Project

<Time-Tracker>

CIS-17B

Name: Aleksandar Videv

Table Of Contents

**1.Introduction………………………………….……..……………..……….……3**

**2.** **Overview………………………………….………………………….….…..….3**

**2.1 Project Summary…………………….………………..………………3**

**2.2 Versioning Scheme…………………….……………………………..3**

**3.** **Project Description…………………….…………………………………….…4**

**4. Gantt Chart…………………….………………………………………........…5**

**5. UML Diagrams…………….……………………….…………………………..7**

6. Pseudo Code……**…………………………..…………………………..……..8**

7.Demonstrating Functionality**………………..………….….……..….….…...10**

7.1 Proof of Concept/ Doxygen**…....………….….….…….…..…...…..10**

7.2 Testing Results**………………..………….….….….….….….……..37**

**1.Introduction**

The Time Manager is a sophisticated application designed to aid users in tracking their time effectively within educational or professional settings. The application leverages user authentication, detailed time tracking, and interactive menus to facilitate productivity and accountability. By offering precise tracking functionalities and user management capabilities, Time Manager ensures that users can focus on their tasks with enhanced efficiency.

**2. Overview**

**2.1 Project Summary**

The Time Manager application is detailed with its coding metrics as follows:

* **GitHub Repository**: [Link](https://github.com/AlexanderVidev/Time-Tracker)
* **Total Lines of Code**: 1207
  + **Executable Lines**: 844
  + **Comment Lines**: 363
* **Variables**: 35+ (15 unique member variables, 20+ local variables)
* **Methods**: 35

**Project Insights:** Developed in C++, the Time Manager leverages various standard libraries to provide robust functionalities such as user data management and timer operations. Approximately 50 hours have been invested in the design and coding phases, excluding additional hours for documentation and debugging. The project is characterized by continuous incremental updates to improve its usability and adapt to the evolving needs of its users.

**2.2 Versioning Scheme**

**Version 1:** Initial version in a single C++ file, includes basic time tracking and user interface functionalities.

**Version 2:** Separates the TimeTracker class into its own header and source files, enhancing code maintainability and scalability.

**Version 3:** Introduces user management functionalities, adding classes for user properties and operations.

**Version 4:** Extends the User class to track detailed time (hours, minutes, seconds) and enhances user management capabilities.

**Version 5:** Incorporates a Menu class to centralize user interaction methods, linking time tracking directly to user accounts.

**Version 6:** Optimizes data handling by implementing fixed-size character arrays for user data, improving binary file I/O operations.

**Version 7:** Adds validated time input and robust error handling for file operations to enhance data integrity and application stability.

**Version 8:** Enhances user data security by introducing encryption and decryption functionalities in the UserManager class, ensuring secure data handling.

**Version 9:** Continues to focus on security with refined encryption processes and improved error handling, maintaining robust user data management.

**Version 9\_1:** Implements comprehensive **Doxygen** documentation for all classes and methods, making the application more accessible for developers and maintainers.

