GEBZE TECHNICAL UNIVERSITY
Faculty of Engineering
Department of Computer Engineering

STUDENT INFORMATION SYSTEM
A Console-Based Student Information System Implemented in C

Course Name: CSE102 – C Programming

Name Surname : Muhammed Huseyin ARABOĞA
Student Number : Spring Semester
Submission Date : 16.01.2026

2.Executive Summary

This project is a final-term project carried out within the scope of the CSE102 – Programming course offered in the Computer Engineering department. The project was developed using the C programming language.
The main purpose of the project is to gain practical experience in modular software development, file-based data persistence (CSV), and input validation using the C programming language. In line with these objectives, the system was designed to perform functions similar to those of a real university system.

The developed system supports operations such as adding, deleting, searching, and updating students; managing courses and instructors; student course enrollments; and grade entry. For each course, a quota can be defined and prerequisites can be added. The project allows records to be stored across multiple academic semesters. In addition, reports such as transcripts are generated for students.

In this project, the software was designed in a manageable and reusable structure by dividing it into small components. Separate modules such as Students, Courses, Instructors, and Grades were designed. In this way, the readability and organization of the code were improved. A menu was designed for the user, and it was intended to be used via the terminal. Data persistence was ensured by using files in CSV format.

During the software development process, various challenges were encountered, and course textbooks and AI-supported tools were used to overcome them. Experience was gained in designing and managing a code structure that had not been encountered before. Ensuring that different modules work together and correctly establishing the connections between these modules constituted an important learning process. At the same time, when examining the development processes of other software written in C, this study made it clearer how much careful planning and structuring are required, especially in large-scale projects.

4.System Design

4.1 System Architecture

The project is a console application developed in the C programming language in accordance with layered architecture and modular design principles. The application is designed as separate modules (student, course, professor, enrollment, grade), each of which undertakes a specific responsibility. Communication between modules is achieved through function calls.

User Interface:
This layer is responsible for receiving user input and displaying output via the console. It deals only with user interaction and does not contain any academic rules. Relevant operations are directed according to user selections.

Domain:
This layer contains the core functions of the system. Student registration processes, adding and removing courses, enrollment conditions, transcript generation, and statistical calculations are carried out in this section.

Common Utilities:
The functions in this layer perform basic operations such as safely receiving user input, clearing the input buffer, and handling string processing.

Data Layer:
This layer manages persistent data storage. The system ensures persistence by using CSV files, which are simple, portable, and readable formats. There is a separate CSV file for each type of data (student.csv, courses.csv, etc.). Data is read from and written to files through load and save functions.

Layered	Components / Files	Responsibility
User Interface	`menu.c / menu.h`	Displays menus, receives user input, and calls the relevant module functions.
Business Logic (Domain)	`student.c/h`, `courses.c/h`, `professor.c/h`, `enrollment.c/h`, `grade.c/h`	Academic rules (capacity, prerequisites), relationship checks, reporting (transcripts, statistics).
Common Utilities	`utils.c/h`	Input reading/cleaning (read_int, clear_input_buffer, clean_newLine), shared validations.
Data Layer (Persistence)	`students.csv`, `courses.csv`, `professors.csv`, `enrollments.csv`, `grades.csv`	`Reading from and writing to files via load/save (CSV format).`

[User]
↓
(menu.c / menu.h) → (utils.c / utils.h)
↓
Domain Modules:
student.c courses.c professor.c enrollment.c grade.c
↓
CSV Persistence (load/save):
students.csv courses.csv professors.csv enrollments.csv grades.csv

4.2 Data Structure Design

4.2.1Student Structure

To store student information, the following structure is defined in the student.h file.

```
typedef struct {
    int student_id;
    char first_name[30];
    char last_name[30];
    char email[50];
    char phone[15];
    int enrollment_year;
    char major[30];
} Students;
```

student_id
It is used to uniquely identify students throughout the student information system, and all operations performed for a student are carried out using the ID. Within the system, student IDs are generated for the years between 2020 and 2024.

first_name & last_name : Fixed character lengths are preferred for student search and listing operations.

email & phone The student's contact information.

enrollment_year :Represents academic information and is used in reporting and transcript operations.

4.2.2Courses Structure

To store course information, the following structure is defined in the courses.h file.

```
typedef struct {
    int course_id;
    char course_code[10];
    char course_name[30];
    int credit;
    int capacity;
    char department[30];
    char prerequisites[30];
} courses;
```

course_id: Uniquely identifies courses in the system. It is the primary key used to establish course–student and course–instructor relationships.

course_code: Represents the short code of the course displayed to the user (e.g., CSE101, MATH202). It is kept independent from course_id to allow the same course to be offered in different semesters.

course_name: Used to store the full name of the course.

credit: Represents the credit value of the course. It is used in GPA calculations and weighting. Within the system, course credits are limited to 3 or 4.

department: Indicates the academic department to which the course belongs.

prerequisites: Represents the prerequisite courses. During enrollment, it is checked whether the student has successfully completed these courses beforehand.

4.2.3 Professor Structure

To store instructor information, the following structure is defined in the professor.h file.

```
typedef struct {
    int professor_id;
    char first_name[30];
    char last_name[30];
    char email[50];
    char phone[15];
    char department[30];
    char title[30];
} professor;
```

professor_id: Used to uniquely identify the instructor across the entire system. It is used in course assignment and instructor search operations.

first & last name: Used to store the instructor's first and last name information.

email & phone: Used to store the instructor's contact information.

department: Indicates the academic department to which the instructor belongs.

title: Represents the instructor's academic title (Professor, Associate Professor, Assistant Professor).

4.2.4 Enrollment Structure

To store enrollment information, the following structure is defined in the enrollment.h file.

```
typedef struct {
    int enrollment_id;
    int student_id;
    int course_id;
    int professor_id;
    char semester[15];
    char status[15];
} enrollment;
```

enrollment_id: Uniquely identifies each enrollment record (a student taking a course in a specific semester) within the system. It enables grade entry and allows a student to take the same course in different semesters.

student_id: The student ID used during the enrollment process.

course_id: Used during enrollment to check whether the course exists in the system, whether the course capacity is available, and whether the course prerequisites are satisfied.

professor_id: Identifies the instructor teaching the course. It plays a critical role in generating reports such as "Course roster" and "Professor course load." It supports the same course being taught by different instructors in different semesters.

semester: Indicates the academic term in which the enrollment takes place (e.g., 2024-FALL). It is the core field that enables multi-semester support in the system design. It allows the same student–course combination to be repeated across different semesters and enables filtered reports such as "courses taken this semester."

status: Indicates the status of the enrollment record.

Enrolled: The student is currently taking the course; a grade may not have been entered yet.

Completed: The course has been completed; in this case, a grade record can be created.

Only Completed records are considered in GPA calculations. The distinction between “active enrollments” and “completed courses” is reflected in reports.

2.4.5 Grade Structure

To store grade information, the following structure is defined in the grade.h file.

```
typedef struct {
    int grade_id;
    int enrollment_id;
    int student_id;
    int course_id;
    char letter_grade[3];
    double numeric_grade;
    char semester[15];
} grades;
```

grade_id: Uniquely identifies each grade record within the system. It prevents grade records from being mixed. Update, delete, and search operations can be performed using grade_id. Duplicate grade_id values are not allowed.

enrollment_id: Indicates which enrollment record the grade belongs to.

student_id: Indicates the student to whom the grade belongs.

course_id: Indicates the course to which the grade belongs.

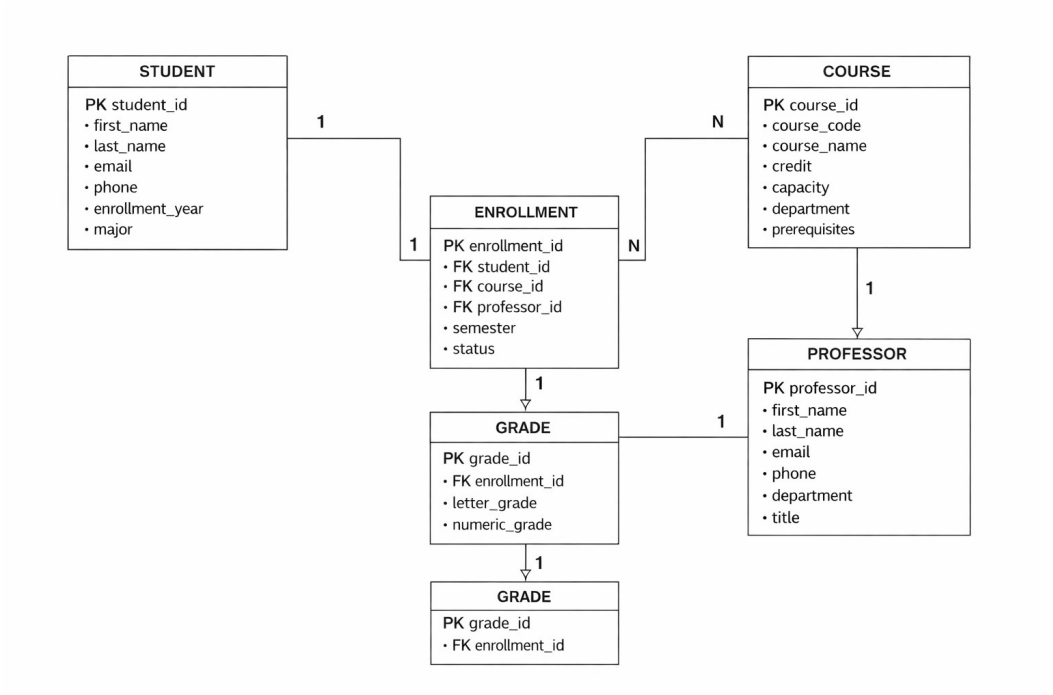
letter_grade: Stores the letter grade (e.g., A, A-, B+, B, B-, C+, C).

numeric_grade: Stores the numeric grade (between 0 and 100).

semester: Indicates the semester in which the grade was received (e.g., 2021-SPRING, 2024-FALL). It is a mandatory field for multi-semester support. It enables grouping courses by semester in transcript output and prevents grades from being mixed when the same course is taken in different semesters.

4.2.6 Entity Relationships Summary

The ER-style diagram shown in the figure illustrates the main data entities of the developed Student Information System and the relationships between these entities.



In the diagram, the Enrollment entity assumes a central role. The Enrollment structure enables the same course to be taken in different semesters, allows the same course to be taught by different professors, and makes semester-based enrollment and reporting possible. In this way, the Student, Course, and Professor entities are related through Enrollment rather than being directly connected to each other.

This structure also allows a course to be taken by many students in the same or different semesters and enables an instructor to have multiple enrollment records covering different courses and semesters.

4.2.7 Memory Layout Considerations

In this project, the memory layout of data structures was designed by considering the memory model of the C language. All core entities (Student, Course, Professor, Enrollment, Grade) are stored in dynamically allocated arrays that occupy contiguous memory locations.

Dynamic memory usage is managed in a controlled manner using separate counter variables for each module (student_count, course_count, etc.). Memory reallocation (realloc) is performed only when the capacity is full, thereby preventing unnecessary memory usage. In addition, relationships between data structures are established through ID-based references rather than direct pointer usage. This design choice simplifies memory management and improves data integrity. When the program terminates, all dynamically allocated memory areas are released using appropriate free operations, preventing memory leaks.

