Unit 5

Industry Use Cases and Project-Based Application

Learning Objectives

- To apply structured programming principles in practical C projects
- To design and develop menu-driven applications
- To implement input validation and real-time user interaction
- To use file I/O for logs and persistent data storage
- To structure code for modularity and maintainability
- To perform manual testing using test cases
- To collaborate using version control with Git

Structure

5.1 Introduction to Project-Based Development

Project-based learning in C is essential to bridge the gap between theoretical knowledge and its practical application. It encourages students to apply structured programming, file handling, modularity, and debugging in real-life scenarios by building mini projects from scratch.

5.1.1 Importance of Practical Application in C

C programming forms the backbone of many systems-level and embedded applications. By working on hands-on projects, students:

- Reinforce core concepts like control structures, arrays, pointers, and file operations
- Understand how C can be used in solving **real-world problems**
- Learn to **debug**, **test**, and **optimize** programs in dynamic environments
- Gain experience in **problem-solving** and **algorithmic thinking**

Project-based development makes learning **engaging** and improves **job readiness**.

5.1.2 Characteristics of Real-World C Projects

Real-world C projects are:

- **Modular**: Broken into smaller, manageable components using functions and headers
- **Well-documented**: Use comments, documentation, and version history
- **Error-resilient**: Handle invalid inputs, file errors, and edge cases
- **File-based**: Use persistent storage through file I/O for user or system data
- **Testable**: Can be verified through unit tests or manual test cases

Examples include:

- **Student Record System**
- **ATM Simulation**
- **Library Management System**
- **Simple Shell or Command Interpreter**

5.1.3 Planning and Scoping a Mini Project

Proper planning is crucial for project success. A typical planning process includes:

1. **Selecting a theme**: E.g., banking, education, inventory
2. **Defining features**: Login system, data storage, reporting
3. **Identifying inputs and outputs**: What will the user enter? What will the program return?

4. **Designing modules**: Divide functionality into logical units
5. **Deciding file structure**: Use .h and .c files for clarity

A sample scope:
Build a menu-driven ATM interface that handles deposits, withdrawals, and balance checks with file-based transaction history.

5.1.4 Time-Bound Development Strategies

To complete a project within a tight schedule (e.g., 1–2 weeks), students should:

- Break the project into **daily tasks**
- Follow **incremental development** (write, test, debug small parts)
- Use **version control (Git)** to track changes
- Apply **agile principles** — work in short sprints
- Avoid overcomplication; prioritize core functionality first

Example timeline for a 7-day mini project:

Day	Task
1	Define problem and plan features
2–3	Code core logic (functions, menu, file I/O)
4	Add validations and error handling
5	Modularize code and test features
6	Integrate version control, finalize documentation
7	Run final tests and prepare submission

5.1.5 Deliverables and Evaluation Criteria

A project submission typically includes:

Deliverables:

- Source code files (.c, .h)
- Makefile (for automated builds)
- README (project description, usage instructions)
- Input/output sample data files
- Final report (design, features, known issues)

Evaluation Criteria:

Criteria	Marks
Correctness and Functionality	30
Code Quality and Structure	20
Use of File Handling / Pointers	15

Criteria	Marks
Use of Modular Programming	15
Documentation and Git Usage	10
Timely Submission and Testing	10

Projects are assessed on **working logic, code style, maintainability, and documentation**.

5.2 Menu-Driven Program Design

Menu-driven programs provide a structured, interactive way for users to perform multiple operations through a **simple menu interface**. This design is common in many CLI (Command Line Interface) systems such as ATMs, grading systems, or inventory management tools.

5.2.1 Designing User Menus and Interfaces

A **user menu** lists available operations, typically numbered for ease of input. Designing an effective user menu involves:

- **Clarity**: Easy-to-understand options
- **Consistency**: Repeated layout and prompt formatting
- **Validation**: Input must be checked for correctness

Example Menu:

```
===== Student Management System =====
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Record
5. Delete Record
6. Exit
Enter your choice:
```

User choices are collected using scanf() or fgets() and then controlled using **switch-case** logic.

5.2.2 switch-case for Menu Control Flow

The **switch-case statement** is ideal for handling menu-based input.