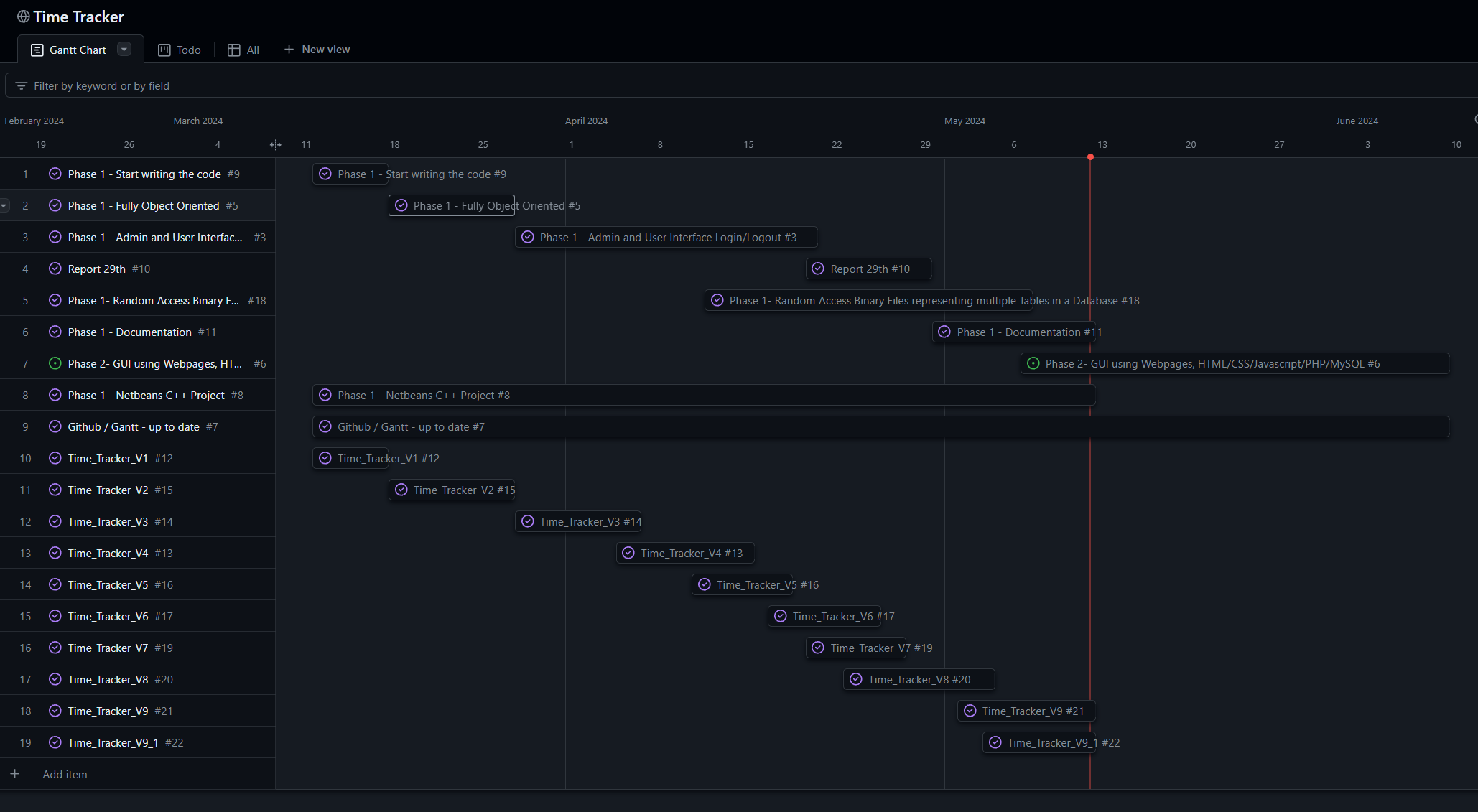
**3.** **Project Description**

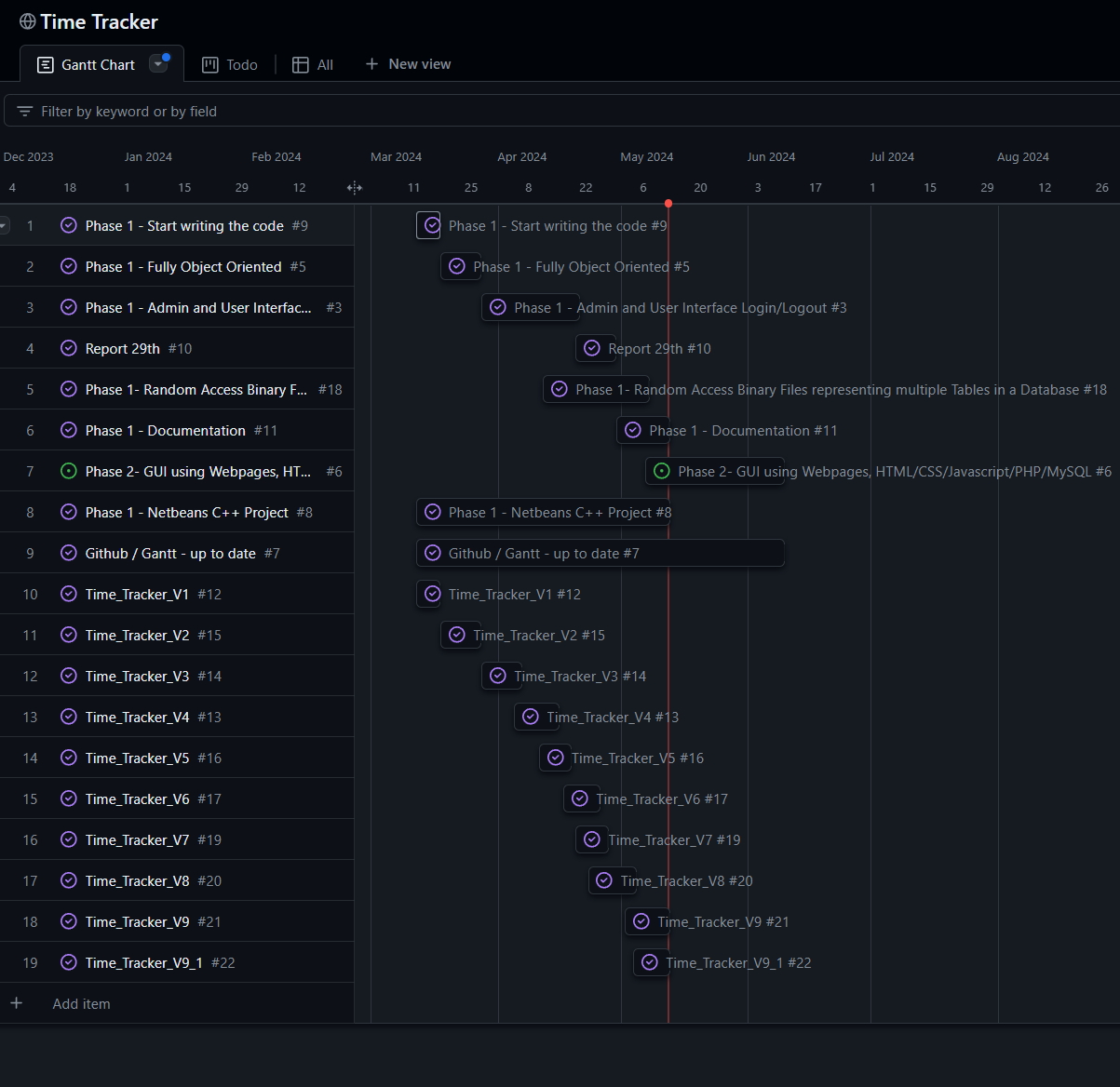
The Time Manager application has been refined with several key features to ensure robust functionality and an improved user experience:

* **Robust Time Tracking**: Enhancements to the Time Tracker module include mechanisms to prevent a timer from being started multiple times, ensuring the accuracy of recorded time and preventing data corruption.
* **Enhanced User Management**: The UserManager module now includes more sophisticated methods for user creation, editing, and deletion. These improvements cater to complex scenarios such as changing user roles or managing class information, making the system highly adaptable to diverse user needs.
* **Normalized Time Input and Display**: Time normalization across the system ensures that time data is consistently accurate and displayed correctly, which is crucial when editing user times or logging session durations.
* **Interactive and Responsive Menus**: The menu system has been structured to provide clear and intuitive pathways for user interaction, whether for system administration or time tracking. New checks and balances have been introduced to guide users effectively through their options, minimizing the likelihood of errors and enhancing the overall user experience.

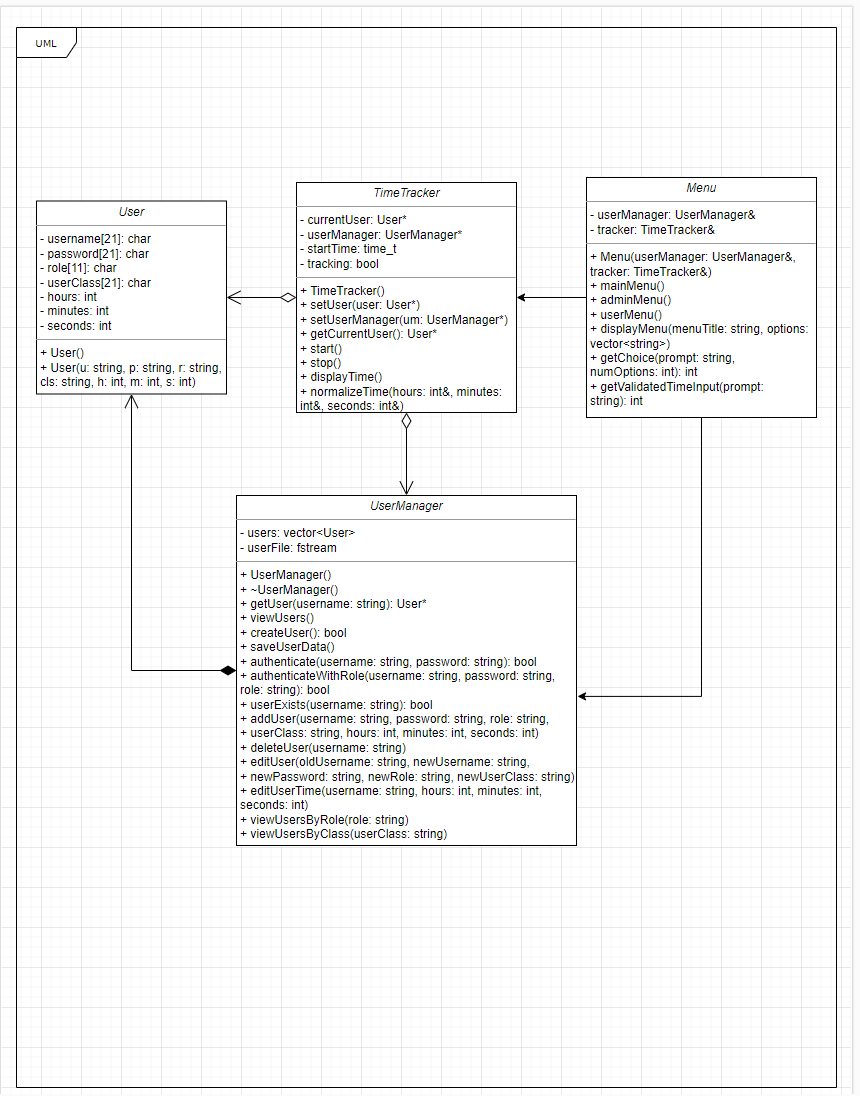
**4. Gantt Chart**

**The Gantt Chart can also be found** [here](https://github.com/users/AlexanderVidev/projects/2/views/1)**.**





**5. UML**

****

**6. Pseudo Code**

Program TimeTrackerSystem

Class User

Define properties for username, password, role, userClass, hours, minutes, seconds

