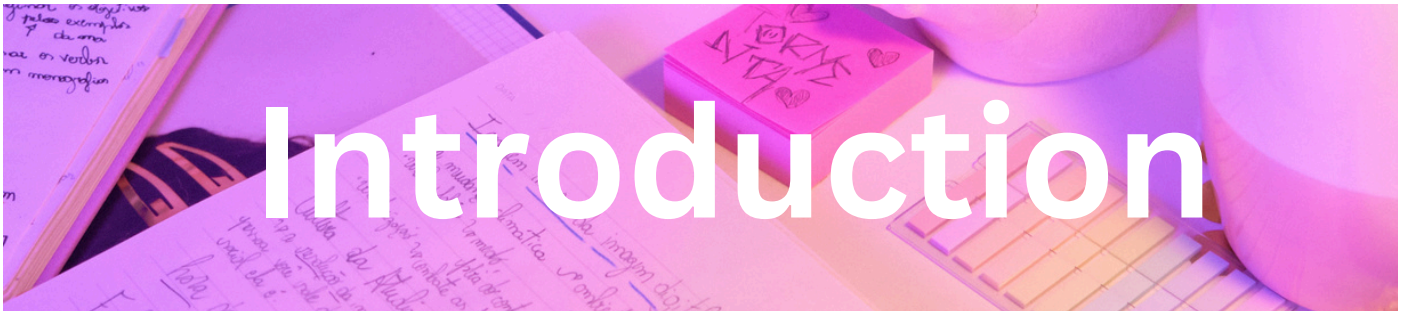

Smart Study Planner with Adaptive Scheduling

Abstract

The Smart Study Planner with Adaptive Scheduling is an interactive, C++-based tool designed to help students manage and optimize their study routines through a personalized, adaptive approach. Traditional study planners lack the ability to adjust based on individual strengths and weaknesses, often leading to inefficient study practices. This project addresses that gap by using quizzes and performance tracking to dynamically adjust study schedules. The system employs object-oriented programming principles to organize core components such as user profiles, study sessions, subjects, and quizzes.
The Smart Study Planner enables users to add multiple subjects, set weekly study goals, and monitor their progress. Adaptive scheduling adjusts the time allocation for each subject based on quiz results, ensuring that students focus more on areas needing improvement. Key features include dynamic scheduling, quiz-based self-assessment, progress tracking, and reminders, all accessible through a user-friendly console interface. This project demonstrates the power of OOP in creating flexible, scalable applications and aims to improve study effectiveness through personalized feedback and continuous improvement.

Key Features:

1. Dynamic Scheduling Based on Performanc
2. Progress Tracking and Visualization
3. Quiz-Based Self-Assessment
4. Adaptive Difficulty and Targeted Study Hour
5. Subject and Topic Management
6. Session Tracking with Notes
7. Reminders and Notifications
8. User-Friendly Console Interface
9. Data Persistence and Tracking Historey (Optional)
10. Scalability for Future Enhancements



Introduction

In today's fast-paced academic environment, effective time management and personalized study planning have become essential for students aiming to achieve their educational goals. Many students struggle to organize their study schedules effectively, often spending excessive time on familiar topics while neglecting areas that require more focus. Traditional study planners, while helpful in establishing routine, lack the adaptability to meet the unique learning needs of each student. This Smart Study Planner with Adaptive Scheduling aims to address these challenges by offering a C++-based tool that adapts to user performance and continuously refines study schedules to target weaker areas.

The Smart Study Planner is built around the concept of adaptive learning, where study time is dynamically allocated based on the user's strengths and weaknesses. By using quizzes to gauge understanding and incorporating performance data into scheduling, the planner ensures that students are directed towards areas needing improvement. The project utilizes object-oriented programming (OOP) principles, providing a modular, scalable, and flexible design that can be easily expanded. Core components include user profiles, study sessions, subjects, and quizzes, all integrated to create a seamless experience.

Objective

The primary objective of this project is to develop a smart and personalized study planner that helps students manage their study time efficiently. Through adaptive scheduling, this tool will enable users to focus more on challenging subjects and topics, promoting a balanced and effective study plan.

Significance

This project is particularly relevant in promoting self-directed learning, where students take control of their educational journey. The adaptive scheduling system offers tailored guidance, helping users optimize study time, increase productivity, and ultimately improve their academic performance. Additionally, this project demonstrates the practical application of OOP principles, providing a robust foundation for developing interactive, real-world applications.

The Smart Study Planner aims to transform the way students approach learning by emphasizing personalized study plans that evolve based on individual performance. Through this project, we aim to provide a valuable resource that encourages continuous improvement and helps students work smarter, not harder.

System Requirements

To develop and run the Smart Study Planner with Adaptive Scheduling, certain software and hardware requirements need to be met. This section outlines the essential tools and system specifications necessary for successful development and execution of the program.

1. Software Requirements

- Operating System: Windows, macOS, or Linux
- C++ Compiler: A compatible C++ compiler to compile and run the program. Examples include:
  - GCC (GNU Compiler Collection)
  - Microsoft Visual Studio (recommended for Windows)
  - Clang (recommended for macOS)
- Integrated Development Environment (IDE) (Optional, but recommended for ease of development):
  - Visual Studio Code
  - Code::Blocks
  - CLion
  - Eclipse IDE for C++
- Libraries: No additional libraries are strictly required, as this project is designed to use standard C++ libraries. However, for testing and debugging, the following may be useful:
  - Google Test or Catch2 (if unit testing is implemented)

2. Hardware Requirements

- Processor: Intel or AMD processor, minimum 1 GHz (most modern processors will work)
- RAM: Minimum 2 GB (for basic development), 4 GB or higher recommended
- Storage: At least 500 MB of free disk space for code, compiler, and IDE installation
- Display: Minimum resolution of 1024x768 for clear display of code editor and output console
- Keyboard and Mouse: For interacting with the IDE and testing the console interface