Example:

```
int choice;
scanf("%d", &choice);

switch (choice) {
    case 1: addStudent(); break;
    case 2: displayAll(); break;
    case 3: searchByID(); break;
    case 4: updateRecord(); break;
    case 5: deleteRecord(); break;
```

```
    case 6: printf("Exiting..."); break;
    default: printf("Invalid choice. Try again.");
}
```

- Each case maps to a specific function
- default handles invalid input
- Combined with a loop to **run continuously** until exit is selected

5.2.3 Case Study: ATM Interface Project
Objective: Simulate ATM operations with file-based balance handling.
Menu Options:
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. View Transaction History
5. Exit

Key Functions:

- checkBalance() — reads from file
- deposit() — adds amount to balance and writes to file
- withdraw() — checks and deducts if funds are sufficient
- logTransaction() — appends operation to a transaction file

Concepts Applied:

- File I/O (fopen, fread, fwrite)
- User input validation
- Modularity and error handling

5.2.4 Case Study: Student Grade Management System
Objective: Manage student records including name, roll number, marks, and grade.
Menu Options:
1. Add Student Record
2. Display Records
3. Calculate Average Marks
4. Find Topper
5. Exit

Key Features:

- Structure array to store student details
- File operations for persistent storage
- Calculation logic for average and maximum

Sample Structure:

```
struct Student {
    int roll;
    char name[50];
    float marks;
};
```

Operations:

- addStudent() — add to file
- displayStudents() — read from file
- findTopper() — logic to compare marks

5.2.5 Exit Conditions and Loop Integration

Menu-driven programs usually run inside a loop to allow **multiple user actions** until the user decides to exit.

Example Structure:

```
int choice;
do {
    displayMenu();
    scanf("%d", &choice);
    switch(choice) {
        case 1: ...; break;
        case 2: ...; break;
        ...
        case N: printf("Exiting..."); break;
        default: printf("Invalid input");
    }
} while(choice != N);
```

Best Practices:

- Display the menu in every loop
- Use clear prompts and confirmations

5.3 Real-Time Input Handling and Validation

Effective input handling is **crucial** in interactive programs. It ensures that the application behaves as expected even when users enter invalid, unexpected, or harmful data. Input validation and safe handling prevent runtime errors, infinite loops, or crashes — especially in file-based or menu-driven systems.

5.3.1 Accepting and Processing User Input

C provides several standard input functions:

Function	Description
scanf()	Reads formatted input (numbers, strings, etc.)
gets()	Reads a line (discouraged due to safety risks)
fgets()	Reads a string safely (preferred over gets())

Function Description

getchar() Reads a single character

Example:

```
int age;
printf("Enter your age: ");
scanf("%d", &age);
For strings:
char name[50];
printf("Enter name: ");
fgets(name, 50, stdin);
```

5.3.2 Input Validation Techniques

Validation ensures input is of **correct type, range, and format**. Techniques include:

1. Range checking:

```
if (age < 0 || age > 120) {
    printf("Invalid age\n");
}
```

2. Character validation (for strings or menus):

```
if (choice < 1 || choice > 5) {
    printf("Please enter a valid menu option (1-5)\n");
}
```

3. Using fgets() + sscanf() for safer parsing:

```
char buffer[100];
int value;
fgets(buffer, sizeof(buffer), stdin);
if (sscanf(buffer, "%d", &value) != 1) {
    printf("Invalid number!\n");
}
```

This approach protects against buffer overflows and invalid format input.

5.3.3 Handling Invalid or Unexpected Input

Unvalidated input can crash programs or create security risks. Defensive programming includes:

- **Checking return values** of scanf(), fgets(), etc.
- Re-prompting users until valid input is received
- Limiting input size to avoid overflow

Example:

```
int age;
while (1) {
    printf("Enter age (0–120): ");
    if (scanf("%d", &age) == 1 && age >= 0 && age <= 120) {
        break;
    } else {
        printf("Invalid input. Try again.\n");
        while (getchar() != '\n'); // flush input buffer
```

```
    }
}
```

5.3.4 Dynamic Input Length Management

When the size of user input is unknown or variable (e.g., full names, addresses), use **dynamic memory**:

```
char *input = malloc(100);
printf("Enter string: ");
fgets(input, 100, stdin);
```

For truly variable input:

- Use getline() (POSIX) or write custom functions to resize memory on the fly
- Ensure null-termination and proper memory deallocation

Tip: Always check allocation success with if (input == NULL)

5.3.5 Buffer Flushing and Safety Considerations

Problems with scanf():

Mixing scanf() and fgets() can cause leftover characters (\n) in the buffer, leading to skipped input.