4.3 File Format Specification

In this project, data is stored in CSV format to ensure persistence. All CSV files contain a header row. The structure of each file is fully compatible with the header (.h) files in which the corresponding data structures are defined.

students.csv
This file contains basic information about students. The file structure is compatible with the Student data structure defined in student.h.
// file header
student_id, first_name, last_name, email, phone, enrollment_year, major
2021001,Ali,Yilmaz,ali.yilmaz@university.edu,5551234,2021,Computer Science

courses.csv
This file contains basic information about courses. The file format is compatible with the Course data structure defined in courses.h.
// file header
course_id, course_code, course_name, credit, capacity, department, prerequisites
101,CS101,Introduction to Programming,4,60,Computer Science,

professors.csv
This file contains basic information about instructors. The file structure is compatible with the Professor data structure defined in professor.h.
// file header
professor_id, first_name, last_name, email, phone, department, title
301,Ahmet,Kaya,ahmet.kaya@university.edu,5559876,Computer Science,Associate Professor

enrollments.csv
This file represents students' course enrollments and is at the center of the system's relational structure. The file structure is compatible with the Enrollment data structure defined in enrollment.h.
Each record represents a student taking a specific course in a specific semester from a specific instructor.
// file header
enrollment_id, student_id, course_id, professor_id, semester, status
5001,2021001,101,301,2023-FALL,Completed

grades.csv
This file contains grade information for completed courses. The file format is compatible with the Grade data structure defined in grade.h. Grades are associated not directly with the student but through the corresponding enrollment_id.
// file header
grade_id, enrollment_id, student_id, course_id, letter_grade, numeric_grade, semester
7001,5001,2021001,101,A,92.5,2023-FALL

4.4 Function Hierarchy

The system is built on a menu-based architecture. The program flow starts with the main function, and user interaction is directed from the main menu to the relevant module functions. Each main module (Student, Course, Professor, Grade, Enrollment) has its own area of responsibility.

main()
├─ mainMenu()
│ ├─ Student Management
│ ├─ Course Management
│ ├─ Professor Management
│ ├─ Enrollment Management
│ ├─ Grade Management
│ └─ System Options
└─ freeAllData()

4.5 Algorithm Descriptions

Search Algorithms

Search operations in the system are performed using the linear search method on dynamic arrays.

Student Search:
Students can be searched by student_id, first and last name, or email.

Course Search:
Courses are searched by course_id or course code (course_code).

Professor Search:
Instructors can be found using professor_id or first and last name.

Enrollment Search:
Enrollments can be filtered by enrollment_id, student ID, or course ID.

Grade Search:
Grade records are searched by the related enrollment_id, student, or course information.

Validation Logic

In student operations, the uniqueness of the student_id field and its validity within the 2020–2024 range are checked, along with the email format and whether the phone number consists only of numeric characters and has a valid length. In course operations, the uniqueness of the course_id field, the conformity of the course code to the defined format, the validity of the credit value, and whether the capacity is positive are verified. In instructor operations, it is checked that the professor_id field is within the 5000–5999 range and is unique. During enrollment, it is examined whether the student and the course exist in the system, the capacity status, whether the semester

information conforms to a valid academic format, and whether prerequisite courses have been completed.

5. Implementation Details

5.1 Module Descriptions

5.1.1 Purpose of each file .c/.h

In this section, the modular structure of the system and the functions of each module are explained.

Student Module (student.c / student.h):
Performs student addition, deletion, update, search, and display operations. It also communicates with other modules to generate transcripts and calculate GPA for students.

Course Module (courses.c / courses.h):
Course addition, editing, deletion, and search operations are carried out under this module. Credit, capacity, and prerequisite information for courses are maintained here. Course information is also used for reporting and creating course lists (rosters).

Professor Module (professor.c / professor.h):
Handles instructor addition, editing, deletion, and search operations. In addition, listing the courses taught by an instructor on a semester basis is provided through this module, enabling the generation of course load reports.

Enrollment Module (enrollment.c / enrollment.h):
Student enrollment in courses and deletion of enrollments are performed through this module. During enrollment, checks are performed to verify whether the student exists in the system, whether the course capacity is sufficient, and whether prerequisite courses have been completed.

Grade Module (grade.c / grade.h):
This module handles grade addition and update operations and applies consistency checks between letter grades and numeric grades. It also calculates course-based grade statistics, grade distributions, and overall grade analyses.

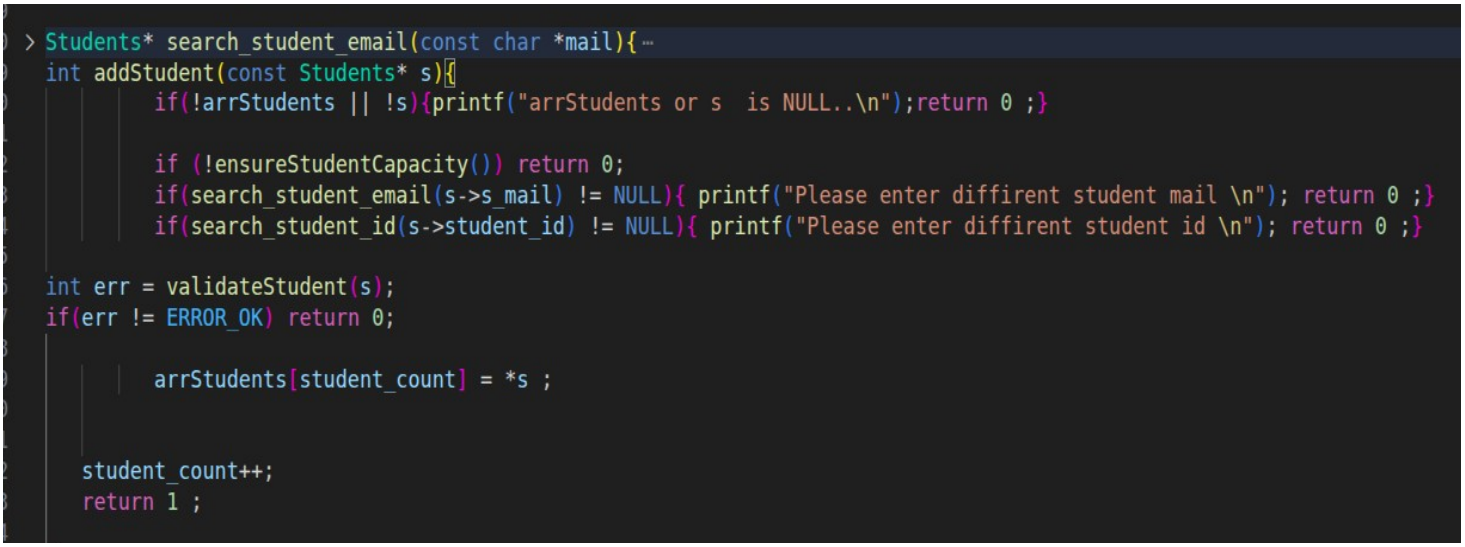
Utilities Module (utils.c / utils.h):
Helper functions such as string cleaning and formatting operations, date validation, and numeric checks are included in this module.

5.1.2 Key Functions

student.c

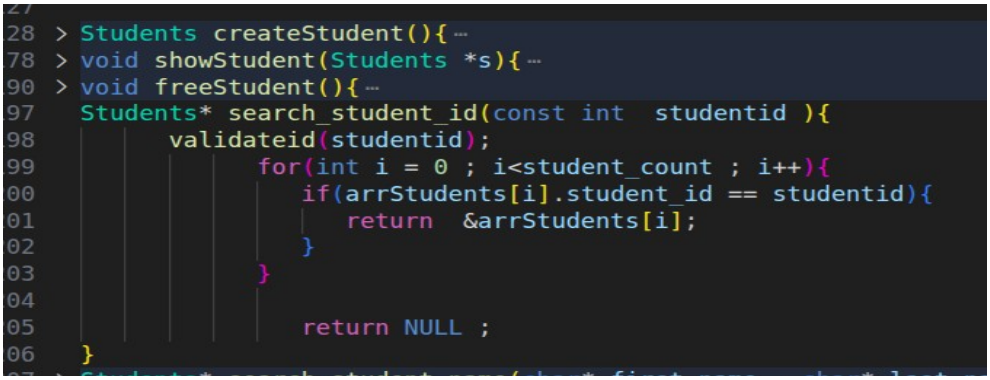
The **addStudent() f**unction is responsible for adding a newly created student record to the dynamically maintained student array in the system. The function first checks whether the arrStudents and s pointers are NULL to ensure memory and parameter safety. Then, the ensureStudentCapacity() function is called; if the array capacity is full, the capacity is increased using realloc to allow the insertion process to continue safely.

To preserve data integrity, the uniqueness of the student's email address (search_student_email) and the uniqueness of the student ID (search_student_id) are checked before insertion. Finally, validateStudent() is used to revalidate all format and range constraints of the student record (ID, email domain, phone format, etc.). When all checks pass successfully, the student record is copied into the array, and the insertion process is completed by incrementing student_count.



```
> Students* search_student_email(const char *mail){…
int addStudent(const Students* s){
        if(!arrStudents || !s){printf("arrStudents or s  is NULL..\n");return 0 ;}

        if (!ensureStudentCapacity()) return 0;
        if(search_student_email(s->s_mail) != NULL){ printf("Please enter diffirent student mail \n"); return 0 ;}
        if(search_student_id(s->student_id) != NULL){ printf("Please enter diffirent student id \n"); return 0 ;}

int err = validateStudent(s);
if(err != ERROR_OK) return 0;

        arrStudents[student_count] = *s ;


   student_count++;
   return 1 ;
```

The **search_student_id()** function enables searching for students registered in the system by student ID. The function uses a linear search approach on the dynamic student array, checking the student_id field of each record. If a match is found, the address of the corresponding student record is returned; if no match is found, a NULL value is returned to indicate that the search was unsuccessful.



```
> Students createStudent(){…
> void showStudent(Students *s){…
> void freeStudent(){…
Students* search_student_id(const int  studentid ){
      validateid(studentid);
            for(int i = 0 ; i<student_count ; i++){
               if(arrStudents[i].student_id == studentid){
                  return  &arrStudents[i];
               }
            }

               return NULL ;
}
```

The **delete_Student()** function ensures the deletion of a registered student from the system and the cleanup of all related data. The function first uses validateid() to check whether the student ID conforms to the expected format (YEAR###) and falls within the supported year range. It then verifies that the dataset is loaded by checking arrStudents and confirms the existence of the student in the system using search_student_id().