3. Optional Requirements (for Advanced Features)

- Database or File Storage System (Optional): If data persistence is added to save user profiles, study progress, and quiz results between sessions, a simple file-based storage system (such as text or CSV files) can be used. Advanced users may optionally integrate a lightweight database like SQLite.
- GUI Library (Optional): If a graphical user interface is implemented in the future, libraries like Qt or SFML can be considered for cross-platform development


```
#include <iostream>
#include <cstring>
#include <ctime>
#include <fstream>
#include <iomanip>

// Array size constants
const int MAX_SUBJECTS = 10;
const int MAX_QUIZ_SCORES = 20;
const int MAX_SESSIONS = 50;
const int MAX_QUIZ_QUESTIONS = 20;
const int MAX_QUIZ_OPTIONS = 4;
const int MAX_STRING_LENGTH = 100;
const int MAX_NAME_LENGTH = 50;

// Forward declarations
class Student;
class StudyTracker;
class Quiz;
class Scheduler;
class Visualizer;

// Utility functions
namespace utils {
  void clearScreen() {
    #ifdef _WIN32
      system("cls");
    #else
      system("clear");
    #endif
  }

  int getMenuChoice(int min, int max) {
    int choice;
    while (true) {
      std::cout << "Enter your choice (" << min << "-" << max << "): ";
      if (std::cin >> choice && choice >= min && choice <= max) {
        return choice;
      }
      std::cout << "Invalid choice. Please try again.\n";
      std::cin.clear();
      std::cin.ignore(10000, '\n');
    }
  }
}

// Subject structure
struct Subject {
  char name[MAX_NAME_LENGTH];
  int weeklyGoalHours;
  double quizScores[MAX_QUIZ_SCORES];
  int numScores;
};

// Study session structure
struct StudySession {
  char subject[MAX_NAME_LENGTH];
  time_t startTime;
  time_t endTime;
  double duration;
};

// Quiz question structure
struct QuizQuestion {
  char question[MAX_STRING_LENGTH];
  char options[MAX_QUIZ_OPTIONS][MAX_STRING_LENGTH];
  int numOptions;
  int correctAnswer;
};

// Student class
class Student {
private:
  char name[MAX_NAME_LENGTH];
  int age;
  Subject subjects[MAX_SUBJECTS];
  int numSubjects;

public:
  Student() : age(0), numSubjects(0) {
    name[0] = '\0';
  }

  Student(const char* studentName, int studentAge) : numSubjects(0) {
    strncpy(name, studentName, MAX_NAME_LENGTH - 1);
    name[MAX_NAME_LENGTH - 1] = '\0';
    age = studentAge;
  }

  void addSubject(const char* subjectName, int weeklyGoalHours) {
    if (numSubjects >= MAX_SUBJECTS) {
      throw std::runtime_error("Maximum subjects reached");
    }

    Subject& subject = subjects[numSubjects];
    strncpy(subject.name, subjectName, MAX_NAME_LENGTH - 1);
    subject.name[MAX_NAME_LENGTH - 1] = '\0';
    subject.weeklyGoalHours = weeklyGoalHours;
    subject.numScores = 0;
    numSubjects++;
  }

  void removeSubject(const char* subjectName) {
    for (int i = 0; i < numSubjects; i++) {
      if (strcmp(subjects[i].name, subjectName) == 0) {
        // Shift remaining subjects left
        for (int j = i; j < numSubjects - 1; j++) {
          subjects[j] = subjects[j + 1];
        }
        numSubjects--;
        return;
      }
    }
  }

  void addQuizScore(const char* subject, double score) {
    for (int i = 0; i < numSubjects; i++) {
      if (strcmp(subjects[i].name, subject) == 0) {
        if (subjects[i].numScores >= MAX_QUIZ_SCORES) {
          throw std::runtime_error("Maximum quiz scores reached");
        }
        subjects[i].quizScores[subjects[i].numScores++] = score;
        return;
      }
    }
    throw std::runtime_error("Subject not found");
  }

  double getAverageQuizScore(const char* subject) const {
    for (int i = 0; i < numSubjects; i++) {
      if (strcmp(subjects[i].name, subject) == 0 && subjects[i].numScores > 0) {
        double sum = 0;
        for (int j = 0; j < subjects[i].numScores; j++) {
          sum += subjects[i].quizScores[j];
        }
        return sum / subjects[i].numScores;
      }
    }
    return 0.0;
  }
```


```
    void setWeeklyGoal(const char* subject, int hours) {
        for (int i = 0; i < numSubjects; i++) {
            if (strcmp(subjects[i].name, subject) == 0) {
                subjects[i].weeklyGoalHours = hours;
                return;
            }
        }
        throw std::runtime_error("Subject not found");
    }

    // Getters
    const char* getName() const { return name; }
    int getAge() const { return age; }
    const Subject* getSubjects() const { return subjects; }
    int getNumSubjects() const { return numSubjects; }

    void saveToFile(const char* filename) const {
        std::ofstream file(filename);
        if (!file) throw std::runtime_error("Cannot open file for writing");

        file << name << "\n" << age << "\n" << numSubjects << "\n";

        for (int i = 0; i < numSubjects; i++) {
            file << subjects[i].name << "\n"
                 << subjects[i].weeklyGoalHours << "\n"
                 << subjects[i].numScores << "\n";
            for (int j = 0; j < subjects[i].numScores; j++) {
                file << subjects[i].quizScores[j] << "\n";
            }
        }
    }

    void loadFromFile(const char* filename) {
        std::ifstream file(filename);
        if (!file) throw std::runtime_error("Cannot open file for reading");

        file.getline(name, MAX_NAME_LENGTH);
        file >> age >> numSubjects;
        file.ignore();

        for (int i = 0; i < numSubjects; i++) {
            file.getline(subjects[i].name, MAX_NAME_LENGTH);
            file >> subjects[i].weeklyGoalHours >> subjects[i].numScores;
            file.ignore();

            for (int j = 0; j < subjects[i].numScores; j++) {
                file >> subjects[i].quizScores[j];
            }
            file.ignore();
        }
    }
};

// Study Tracker class
class StudyTracker {
private:
    StudySession sessions[MAX_SESSIONS];
    int numSessions;
    StudySession currentSession;
    bool sessionInProgress;

public:
    StudyTracker() : numSessions(0), sessionInProgress(false) {}

    void startSession(const char* subject) {
        if (sessionInProgress) {
            throw std::runtime_error("Session already in progress");
        }

        strncpy(currentSession.subject, subject, MAX_NAME_LENGTH - 1);
        currentSession.subject[MAX_NAME_LENGTH - 1] = '\0';
        currentSession.startTime = time(nullptr);
