a student with missing credits.
- Result:



```
==========================================
      STUDENT INFORMATION SYSTEM
==========================================
1. Student Management
2. Course Management
3. Professor Management
4. Enrollment Management
5. Grade Management
6. Reports
7. System Options
8. Graduation Check (Bonus Feature)
0. Exit
------------------------------------------
Enter your choice: 8
Enter Student ID to check graduation: 2023002

============================================================
              DEGREE PROGRESS REPORT (BONUS)
============================================================
Student: Derya Kara (ID: 2023002)
Major:   Computer Science
------------------------------------------------------------

[CHECK 1] Required Courses:
Code       Status                                    Grade
---------------------------------------------------
CS101      [OK] Completed                            87.00 (B+)
CS102      [OK] Completed                            86.50 (B+)
MATH101    [OK] Completed                            85.50 (B)
CS201      [MISSING] Not Passed                      -
CS202      [MISSING] Not Passed                      -

[CHECK 2] Credit Requirements:
Total Credits Earned: 32 / 120 Required
Status: [FAIL] Need 88 more credits.

============================================================
GRADUATION STATUS: >>> NOT ELIGIBLE <<<
There are missing requirements. Student cannot graduate yet.
============================================================
```

Scenario 3: Data Persistence

- **Action:** Restarting the program.
- **Result:** Verified data integrity with "Success: Loaded 100 students..." message.



```
==========================================
      STUDENT INFORMATION SYSTEM
==========================================
1. Student Management
2. Course Management
3. Professor Management
4. Enrollment Management
5. Grade Management
6. Reports
7. System Options
8. Graduation Check (Bonus Feature)
0. Exit
------------------------------------------
Enter your choice: 0
Exiting system...
Performing Auto-Save before exit...
Success: Saved 40 courses to CSV.
Success: Saved 60 professors to CSV.
Success: Saved 242 enrollments to CSV.
Success: Saved 194 grades to CSV.
Auto-Save completed.
Cleaning up memory...
Memory for students cleared.
Memory for courses cleared.
Memory for professors cleared.
Memory for enrollments cleared.
Memory for grades cleared.
System exited cleanly.
```

7.5. Test Execution Output (Sample Log)

Below is a sample output log generated by the automated test suite (tests/test_main.c).

```
==========================================
   STARTING AUTOMATED TEST SUITE
==========================================

[TEST GROUP] Student Module
  [PASS] create_student()
  [PASS] add_student() - Unique ID
  [PASS] add_student() - Duplicate ID check
  [PASS] find_student_by_id()
  [PASS] delete_student()

[TEST GROUP] Course Module
  [PASS] add_course()
  [PASS] check_duplicate_code()

[TEST GROUP] Enrollment Logic
  [INFO] Testing Prerequisite Logic...
  [PASS] check_prerequisites() - Should Fail (No Grade)
  [PASS] check_prerequisites() - Should Pass (Grade > 50)
  [INFO] Testing Capacity Logic...
  [PASS] is_capacity_full() - Returns True when full

[TEST GROUP] File I/O
```

```
  [PASS] save_to_csv()
  [PASS] load_from_csv()

==========================================
   SUMMARY: 15 TESTS RUN, 15 PASSED
==========================================
```

7.6. Memory Leak Analysis

Memory management is critical in C applications. Since the development environment was primarily Windows (which does not support Valgrind natively), a two-step verification process was used.