To maintain data integrity, all records related to the student are removed before deletion: the student's grade records are deleted using deleteGradeBy_StudentId(), and enrollment records are removed using deleteEnrollment_Student(). When deleting the student record from the array, instead of shifting elements for performance reasons, the last element of the array is moved into the position of the deleted element, and student_count is decremented. Finally, the number of grade records and enrollment records deleted is reported to the user as an informational message.

```c
int delete_Student(int student_id){

   if(!validateid(student_id)){
      printf("Student ID must be YEAR### (e.g., 2020001) and year must be 2020-2024.\n");
      return 0 ;
   }

   if(!arrStudents) { printf("Error. Plese load students. \n");return 0 ;}
   Students *a = search_student_id(student_id);
if(!a){
   printf("No such a student . \n");
   return 0 ;
}

int deletedGrade = deleteGradeBy_StudentId(student_id);
int deletedEnrollmet = deleteEnrollment_Student(student_id);
int found = 0 ;
   for(int i = 0 ; i<student_count ; i++){
      if(arrStudents[i].student_id == student_id){
            arrStudents[i] = arrStudents[student_count -1 ];
            student_count-- ;
            found = 1 ;
            break;
      }
   }
   if(!found){
            printf("Student could not be deleted.\n");
            return 0 ;
         }

printf("Deleted student. Remove %d  grades , and %d enrollment record\n",deletedGrade,deletedEnrollmet);
return 1 ;
}
```

The **calculate_GPA()** function computes a student's Grade Point Average (GPA) using a credit-weighted average approach. The function first checks whether the required datasets (grades, students, enrollments) are loaded, and validateid() is used to verify that the student ID is in a valid format.

During the calculation process, the function iterates over the arrGrades array to select the grade records belonging to the specified student. For each grade record, the status of the corresponding enrollment record is checked via arrEnrollments, and only courses with the status Completed are included in the GPA calculation. Then, the course credit information is retrieved from the Course module using search_course_id(). The numeric grade value is converted to a 4.0 scale using numeric_grade / 25.0, and the result is accumulated as (grade_point × credit) and divided by the total credits.

```
double calculate_GPA(int student_id){

    if(!arrGrades || !arrStudents || !arrEnrollments ) return 0 ;
    if(validateid(student_id) != 1){
       printf("Student ID must be YEAR### (e.g., 2020001) and year must be 2020-2024.");
return 0 ;
    }

double sum_credit= 0 ;
int total_credi = 0;
        int i = 0 , j = 0;
           for(i = 0 ; i<grade_count; i++){
              if(arrGrades[i].student_id !=student_id) continue; ;

              int completed = 0;
              for(j = 0 ;j<enrollment_count ; j++){
                 if(arrEnrollments[j].enrollment_id == arrGrades[i].enrollment_id){
                 if (strcmp(arrEnrollments[j].status, "Completed") == 0) {
                    completed = 1;
               }
               break;
              }
           }
    if(!completed) continue;

    courses* c = search_course_id((arrGrades[i].course_id));
    if(!c) continue;
           double gp = arrGrades[i].numeric_grade / 25.0;
          sum_credit += gp * c->credit ;
          total_credi += c->credit ;

}
    if(total_credi == 0)return 0 ;


return sum_credit/(double)total_credi;
```

Course Module (courses.c / courses.h)

The **`init_Courses()`** function is used to initialize the dynamic data structure that will store course data. This function is called at system startup or before loading course data, and it allocates the initial memory for the course array using `malloc`. The initial capacity is set to a fixed value, and by resetting the `course_count` variable to zero, the system is ensured to start in a clean state. If memory allocation fails, the program is terminated in a controlled manner to prevent potential memory errors later.

The **`ensureCourseCapacity()`** function checks whether the current number of courses exceeds the defined maximum capacity. If the capacity is full, the course array is moved to a larger memory area using `realloc`, and the maximum capacity is increased. If `realloc` fails, a warning is displayed to the user.

```c
static int ensureCourseCapacity(void)
{
    if (course_count < maxCourse) return 1;

    int newCap = maxCourse * 2;

    courses *tmp = realloc(arrCourses, newCap * sizeof(courses));
    if (!tmp) {
        printf("Error: realloc failed for Courses\n");
        return 0;
    }
    arrCourses = tmp;
    maxCourse = newCap;
    return 1;
}
void init_Courses(){
    maxCourse = 20 ;
     arrCourses = (courses*)malloc(maxCourse* sizeof(courses));
    if(!arrCourses){
        printf("Memory allecation is failed");
        exit(1);
    }
    course_count = 0;
}
```

addPrerquistis()

The `addPrerquistis(int course_id, const char *preq)` function allows the definition of prerequisite information for a specific course. The function first checks whether the given course exists in the system. Then, it validates the format of the prerequisite course codes provided and verifies that they are defined in the system.

When all validations are successful, the prerequisite information is assigned to the relevant course record.

```c
int addPrerquistis(int course_id ,const char* preq){

    if(!arrCourses){
        printf(" Please load courses data.\n");
        return 0 ;
    }
    courses *c = search_course_id(course_id);

    if(!c){
        printf("Course not found. \n");
        return 0 ;
    }
    courses temp = *c ;

        if(!preq || preq[0] == '\0' || strcmp(preq,"0") == 0){
    c->prerquistis[0] = '\0';
    printf("Prequistis cleaned.\n");
    return 1 ;
}
        strncpy(temp.prerquistis, preq, sizeof(temp.prerquistis) - 1);
        temp.prerquistis[sizeof(temp.prerquistis) - 1] = '\0';


if(!validatePrequis(&temp)){
    printf("Invalid prerquisite format \n");
    return 0 ;
}
strncpy(c->prerquistis, preq, sizeof(c->prerquistis) - 1);
c->prerquistis[sizeof(c->prerquistis) - 1] = '\0';
return 1 ;

}
```

`updateCourse()`

The `updateCourse()` function is used to update the information of a course registered in the system. The function first checks whether the course record corresponding to the given course ID (`course_id`) exists in the system. If the course cannot be found, the operation is terminated to prevent an invalid update.

When the course is found, fields such as the course name, credit, capacity, department, and prerequisite are updated according to user input. After the update process, `validate_Course()` is called to re-validate rules such as credit value (3/4), capacity (>0), course code level (100–400), and prerequisite format. In this function, the `course_id` and `course_code` fields are kept fixed, while the course name, credit, capacity, department, and prerequisite information are treated as updatable fields.

```c
int updateCourse(int course_id){
    courses *c  = search_course_id(course_id);
        if(c == NULL){
            printf("Course not found \n :");
           return 0 ;
        }
        printf("\n------Current course details : ------\n");
        showCourse(c);
        printf("-----Enter New Data ( ID and Code will not change )-----\n");
        courses intput = createCourse(0);
        courses  new =  *c ;
   strncpy(new.course_name , intput.course_name,sizeof(new.course_name) -1 );
   new.course_name[sizeof(new.course_name) -1 ] = '\0';
new.credit = intput.credit;
new.capasity = intput.capasity;

   strncpy(new.department , intput.department,sizeof(new.department) -1 );
   new.department[sizeof(new.department) -1 ] = '\0';

 strncpy(new.prerquistis , intput.prerquistis,sizeof(new.prerquistis) -1 );
   new.prerquistis[sizeof(new.prerquistis) -1 ] = '\0';
         courses *temp = search_course_name(new.course_name);
         if(temp != NULL  && temp !=c){
            printf("Duplicate course name : \n");
            return 0 ;
         }

    Error_Code e = validate_Course(&new);
    if(e  != COURSE_OK)
    {
        printf("%s\n",validation_errorMSG(e));
        return 0 ;
    }
         *c = new;
         return 1 ;

        }
```

Enrollment Module (enrollment.c / enrollment.h)

listStudentEnrollment()

The `listStudentEnrollment()` function is used to list all course enrollments of a specific student. The function iterates over the `arrEnrollments` array to find records that match the given `student_id` and optionally applies semester-based filtering using the `semesterOptional` parameter. For each record, the course code information is retrieved via `search_course_id()`, and the results are printed in a table format (enrollment_id, course_id, professor_id, course code, semester, date, status). If no records are found, an error message is displayed to the user.

```c
void listStudentEnrollment(int student_id ,const char *semesterOptinal){

        if(!arrEnrollments){
            printf("Printf arrEnrollment is null . Call init arrEnrollment .\n");
          return ; }
        int found = 0 ;
int i = 0 ;
printf("\n==== Student Enrollments (student id : %d ) ====\n",student_id);
        if (semesterOptinal && *semesterOptinal)
        printf("Semester filter: %s\n", semesterOptinal);
 printf("+----------+--------+----------+-----------------+------------+------------+------------+\n");
    printf("| EnrollID | CID    | ProfID   | Course (Code)   | Semester   | Date       | Status     |\n");
    printf("+----------+--------+----------+-----------------+------------+------------+------------+\n");

for(i = 0 ; i<enrollment_count ; i++){

    if(arrEnrollments[i].student_id != student_id)
    continue;

    if(semesterOptinal && semesterOptinal[0] !='\0'){
        if(strcmp(arrEnrollments[i].semester,semesterOptinal) != 0 )
        continue;
    }

courses *c = search_course_id(arrEnrollments[i].course_id);

const char *code = (c ? c->course_code : "-");

printf("| %-8d | %-6d | %-8d | %-15s | %-10s | %-10s | %-9s |\n",
       arrEnrollments[i].enrollment_id,
       arrEnrollments[i].course_id,
       arrEnrollments[i].professor_id,
       code,
       arrEnrollments[i].semester,
       arrEnrollments[i].enrollment_date,
       arrEnrollments[i].status);
    found = 1 ;

}printf("+----------+--------+----------+-----------------+------------+------------+------------+\n");
    if(!found){
        if(semesterOptinal && semesterOptinal[0] != '\0')
        printf("No enrollments found for this stdudent in semester %s \n",semesterOptinal);
        else{
            printf("No enrollments found for this student \n");
```

addEnrollment(),

`addEnrollment()` is the main function that enables a student to be enrolled in a course. The function sequentially checks the capacity status (`ensureEnrollmentCapacity`), validates the enrollment data (`validateEnrollment`), prevents `enrollment_id` conflicts, verifies the existence of the student, course, and instructor, and prevents duplicate enrollment in the same course within the same semester (`checkStudent_Enrollment`). It then applies prerequisite checking (`check_prerquisties`) and course capacity control (`checkCourse_capacity`). If a new enrollment is requested to be initialized as “Completed,” the function automatically converts the status to “Enrolled” to maintain process consistency.