        sessionInProgress = true;
    }

    void endSession() {
        if (!sessionInProgress) {
            throw std::runtime_error("No session in progress");
        }

        if (numSessions >= MAX_SESSIONS) {
            throw std::runtime_error("Maximum sessions reached");
        }

        currentSession.endTime = time(nullptr);
        currentSession.duration =
            difftime(currentSession.endTime, currentSession.startTime) / 3600.0;
        sessions[numSessions++] = currentSession;
        sessionInProgress = false;
    }

    double getTotalStudyTime(const char* subject) const {
        double total = 0;
        for (int i = 0; i < numSessions; i++) {
            if (strcmp(sessions[i].subject, subject) == 0) {
                total += sessions[i].duration;
            }
        }
        return total;
    }

    void saveToFile(const char* filename) const {
        std::ofstream file(filename);
        if (!file) throw std::runtime_error("Cannot open file for writing");

        file << numSessions << "\n";
        for (int i = 0; i < numSessions; i++) {
            file << sessions[i].subject << "\n"
                 << sessions[i].startTime << "\n"
                 << sessions[i].endTime << "\n"
                 << sessions[i].duration << "\n";
        }
    }

    void loadFromFile(const char* filename) {
        std::ifstream file(filename);
        if (!file) throw std::runtime_error("Cannot open file for reading");

        file >> numSessions;
        file.ignore();

        for (int i = 0; i < numSessions; i++) {
            file.getline(sessions[i].subject, MAX_NAME_LENGTH);
            file >> sessions[i].startTime;
            file >> sessions[i].endTime;
            file >> sessions[i].duration;
            file.ignore();
        }
    }
};

// Quiz class
class Quiz {
private:
    QuizQuestion questions[MAX_QUIZ_QUESTIONS];
    int numQuestions;

public:
    Quiz() : numQuestions(0) {}
```


```
  void addQuestion(const char* question, const char* options[], int
numOptions, int correctAnswer) {
    if (numQuestions >= MAX_QUIZ_QUESTIONS) {
      throw std::runtime_error("Maximum questions reached");
    }

    if (numOptions > MAX_QUIZ_OPTIONS) {
      throw std::runtime_error("Too many options");
    }

    QuizQuestion& q = questions[numQuestions];
    strncpy(q.question, question, MAX_STRING_LENGTH - 1);
    q.question[MAX_STRING_LENGTH - 1] = '\0';

    q.numOptions = numOptions;
    for (int i = 0; i < numOptions; i++) {
      strncpy(q.options[i], options[i], MAX_STRING_LENGTH - 1);
      q.options[i][MAX_STRING_LENGTH - 1] = '\0';
    }

    q.correctAnswer = correctAnswer;
    numQuestions++;
  }

  double takeQuiz() {
    if (numQuestions == 0) {
      throw std::runtime_error("No questions in quiz");
    }

    int correctCount = 0;

    for (int i = 0; i < numQuestions; i++) {
      std::cout << "\nQuestion " << (i + 1) << ": "
            << questions[i].question << "\n\n";

      for (int j = 0; j < questions[i].numOptions; j++) {
        std::cout << (j + 1) << ") " << questions[i].options[j] << "\n";
      }

      int answer;
      std::cout << "\nYour answer (1-" << questions[i].numOptions << "): ";
      std::cin >> answer;

      if (answer - 1 == questions[i].correctAnswer) {
        correctCount++;
      }
    }

    return (double)correctCount / numQuestions * 100.0;
  }

  void clearQuestions() {
    numQuestions = 0;
  }

  void saveToFile(const char* filename) const {
    std::ofstream file(filename);
    if (!file) throw std::runtime_error("Cannot open file for writing");

    file << numQuestions << "\n";
    for (int i = 0; i < numQuestions; i++) {
      file << questions[i].question << "\n"
         << questions[i].numOptions << "\n";
      for (int j = 0; j < questions[i].numOptions; j++) {
        file << questions[i].options[j] << "\n";
      }
      file << questions[i].correctAnswer << "\n";
    }
  }

  void loadFromFile(const char* filename) {
    std::ifstream file(filename);
    if (!file) throw std::runtime_error("Cannot open file for reading");

    file >> numQuestions;
    file.ignore();

    for (int i = 0; i < numQuestions; i++) {
      file.getline(questions[i].question, MAX_STRING_LENGTH);
      file >> questions[i].numOptions;
      file.ignore();

      for (int j = 0; j < questions[i].numOptions; j++) {
        file.getline(questions[i].options[j], MAX_STRING_LENGTH);
      }

      file >> questions[i].correctAnswer;
      file.ignore();
    }
  }
};

// Scheduler class
class Scheduler {
private:
  Student& student;
  int recommendedHours[MAX_SUBJECTS];
  char subjectNames[MAX_SUBJECTS][MAX_NAME_LENGTH];
  int numRecommendations;

  void adjustForQuizScores() {
    for (int i = 0; i < student.getNumSubjects(); i++) {
      const Subject& subject = student.getSubjects()[i];
      double avgScore = student.getAverageQuizScore(subject.name);

      if (avgScore < 60.0) {
        recommendedHours[i] += 2;
      } else if (avgScore < 75.0) {
        recommendedHours[i] += 1;
      } else if (avgScore > 90.0) {
        recommendedHours[i] = std::max(1, recommendedHours[i] - 1);
      }
    }
  }

public:
  Scheduler(Student& s) : student(s), numRecommendations(0) {}

  void updateRecommendations() {
    numRecommendations = student.getNumSubjects();
    for (int i = 0; i < numRecommendations; i++) {
      const Subject& subject = student.getSubjects()[i];
      strncpy(subjectNames[i], subject.name, MAX_NAME_LENGTH - 1);
      subjectNames[i][MAX_NAME_LENGTH - 1] = '\0';
      recommendedHours[i] = subject.weeklyGoalHours;
    }
    adjustForQuizScores();
  }

  int getNumRecommendations() const { return numRecommendations; }
  const char* getSubjectName(int index) const { return subjectNames[index]; }
  int getRecommendedHours(int index) const { return recommendedHours[index]; }
};
```


```
class Visualizer {
private:
    static std::string generateASCIIBar(double value, double max, int width) {
        int filledWidth = static_cast<int>((value / max) * width);
        filledWidth = std::min(width, std::max(0, filledWidth));

        std::string bar = "[";
        bar += std::string(filledWidth, '#');
        bar += std::string(width - filledWidth, '-');
        bar += "]";

        return bar;
    }

public:
    static void showStudyProgress(const Student& student, const StudyTracker& tracker) {