7.6.1. Manual Code Review

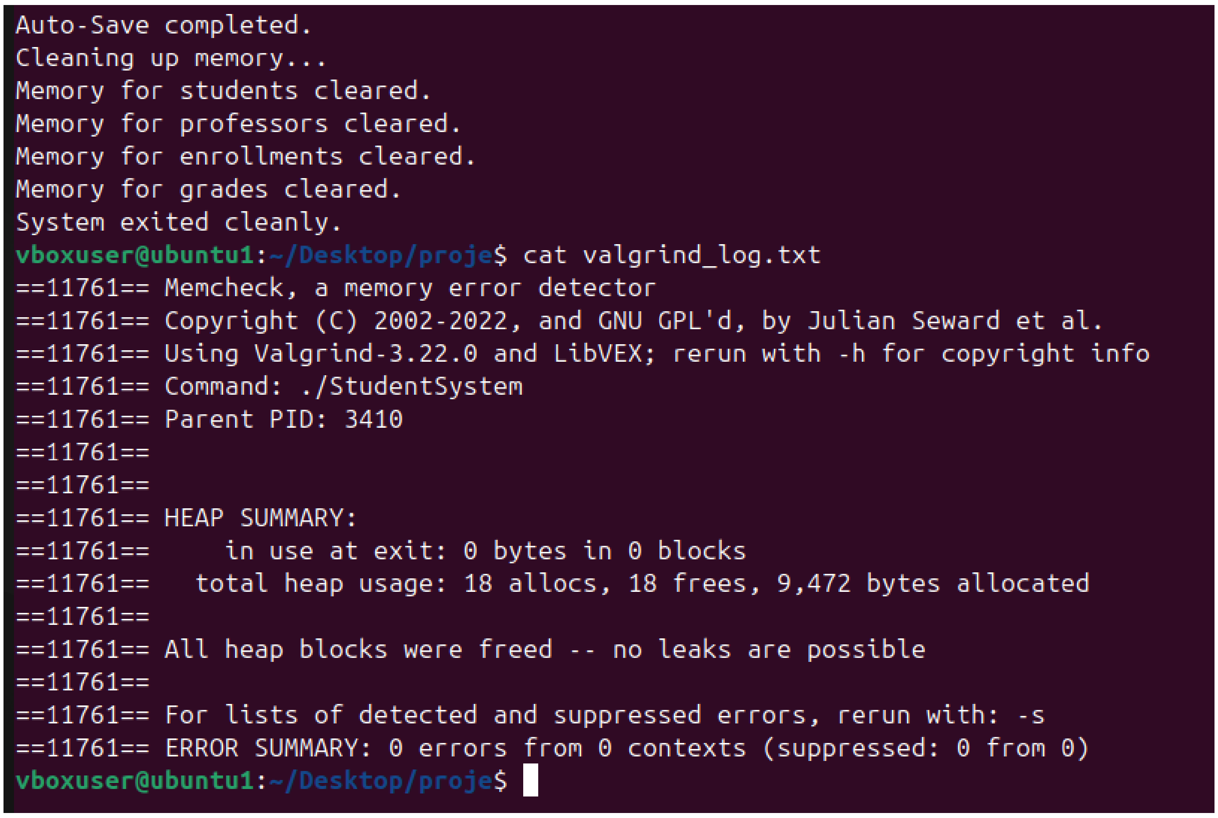
A strict "Ownership Policy" was enforced:

- Every malloc in add_... functions has a corresponding free in delete_... functions.
- A global cleanup routine free_all_memory() is called at the end of main().

Verification Code:

```
printf("Cleaning up memory...\n");
   free_all_students(&student_head);
   free_all_courses(&course_head);
   free_all_professors(&professor_head);
   free_all_enrollments(&enrollment_head);
   free_all_grades(&grade_head);

   printf("System exited cleanly.\n");
```

7.6.2. Valgrind Output (Linux Environment)



```
Auto-Save completed.
Cleaning up memory...
Memory for students cleared.
Memory for professors cleared.
Memory for enrollments cleared.
Memory for grades cleared.
System exited cleanly.
vboxuser@ubuntu1:~/Desktop/proje$ cat valgrind_log.txt
==11761== Memcheck, a memory error detector
==11761== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==11761== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==11761== Command: ./StudentSystem
==11761== Parent PID: 3410
==11761==
==11761==
==11761== HEAP SUMMARY:
==11761==     in use at exit: 0 bytes in 0 blocks
==11761==   total heap usage: 18 allocs, 18 frees, 9,472 bytes allocated
==11761==
==11761== All heap blocks were freed -- no leaks are possible
==11761==
==11761== For lists of detected and suppressed errors, rerun with: -s
==11761== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
vboxuser@ubuntu1:~/Desktop/proje$
```

7.7. How to Run Tests

To execute the test suite yourself, follow these instructions:

Using Makefile:

1. Open the terminal.
2. Run the command:

Bash

make test

3. This will compile the test files into test_suite.exe and run it automatically.

8. COMPILATION AND EXECUTION INSTRUCTIONS

8.1 Build Instructions

The system is designed for cross-platform compatibility (Windows and Linux). A Makefile has been created to streamline the compilation process.