Initialize user properties with default values

Provide constructors to set user properties

Class UserManager

Define a list to store Users

Define methods for user management:

Load users from file

Write users to file

Encrypt and decrypt user data

Get a user by username

Create a new user

Authenticate a user by username and password, and role

Check if a user exists

Add a user

Edit a user's details

Edit a user's time

Delete a user

View all users or filter by role or class

Class TimeTracker

Define properties for currentUser, userManager, startTime, tracking status

Methods for time tracking:

Start tracking time for the current user

Stop tracking and calculate elapsed time

Display the current logged time

Normalize time to handle overflow of minutes and seconds

Class Menu

Define properties for UserManager and TimeTracker

Display main menu and handle user interactions:

Main menu to choose between Admin and User login, account creation, and exit

Admin menu to manage users and their times

User menu for time tracking and editing own details

Validate time input to ensure correct time format

Main

Instantiate UserManager and TimeTracker

Initialize TimeTracker with UserManager

Display the main menu and process user commands until exit

End Program

**7.Demonstrating Functionality**

**7.1 Proof of Concept**

**Menu.h**

|  |
| --- |
| /\*\*  \* @file Menu.h  \* @brief Defines the Menu class which manages user interactions and menu navigation.  \* @author  \* @date May 3, 2024  \*/  #ifndef MENU\_H  #define MENU\_H  #include "UserManager.h"  #include "TimeTracker.h"  #include <string>  **using** **namespace** std;  /\*\*  \* @class Menu  \* @brief Handles the main menu and submenus for user interactions.  \*  \* This class is responsible for displaying different menus based on the user type (admin/user) and managing the user's navigation through these menus.  \*/  **class** **Menu** {  **public:**  /\*\*  \* @brief Constructor for the Menu class.  \* @param userManager Reference to the UserManager to manage user data.  \* @param tracker Reference to the TimeTracker for tracking time.  \*/  Menu(UserManager& userManager, TimeTracker& tracker);  /\*\*  \* @brief Displays the main menu and handles user interactions.  \*/  **void** **mainMenu**();  **private:**  UserManager& userManager; ///< UserManager object to manage user data.  TimeTracker& tracker; ///< TimeTracker object to handle time tracking.  **void** **adminMenu**(); ///< Displays the admin menu and handles admin interactions.  **void** **userMenu**(); ///< Displays the user menu and handles regular user interactions.  **void** **displayMenu**(**const** string& menuTitle, **const** vector<string>& options); ///< Generic menu display function.  **int** **getChoice**(**const** string& prompt, **int** numOptions); ///< Retrieves and validates user menu choice.  **int** **getValidatedTimeInput**(**const** string& prompt); ///< Ensures the user inputs valid time data.  };  #endif // MENU\_H |

**TimeTracker.h**

|  |
| --- |
| /\*\*  \* @file TimeTracker.h  \* @brief Defines the TimeTracker class for managing time tracking for users.  \* @author alex  \* @date March 26, 2024  \*/  #ifndef TIMETRACKER\_H  #define TIMETRACKER\_H  #include "User.h"  #include "UserManager.h"  #include <ctime> // Includes standard time functions  #include <string>  /\*\*  \* @class TimeTracker  \* @brief Manages the tracking of user time sessions.  \*  \* This class is responsible for starting, stopping, and displaying the time tracking  \* of a user's session. It utilizes time\_t to track the start time of a session.  \*/  **class** **TimeTracker** {  **private:**  User\* currentUser; ///< Pointer to the current user in session.  UserManager\* userManager; ///< Pointer to the UserManager to access user management functions.  **time\_t** startTime; ///< Stores the start time of the current tracking session using time\_t instead of chrono.  **bool** tracking; ///< Flag to check if time tracking is currently active.  /\*\*  \* @brief Normalizes the time to ensure minutes and seconds are within typical range.  \* @param hours Reference to the hours to adjust.  \* @param minutes Reference to the minutes to adjust.  \* @param seconds Reference to the seconds to adjust.  \*  \* Ensures that seconds are less than 60 by rolling over excess into minutes, and similarly for minutes to hours.  \*/  **void** **normalizeTime**(**int**& hours, **int**& minutes, **int**& seconds);  **public:**  /\*\*  \* @brief Constructor, initializes pointers to nullptr.  \*/  TimeTracker() : currentUser(nullptr), userManager(nullptr) {}  /\*\*  \* @brief Sets the current user for the time tracking session.  \* @param user Pointer to the user to be set as current user.  \*/  **void** setUser(User\* user) { currentUser = user; }  /\*\*  \* @brief Sets the user manager handling the users.  \* @param um Pointer to the UserManager.  \*/  **void** setUserManager(UserManager\* um) { userManager = um; }  /\*\*  \* @brief Gets the current user involved in the time tracking session.  \* @return Pointer to the current user.  \*/  User\* getCurrentUser() **const** { **return** currentUser; }  **void** start(); ///< Starts the time tracking session.  **void** **stop**(); ///< Stops the current time tracking session.  **void** **displayTime**(); ///< Displays the accumulated time for the current session.  };  #endif // TIMETRACKER\_H |