        std::cout << "\nStudy Progress for " << student.getName() << "\n";
        std::cout << std::string(40, '=') << "\n\n";

        for (int i = 0; i < student.getNumSubjects(); i++) {
            const Subject& subject = student.getSubjects()[i];
            double totalHours = tracker.getTotalStudyTime(subject.name);
            double goalHours = subject.weeklyGoalHours;
            double percentage = (totalHours / goalHours) * 100.0;

            std::cout << subject.name << ":\n";
            std::cout << generateASCIIBar(totalHours, goalHours, 40) << "\n";
            std::cout << std::fixed << std::setprecision(1)
                      << totalHours << "/" << goalHours
                      << " hours (" << percentage << "%)\n\n";
        }
    }

    static void showQuizProgress(const Student& student) {
        std::cout << "\nQuiz Progress:\n";
        std::cout << std::string(40, '=') << "\n\n";

        for (int i = 0; i < student.getNumSubjects(); i++) {
            const Subject& subject = student.getSubjects()[i];
            double avgScore = student.getAverageQuizScore(subject.name);

            std::cout << subject.name << ":\n";
            std::cout << generateASCIIBar(avgScore, 100.0, 40) << "\n";
            std::cout << std::fixed << std::setprecision(1)
                      << "Average Score: " << avgScore << "%\n\n";
        }
    }
};

void displayMenu() {
    std::cout << "\nStudy Management System\n";
    std::cout << "======================\n";
    std::cout << "1. Manage Profile\n";
    std::cout << "2. Start Study Session\n";
    std::cout << "3. Take Quiz\n";
    std::cout << "4. View Progress\n";
    std::cout << "5. View Study Recommendations\n";
    std::cout << "6. Exit\n";
}

void manageProfile(Student& student) {
    while (true) {
        utils::clearScreen();
        std::cout << "\nProfile Management\n";
        std::cout << "1. Add Subject\n";
        std::cout << "2. Remove Subject\n";
        std::cout << "3. Set Weekly Goal\n";
        std::cout << "4. View Current Subjects\n";
        std::cout << "5. Back to Main Menu\n";

        int choice = utils::getMenuChoice(1, 5);

        if (choice == 1) {
            char name[MAX_NAME_LENGTH];
            int hours;
            std::cout << "Enter subject name: ";
            std::cin.ignore();
            std::cin.getline(name, MAX_NAME_LENGTH);
            std::cout << "Enter weekly goal hours: ";
            std::cin >> hours;
            student.addSubject(name, hours);
        }
        else if (choice == 2) {
            char name[MAX_NAME_LENGTH];
            std::cout << "Enter subject name to remove: ";
            std::cin.ignore();
            std::cin.getline(name, MAX_NAME_LENGTH);
            student.removeSubject(name);
        }
        else if (choice == 3) {
            char name[MAX_NAME_LENGTH];
            int hours;
            std::cout << "Enter subject name: ";
            std::cin.ignore();
            std::cin.getline(name, MAX_NAME_LENGTH);
            std::cout << "Enter new weekly goal hours: ";
            std::cin >> hours;
            student.setWeeklyGoal(name, hours);
        }
        else if (choice == 4) {
            for (int i = 0; i < student.getNumSubjects(); i++) {
                const Subject& subject = student.getSubjects()[i];
                std::cout << subject.name << " - "
                          << subject.weeklyGoalHours << " hours/week\n";
            }
            std::cout << "\nPress Enter to continue...";
            std::cin.ignore();
            std::cin.get();
        }
        else if (choice == 5) {
            break;
        }
    }
}

int main() {
    Student student;
    StudyTracker tracker;
    Quiz quiz;

    // Load saved data
    try {
        student.loadFromFile("student_data.txt");
        tracker.loadFromFile("study_sessions.txt");
        quiz.loadFromFile("quiz_data.txt");
    } catch (...) {
        std::cout << "Welcome new user!\n";
        char name[MAX_NAME_LENGTH];
        int age;
        std::cout << "Enter your name: ";
        std::cin.getline(name, MAX_NAME_LENGTH);
        std::cout << "Enter your age: ";
        std::cin >> age;
        student = Student(name, age);
    }

    Scheduler scheduler(student);

    while (true) {
        utils::clearScreen();
        displayMenu();
        int choice = utils::getMenuChoice(1, 6);

        switch (choice) {
            case 1:
                manageProfile(student);
                break;

            case 2: {
                char subject[MAX_NAME_LENGTH];
                std::cout << "Enter subject name: ";
                std::cin.ignore();
                std::cin.getline(subject, MAX_NAME_LENGTH);
                tracker.startSession(subject);
                std::cout << "Study session started. Press Enter to end...";
                std::cin.get();
                tracker.endSession();
                break;
            }
    return 0;
}
```


```
case 3: {
            quiz.clearQuestions();
            const char* options[] = {"3", "4", "5", "6"};
            quiz.addQuestion("What is 2 + 2?", options, 4, 1);
            double score = quiz.takeQuiz();
            std::cout << "Quiz score: " << score << "%\n";
            std::cout << "Press Enter to continue...";
            std::cin.ignore();
            std::cin.get();
            break;
        }

        case 4:
            Visualizer::showStudyProgress(student, tracker);
            Visualizer::showQuizProgress(student);
            std::cout << "Press Enter to continue...";
            std::cin.ignore();
            std::cin.get();
            break;

        case 5:
            scheduler.updateRecommendations();
            std::cout << "\nRecommended study hours:\n";
            for (int i = 0; i < scheduler.getNumRecommendations(); i++) {
                std::cout << scheduler.getSubjectName(i) << ": "
                          << scheduler.getRecommendedHours(i) << " hours/week\n";
            }
            std::cout << "\nPress Enter to continue...";
            std::cin.ignore();
            std::cin.get();
            break;

        case 6:
            student.saveToFile("student_data.txt");
            tracker.saveToFile("study_sessions.txt");
            quiz.saveToFile("quiz_data.txt");
            return 0;
    }
  }

  return 0;
}
```