A new enrollment record cannot initially be created with the status “Completed,” because a course can only be considered completed if a corresponding grade record exists in the system. Therefore, the system automatically converts enrollments entered as “Completed” into the “Enrolled” status at creation time to preserve the consistency of the enrollment–grade relationship.

```c
int addEnrollment(const enrollment *e){
    if(!arrEnrollments || !e  ) {
        printf("arrEnrollment or e is Null  \n");
        return 0 ;
    }
      if (!ensureEnrollmentCapacity()) return 0;
int err = validateEnrollment(e);
    if (err != ENROLLMENT_OK) {
        printf("%s\n", validation_enrollment_msg(err));
        return 0;
    }
    if(search_enrollment_id(e->enrollment_id) !=NULL){
    printf("Duplicate enrollment \n");
    return 0;
}
if(search_student_id(e->student_id) == NULL){
    printf("Student not found . ID : %d\n",e->student_id);
    return 0 ;
}
if(search_course_id(e->course_id) == NULL){
    printf("Course not found  ,  ID %d\n ",e->course_id);
    return 0 ;}
if(search_prof_id(e->professor_id) == NULL){
    printf("Professor not found, ID   %d \n",e->professor_id);
    return 0 ;}
if(checkStudent_Enrollment(e->student_id,e->course_id,e->semester)){
    printf("This student already enrolled int this course for this semester \n");
    return 0;
}
if(!check_prerquisties(e->student_id,e->course_id)){
    printf("Prerquisties are not satisfied.\n");
    return 0 ;
}
if(!checkCourse_capacity(e->course_id,e->semester)){
    printf("Course capacity is full\n");
    return 0 ;}
        enrollment  temp = *e;
        if(strcmp(temp.status,"Completed") == 0 ){
            printf("Warning : New enrollment can not start as Completed. \n");
            printf("Student status change Enrolled. If you want to Completed you need to add grade.\n");
            strcpy(temp.status ,"Enrolled");
        }
arrEnrollments[enrollment_count] = temp;
enrollment_count++;
printf("Enrollment added \n");
printf("Remaing capacity : %d \n",remaingCourseCapasty(e->course_id,e->semester));
return 1 ;
```

Grade Module (grade.c / grade.h)

The `int course_statistics(int course_id, const char *semester, double *avg, double *min, double *max)` function calculates the average, minimum, and maximum grade values for a given course and semester using the grades entered in the system.The `arrGrades` array is scanned, and only records matching the specified `course_id` and `semester` are considered. The calculated statistics are returned through the `avg`, `min`, and `max` output parameters. If no grades are found for the given course and semester, the function returns 0 to indicate that statistics could not be generated.

```c
int course_statistics(int course_id, const char *semester,double *avg, double *min, double *max)
{
 if(!arrGrades) {printf("Error: arrGrades is NULL. Call init_Grades() first.\n");return 0;
    }
if(!semester || semester[0] == '\0') {printf("Error: semester is empty.\n");
   return 0;
    }
if(!avg|| !min || !max) {
printf("Error: avg/min/max pointers cannot be NULL.\n");
return 0;
 }
double sum = 0.0;    double mn = DBL_MAX;
double mx = -DBL_MAX;
 int cnt = 0;
int i = 0 ;
for (i = 0; i < grade_count; i++) {
if (arrGrades[i].course_id == course_id &&
strcmp(arrGrades[i].semester, semester) == 0)
        {
                double x = arrGrades[i].numeric_grade;
            sum += x;
if(x < mn) {mn = x;}
if(x > mx) {mx = x;}

        cnt++;
   }
    }
 if(cnt == 0) {
  printf("No grades found for course %d in %s\n", course_id, semester);        return 0;
    }
*avg = sum / cnt;
*min = mn;
*max = mx;

return 1; // başarı
}
```

The `void grade_distribution(int course_id, const char *semester)` function calculates and displays the distribution of letter grades for a given course and semester. The `arrGrades` array is scanned, and only grades matching the specified `course_id` and `semester` are evaluated. Counters are maintained for each letter grade (A, A-, B+, B, B-, C+, C, etc.), and the total number of students is calculated.

```c
void grade_distribution(int course_id, const char *semester){

    if (!arrGrades) {
        printf("Error: arrGrades is NULL. Call init_Grades() first.\n"); return;  }
    if (!semester || semester[0] == '\0') {
        printf("Error: semester is empty.\n"); return; }
    int A_counter=0, A_ncounter=0, B_pcounter=0, B_counter=0, B_ncounter=0, C_pcounter=0, C_counter=0, Total=0;
int i = 0;
    for (i = 0; i<grade_count;i++) { if (arrGrades[i].course_id == course_id &&
            strcmp(arrGrades[i].semester, semester) == 0)
        {
            Total++;
            if(strcmp(arrGrades[i].letter_grade, "A")  == 0) A_counter++;
            else if(strcmp(arrGrades[i].letter_grade, "A-") == 0) A_ncounter ++;
            else if(strcmp(arrGrades[i].letter_grade, "B+") == 0) B_pcounter++;
            else if(strcmp(arrGrades[i].letter_grade, "B")  == 0) B_counter++;
            else if(strcmp(arrGrades[i].letter_grade, "B-") == 0) B_ncounter++;
            else if(strcmp(arrGrades[i].letter_grade, "C+") == 0) C_pcounter++;
            else if(strcmp(arrGrades[i].letter_grade, "C")  == 0) C_counter++; } }
    if (Total == 0) {
        printf("No grades found for course %d in %s\n", course_id, semester); return; }
    printf("Grade Distribution | Course %d | %s | Total=%d\n", course_id, semester, Total);
        printf("A : %d\n", A_counter);
        printf("A-: %d\n", A_ncounter);
    printf("B+: %d\n", B_pcounter);
        printf("B : %d\n", B_counter);
    printf("B-: %d\n", B_ncounter);
        printf("C+: %d\n", C_pcounter);
     printf("C : %d\n", C_counter);
}

```

5.2 Challenges Faced and Solutions

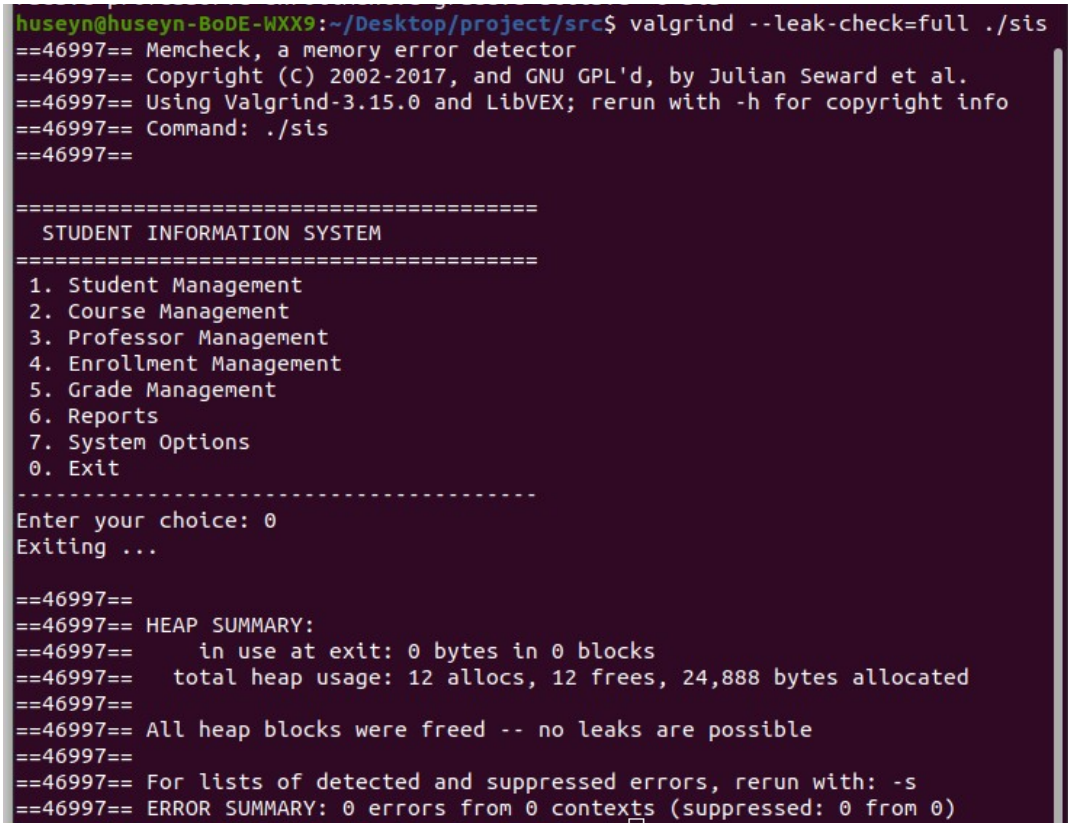
I had never worked on a project of this scale before. For the first time, I wrote a large amount of code and developed a system in which many files are interconnected. At the beginning, maintaining code organization and correctly establishing relationships between modules was challenging for me.

Especially in the C language, I encountered many issues related to memory management, pointer usage, and data transfer. By experimenting, creating different scenarios, and carefully examining the error messages produced by the program, I tried to identify the source of the errors. In faulty cases, I printed meaningful error messages to the screen, followed the problem step by step, and was able to fix issues such as incorrect memory usage, missing checks, and logical errors.

During this process, I benefited from the course textbook, online resources, and AI-assisted tools to find solutions to the problems I encountered. By the end of the project, not only my coding skills but also my debugging, trial-and-error, and problem-solving abilities had significantly improved.

5.3 Memory Management

In the student, course, instructor, enrollment, and grade modules, data is stored using dynamic memory. For each module, an array is initially created using `malloc`, and memory allocation is performed through the corresponding initialization functions (`init_Students`, `init_Courses`, `init_Prof`, `init_Enrollments`, `init_Grades`). When the amount of data increases and the capacity becomes insufficient, the array sizes are expanded using `realloc`. During memory allocation operations, failure cases are checked and appropriate error messages are generated to ensure the safe execution of the program. At the end of the program, all allocated memory areas are released using `free`. With this approach, memory leaks are prevented, and the project was tested using the Valgrind tool to verify that memory management works correctly.



5.4 Input Validation

In this project, a general input validation structure was implemented to ensure that the data received from the user is correct and consistent. All data entered into the system is checked in terms of type, format, and logical validity.