**User.h**

|  |
| --- |
| /\*\*  \* @file User.h  \* @brief Defines the User class for managing user details in the system.  \* @author alex  \* @date March 30, 2024  \*/  #ifndef USER\_H  #define USER\_H  #include <cstring> // Includes functions for memory manipulation  #include <string>  **using** **namespace** std;  /\*\*  \* @class User  \* @brief Holds information about a system user, including credentials and time logged.  \*  \* This class stores details such as username, password, role, class information,  \* and the time duration for which the user has been active. It provides constructors  \* for initializing these values.  \*/  **class** **User** {  **public:**  **char** username[**21**]; ///< Username of the user, fixed size to ensure data consistency.  **char** password[**21**]; ///< User's password, securely stored.  **char** role[**11**]; ///< Role of the user (e.g., admin, student).  **char** userClass[**21**]; ///< Class information, relevant for student users.  **int** hours; ///< Hours part of the time logged by the user.  **int** minutes; ///< Minutes part of the time logged by the user.  **int** seconds; ///< Seconds part of the time logged by the user.  /\*\*  \* @brief Default constructor that initializes the user's properties to zero or empty.  \*  \* Initializes numeric time values to zero and character arrays to empty strings,  \* ensuring that all user properties start from a clean state.  \*/  User() : hours(**0**), minutes(**0**), seconds(**0**) {  memset(username, **0**, **sizeof**(username)); // Clear username array  memset(password, **0**, **sizeof**(password)); // Clear password array  memset(role, **0**, **sizeof**(role)); // Clear role array  memset(userClass, **0**, **sizeof**(userClass)); // Clear class information array  }  /\*\*  \* @brief Parameterized constructor for creating a user with detailed information.  \* @param u Username of the user.  \* @param p Password of the user.  \* @param r Role of the user.  \* @param cls Class of the user, optional with default empty.  \* @param h Hours logged, optional with default 0.  \* @param m Minutes logged, optional with default 0.  \* @param s Seconds logged, optional with default 0.  \*  \* This constructor initializes the user with provided details, ensuring  \* that string lengths are handled correctly to fit into fixed-size character arrays.  \*/  User(string u, string p, string r, string cls = "", **int** h = **0**, **int** m = **0**, **int** s = **0**) {  strncpy(username, u.c\_str(), **20**); // Copy username, ensuring it does not exceed buffer size  username[**20**] = '\0'; // Null terminate username array  strncpy(password, p.c\_str(), **20**); // Copy password similarly  password[**20**] = '\0'; // Null terminate password array  strncpy(role, r.c\_str(), **10**); // Copy role  role[**10**] = '\0'; // Null terminate role array  strncpy(userClass, cls.c\_str(), **20**); // Copy class information  userClass[**20**] = '\0'; // Null terminate class array  hours = h; // Set hours  minutes = m; // Set minutes  seconds = s; // Set seconds  }  };  #endif // USER\_H |