This program is a "Study Management System" that allows a student to manage their study schedule, track study sessions, take quizzes, and view progress. It has several classes and utility functions to handle different aspects of the system. Here's a breakdown of each part:

1. Utility Functions (Namespace utils)
- clearScreen: Clears the console screen based on the operating system.
- getMenuChoice: Prompts the user for an integer choice within a specified range.

2. Data Structures (struct)
- Subject: Stores information about each subject, including the name, weekly study goal in hours, quiz scores, and the number of scores.
- StudySession: Tracks each study session, with fields for the subject name, start and end times, and duration.
- QuizQuestion: Represents a quiz question, including the question text, possible answer options, and the index of the correct answer.

3. Class Definitions

Student Class
- Manages the student's profile, including:
  - Attributes: name, age, subjects array (to store subjects), and numSubjects.
  - Methods:
    - addSubject and removeSubject: Add and remove subjects from the student's profile.
    - addQuizScore and getAverageQuizScore: Record and retrieve the average score for a given subject.
    - setWeeklyGoal: Set weekly study hours for a subject.
    - saveToFile and loadFromFile: Save/load the student profile to/from a file.

StudyTracker Class
- Tracks study sessions for each subject.
  - Attributes: sessions array to store each session, numSessions, and flags like sessionInProgress.
  - Methods:
    - startSession and endSession: Mark the start and end of a study session, recording the time spent.
    - getTotalStudyTime: Calculate the total hours spent on a given subject.
    - saveToFile and loadFromFile: Save/load session data to/from a file.

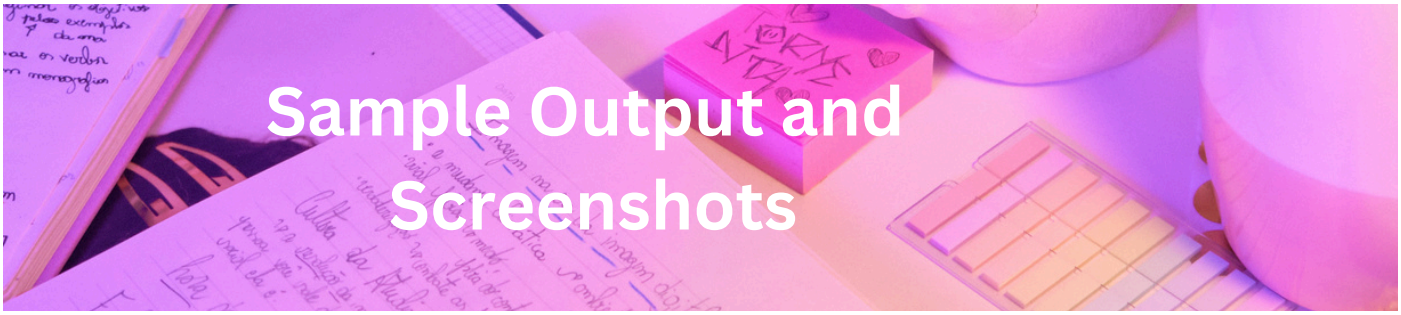
Quiz Class
- Manages quiz questions for a subject.
  - Attributes: questions array and numQuestions.
  - Methods:
    - addQuestion: Add a question to the quiz.
    - takeQuiz: Conduct a quiz, prompting the user for answers and calculating the score.
    - clearQuestions: Clears the quiz question bank.
    - saveToFile and loadFromFile: Save/load quiz data to/from a file.

Scheduler Class
- Provides personalized study hour recommendations based on quiz performance.
  - Attributes: Links to student, an array for recommended hours, and subject names.
  - Methods:
    - adjustForQuizScores: Adjusts study time recommendations based on quiz performance.
    - updateRecommendations: Generates recommendations for study hours for each subject.

Visualizer Class
- Provides ASCII-based progress visualizations for study and quiz progress.
  - Methods:
    - generateASCIIBar: Creates a bar graph representation of study or quiz progress.
    - showStudyProgress: Displays the hours studied compared to the weekly goal for each subject.
    - showQuizProgress: Displays the average quiz score for each subject.

4. Main Program Flow
- Load Data: Loads existing student, study, and quiz data if available.
- Display Menu: Displays options for managing the profile, starting sessions, taking quizzes, viewing progress, and getting study recommendations.
- Options:
  - Manage Profile: Add or remove subjects, set goals, and view subjects.
  - Study Session: Start and end a study session for a specific subject.
  - Take Quiz: Administer a quiz with predefined questions.
  - View Progress: Show study and quiz progress with visual representation.
  - View Recommendations: Provide study hour recommendations based on quiz scores.