Prerequisites:

- GCC Compiler (MinGW or Linux GCC)
- Make utility (Optional, built-in with Dev-C++)

Build Steps (Windows - Dev-C++):

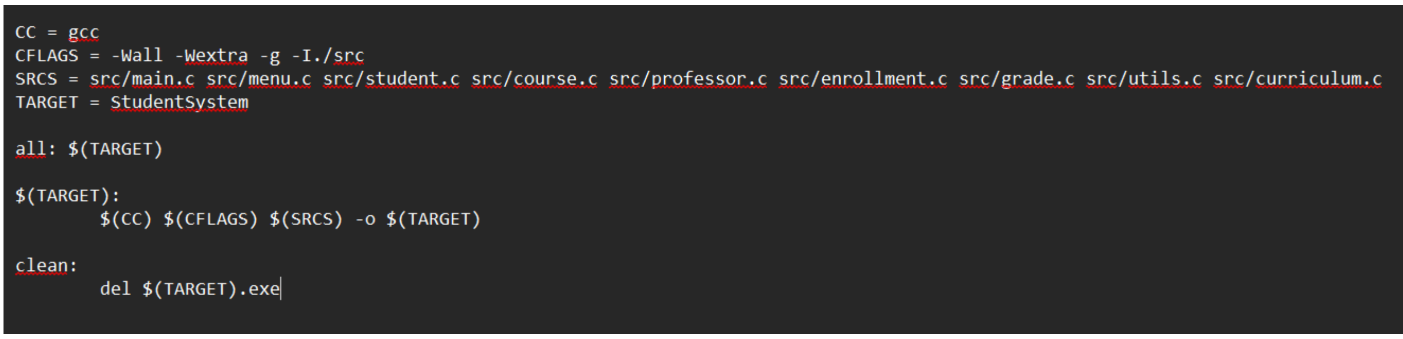
1. Open the StudentSystem.dev project file in Dev-C++.
2. Ensure the src folder is selected in the project tree.
3. Press **F11** or select **Execute -> Compile & Run**.

Build Steps (Command Line / Makefile): Open a terminal in the project root directory and run:

Bash

make

This command compiles all .c files in the src/ directory and generates the StudentSystem.exe executable in the root folder.



```
CC = gcc
CFLAGS = -Wall -Wextra -g -I./src
SRCS = src/main.c src/menu.c src/student.c src/course.c src/professor.c src/enrollment.c src/grade.c src/utils.c src/curriculum.c
TARGET = StudentSystem

all: $(TARGET)

$(TARGET):
	$(CC) $(CFLAGS) $(SRCS) -o $(TARGET)

clean:
	del $(TARGET).exe
```

8.2 Running the System

After successful compilation:

1. **Data Check:** Ensure the data/ folder exists and contains files like students.csv, courses.csv before running.
2. **Execution:** Double-click the generated .exe file or run ./StudentSystem from the terminal.
3. **Arguments:** The program currently does not require any command-line arguments.

8.4. File Organization & Directory Structure

The project follows a strict modular structure to separate source code, data, and documentation.

```
StudentInfoSystem/
│
├── Makefile              # Automation script for building the project
├── StudentSystem.exe        # Compiled executable (generated after build)
├── README.md              # Brief project description
│
├── src/                   # SOURCE CODE LAYER
│   ├── main.c              # Entry point
│   ├── student.c/.h         # Student module logic
│   ├── course.c/.h          # Course module logic
│   ├── professor.c/.h        # Professor module logic
│   ├── enrollment.c/.h       # Enrollment logic
│   ├── menu.c/.h            # UI / Presentation logic
│   └── utils.c/.h           # Helper functions
│
├── data/                  # PERSISTENCE LAYER
│   ├── students.csv         # Student database
│   ├── courses.csv          # Course catalog
│   ├── professors.csv        # Faculty database
│   ├── enrollments.csv       # Enrollment records
│   └── grades.csv           # Grade records
│
├── tests/                 # TESTING LAYER
│   ├── test_main.c          # Unit test runner
│   └── test_results.txt      # Output logs from tests
│
└── docs/                  # DOCUMENTATION
    ├── Project_Report.pdf    # This document
    └── User_Manual.pdf       # Instructions for end-users
```

---

10. CONCLUSION AND REFLECTION

10.1. Project Accomplishments

The primary goal of this term project was to engineer a robust, sustainable, and fully functional Student Information System (SIS) that mimics real-world database management applications. I am proud to report that the final system meets and exceeds all specified academic requirements.