**UserManager.h**

|  |
| --- |
| /\*\*  \* @file UserManager.h  \* @brief Defines the UserManager class for managing user accounts and interactions with user data files.  \* @author alex  \* @date March 30, 2024  \*/  #ifndef USERMANAGER\_H  #define USERMANAGER\_H  #include "User.h"  #include <vector>  #include <fstream>  #include <sstream>  **using** **namespace** std;  /\*\*  \* @class UserManager  \* @brief Manages user operations such as authentication, user data retrieval, and user data persistence.  \*  \* Handles user-related tasks including adding, deleting, and editing users, as well as  \* authentication and viewing user details. It interfaces with a file system to persist user data.  \*/  **class** **UserManager** {  **private:**  vector<User> users; ///< Container for storing all user objects in memory.  fstream userFile; ///< File stream for user data file operations.  **void** **loadUsers**(); ///< Loads users from the file into the vector.  **void** **writeUsersToFile**(); ///< Writes the user data from the vector back into the file.  /\*\*  \* @brief Encrypts user data using XOR encryption.  \* @param user User object to encrypt.  \*/  **void** **encryptUser**(User& user); ///< Encrypts user data to ensure privacy.  /\*\*  \* @brief Decrypts user data using XOR decryption.  \* @param user User object to decrypt.  \*/  **void** **decryptUser**(User& user); ///< Decrypts user data to make it readable.  **public:**  UserManager(); ///< Constructor that initializes and loads users from a file.  ~UserManager(); ///< Destructor that ensures any remaining user data is saved.  /\*\*  \* @brief Retrieves a user object by username.  \* @param username The username of the user to retrieve.  \* @return Pointer to the User object if found, nullptr otherwise.  \*/  User\* **getUser**(**const** string& username);  **void** **viewUsers**(); ///< Displays all users' information.  **bool** **createUser**(); ///< Interactively creates a new user from console input.  **void** **saveUserData**(); ///< Saves all user data to the file.    /\*\*  \* @brief Authenticates a user based on username and password.  \* @param username The username to authenticate.  \* @param password The password to authenticate.  \* @return True if authentication is successful, false otherwise.  \*/  **bool** **authenticate**(string username, string password);  /\*\*  \* @brief Authenticates a user with an additional role check.  \* @param username The username to authenticate.  \* @param password The password to authenticate.  \* @param role The role to validate against.  \* @return True if authentication and role check are successful, false otherwise.  \*/  **bool** **authenticateWithRole**(**const** string& username, **const** string& password, **const** string& role);  /\*\*  \* @brief Checks if a user exists by username.  \* @param username The username to check.  \* @return True if the user exists, false otherwise.  \*/  **bool** **userExists**(**const** string& username);  /\*\*  \* @brief Adds a new user with detailed information.  \* @param username The new user's username.  \* @param password The new user's password.  \* @param role The new user's role.  \* @param userClass The class information, relevant for students.  \* @param hours Initial hours logged, optional.  \* @param minutes Initial minutes logged, optional.  \* @param seconds Initial seconds logged, optional.  \*/  **void** **addUser**(string username, string password, string role, string userClass, **int** hours = **0**, **int** minutes = **0**, **int** seconds = **0**);  /\*\*  \* @brief Deletes a user by username.  \* @param username The username of the user to delete.  \*/  **void** **deleteUser**(string username);  /\*\*  \* @brief Edits existing user details.  \* @param oldUsername The current username of the user.  \* @param newUsername The new username, if changing.  \* @param newPassword The new password, if changing.  \* @param newRole The new role, if changing.  \* @param newUserClass The new class information, if applicable.  \*/  **void** **editUser**(string oldUsername, string newUsername, string newPassword, string newRole = "", string newUserClass = "");  /\*\*  \* @brief Edits the time logged for a specific user.  \* @param username The username of the user whose time is being edited.  \* @param hours The new hours to set.  \* @param minutes The new minutes to set.  \* @param seconds The new seconds to set.  \*/  **void** **editUserTime**(**const** string& username, **int** hours, **int** minutes, **int** seconds);  /\*\*  \* @brief Views users filtered by their role.  \* @param role The role to filter users by (e.g., "student", "instructor").  \*/  **void** **viewUsersByRole**(**const** string& role);  /\*\*  \* @brief Views users filtered by their class.  \* @param userClass The class to filter users by.  \*/  **void** **viewUsersByClass**(**const** string& userClass);  };  #endif // USERMANAGER\_H |