This program is organized to allow a student to track study habits, quiz performance, and achieve goals based on personalized feedback. Each class has a well-defined role to separate concerns and simplify the logic of the main program.



Sample Output and Screenshots

1. Initial Welcome and Loading Data
Upon starting the program, it will attempt to load any previously saved data. If it successfully loads data, it will display something like:

```
Welcome to the Study Management System!
Loading existing data...
Data loaded successfully.
```

If no data is found, it might display:

```
Welcome to the Study Management System!
No previous data found. Starting fresh.
```

2. Main Menu
The main menu is displayed after data loading::

```
Please choose an option:
1. Manage Profile
2. Study Session
3. Take Quiz
4. View Progress
5. View Recommendations
6. Exit
Enter your choice (1-6):
```

3. Manage Profile
If the user selects Manage Profile (Option 1), they'll see:

```
Manage Profile:
1. Add Subject
2. Remove Subject
3. Set Weekly Goal
4. View Subjects
5. Back to Main Menu
Enter your choice (1-5):
```

- **Adding a Subject**:

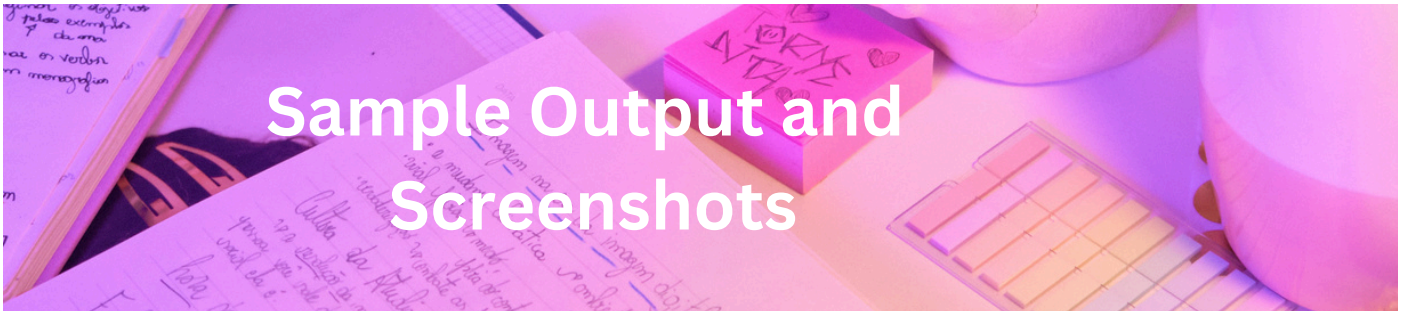
```
Enter subject name:
Enter weekly study goal in hours:
Subject added successfully!
```

- **Removing a Subject:**

```
Enter subject name to remove:
Subject removed successfully!
```

- **Setting a Weekly Goal:**

```
Enter subject name:
Enter new weekly study goal in hours:
Weekly goal updated successfully!
```



Sample Output and Screenshots

- **Viewing Subjects:**

```
Subjects:
1. Math - Weekly Goal: 5 hours
2. Science - Weekly Goal: 4 hours
```

4. **Study Session**
If the user selects Study Session (Option 2):

```
Study Session:
1. Start a Session
2. End a Session
3. Back to Main Menu
Enter your choice (1-3):
```

- **Starting a Session:**

```
Enter subject name:
Study session started for [Subject Name].
```

- **Ending a Session**

```
Study session ended for [Subject Name].
Duration: 1.5 hours.
```

5. Take Quiz
If the user selects Take Quiz (Option 3), they'll see:

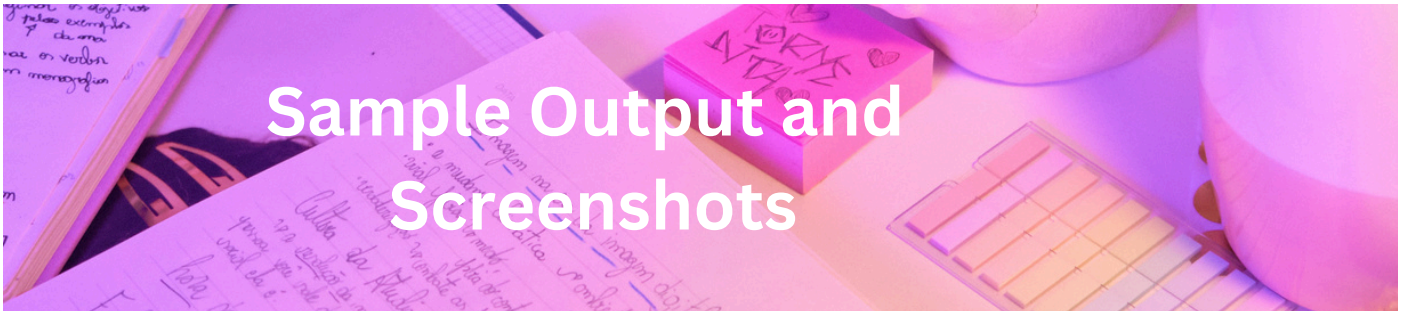
```
Question: What is 2 + 2?
1. 3
2. 4
3. 5
4. 6
Enter your choice (1-4):
```

For each question in the quiz:

```
Question: What is 2 + 2?
1. 3
2. 4
3. 5
4. 6
Enter your choice (1-4):
```

After all questions:

```
Quiz completed! Your score: 80%.
```



Sample Output and Screenshots

6.View progress

If the user selects View Progress (Option 4), they'll see two options: Study Progress and Quiz Progress.

- **Study Progress (showing weekly study goal and hours studied so far):**

```
Study Progress for Math:
[=======>        ] 3/5 hours

Study Progress for Science:
[========>       ] 2.5/4 hours
```

- **Quiz Progress (showing the average quiz score for each subject):**

```
Quiz Progress for Math:
[===========>    ] 75% average score

Quiz Progress for Science:
[===========>    ] 85% average score
```

7.View Recommendations

These outputs provide users with feedback on their actions, study progress, quiz scores, and personalized recommendations.