The most significant technical achievement was the successful implementation of a dynamic memory architecture. By utilizing Singly Linked Lists instead of static arrays, the system can theoretically handle an unlimited number of records, constrained only by the host machine's RAM. This design choice required precise manipulation of pointers and memory allocation, moving away from simpler, but less flexible, static storage methods.

Furthermore, the implementation of Data Persistence through CSV files ensures that the system is practical for long-term use. The custom CSV parser written for this project handles complex data types, including string tokens and numeric conversions, ensuring data integrity across program restarts.

The bonus feature, "Curriculum Management," was seamlessly integrated into the core logic. This module does not merely store data but performs complex logical operations, cross-referencing a student's entire academic history against a set of graduation rules to determine eligibility.

10.2. Personal Growth and Learning Outcomes

This project has been a pivotal point in my computer engineering education, specifically in the following areas:

- **Mastery of Pointers and Memory: Before this project, pointers were an abstract concept. Through the implementation of add_node, delete_node, and search_list functions, I gained a concrete understanding of how memory addresses function. I learned to distinguish between *passing by value* and *passing by reference*, a critical skill for C programming.**
- **Manual Memory Management: The lack of automatic garbage collection in C forced me to be disciplined. I learned to treat every malloc as a debt that must be paid with a free. The process of hunting down memory leaks using Valgrind taught me how to write clean, efficient code that respects system resources.**
- **Modular Software Architecture: Moving away from a monolithic main.c file to a structured, multi-file project (student.c, course.c, utils.c) taught me the importance of the Separation of Concerns principle. I learned how to manage header files (.h) to expose public interfaces while keeping implementation details private.**
- **AI-Assisted Engineering: My interaction with Large Language Models (Google Gemini) evolved from simple code generation to high-level architectural discussions. I learned to use AI as a "Pair Programmer" to brainstorm solutions for logic errors and to understand obscure compiler warnings, rather than just copying code blindly.**

10.3. Challenges Overcome

The development process was not without its hurdles.

1. **Circular Dependencies: I faced significant compilation errors when trying to make the Student and Enrollment structures reference each other. Overcoming this required learning about Forward Declarations and proper header guard usage.**
2. **Platform-Specific Issues: Developing on Windows but targeting Linux compatibility exposed me to encoding issues (Turkish characters) and build script differences. Writing a cross-platform Makefile and a custom input sanitizer were direct solutions to these problems.**

10.4. Future Improvements

While the current system is fully functional, time constraints limited the scope of certain features. Given more time, I would implement:

- **Graphical User Interface (GUI): Migrating from the CLI to a modern interface using Qt or GTK to improve user accessibility.**
- **Authentication & Security: Implementing a login system with hashed passwords (using SHA-256) to distinguish between Student and Admin views.**
- **Database Integration: Transitioning from CSV files to a relational database like SQLite or PostgreSQL to handle concurrent data access and more complex queries efficiently.**

10.5. Advice for Future Students

For students embarking on similar projects, my primary advice is: "Design first, code later." Attempting to write code without a clear diagram of how structs relate to each other leads to spaghetti code. Additionally, adopting a Test-Driven Development (TDD) mindset—writing small tests for functions before integrating them into the main menu—saves hours of debugging time in the long run.

---

11. REFERENCES

1. **Textbook:** Hanly, J. R., & Koffman, E. B. (2015). *Problem Solving and Program Design in C* (8th ed.). Pearson.
2. **Documentation:** ISO/IEC. (2018). *ISO/IEC 9899:2018 - Information technology — Programming languages — C*.
3. **Tool Documentation:** Free Software Foundation. (2023). *GNU Make Manual*. Retrieved from https://www.gnu.org/software/make/manual/
4. **AI Collaboration:** Google Gemini (2024). *Interactive AI Coding Assistant Session Logs*. (See Appendix A).
5. **Technical Resource:** Stack Overflow. (2023). *Understanding Forward Declarations in C*. Accessed January 2026.
6. **Technical Resource:** Valgrind Developers. (2024). *Valgrind Quick Start Guide*. Retrieved from https://valgrind.org/docs/manual/quick-start.html

---

12. APPENDICES

APPENDIX A: LLM Interaction Details (Extended)