**Menu.cpp**

|  |
| --- |
| /\*\*  \* @file Menu.cpp  \* @brief Implementation for the Menu class which handles all menu-driven interactions.  \* @author Aleksandar Videv  \* @date May 3, 2024  \*/  #include "TimeTracker.h"  #include "Menu.h"  #include <iostream>  #include <string>  #include <vector>  #include <sstream> // Required for istringstream  #include <iomanip> // Required for get\_time  **using** **namespace** std;  /\*\*  \* @brief Constructs the Menu object.  \* @param userManager A reference to UserManager to manage user data.  \* @param tracker A reference to TimeTracker to manage time tracking.  \*/  Menu::Menu(UserManager& userManager, TimeTracker& tracker) : userManager(userManager), tracker(tracker) {}  /\*\*  \* @brief Displays the main menu and handles user input for various functionalities.  \*/  **void** Menu::mainMenu() {  string input;  **bool** exit = false;  // Loop until the user chooses to exit the program  **while** (!exit) {  cout << "**\n**Main Menu**\n**"  << "1. Admin Login**\n**"  << "2. User Login**\n**"  << "3. Create Account**\n**"  << "4. Exit**\n**"  << "Enter choice: ";  getline(cin, input);  // Handle user choice using a switch statement  **switch** (input[**0**]) {  **case** '1': // Admin login  **case** '2': // User login  {  string username, password;  cout << (input[**0**] == '1' ? "Admin" : "User") << " Login**\n**"  << "Enter username: ";  getline(cin, username);  cout << "Enter password: ";  getline(cin, password);  // Authenticate user based on role (admin or user)  **bool** isAuthenticated = userManager.authenticateWithRole(username, password, input[**0**] == '1' ? "instructor" : "student");  **if** (isAuthenticated) {  // Fetch user details and set in tracker if authenticated  User\* loggedInUser = userManager.getUser(username);  tracker.setUser(loggedInUser);  // Redirect to appropriate menu based on role  **if** (input[**0**] == '1') {  adminMenu(); // Access admin-specific functions  } **else** {  userMenu(); // Access general user functions  }  } **else** {  cout << "Incorrect credentials or access level.**\n**";  }  **break**;  }  **case** '3': // Create Account  // Initiate user creation process  userManager.createUser();  **break**;  **case** '4': // Exit  // Set flag to true to exit loop and end the program  cout << "Exiting program.**\n**";  exit = true;  **break**;  **default:**  // Handle unexpected input  cout << "Invalid choice. Please try again.**\n**";  }  }  }  /\*\*  \* @brief Displays the administrative menu for admin operations like user management.  \*/  **void** Menu::adminMenu() {  string choice;  **do** {  cout << "**\n**Admin Menu**\n**"  << "1. View Users**\n**"  << "2. Add User**\n**"  << "3. Delete User**\n**"  << "4. Edit User**\n**"  << "5. Edit User Time**\n**"  << "0. Exit to Main Menu**\n**"  << "Enter choice: ";  getline(cin, choice);  cout << endl;  // Handle the choice for different administrative tasks  **switch** (choice[**0**]) {  **case** '1': // View Users  cout << "1. View All Users**\n**"  << "2. View by Role**\n**"  << "3. View by Class**\n**" // Added option to view by class  << "Enter choice for view: ";  getline(cin, choice);  **if** (choice == "1") {  userManager.viewUsers();  } **else** **if** (choice == "2") {  cout << "Choose role (s = Student, i = Instructor): ";  string role;  getline(cin, role);  userManager.viewUsersByRole(role); // View users by normalized role  } **else** **if** (choice == "3") {  cout << "Enter class to filter by: "; // Prompt for class  string userClass;  getline(cin, userClass);  userManager.viewUsersByClass(userClass); // View users by class  }  **break**;  **case** '2': // Add User  userManager.createUser();  **break**;  **case** '3': // Delete User  {  string username;  cout << "Enter username to delete: ";  getline(cin, username);  userManager.deleteUser(username);  **if** (userManager.userExists(username)) {  cout << "User deleted successfully.**\n**";  }  }  **break**;  **case** '4': //Edit User  {  string username, newUsername, newPassword, newRole, newUserClass;  **char** roleChoice;  cout << "Enter username to edit: ";  getline(cin, username);  **if** (!userManager.userExists(username)) {  cout << "User not found.**\n**";  **break**;  }  cout << "Enter new username (leave empty if unchanged): ";  getline(cin, newUsername);  cout << "Enter new password (leave empty if unchanged): ";  getline(cin, newPassword);  // Adding a choice for the role  cout << "Choose new role (1 for Student, 2 for Instructor, leave empty if unchanged): ";  getline(cin, newRole);  **bool** promptForClass = false;  **if** (!newRole.empty()) {  roleChoice = newRole[**0**];  **switch** (roleChoice) {  **case** '1':  newRole = "student";  promptForClass = true; // Prompt for class if role is student  **break**;  **case** '2':  newRole = "instructor";  newUserClass = ""; // Clear class when role is instructor  **break**;  **default:**  cout << "Invalid role selected. No changes to role will be made.**\n**";  newRole = ""; // Reset to empty to avoid changing the role  promptForClass = true; // Still prompt for class if the role change is invalid  **break**;  }  }  // Prompt for class if role is student or role change is skipped  **if** (promptForClass) {  cout << "Enter new class (leave empty if unchanged): ";  getline(cin, newUserClass);  }  userManager.editUser(username, newUsername, newPassword, newRole, newUserClass);  }  **break**;  **case** '5': // Edit User Time  {  string username;  cout << "Enter username of the user to edit time: ";  getline(cin, username);  // Check if user exists before asking for time details  **if** (!userManager.userExists(username)) {  cout << "User not found.**\n**";  **break**; // Exit if user doesn't exist  }  **int** hours = getValidatedTimeInput("Enter new hours (0-9999): ");  **int** minutes = getValidatedTimeInput("Enter new minutes (0-59): ");  **int** seconds = getValidatedTimeInput("Enter new seconds (0-59): ");  userManager.editUserTime(username, hours, minutes, seconds);  }  **break**;  **case** '0':  **return**; // Exit admin menu  **default:**  cout << "Invalid choice. Please try again.**\n**";  }  } **while** (true);  }  /\*\*  \* @brief Displays the user menu for regular user functions.  \*/  **void** Menu::userMenu() {  string choice;  // Continuously display the menu until the user decides to exit  **do** {  cout << "**\n**User Menu**\n**"  << "1. Start Timer**\n**"  << "2. Stop Timer**\n**"  << "3. Display Logged Time**\n**"  << "4. Edit Details**\n**"  << "0. Exit to Main Menu**\n**"  << "Enter choice: ";  getline(cin, choice);    // Handle the user's menu selection  **switch** (choice[**0**]) {  **case** '1':  // Starts the time tracking for the current session  tracker.start();  **break**;  **case** '2':  // Stops the time tracking and logs the session duration  tracker.stop();  **break**;  **case** '3':  // Displays the total time logged in the current session  tracker.displayTime();  **break**;  **case** '4':  {  // Attempts to retrieve the current logged-in user  User\* user = tracker.getCurrentUser();  **if** (user) {  string newUsername, newPassword;  cout << "Enter new username (leave empty if unchanged): ";  getline(cin, newUsername);  cout << "Enter new password (leave empty if unchanged): ";  getline(cin, newPassword);  // Submits the updated user details to the UserManager  userManager.editUser(user->username, newUsername, newPassword);  } **else** {  // Notifies if no user is currently logged in  cout << "No user logged in.**\n**";  }  }  **break**;  **case** '0':  // Exits the user menu and returns to the main menu  **return**;  **default**:  // Handles invalid choices entered by the user  cout << "Invalid choice. Please try again.**\n**";  }  } **while** (true);  }  /\*\*  \* @brief Prompts the user for time input and validates it.  \* @param prompt The prompt displayed to the user asking for input.  \* @return The validated time entered by the user.  \*/  **int** Menu::getValidatedTimeInput(**const** string& prompt) {  string input;  **int** time;  // Loop indefinitely until a valid time is entered  **while** (true) {  cout << prompt;  getline(cin, input); // Read input from user    try {  time = stoi(input); // Attempt to convert the user input to an integer  // Check if the input falls within the valid range for hours or minutes/seconds  **if** (prompt.find("hours") != string::npos && time >= **0** && time <= **9999**) {  // If the prompt is for hours and the value is within the valid range, break the loop  **break**;  } **else** **if** (time >= **0** && time <= **59**) {  // If the prompt is for minutes/seconds and the value is within the valid range, break the loop  **break**;  } **else** {  // If the input is not within any valid range, prompt the user again  cout << "Invalid time. Please enter a valid number within the range.**\n**";  }  } **catch** (**const** invalid\_argument& e) {  // Handle cases where conversion to integer fails  cout << "Invalid input. Please enter a valid number.**\n**";  } **catch** (**const** out\_of\_range& e) {  // Handle cases where the integer is out of the acceptable range  cout << "Number out of range. Please enter a smaller number.**\n**";  }  }  **return** time; // Return the validated time value  } |