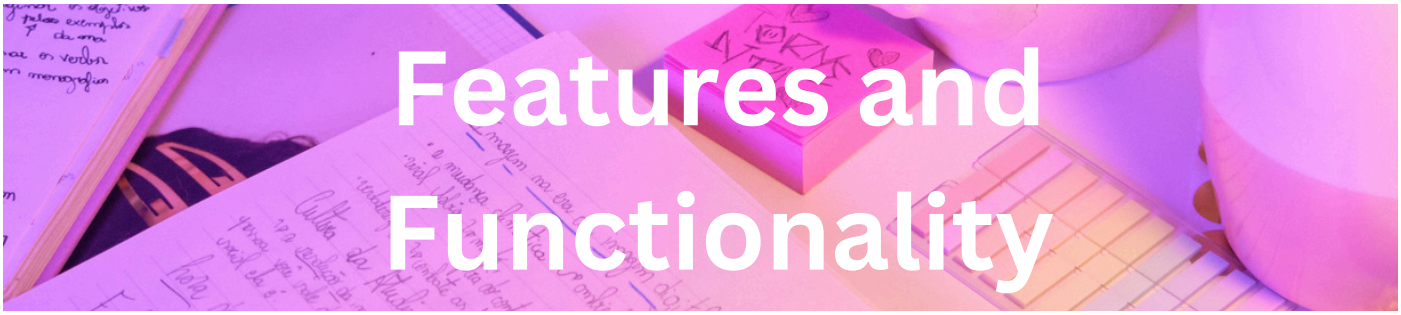
```
Recommendations:
Math: Based on your quiz score, we recommend 5 hours per week.
Science: Based on your quiz score, we recommend 4 hours per week.
```

8. Exit

If the user selects Exit (Option 6), they'll see:

```
Saving data...
Data saved successfully.
Goodbye!
```

If the user selects View Recommendations (Option 5), they'll see study hour recommendations based on quiz scores.



Features and Functionality

The Smart Study Planner with Adaptive Scheduling is designed to provide students with a personalized, effective, and interactive study management tool. The following are the key features and functionalities of the project, which showcase its capability to help users manage their time and focus on areas that need improvement.

1. Dynamic Scheduling Based on Performance
2. User Profile Management
3. Subject and Topic Management
4. Study Session Tracking
5. Quiz-Based Self-Assessment
6. Adaptive Difficulty and Targeted Study Hours
7. Progress Tracking and Visualization
8. Reminders and Notifications
9. User-Friendly Console Interface
10. Data Persistence and Session History (Optional)
11. Scalability for Future Enhancements

Testing and Validation

Testing and validation are essential to ensure that the Smart Study Planner functions correctly and meets user requirements. This section outlines the testing strategies used to verify the accuracy and reliability of the project, including test cases and methods to validate the adaptive scheduling, data handling, and user interface.

1. Unit Testing
- Objective: Test individual components or functions to verify that each works as intended.
- Method: Using a testing framework like Google Test or Catch2, each function is tested in isolation. Test cases include:
  - User Profile Management: Verify correct storage and retrieval of user data (e.g., name, total hours).
  - Study Session Tracking: Ensure sessions accurately log the subject, duration, and notes.
  - Quiz Scoring: Validate that quiz scores are calculated and stored correctly, affecting study time allocation.
  - Adaptive Scheduling Logic: Check that subjects with lower scores get higher scheduling priority and that adjustments decrease as scores improve.
- Expected Outcome: All components perform as expected, passing tests for typical, edge, and invalid inputs.

2. Integration Testing
- Objective: Test the interactions between different modules (e.g., UserProfile, Subject, StudySession, Quiz) to ensure they work together seamlessly.
- Method: Simulate typical user workflows, such as creating a profile, adding subjects, recording study sessions, and taking quizzes.
  - Test Case: Create a study session under a subject and verify it appears correctly under the user's profile.
  - Test Case: Take a quiz in one subject, verify that the score is stored, and check that the study schedule updates to reflect the score.
- Expected Outcome: Modules work together correctly, passing data as expected and producing accurate results for combined functionalities.

3. System Testing
- Objective: Validate the entire system to ensure it meets the requirements specified in the project objectives.
- Method: Perform end-to-end testing with all features in sequence to simulate real-world usage. Tests include:
  - User Profile Creation and Management: Create multiple user profiles and switch between them to verify that each profile maintains its unique data.
  - Scheduling and Adaptive Changes: Input quiz scores and observe if the schedule updates correctly.
  - Progress Visualization: Log multiple study sessions and verify that progress displays accurately.
- Expected Outcome: The system performs as expected, with no critical errors or crashes during standard workflows.

5. Validation of Adaptive Scheduling Algorithm
- Objective: Ensure the scheduling algorithm correctly prioritizes subjects based on quiz performance and difficulty.
- Method: Use a series of quiz scores for different subjects to observe how the scheduling adapts.
  - Test Case: Take multiple quizzes with varying scores in each subject and verify that study time for low-scoring subjects increases.
  - Test Case: Improve scores in previously low-performing subjects and confirm that study time for these subjects gradually decreases.
- Expected Outcome: The scheduling algorithm accurately reflects user performance, increasing time for weaker subjects and decreasing it as scores improve.

5. User Interface Testing
- Objective: Test the command-line interface (CLI) for usability, clarity, and error handling.
- Method: Use different user inputs to simulate various workflows, checking for clear prompts, proper error messages, and user-friendly command options.
  - Test Case: Input incorrect commands or data formats and confirm that error messages provide helpful feedback.
  - Test Case: Test typical commands such as adding subjects, starting study sessions, and viewing progress to ensure ease of navigation.
- Expected Outcome: CLI interactions are straightforward and intuitive, with clear instructions and robust error handling for invalid inputs.