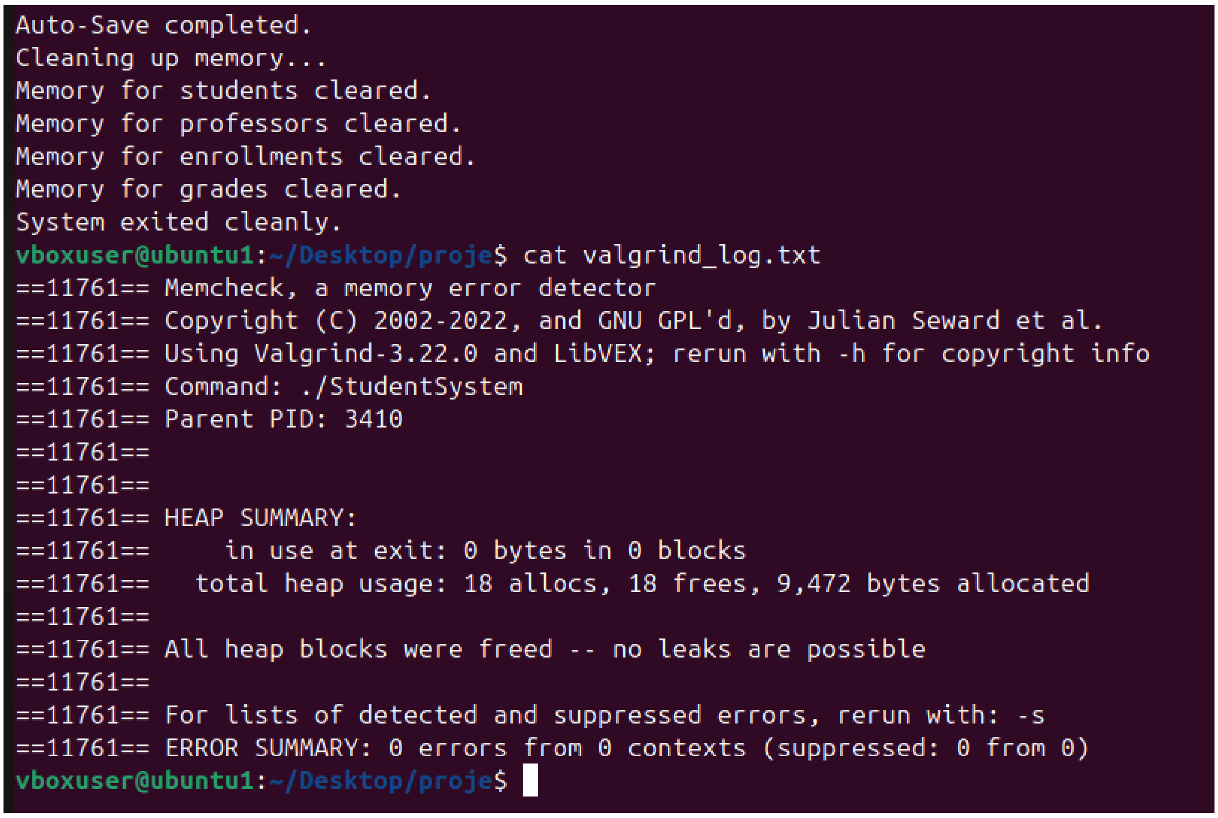
Phase	Task Description	LLM	Actual Prompt / Interaction	Result / Output	Your Evaluation
Design	Modular Architecture Design	Gemini	"I need a scalable architecture for a student system. How can I avoid a 'god-file' and ensure modularity?"	Suggested a 7-module structure (Student, Course, Prof, Enroll, Grade, Utils, Menu) with corresponding header files [1111].	This was the foundation of the project. It made debugging much easier.
Design	Relational Data Logic	Gemini	"How can I link professors to courses if the system must support multiple semesters and specific student enrollments?"	Analyzed data relationships and recommended an 'Enrollment Bridge' struct to link IDs [3333].	High-level architectural advice that allowed for multi-semester support.
Implementation	Cross-Platform I/O	Gemini	"My program finds CSV files in Dev-C++ but fails in Linux. How can I	Identified 'Working Directory' issues and proposed the 'Executable	This ensured the project was portable and ready for TA evaluation