Solution: Use getchar() or clear input buffer manually:

```
int ch;
while ((ch = getchar()) != '\n' && ch != EOF);
```

Avoid:

- gets() (unsafe and removed in C11)
- Using %s in scanf() without width limit

Use:

```
scanf("%49s", name); // safe string read
```

Best Practices:

- Always set buffer limits
- Use safer alternatives like fgets() + sscanf()
- Flush the input buffer where necessary
- Validate inputs before using them

5.4 File-Based Logging Systems

Logging is a technique for **recording runtime information** such as inputs, operations, errors, or events into a file. It is widely used for **debugging, auditing, and analyzing** the flow of an application over time.

5.4.1 Implementing Basic Logging with fprintf()

The fprintf() function allows formatted writing to a file — making it ideal for logs.

Syntax:

```
FILE *logFile = fopen("log.txt", "a");
```

```
fprintf(logFile, "User logged in\n");
fclose(logFile);
```

- Open the file in **append mode ("a")** so new logs are added at the end
- Log files can store **text entries** describing system events or application states

Example:

```
void logEvent(const char* event) {
    FILE *fp = fopen("event_log.txt", "a");
    if (fp != NULL) {
        fprintf(fp, "EVENT: %s\n", event);
        fclose(fp);
    }
}
```

5.4.2 Timestamps and Event Tracking

Timestamps allow you to track **when** events occur. Use the time.h library to get the current date and time.

Example with Timestamp:

```
#include <time.h>

void logWithTimestamp(const char* message) {
    FILE *fp = fopen("log.txt", "a");
    time_t now = time(NULL);
    char *timeStr = ctime(&now);    // includes newline

    if (fp != NULL) {
        timeStr[strlen(timeStr)-1] = '\0'; // remove newline
        fprintf(fp, "[%s] %s\n", timeStr, message);
        fclose(fp);
    }
}
```

This helps track:

- User login times
- Operation sequences
- File access or modification logs

5.4.3 Error and Transaction Logs

Separate logs can be maintained for:

- **Error logs**: Store system or runtime failures
- **Transaction logs**: Record operations like deposits, withdrawals, edits, etc.

Error Log Example:

```
void logError(const char* error) {
    FILE *fp = fopen("error_log.txt", "a");
```

```
    if (fp != NULL) {
        fprintf(fp, "ERROR: %s\n", error);
        fclose(fp);
    }
}
```

In real projects:

- Errors should include **function name**, **timestamp**, and **error details**
- Transactions should be **structured** for readability

5.4.4 Managing Log File Size and Rotation (Introductory)

Log rotation refers to archiving or truncating logs to prevent unlimited growth. While advanced rotation is handled by external tools (like logrotate in Linux), you can introduce basic techniques in C:

Approach:

- Use ftell() to get file size
- Rename or truncate file when size exceeds a threshold

```
FILE *fp = fopen("log.txt", "a+");
fseek(fp, 0, SEEK_END);
long size = ftell(fp);

if (size > 10000) { // 10 KB limit
    fclose(fp);
    rename("log.txt", "log_old.txt");
    fp = fopen("log.txt", "w"); // start fresh
}
```

5.4.5 Analyzing Logs for Debugging

Logs help in **post-mortem debugging** — understanding how a problem occurred **after** it happens.

Logs can reveal:

- Which inputs led to a crash
- Sequence of function calls
- Missing or corrupted data paths
- File access failures
- Time between operations (performance)

Tips for Debugging Logs:

- Use **clear tags** like INFO, ERROR, WARNING
- Use **consistent formatting**
- Always **timestamp** important events

- Keep logs **short but meaningful**

Example Log Output:
[2025-10-12 10:31:01] INFO: User logged in
[2025-10-12 10:31:04] ERROR: Invalid account number
[2025-10-12 10:31:06] TRANSACTION: Withdrawal Rs. 500

5.5 Code Structuring for Modularity and Readability

Modular and readable code is **easier to debug, maintain, and extend**. It supports collaboration and reduces the risk of introducing errors during updates. Good structuring involves splitting logic across files, using functions and headers, maintaining naming consistency, and organizing the file system.

5.5.1 Splitting Code into Functions and Modules

Dividing large code blocks into **functions** improves modularity and reuse.
Benefits:

- Functions can be **tested independently**
- Avoids **code repetition**
- Easier to understand and modify logic

Example:
Instead of writing:

```
int main() {
    // login logic
    // transaction logic
    // logout logic
}
```

Split into:

```
void login();
void performTransaction();
void logout();

int main() {
    login();
    performTransaction();
    logout();
}
```

For larger projects, group functions by **feature or functionality** into separate modules/files:

- login.c
- transaction.c
- utility.c

5.5.2 Use of Header Files for Declarations

Header files (.h) contain:

- **Function declarations (prototypes)**
- **Constants / macros**
- **Global structure definitions**

Why use headers?