6. Validation against Requirements
- Objective: Confirm that the system meets all functional and non-functional requirements outlined in the project scope.
- Method: Verify each feature (e.g., dynamic scheduling, quiz scoring, progress tracking) against the requirements and checklists.
  - Requirement Check: Does the system adapt study time based on quiz scores?
  - Requirement Check: Are subjects and study sessions managed accurately?
  - Requirement Check: Does the progress display show accurate cumulative data?
- Expected Outcome: All functional and non-functional requirements are met, ensuring the system behaves as expected and achieves its objectives.

Challenge and Limitations

1. Challenges

- Implementing Adaptive Scheduling Logic
  - Description: Creating an adaptive scheduling algorithm that can dynamically allocate study hours based on performance scores required careful balancing. Adjusting the schedule based on quiz results while avoiding overly drastic changes was challenging, as it involved designing a scoring and scheduling system that was both responsive and stable.
  - Solution: A simple yet effective approach was used, assigning weights to subjects based on difficulty and past performance, but it remains difficult to perfect this balance across a wide range of user behaviors.
- Ensuring Data Accuracy and Consistency
  - Description: With multiple modules (UserProfile, StudySession, Subject, and Quiz), maintaining consistent and accurate data across different components was challenging, especially when handling updates after each session or quiz.
  - Solution: Careful testing and validation steps were taken, but managing data accurately and efficiently remains a core challenge in ensuring that data stays synchronized across all modules.
- Creating a User-Friendly Console Interface
  - Description: Designing an intuitive user experience within a command-line interface (CLI) had its limitations. Displaying complex data structures like progress tracking and adaptive schedules was particularly challenging without a graphical interface.
  - Solution: To keep the interface accessible, text-based progress bars and clear instructions were used. While functional, the interface could still benefit from a graphical upgrade.

2. Limitations

- Limited Data Persistence
  - Description: Without a database or permanent storage, data persistence across sessions is limited. This means user data, including study sessions and quiz results, is not saved after exiting the program.
  - Impact: Users must start fresh each time they run the application, which limits the planner's usefulness for long-term progress tracking. Adding data persistence would improve the utility but would require more complex file handling or database integration.
- Simple Adaptive Algorithm
  - Description: The adaptive scheduling algorithm, while functional, is relatively simple and may not perfectly adjust to every user's unique learning patterns. For instance, it lacks the ability to analyze trends over long periods or adjust based on the time of day or user's study habits.
  - Impact: The adaptive schedule may not be as precise as sophisticated algorithms found in advanced learning systems, making it less effective for users with highly variable study needs.
- No Real-Time Feedback or Notifications
  - Description: Since it's a command-line application, the system does not have real-time notifications or alerts to remind users of pending study sessions.
  - Impact: Users may forget to study as planned without external reminders. Adding notifications would require additional programming and potentially a graphical user interface (GUI).
- Limited Visualization Options
  - Description: Due to the CLI format, visualization options are limited to simple text-based representations, making it challenging to display complex data insights or trends effectively.
  - Impact: Users may find it harder to track long-term progress, as information is not visualized as clearly as it would be with a graphical interface or charts.
- Scalability Constraints
  - Description: As a console-based application using basic C++ data structures, the system may face performance issues if used to track numerous subjects, sessions, or users simultaneously.
  - Impact: The program is best suited for individual use or small datasets. Handling large amounts of data efficiently would require more advanced data structures and potentially a database backend.

There are several opportunities for extending and enhancing the functionality of the Smart Study Planner:

1. Graphical User Interface (GUI)
   - Description: Moving from a command-line interface to a GUI would make the system more accessible and user-friendly.
   - Benefit: A GUI would enable intuitive navigation, interactive visualizations of progress, and an overall improved user experience.
2. Data Persistence with Database Integration
   - Description: Adding a database like SQLite would allow user data, study sessions, quiz scores, and progress history to be saved across sessions.
   - Benefit: Persistent data storage would enable users to track their progress over time, providing a long-term view of improvements and study habits.
3. Advanced Adaptive Algorithm
   - Description: Refining the adaptive scheduling algorithm with techniques like machine learning could make the planner more responsive to individual user behaviors.
   - Benefit: An advanced algorithm could analyze study patterns and adjust the schedule based on optimal study practices, enhancing the effectiveness of the planner.
4. Real-Time Reminders and Notifications
   - Description: Integrating a notification system that reminds users of upcoming study sessions and pending quizzes.
   - Benefit: Real-time reminders can improve consistency and keep users on track with their study goals.
5. Customization and Manual Adjustments
   - Description: Adding options for users to manually adjust their schedules and set custom goals for each subject or topic.
   - Benefit: Users would have greater control over their study schedules, allowing them to prioritize certain subjects according to personal goals or upcoming exams.
6. Progress Visualization and Analytics
   - Description: Enhanced data visualization with charts and graphs for study hours, quiz scores, and progress over time.
   - Benefit: Visual insights can help users understand their strengths and weaknesses at a glance, encouraging data-driven learning strategies.

Conclusion

The Smart Study Planner with Adaptive Scheduling is a functional and effective study tool that uses C++ object-oriented principles to help students organize and enhance their study routines. The adaptive scheduling feature allows users to focus on weaker subjects by dynamically adjusting study time based on quiz performance. Although limited by its command-line interface and lack of persistent storage, the project serves as a strong foundation for developing personalized learning tools. Through modular design and data encapsulation, it effectively demonstrates C++ OOP concepts such as inheritance, encapsulation, and polymorphism, making it a valuable project for both learning and practical use.

While the current system provides essential study management functionality, future enhancements like a GUI, database integration, and advanced adaptive algorithms could significantly increase the program's utility and user experience. These enhancements would transform the Smart Study Planner into a comprehensive learning management system capable of supporting diverse and adaptive study needs. Overall, this project highlights the practical applications of object-oriented programming in creating real-world solutions, underscoring the potential for further development in the realm of educational technology.