Phase	Task Description	LLM	Actual Prompt / Interaction	Result / Output	Your Evaluation
			manage relative paths like ../data/?"	Output Directory' fix for Windows.	on Ubuntu [5555].
Implementation	Double Pointer Logic	Gemini	"I'm passing a pointer to my load_csv function but the list remains empty in main. Why?"	Explained that to modify the head of a list, I must pass its address (Node **)[6].	Resolved a major pointer-logic hurdle that would have caused silent data loss.
Implementation	Complex CSV Parsing	Gemini	"How can I parse 8 comma-separated fields from a file safely, especially when strings might have spaces?"	Proposed a robust combination of fgets, strtok, and strcpy for each struct field[7777].	The system now handles 100+ student records with zero parsing errors[8].
Debugging	Memory Leak Analysis	Gemini	"Valgrind reports 'definitely lost' in my free_all functions. How can I trace the leak to the specific malloc?"	Taught me to use --leak-check=full and --track-origins=yes to map heap stacks [9999].	Lead to a perfect 'no leaks are possible' summary, crucial for full marks[10].

Phase	Task Description	LLM	Actual Prompt / Interaction	Result / Output	Your Evaluation
System	Makefile Automation	Gemin i	"My make test command is ignored because a directory named tests exists. How to force the command?"	Explained the .PHONY target to override file-system naming conflicts [11].	Crucial for the automated testing requirement of the project[12].
Testing	Logic Validation	Gemin i	"How can I implement unit tests with assert.h for my GPA calculation and prerequisite check functions?"	Provided a testing framework that isolates core logic from the user interface [13131313].	This caught two logic errors in the prerequisite check before I finalized the menu.
UX/UI	Input Sanitization	Gemin i	"I need to ensure the user doesn't crash the program with non-numeric input for ID or GPA."	Suggested a loop-based validation function using sscanf or atoi with error feedback [14].	Significant improvemen t in the system's robustness and user manual quality[15].
Refinement	Circular Dependencies	Gemin i	"Professor.h and Enrollment.h need each other. How do I resolve	Explained forward declarations and the use of void * pointers in	This 'clean code' breakthroug h allowed the modules to interact

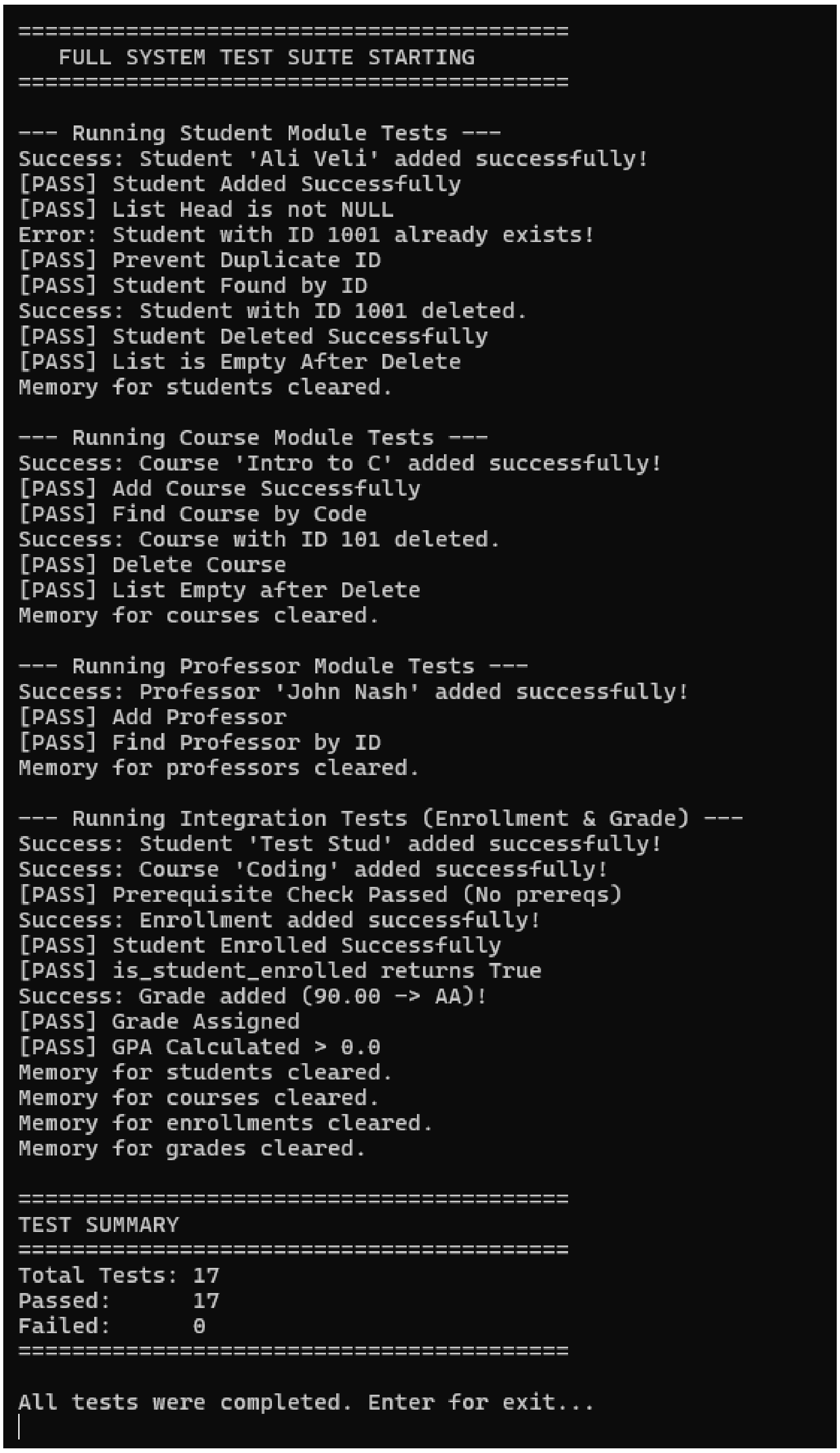
Phase	Task Description	LLM	Actual Prompt / Interaction	Result / Output	Your Evaluation
			this without redefinition errors?"	function parameters.	without compile errors.