- Promote **code reuse**
- Prevent **circular dependencies**
- Separate **interface** from **implementation**

Example:

login.h

```
void login();
```

main.c

```
#include "login.h"
```

Use #include to bring function declarations into other files. Header files **must be protected** using **include guards** to avoid double inclusion.

```
#ifndef LOGIN_H
#define LOGIN_H
// content
#endif
```

5.5.3 Naming Conventions and Commenting

Clear naming and documentation are essential for readability.

Naming Best Practices:

- Use **meaningful names**: calculateGrade() over cg()
- Use **lower_snake_case** or **camelCase** consistently
- Constants/macros in **UPPERCASE**

Commenting Best Practices:

- Write **header comments** for each file and function
- Use **inline comments** to explain logic blocks
- Avoid over-commenting obvious code

```
// Calculates GPA based on marks array
float calculateGPA(int marks[], int n) {
    ...
}
```

5.5.4 Folder Structure for Source, Headers, and Output

Maintain a clean project directory layout:

project/

```
|
├── src/                    # All .c files (implementation)
|       ├── main.c
|       ├── login.c
|       ├── transaction.c
|
├── include/                # All .h files (headers)
|       ├── login.h
|       ├── transaction.h
|
├── build/                  # All compiled object/output files
|
├── logs/                   # Log files
|
├── Makefile
├── README.md
```

This makes it easy to:

- Locate relevant files
- Build using Makefile
- Share or version control the project

5.5.5 Managing Large Codebases Efficiently

As projects grow:

- Use **modular functions grouped by functionality**
- Apply **version control** (e.g., Git) to track changes
- Use **build automation** (Makefile) to compile large sets of files
- Write **unit tests** for major modules
- Maintain clear **documentation** (README, in-code comments)

Maintainability Tips:

- Avoid global variables
- Separate logic, I/O, and data storage
- Keep functions short and single-purpose
- Document assumptions and dependencies

5.6 Application Testing Using Manual Test Cases

Manual testing involves **running a program with predefined inputs** and checking if it behaves as expected. It helps identify logical errors, boundary issues, or unexpected behavior before deploying or submitting a project.

5.6.1 Importance of Testing in Software Projects

Testing ensures:

- The program works as intended
- Bugs are caught before deployment
- Edge cases are handled
- Code is reliable, secure, and user-friendly

In academic or industry settings, **untested code is treated as unreliable** — testing adds credibility and helps avoid runtime failures.

5.6.2 Creating Test Plans and Test Case Tables

A **test plan** defines:

- What needs to be tested (features/functions)
- How the testing will be done
- What inputs will be used
- What results are expected

A **test case table** summarizes all this in a structured format.

Example Test Case Table for Student Grade System:

Test Case ID	Function Tested	Input	Expected Output	Actual Output	Status
TC01	Add Student	Name: John, Marks: 85	Record Added	Record Added	Pass
TC02	Calculate Grade	Marks: 92	Grade: A	Grade: A	Pass
TC03	Search Student	Roll No: 999	Record Not Found	Record Not Found	Pass
TC04	Delete Student	Roll No: 1	Record Deleted	Error Msg	Fail

5.6.3 Testing Boundary and Edge Cases

Edge cases often cause unexpected behavior if not handled. Common edge scenarios:

- **Zero or negative values** (e.g., deposit -100)
- **Maximum/minimum values** (e.g., int overflow)
- **Empty or null input** (e.g., empty name field)

- **Invalid characters or formats** (e.g., letters in numeric fields)

Example:
If a student's marks are 0 or 100, ensure that:

- Grade calculation still works
- The program does not crash or misbehave

Boundary Testing Checklist:

- Test just below, at, and just above limits (e.g., marks 59, 60, 61)
- Validate menu choices (e.g., input 0, 1, 6, or non-numeric)

5.6.4 Input/Output Validation and Comparison

Manually comparing:

- **Expected outputs** with **actual outputs**
- Ensures correctness of formatting, calculation, flow

Use techniques like:

- Printing debug information
- Logging I/O to files
- Checking printed messages for clarity

Example:

- Input: Name = "", Marks = -10
- Expected Output: "Invalid input"
- Check if actual output matches