In Student operations, the validity and uniqueness of the student ID, non-empty name fields, and the correct format of email and phone information are verified. In Course operations, the course ID, credit value, capacity value, and prerequisite formats are checked. In Professor operations, the validity of the ID number and contact information is verified.In Enrollment operations, it is checked that the student and the course exist in the system, that the capacity is not full, and that prerequisites have been completed. In Grade operations, the uniqueness of the grade ID, the consistency between the letter grade and the numeric grade, and that grades are entered only for valid enrollments are verified.Validation processes are carried out using separate `validate` functions written for each module, and in error cases, helper message functions that return meaningful messages to the user are used. Below, the `validateStudent`, `validation_msg`, `validateGrade`, and `validationGrade_msg` functions are provided as examples of this structure.

```
int validateStudent(const Students *s){
                if(s == NULL){ return  ERROR_NULL;}
      int year = s->student_id / 1000 ;
      if(s->student_id <1000000 || s->student_id >9999999 )return  ERROR_STUDENT_ID ;
         if(year <2020 || year >2024 ){ return ERROR_STUDENT_ID ;}
      if(strlen(s->first_name)  == 0  || strlen(s->last_name)== 0 ) return ERROR_STUDENT_NAME;
   if (!is_valid_name(s->first_name) || !is_valid_name(s->last_name))
    return ERROR_STUDENT_NAME;
      if (strchr(s->s_mail, '@') == NULL) return ERROR_STUDENT_MAIL;
       if (strstr(s->s_mail, "@university.edu") == NULL) {
        return ERROR_STUDENT_MAIL ;
    }
        if (strlen(s->s_phone) != 8) {
                return ERROR_STUDENT_PHONE;
    }
// Burada ilk  3 index 5 5 5 - olmalı onun kontrolu.
      if(s->s_phone[0] != '5'
|| s->s_phone[1] != '5' || s->s_phone[2] != '5' || s->s_phone[3] != '-')return ERROR_STUDENT_PHONE ;
         //Son 4 index rakam olmalı.
for (int i = 4; i <= 7; i++) {
    if (!isdigit((unsigned char)s->s_phone[i])) {
        return ERROR_STUDENT_PHONE;}
   }
   if (s->enrollment_year < 1900 || s->enrollment_year > 2100) {
        return ERROR_STUDENT_YEAR; }
         if (s->gpa < 0.0 || s->gpa > 4.0) {
        return ERROR_STUDENT_GPA;
    }
    if (strcmp("Computer Engineering", s->major) != 0 &&
    strcmp("Industrial Engineering", s->major) != 0 &&
    strcmp("Mechanical Engineering", s->major) != 0 &&
    strcmp("Electrical Engineering", s->major) != 0 &&
    strcmp("Computer Science", s->major) != 0){
    return ERROR_STUDENT_MAJOR;
}
return ERROR_OK;
```

```
const char* validation_error_msg(ErrorCode e) {
    switch (e) {
         case  ERROR_NULL : return " Student is null try again. \n.";
        case ERROR_OK: return "OK";
        case ERROR_STUDENT_ID: return "Student ID must be YEAR### (e.g., 2020001) and year must be 2020-2024.";
        case ERROR_STUDENT_NAME: return "First/Last name cannot be empty and must be invalid..";
        case ERROR_STUDENT_MAIL: return "Email must be a university email (must include @university.edu).";
        case ERROR_STUDENT_PHONE: return "Phone must be exactly in format 555-#### (e.g., 555-0101).";
        case ERROR_STUDENT_YEAR: return "Enrollment year must be between 1900 and 2100.";
        case ERROR_STUDENT_GPA: return "GPA must be between 0.0 and 4.0.";
        case ERROR_STUDENT_MAJOR: return "Major must be one of allowed majors (e.g., Computer Engineering).";
        default: return "Unknown validation error.";
    }
```

```
> void grade_distribution(int course_id, const char *semester){…

   Grade_Error validateGrade(const grades *g ){
       if(!g) return GRADE_NULL ;
       if(g->grade_id<10000 || g->grade_id>99999)
       return GRADE_ID;

          if (g->enrollment_id <= 0) {
           return GRADE_ENROLLMENT;
       }
   if (g->student_id < 1000000 || g->student_id > 9999999) return GRADE_STUDENT;
      int year = g->student_id / 1000 ;
       if(year <2020 || year >2024 ){ return GRADE_STUDENT ;}

         if(g->course_id<= 0){
           return  GRADE_COURSE;
       }
   if (!validateSemester(g->semester))
       return GRADE_SEMESTER;
       if(!(strcmp(g->letter_grade,"A")== 0 ||  strcmp(g->letter_grade,"A-")== 0
       || strcmp(g->letter_grade,"B+")== 0 ||strcmp(g->letter_grade,"B")== 0 ||
       strcmp(g->letter_grade,"B-")== 0  || strcmp(g->letter_grade,"C+")== 0
        || strcmp(g->letter_grade,"C")== 0) )
           return GRADE_LETTER ;
   if(g->numeric_grade < 0.0 || g->numeric_grade > 100.0)
       return GRADE_NUMERIC;
       if(!checkletter_numeric(g->letter_grade,g->numeric_grade))
       return GRADE_NUMERIC;
          return  GRADE_OK;
   }
```

```
const char * validationGrade_msg(Grade_Error err){
    switch (err)
    {
    case GRADE_NULL :  return "Grade pointer is null";
    case GRADE_OK  :    return "Grade data is valid ";
    case GRADE_ID :       return "Grade must be 5 digit and >0";
    case GRADE_ENROLLMENT :       return "Enrollment id must be greater than 0";
    case GRADE_STUDENT :  return"Invalid student id , id must be 7 digit and 2020-2024";
    case GRADE_COURSE :    return "Invalid course id , id must be grater than 0";
    case GRADE_LETTER :     return"Invalid letter grade , A , A- , B+, B- ,B, C+,C";
    case GRADE_NUMERIC :      return"Invalid numeric grade , grade must be 0<grade<100";
    case GRADE_SEMESTER :   return "Invalid semester format , (YYYY-FALL) or (YYYY-SUMMER), (YYYY-SPRING)";
    default:        return "Unknwn enrollment error" ;
    }
```

6 .LLM Collaboration Documentation

6.1 LLM Usage Philosophy

In this project, Large Language Model (LLM) tools were used not as ready-made solution providers, but as assistants that support and guide the software development process. The primary purpose of LLMs was not to generate code directly, but to contribute to critical processes such as validating ideas, analyzing problems, and debugging errors.

LLMs were particularly utilized in the following stages:

- System architecture and data structure design,
- Reviewing sample function designs and restructuring them to fit the project,
- Analyzing possible causes of runtime errors (segmentation faults, memory errors, etc.),
- Developing approaches for validating user inputs and ensuring the consistency of system outputs,
- Organizing menu/panel structures and improving the readability of user-facing outputs (e.g., transcript screens, GPA calculation results).

All suggestions provided by the LLM were evaluated, and no output was directly copied without being tested and verified beforehand.

In some cases, solutions proposed by the LLM were consciously rejected or redesigned because they conflicted with project requirements or good programming practices. For example, some string-based approaches suggested for prerequisite checks were considered unreliable, and a more controlled comparison method was preferred.

Areas Where LLMs Were Not Used

Final design of business logic:
Fundamental rules—such as how student–course–enrollment–grade relationships are established, how a course is handled based on its “Enrolled” or “Completed” status, and the rule that grades are entered only for completed courses—were entirely determined by the developer.
In GPA calculation, the weighted consideration of course credits, evaluation of only courses with valid grades, and handling of missing grade cases were designed and implemented by the developer. LLMs were not used in the final logic of these calculations.

The use of `malloc`, `realloc`, and `free`, capacity expansion strategies, and checks to prevent memory leaks were manually implemented by the developer.
The content of error messages and the feedback presented to the user,
as well as the relationship between the `main` function and the menu structures, program control flow, and user interaction, were entirely planned and implemented by the developer.

6.2 Detailed LLM Interaction Table

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Design	Selecting an appropriate data structure for storing students	ChatGPT	“For storing students, which is more suitable: an array or a linked list for this project?”	The LLM recommended using a dynamic array due to the limited dataset size and CSV-based workflow, highlighting better cache locality and simpler implementation. It also explained when a linked list might be preferable.	The recommendation aligned well with project requirements. Dynamic arrays were chosen for students, courses, and professors. The trade-offs of linked lists were clearly understood and consciously avoided.	It was understood that using a linked list would require larger, more dynamic, and complex datasets, whereas for the scale of this project, dynamic arrays provide a simpler and safer solution.

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Design	Designing data structures for CSV-based system	ChatGPT	“Create a sample struct for me to access data from CSV files.”	Suggested CSV-oriented struct design with fixed-size fields and dynamic arrays.	The overall design idea was adopted, but `#define MAX_*` macros and per-struct `is_active` flags were not used. Instead, data management was implemented through a centralized `System` structure with dynamic arrays and capacity tracking.	Learned that in CSV-based systems, how data is managed at the system level is more critical than individual struct definitions. Design patterns must be adapted, not copied.

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Planning	Identifying the right questions for each project stage	ChatGPT	“Could you give me the questions I need to ask according to the stages of my project assignment? I don’t want answers.”	Provided a stage-by-stage checklist covering design, data structures, CSV I/O, validation, GPA, memory management, and documentation.	The output was used only as a planning guide, not for direct implementation. Questions were referenced selectively during development.	Learned that LLMs can be effectively used to structure thinking and guide the development process, not just to generate solutions.

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Testing / Debugging	Understanding the use of `assert.h` in C	ChatGPT	“Briefly explain the include <assert.h> library in C and give me an example.”	Explained that `assert()` is used to detect logic errors during development and should not be used for user input validation.	`assert()` was investigated because it appeared in the assignment materials. It was applied only in test files and not in the main program logic.	Learned that `assert()` is intended for testing and debugging purposes and should be avoided in production code paths.

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Conceptual	Understanding the purpose of CSV files	ChatGPT	“What is the purpose of a CSV file and how does it differ from other files?”	Explained CSV as a plain-text format for tabular data and compared it with binary files, databases, and JSON/XML.	Used as conceptual guidance to justify the CSV-based file design used throughout the project. Fully consistent with the implemented file I/O approach.	Learned why CSV is suitable for small to medium-sized C projects requiring simple and transparent data storage.

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Design	Defining the public API of `student.h`	ChatGPT	“Which functions should be in the Student.h file?”	Suggested putting struct definitions, extern globals, and student CRUD/search/validation prototypes in `student.h`.	The modular guideline was useful, but some suggested CRUD APIs (especially delete) did not account for SIS relationships, potentially breaking enrollment/grade consistency. Therefore, deletion/update logic was designed more carefully to prevent orphan records and preserve data integrity.	Header/API design must consider cross-module relationships and data integrity, not only module boundaries.

Phase	Task Description	LLM Used	Actual Prompt	Result / Output	Evaluation	What I Learned
Design	Defining the content of `utils.c / utils.h`	ChatGPT	“There should be a file like utils.c-h among the project files. Could you give me some example functions I can use here?”	Suggested a utility module covering input handling, string utilities, validation helpers, and menu-related functions.	The suggested functions were **evaluated and tested**. Based on the test results, only the utilities that matched the project requirements and reduced code duplication were integrated into the system.	Learned that centralizing shared helper functions improves maintainability and keeps core modules focused.

Phase	Task Description	LLM Used	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Testing / Debugging	Checking memory leaks using Valgrind	Claude	“How can I check my C program for memory leaks using Valgrind?”	Explained how to run Valgrind from the terminal and interpret memory leak reports.	The provided commands were executed in the terminal to test the program for memory leaks.	Learned that tools like Valgrind are essential for validating correct memory management in C programs.

Aşama	Görev Tanımı	Kullanılan LLM	Kullanılan Prompt (yaklaşık)	LLM Çıktısı (Özet)	Değerlendirme	Öğrenilenler
Debug / Code Review	`student.c` ve `student.h` dosyalarının incelenmesi	ChatGPT	“I’m sending you `student.c` and `student.h`. Please review them and point out possible errors or risky design decisions.”	Header–source tutarlılığı, fonksiyon prototiplerinin kapsamı, isimlendirme uyumu ve öğrenci işlemlerinin diğer modüllerle (enrollment/grade) olan ilişkileri hakkında geri bildirim verildi.	Geri bildirimler, proje dosyalarındaki gerçek kod dikkate alınarak incelendi. Header’da yalnızca public API’nin yer almasının ve öğrenci işlemlerinin modüller arası veri bütünlüğünü bozmayacak şekilde ele alınmasının doğru bir yaklaşım olduğu teyit edildi.	Kod incelemesinde yalnızca sözdizimi değil, **API tasarımı, modül sınırları ve veri bütünlüğü** gibi konuların da önemli olduğu öğrenildi.

Phase	Task Description	LLM Used	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Implementation	Designing table-like terminal output for transcript and course roster	ChatGPT	“How can I display transcript and course roster information in a clean table format in a terminal application?”	Provided example table layouts using aligned columns and separator lines suitable for terminal output.	The examples were not copied directly. Instead, the layout ideas were adapted to the terminal environment using `printf` and separator lines, then tested within the project.	Learned that in terminal-based systems, output formatting and readability are essential parts of usability and should be intentionally designed.