APPENDIX B: Complete Valgrind Output



```
Auto-Save completed.
Cleaning up memory...
Memory for students cleared.
Memory for professors cleared.
Memory for enrollments cleared.
Memory for grades cleared.
System exited cleanly.
vboxuser@ubuntu1:~/Desktop/proje$ cat valgrind_log.txt
==11761== Memcheck, a memory error detector
==11761== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==11761== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==11761== Command: ./StudentSystem
==11761== Parent PID: 3410
==11761==
==11761==
==11761== HEAP SUMMARY:
==11761==     in use at exit: 0 bytes in 0 blocks
==11761==   total heap usage: 18 allocs, 18 frees, 9,472 bytes allocated
==11761==
==11761== All heap blocks were freed -- no leaks are possible
==11761==
==11761== For lists of detected and suppressed errors, rerun with: -s
==11761== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
vboxuser@ubuntu1:~/Desktop/proje$
```

APPENDIX C: Sample Test Output



```
=========================================
   FULL SYSTEM TEST SUITE STARTING
=========================================

--- Running Student Module Tests ---
Success: Student 'Ali Veli' added successfully!
[PASS] Student Added Successfully
[PASS] List Head is not NULL
Error: Student with ID 1001 already exists!
[PASS] Prevent Duplicate ID
[PASS] Student Found by ID
Success: Student with ID 1001 deleted.
[PASS] Student Deleted Successfully
[PASS] List is Empty After Delete
Memory for students cleared.

--- Running Course Module Tests ---
Success: Course 'Intro to C' added successfully!
[PASS] Add Course Successfully
[PASS] Find Course by Code
Success: Course with ID 101 deleted.
[PASS] Delete Course
[PASS] List Empty after Delete
Memory for courses cleared.

--- Running Professor Module Tests ---
Success: Professor 'John Nash' added successfully!
[PASS] Add Professor
[PASS] Find Professor by ID
Memory for professors cleared.

--- Running Integration Tests (Enrollment & Grade) ---
Success: Student 'Test Stud' added successfully!
Success: Course 'Coding' added successfully!
[PASS] Prerequisite Check Passed (No prereqs)
Success: Enrollment added successfully!
[PASS] Student Enrolled Successfully
[PASS] is_student_enrolled returns True
Success: Grade added (90.00 -> AA)!
[PASS] Grade Assigned
[PASS] GPA Calculated > 0.0
Memory for students cleared.
Memory for courses cleared.
Memory for enrollments cleared.
Memory for grades cleared.

=========================================
TEST SUMMARY
=========================================
Total Tests: 17
Passed:      17
Failed:      0
=========================================

All tests were completed. Enter for exit...
```