Pro Tip: Use logging to capture outputs for review

5.6.5 Logging Test Results

Record test outcomes in a **log file** using fprintf():

```
FILE *log = fopen("test_log.txt", "a");
if (log != NULL) {
    fprintf(log, "Test Case: TC03 | Input: Roll = 999 | Output: Not Found | Status: PASS\n");
    fclose(log);
}
```

You can log:

- Inputs used
- Expected vs actual outputs
- Whether the test passed or failed
- Timestamp (optional)

Use structured formats so logs can be **analyzed or searched** easily.
Example Log Entry:
[2025-10-13 10:00:15] TC01: Add Student | Input: John, 85 | Output: Record Added | Status: PASS

5.7 Git Integration and Version Control

Version control systems like Git allow developers to **track, manage, and collaborate** on code systematically. For C projects, especially those involving teamwork or iterative development, Git ensures that changes are traceable, recoverable, and organized.

5.7.1 Initializing a Git Repository for the Project

To start using Git in your C project:
Steps:

1. Navigate to your project directory:
2. cd path/to/project
3. Initialize Git:
4. git init

This creates a hidden .git folder, which tracks all future changes.

3. Add your project files:
4. git add .
5. Make your first commit:
6. git commit -m "Initial commit - project setup"

Now your project is under version control.

5.7.2 Tracking Changes and Project Snapshots

Git tracks all changes across your files over time.

- To check file status:
- git status
- To view differences before committing:
- git diff
- Each commit is a **snapshot** of the code:
- git commit -m "Fixed input validation in login()"
- To view history:
- git log

This helps:

- Revert to previous versions
- Understand project evolution
- Identify when bugs were introduced

5.7.3 Commit Messages and Documentation

Clear and consistent commit messages are essential for understanding changes.

Good Practices:

- Use **present tense**: "Add login validation"
- Be **specific**: "Fix null pointer bug in transaction.c"
- Group related changes into one commit

Bad Message Example:

Update

Good Message Example:

Refactor calculateGPA() for edge case handling

Use README.md and code comments alongside commits to explain **why** a change was made, not just what was changed.

5.7.4 Creating Branches for Features or Fixes

Branches allow working on different parts of the project without affecting the main version (main or master).

Example:

git branch feature-logging

git checkout feature-logging

Now you can add a logging feature separately.

Merge it back when complete:

git checkout main

git merge feature-logging

This allows:

- **Parallel development**
- **Bug isolation**
- **Safer experimentation**

5.7.5 Pushing to Remote Repositories (GitHub/GitLab)

To collaborate or back up your project remotely:

1. Create a repository on **GitHub** or **GitLab**
2. Link your local repo:

git remote add origin https://github.com/username/project.git

3. Push your code:

git push -u origin main
From now on, use:
git push
You can also:

- **Clone** a project:
- git clone https://github.com/username/project.git
- **Pull** updates from teammates:

git pull

5.8 Mini Project Execution

A mini project provides an opportunity to **apply theoretical knowledge in a practical scenario**. It involves planning, implementing, testing, and presenting a working software solution — often simulating real-world application workflows.

5.8.1 Project Planning and Task Breakdown

Planning is the foundation of any successful project. This stage includes:
Key Steps:

- **Define the objective**: What is the problem the project solves?
- **Decide the features**: Login, database handling, search, reports, etc.
- **Create a timeline**: Use weeks or sessions for tracking progress
- **Identify roles** (for team projects): Who will code, test, document?

Example Task Breakdown (ATM Project):

Task	Description	Owner	Deadline
Design Menu	Create ATM interface	Student A	Day 1
File I/O	Log transactions	Student B	Day 2
Testing	Run test cases	Student C	Day 4

Break large tasks into **functions or modules**:

- login(), withdraw(), deposit(), generateReceipt()

Use a checklist or Kanban board for progress tracking.

5.8.2 Implementation with Milestone Tracking

Start coding in phases with clear milestones:

1. **Prototype** core features (basic functionality)
2. **Expand** with additional logic and validation
3. **Polish** UI, file I/O, and structure

Milestone Examples:

- **Milestone 1**: Menu and navigation done

- **Milestone 2**: Core functionality implemented
- **Milestone 3**: Logging and file handling added
- **Milestone 4**: Fully tested and documented

Use Git for version control and tags like:
git tag v1.0-milestone1
This helps track **progress over time** and rollback if needed.

5.8.3 Final Code Review and Refactoring
Before submission:

- **Review all code for consistency**
- **Remove redundant code**
- **Refactor** long functions into smaller ones

Code Review Checklist:

- Are variables named clearly?
- Are functions reusable and short?
- Are edge cases handled?
- Are there unnecessary goto or deep nested loops?
- Are meaningful comments added?