Phase	Task Description	used LLM	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Debug / Code Review	`student.c` ve `student.h` dosyalarının incelenmesi	ChatGPT	“I’m sending you `student.c` and `student.h`. Please review them and point out possible errors or risky design decisions.”	Feedback was provided regarding header–source consistency, the scope of function prototypes, naming consistency, and the relationship between student operations and the enrollment/grade modules.	The feedback was validated directly on the code. In particular, the consistency between prototypes in `student.h` and `student.c`, and the need to handle student deletion and update operations in a way that does not break data integrity with enrollment and grade records, were noted and the implementation was adjusted accordingly.	It was learned that code review is not limited to syntax alone, but that API design, module boundaries, and relational data integrity are also critical.

Phase	Task Description	LLM Used	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Debugging / Calculation	Reviewing the GPA calculation function	ChatGPT	“Here is my GPA calculation function. Can you review it and suggest improvements?”	Suggested calculating GPA using letter grades.	The LLM calculated GPA using letter grades; since this was inconsistent with the project definition, the function was **converted to a numeric calculation model**, tested, and then used.	Learned that calculation logic must be validated against project-specific grading definitions.

Phase	Task Description	LLM Used	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Implementation / Debugging	Designing the `addGrade` function	ChatGPT	“How should I implement an add grade function in a Student Information System?”	Provided an example function that assigns grades without checking course status.	The LLM assigned grades without checking course status; therefore, the function was modified to **check Enrolled / Completed status** before assigning a grade.	Learned that grade assignment must strictly follow state-based business rules.

Phase	Task Description	LLM Used	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Implementation / Design	Understanding the purpose of `backupAllData()`	Claude	“I don’t understand what the function `int backupAllData();` is supposed to do. How should it work in a project like this?”	Explained the general responsibility of a centralized backup function.	After understanding the purpose, the function was implemented according to the project structure.	Learned that system-level functions can span multiple modules rather than belonging to a single component.

Phase	Task Description	LLM Used	Actual Prompt (approx.)	Result / Output	Evaluation	What I Learned
Design / Testing	Evaluating the impact of Turkish characters in the code	Claude	“The assignment states that student and instructor names may contain Turkish characters. How does this affect my C code, and what should I do?”	Explained how character encoding can affect string handling and terminal output.	Based on the explanation, string handling assumptions were reviewed and terminal output was tested to ensure correct display of Turkish characters.	Learned that character encoding considerations are important when handling non-ASCII characters in C programs.

6.3 Critical Analysis

Large Language Model (LLM) tools played a supportive role during the design, implementation, testing, and debugging phases. However, the interactions showed that LLM outputs are not always fully aligned with project requirements and must always be evaluated by the developer.

Situations Where LLMs Were Most Useful:
LLMs provided helpful guidance on the design of CSV-based data structures, the scope of the `utils` module, and the general structure of header files such as `student.h`. However, these suggestions mostly offered general and idealized solutions. For example, some functions proposed for `student.h` did not sufficiently consider the student–enrollment–grade relationship within the project and were therefore incomplete in terms of data integrity.
The use of Valgrind to test for memory leaks was learned, and this knowledge was applied in the terminal to verify memory management.

Situations Where LLMs Provided Insufficient or Incorrect Suggestions:
In GPA calculation, although the project definition specified GPA as a numerical value, the LLM suggested a calculation based on letter grades. This suggestion was consciously rejected, and the calculation logic was converted to a numerical model.
Similarly, in suggestions related to the `addGrade` function, it was observed that the student's course status (Enrolled or Completed) was not checked. Since this would violate the system's business rules, the suggested logic was corrected and status checking was added.

How LLM Responses Were Improved:
No code or suggestion provided by the LLM was used directly. All outputs were evaluated by comparing them with the project documentation and the existing code structure, tested, and then used.

Lessons Learned from Collaboration with Artificial Intelligence:
During the project process, all project files were shared before asking questions to the LLM, and the conversation was conducted within this context. However, in some cases, development continued by trusting the code suggested by the LLM, and it later became apparent during the compilation phase that this code was not fully compatible with the existing structure. This delayed the detection of incorrect suggestions and led to some parts having to be rewritten from scratch. As a result, it was experienced that using LLM outputs without compiling or testing them at an early stage can lead to a loss of time. This experience highlighted the importance of using LLM suggestions by testing them in small steps and continuously validating them against the existing code structure.

7. Testing Report

7.1 Testing Strategy

In this project, the testing process was carried out in a planned and incremental manner to verify the correctness, stability, and reliability of the developed Student Information System. The tests were not limited to checking whether the system works, but also considered incorrect usage scenarios and boundary cases. Realistic CSV datasets were used in the tests to evaluate core functionalities—such as the multi-semester structure, prerequisite checks, course enrollment processes, and grade calculations—under real-world usage scenarios. In addition, verifying whether invalid or incomplete user inputs were handled correctly by the system was an important part of the testing process.

From a memory management perspective, the Valgrind tool was used to check for memory leaks and invalid memory accesses in all components that use dynamic memory.

7.2 What Needs to Be Tested

The test plan was prepared in accordance with the modular structure of the system, with separate test files created for each main component. For each module, tests were designed to cover three main objectives:

CSV File Operations (Load/Save)

- Loading of `students.csv`, `courses.csv`, `professors.csv`, `enrollments.csv`, and `grades.csv` files
- Verifying that reloading a file after saving produces the same data (round-trip testing)

Input Validation

- ID formats and ranges (e.g., `student_id` format, `professor_id` range 5000–5999)
- Email format
- Phone number format
- Semester format (YYYY-FALL/SPRING, etc.)
- Date format (YYYY-MM-DD)
- Grade fields: consistency between letter grades and numeric grades

CRUD Operations (Create/Read/Update/Delete)

- Add, search, update, and delete operations in each module
- Consistency of related records after deletion (e.g., the effect on enrollments/grades when a student is deleted)

Entity Relationships / Data Integrity

- Verifying that the student, course, and professor exist when adding an enrollment
- Verifying that the enrollment exists when adding a grade
- Ensuring that grades can be entered only for “Completed” enrollments

Memory Safety

- Proper management of dynamic arrays (`malloc`/`calloc`/`realloc`)
- Ensuring that `free` operations are performed when the program terminates
- Checking for memory leaks using Valgrind

Test Coverage Goals

When determining test coverage in this project, the focus was not on writing separate tests for every single function, but rather on the components that directly affect the correct operation of the system. The goal was to ensure that the system both produces correct results and behaves as expected in response to invalid usage.

During test planning, priority was given to functions that receive user input and are prone to errors. In particular, validations in student, course, enrollment, and grade operations were tested in more detail, as these represent the most critical parts of the system. This approach verified that cases such as invalid IDs, incorrectly formatted emails, and invalid semester information are correctly prevented by the system.

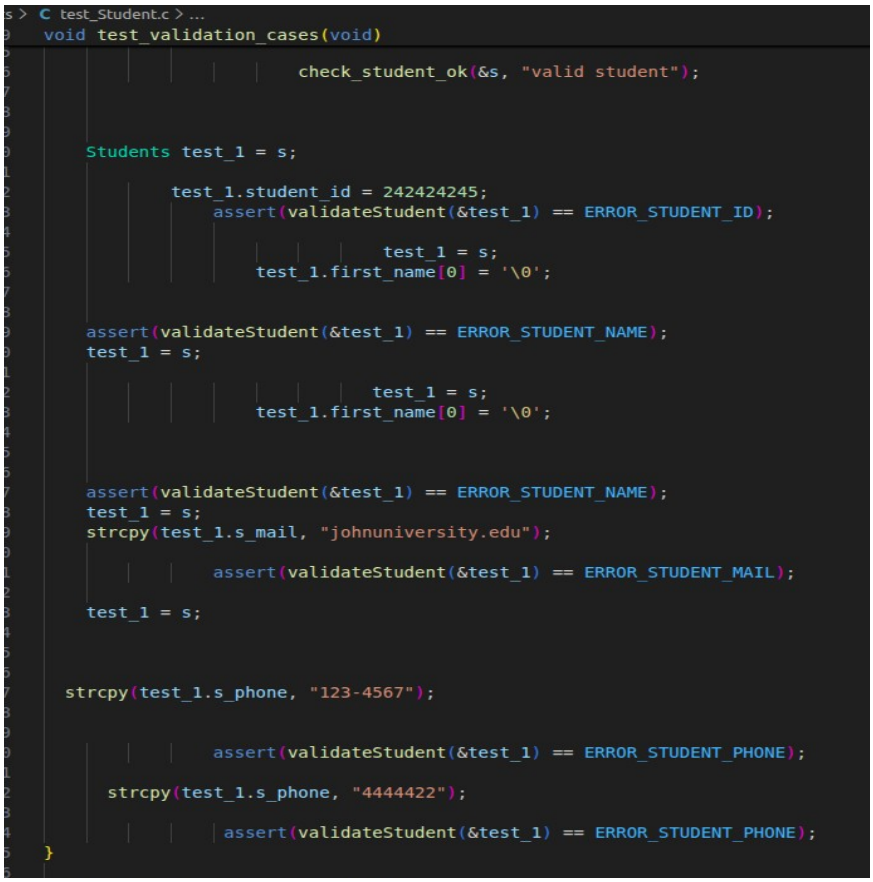
For operations involving inter-module dependencies, an integration-focused approach was adopted. Especially in enrollment and grade operations, tests were designed to verify that multiple modules work together correctly.

Finally, all components involving dynamic memory usage were checked using the Valgrind tool to ensure that the program runs without causing memory leaks.

7.3 Unit Tests

Sample Test 1: Invalid Student Information Validation

In this test, it was verified whether the information entered before adding a student was properly validated. The test started with a valid student record, and then the student ID, name, email, and phone information were made invalid one by one to check whether the validation function produced the expected error codes.



Sample Test 2: Student Add–Search–Delete and Duplicate Registration Prevention
In this test, a valid student was first added to the system. Then, an attempt was made to add the same student again using the same student ID to verify whether the system prevents duplication. Next, a second student was added, and a search by ID was performed to confirm that the correct student information was retrieved. Finally, one student was deleted, and after the deletion, searching with the same student ID returned NULL, demonstrating that the student was successfully removed.

```c
void test_add_search_delete(void)
{
        printf("-> add/search/delete\n");
                Students s1 = sample_student();
                    assert(addStudent(&s1) == 1);

                   assert(student_count == 1);
                       Students dup_id = s1;
                          strcpy(dup_id.s_mail, "diff@university.edu");
                   assert(addStudent(&dup_id) == 0);
    Students s2 = {0};
     s2.student_id = 2021002;
    strcpy(s2.first_name, "Hüseyin");
    strcpy(s2.last_name, "ARABOĞA");
                strcpy(s2.s_mail, "hüseyinn@university.edu");
                strcpy(s2.s_phone, "555-3622");
        s2.enrollment_year = 2023;
        strcpy(s2.major, "Electrical Engineering");
    check_student_ok(&s2, "student #2");
    assert(addStudent(&s2) == 1);
    assert(student_count == 2);
            Students *f = search_student_id(2021002);
            assert(f && strcmp(f->first_name, "Hüseyin") == 0);
        assert(delete_Student(2020001) == 1);
                assert(search_student_id(2020001) == NULL);
}
```

Sample Test 3: Validation of Course Search Functions (Search by ID and Course Code)

```c
void test_search_course_id(void)
{

    print_case("search_course_id");
    courses *found = search_course_id(101);
    assert(found != NULL);
    assert(found->course_id == 101);
    assert(strcmp(found->course_code, "CS101") == 0);
   assert(search_course_id(9999) == NULL);
        }

void test_search_course_code(void)
{
    print_case("search_course_code");
courses *found = search_course_code("CS201");

        assert(found != NULL);
         assert(strcmp(found->course_code, "CS201") == 0);
     assert(strcmp(found->course_name, "Data Structures") == 0);
    assert(search_course_code("CS999") == NULL);
        }
```

In this test, it was verified whether courses can be correctly found in the system using both course ID and course code. It was confirmed that the search functions return valid results for courses that

were previously added to the system, and return NULL for course IDs and codes that do not exist in the system. This test was conducted to verify the reliability of course search operations.

Sample Test 4: Validation of Course Deletion (Delete Course Test)

```
void test_delete_Course(void)
{
    print_case("delete_Course");

    int initial = course_count;
                  assert(delete_Course(101) == 1);
                  assert(course_count == initial - 1);
                  assert(search_course_id(101) == NULL);
                  assert(search_course_id(201) != NULL);
    assert(delete_Course(9999) == 0);
}
```

In this test, it was verified whether the course deletion operation works correctly. A course that had previously been added to the system was deleted, and after the deletion, it was confirmed that the course count decreased and that the deleted course could no longer be found in the system. Additionally, an attempt was made to delete a course using a course ID that does not exist in the system, and in this case, the operation was observed to fail.

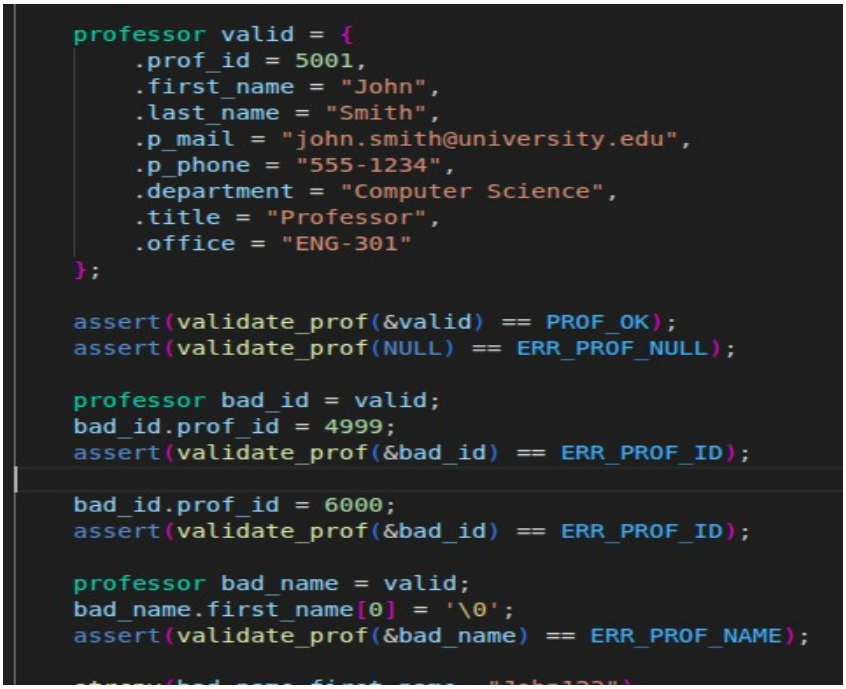
Sample Test 5: Validation of the Course Prerequisite Field

```
void test_prerequisites(void)
{
    print_case("prerequisite field");

    courses *with_p = search_course_code("CS301");

    assert(with_p != NULL);

    assert(strcmp(with_p->prerquistis, "CS201") == 0);


    courses *no_p = search_course_code("MATH101");

    assert(no_p != NULL);

    assert(no_p->prerquistis[0] == '\0');
}
```

In this test, it was verified whether prerequisite information for courses is correctly stored in the system. It was confirmed that, for a course with prerequisites, the relevant field contains the correct

course code, and for a course without prerequisites, this field is empty. This test was conducted to demonstrate that academic dependencies between courses are represented correctly.

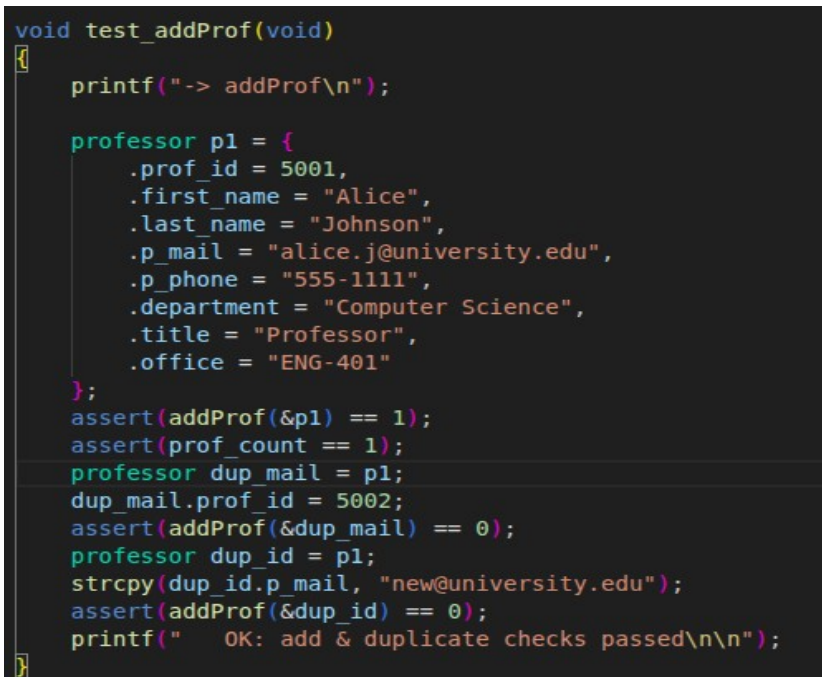
Sample Test 6: Instructor Information Validation (Professor Validation)



```
professor valid = {
    .prof_id = 5001,
    .first_name = "John",
    .last_name = "Smith",
    .p_mail = "john.smith@university.edu",
    .p_phone = "555-1234",
    .department = "Computer Science",
    .title = "Professor",
    .office = "ENG-301"
};

assert(validate_prof(&valid) == PROF_OK);
assert(validate_prof(NULL) == ERR_PROF_NULL);

professor bad_id = valid;
bad_id.prof_id = 4999;
assert(validate_prof(&bad_id) == ERR_PROF_ID);

bad_id.prof_id = 6000;
assert(validate_prof(&bad_id) == ERR_PROF_ID);

professor bad_name = valid;
bad_name.first_name[0] = '\0';
assert(validate_prof(&bad_name) == ERR_PROF_NAME);
```

In this test, the validation of instructor information was targeted. It was verified that the operation succeeds for a valid record, that the instructor ID must be within the range 5000–5999, and that for IDs outside this range, empty name fields, and NULL inputs, the function returns the appropriate error codes.

Sample Test 7: Instructor Add and Duplicate Registration Checks (Add & Duplicate Test)



```
void test_addProf(void)
{
    printf("-> addProf\n");

    professor p1 = {
        .prof_id = 5001,
        .first_name = "Alice",
        .last_name = "Johnson",
        .p_mail = "alice.j@university.edu",
        .p_phone = "555-1111",
        .department = "Computer Science",
        .title = "Professor",
        .office = "ENG-401"
    };
    assert(addProf(&p1) == 1);
    assert(prof_count == 1);
    professor dup_mail = p1;
    dup_mail.prof_id = 5002;
    assert(addProf(&dup_mail) == 0);
    professor dup_id = p1;
    strcpy(dup_id.p_mail, "new@university.edu");
    assert(addProf(&dup_id) == 0);
    printf("   OK: add & duplicate checks passed\n\n");
}
```

In this test, it was verified whether the instructor addition operation works correctly and whether duplicate registrations are prevented. It was confirmed that attempts to add an instructor using the same email address or the same instructor ID are rejected by the system.

Sample Test 8: Validation of Instructor Deletion (Delete Professor Test)

```c
void test_delete_prof(void)
{
    printf("-> delete_prof\n");

    int before = prof_count;
    assert(delete_prof(5001) == 1);
    assert(prof_count == before - 1);
    assert(search_prof_id(5001) == NULL);

    assert(delete_prof(9999) == 0);

    printf("   OK: delete behavior verified\n\n");
}

void test_save_and_load(void)
{
    printf("-> save / load\n");

    professor p = {
        .prof_id = 5004,
        .first_name = "David",
        .last_name = "Wilson",
        .p_mail = "david.w@university.edu",
        .p_phone = "555-4444",
        .department = "Engineering",
        .title = "Professor",
        .office = "ENG-501"
    };
    addProf(&p);

    assert(save_Prof("test_professors.csv") == 1);

    int saved = prof_count;
    freeProf();
    assert(arrProf == NULL);
    assert(prof_count == 0);

    init_Prof();
    assert(loadPorf("test_professors.csv") == saved);

    professor *loaded = search_prof_id(5004);
    assert(loaded != NULL);
    assert(strcmp(loaded->first_name, "David") == 0);

    printf("   OK: save/load cycle successful\n\n");
}

void test_prof_Error_MSG(void)
```

In this test, it was verified whether the instructor deletion operation works correctly. It was confirmed that an existing instructor was deleted, the counter value was updated accordingly, and that deletion attempts using an ID that does not exist in the system were rejected.

Sample Test 9: Enrollment Information Validation

```
enrollment e = make_enrollment(
    1, 2024001, 101, 5500,
    "2024-FALL", "2024-09-01", "Enrolled"
);

assert(validateEnrollment(NULL) == ENROLLMENT_NULL);
assert(validateEnrollment(&e) == ENROLLMENT_OK);

e.enrollment_id = 0;
assert(validateEnrollment(&e) == ENROLLMENT_ID);
e.enrollment_id = 1;

e.student_id = 999999;
assert(validateEnrollment(&e) == ENROLLMENT_STUDENT_ID);
e.student_id = 2025001;
assert(validateEnrollment(&e) == ENROLLMENT_STUDENT_ID);
e.student_id = 2024001;
```

In this test, it was verified whether enrollment information is validated before being added to the system. It was confirmed that the function returns appropriate error codes for cases such as NULL input, invalid enrollment ID, invalid student/course/instructor IDs, and incorrect semester format.