Run through **peer reviews** or **self-review** with a checklist.

5.8.4 Test and Debugging Phase
Ensure the project:

- Compiles with **no warnings** (gcc -Wall)
- Handles all **valid and invalid inputs**
- Has been tested using a **test case table**
- Has proper **error messages and recovery**

Use:

- gdb for debugging segmentation faults
- Logs for tracking values during execution
- valgrind (optional) for memory issues

Write **at least 5–10 test cases** covering:

- Normal operations
- Invalid inputs

- Boundary values
- File corruption or missing file scenarios

5.8.5 Project Presentation and Submission Guidelines

Prepare the final version for submission and presentation.

Deliverables:

- **Source code** (well-structured folders)
- **Executable file** (compiled .exe or Linux binary)
- **README** with:
    - Project title and description
    - How to compile and run
    - Features implemented
    - Known issues or limitations
- **Documentation file (PDF)** with:
    - Problem statement
    - Code architecture diagram
    - Screenshot of outputs
    - Sample test cases

Presentation Tips:

- Keep demo short (3–5 minutes)
- Show real-time input/output
- Highlight key features
- Explain logic with flowcharts or pseudocode

5.9 Summary

This module emphasized the real-world application of C programming through project-based learning. Students explored the importance of planning, structuring, and managing C programs in a modular, readable, and testable format. Concepts such as menu-driven design, real-time input handling, logging, and Git-based version control were applied in the context of mini projects. By engaging in code implementation, milestone tracking, manual testing, and collaborative workflows, students gained practical experience in deploying structured and functional software solutions aligned with industry standards.

5.10 Keywords

- **Menu-driven programming** – User-driven execution model using options and switch-case statements
- **Input validation** – Ensuring correctness and safety of user inputs
- **Logging** – Recording events and errors during program execution to a file
- **Modular programming** – Dividing code into reusable functions and source files
- **Git** – Distributed version control system for tracking code changes
- **Commit** – Saving a snapshot of changes in Git with a message
- **Branching** – Creating isolated development environments in Git
- **Milestone** – A defined project stage or goal in software development
- **Refactoring** – Improving code without changing its behavior
- **Test case** – A predefined set of inputs and expected outputs used for verification

5.11 Self-Assessment Questions (Subjective & Case-Based)

Subjective Questions

1. Explain the benefits of using a menu-driven structure in a C project.
2. Describe how input validation improves the reliability of a program.
3. What are the advantages of using version control systems like Git in collaborative development?
4. How can a developer manage code modularity in large-scale projects?
5. Discuss the importance of manual testing and how it differs from automated testing.

Case-Based Questions

1. You are developing a mini project for a **Library Management System**. How would you structure your code to ensure modularity and maintainability? What are the key milestones you would track?
2. During testing, your **ATM simulation program** crashes when the user inputs special characters in the PIN entry. Identify how this could be resolved using input validation techniques.

3. A teammate accidentally deletes a crucial function from your source code. How can Git help recover this, and what commit strategy would prevent such issues in the future?

5.12 Case Study

Case Study: Student Grade Management System (SGMS)

A team of students developed a Student Grade Management System using the C programming language. The system allowed users to:

- Add new student records
- Input and update marks
- Generate grades
- Store all data in a text file
- Maintain an activity log of operations

The team used the following practices:

- Created a **menu-driven interface** using switch-case
- Applied **input validation** to avoid incorrect mark entries
- Logged all operations (add, update, delete) using fprintf() with timestamps
- Maintained **separate files** for declarations (.h files), logic (.c), and logs (log.txt)
- Tracked their progress with Git branches (e.g., feature-logging, bugfix-grade-calc)
- Followed a **milestone plan** with weekly targets for features, testing, and documentation

The project was evaluated based on functionality, code readability, testing completeness, and Git history.

Outcome: The structured approach helped the team complete the project on time with minimal bugs and high code quality.

5.13 References

1. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.
2. Git Documentation – https://git-scm.com/doc
3. GNU GCC Compiler – https://gcc.gnu.org/
4. Stack Overflow Community – https://stackoverflow.com/
5. GeeksforGeeks – https://www.geeksforgeeks.org/c-programming-language/

6. TutorialsPoint C Guide – https://www.tutorialspoint.com/cprogramming/index.htm
7. GitHub Docs – https://docs.github.com/en