Sample Test 10: Validation of Grade Search and Deletion (Grade Search & Delete)
In this test, it was verified that grade records can be correctly searched by grade ID and enrollment ID, and that deletion by enrollment ID works correctly. After the deletion operation, it was confirmed that the grade count decreased and that the related record could no longer be found.

```
void test_search_and_delete(void)
{
    printf("-> search & delete\n");

    reset_grades();

    arrGrades[0] = make_grade(10001, 1, 2020001, 101, "A", 95.0, "2023-FALL");
    grade_count = 1;

    assert(search_grade_id(10001) != NULL);
    assert(seachGrade_byEnrollmentid(1) != NULL);
    assert(seachGrade_byEnrollmentid(999) == NULL);

    assert(deleteGradeBy_EnrollmentId(1) == 1);
    assert(grade_count == 0);
    assert(seachGrade_byEnrollmentid(1) == NULL);

    assert(deleteGradeBy_EnrollmentId(999) == 0);
```

7.5 Test Results

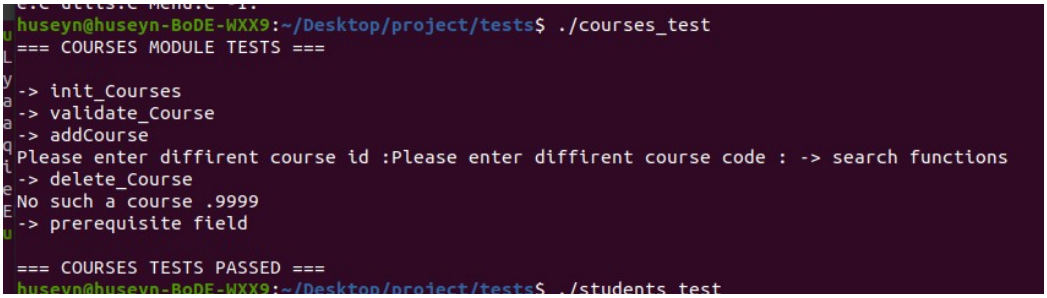
In this section, the prepared unit and integration tests were executed to verify whether the system behaves as expected. As a result of the test executions, it was observed that both valid scenarios and

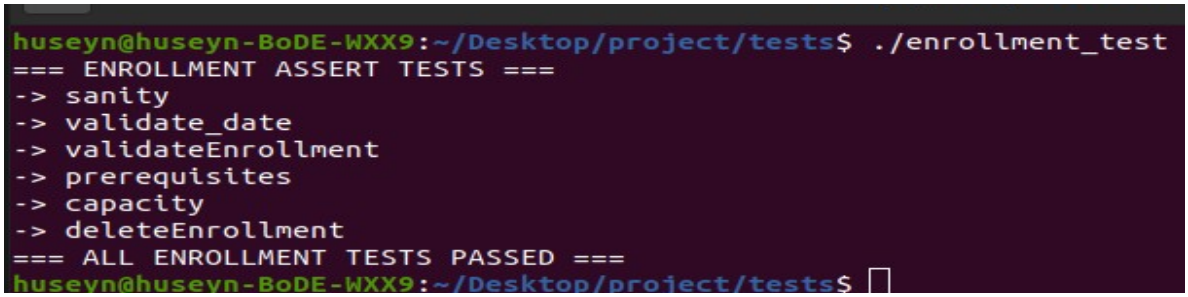
invalid inputs were handled correctly for each module. No crashes or unexpected behavior were encountered during test execution.

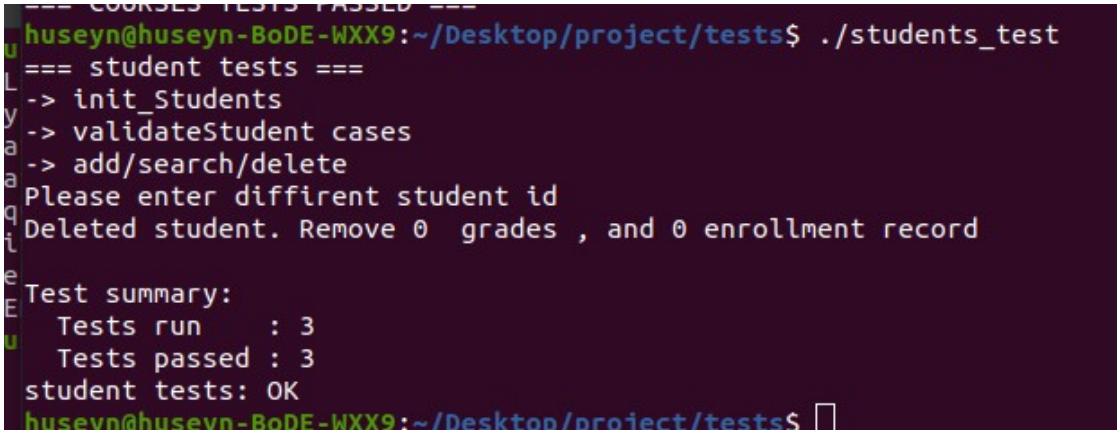
When the test results are evaluated on a module basis:

- **Student tests:** passed (validation, add/search/delete, save/load)
- **Course tests:** passed (add/uniqueness, search, delete, prerequisite field)
- **Professor tests:** passed (validation, add duplicate, delete, save/load)
- **Enrollment tests:** passed (validateEnrollment, etc.)
- **Grade tests:** passed (search/delete, save/load)

Some visuals of the test terminal results.

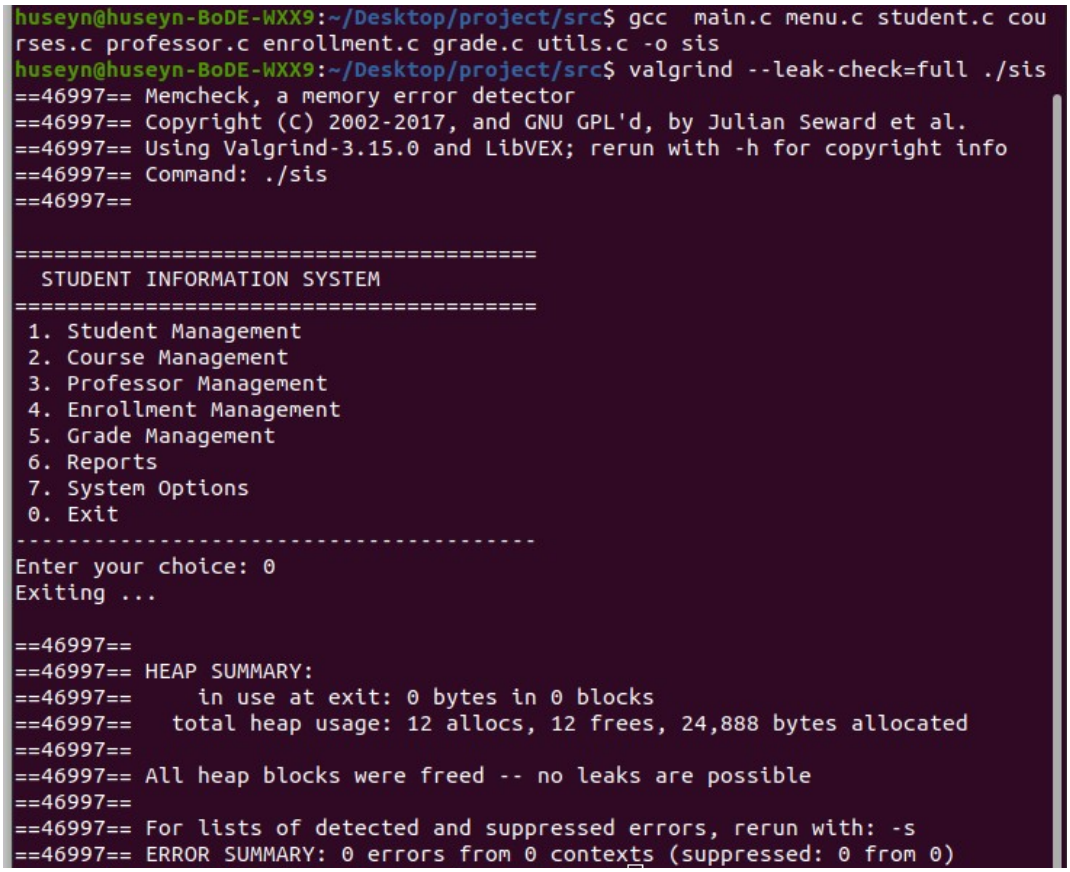






7.6 Memory Leak Tests (Valgrind)

Since dynamic memory is used in this project, the program was tested using the Valgrind tool for memory leaks and invalid memory accesses. During testing, the system was run under Valgrind, and scenarios such as data loading, add–delete operations, enrollment–grade operations, and program exit were executed. The objective was to ensure that all dynamically allocated memory areas were released when the program terminated.



7.7 How to Run Tests

In this project, unit and integration tests are executed using test files prepared for each module. Before running the tests, it must be ensured that the required source files (`.c` and `.h`) are present in the project directory. The tests are carried out by compiling and running them via the terminal. Example compilation.

gcc -o grade_test3 test_grade.c grade.c enrollment.c student.c courses.c professor.c utils.c menu.c -I.

gcc -o enrollment_test test_enrollment.c enrollment.c student.c courses.c professor.c grade.c utils.c menu.c -I.

8. Compilation and Execution Instructions

This project was developed as a multi-file C program and is compiled using the GNU Compiler Collection (gcc). The compilation process is carried out via the `build.sh` script located in the project root directory.

To compile the project, the following steps should be followed:

1. Navigate to the project directory:
   ```
   $ cd project
   ```
2. Grant execution permission to the build script:
   ```
   $ chmod +x build.sh
   ```
3. Compile the main system:
   ```
   $ ./build.sh
   ```

As a result of the compilation process, all source files are compiled separately, the required object files are generated, and an executable main program is produced.

After the compilation is completed, the system is run with the following command:
```
$ ./sis
```

When the program is executed, the main menu is displayed. For the system to function correctly, the CSV files must first be loaded into memory using the **System Options → Load All Data** option. Otherwise, the system will operate with empty data.

project/

├── src/

│ ├── main.c

│ ├── menu.c

│ ├── menu.h

│ ├── student.c

│ ├── student.h

│ ├── courses.c

│ ├── courses.h

│ ├── professor.c

│ ├── professor.h

│ ├── enrollment.c

```
│   ├── enrollment.h
│   ├── grade.c
│   ├── grade.h
│   ├── utils.c
│   └── utils.h
├── tests/
│   ├── grade_test.c
│   ├── student_test.c
│   ├── course_test.c
│   └── test_main.c
├── data/
│   ├── students.csv
│   ├── courses.csv
│   ├── professors.csv
│   ├── enrollments.csv
│   └── grades.csv
├── build.sh
└── README.md
```

10. Conclusion and Reflection

Within the scope of this project, a multi-file and menu-driven Student Information System was developed using the C programming language. Throughout the project, student, course, instructor, enrollment, and grade operations were brought together under a single system, and persistent data storage was achieved using CSV files.

The project process was challenging but educational for me. Since I had not previously developed such a large and multi-modular C project, establishing connections between files and ensuring that the code compiled smoothly as a whole took a significant amount of time. Various errors related to dynamic memory usage and pointer management were encountered, and these were resolved through trial and error and by analyzing error messages.

During the input validation phase, it became clear how important checking user-provided data is for system stability. Memory leaks were examined using the Valgrind tool, and the necessary adjustments were made.

During the project development process, support was received from Large Language Models (LLMs). LLMs were particularly useful for idea generation, reviewing example code, and debugging. However, in some cases, the provided code could not be used directly and had to be

rewritten because it caused compilation errors or logical problems. This demonstrated that LLM outputs must be questioned and always tested.

In conclusion, this project demonstrated how much attention and planning are required to develop large-scale software using the C language. Through this project, significant experience was gained in modular programming, memory management, file operations, and test writing. It is believed that these experiences will form a solid foundation for more comprehensive software projects in the future.