**TimeTracker.cpp**

|  |
| --- |
| /\*\*  \* @file TimeTracker.cpp  \* @brief Manages the tracking of time for users, including starting, stopping, and displaying logged time.  \* @author alex  \* @date March 26, 2024  \*/  #include "TimeTracker.h"  #include <iostream>  #include <ctime>  **using** **namespace** std;  /\*\*  \* @brief Starts the timer for the current user session.  \*  \* This function records the start time if no timer is currently running and a user is logged in.  \* If the timer is already running or no user is logged in, it outputs an appropriate message.  \*/  **void** TimeTracker::start() {  **if** (!currentUser) {  // Check if a user is logged in before starting the timer  cout << "No user logged in.**\n**";  **return**;  }  **if** (tracking) {  // Prevent starting a new timer if one is already running  cout << "Error: Timer is already running.**\n**";  **return**;  }  startTime = time(nullptr); // Capture the current time as the start time  tracking = true; // Set tracking to true to indicate the timer is running  cout << "Timer started.**\n**";  }  /\*\*  \* @brief Stops the timer and logs the time elapsed since it was started.  \*  \* If the timer is running and a user is logged in, this function calculates the elapsed time, updates the user's  \* time record, and saves the changes. If no timer is running or no user is logged in, it outputs an error message.  \*/  **void** TimeTracker::stop() {  **if** (!tracking || !currentUser) {  // Ensure that a timer is running and a user is logged in before stopping the timer  cout << "Timer not started or no user logged in.**\n**";  **return**;  }  **time\_t** endTime = time(nullptr); // Capture the current time as the end time  **double** elapsed\_seconds = difftime(endTime, startTime); // Calculate elapsed time in seconds  **int** elapsedHours = **static\_cast**<**int**>(elapsed\_seconds) / **3600**;  **int** elapsedMinutes = (**static\_cast**<**int**>(elapsed\_seconds) / **60**) % **60**;  **int** elapsedSeconds = **static\_cast**<**int**>(elapsed\_seconds) % **60**;  // Update the user's logged time  currentUser->hours += elapsedHours;  currentUser->minutes += elapsedMinutes;  currentUser->seconds += elapsedSeconds;  // Normalize the time to ensure proper time format  normalizeTime(currentUser->hours, currentUser->minutes, currentUser->seconds);  userManager->saveUserData(); // Save the updated user data  tracking = false; // Set tracking to false as the timer is stopped  cout << "Timer stopped. Time logged: "  << elapsedHours << "h " << elapsedMinutes << "m " << elapsedSeconds << "s**\n**";  }  /\*\*  \* @brief Displays the total logged time for the current user.  \*  \* Outputs the total time logged by the user both from past sessions and any ongoing session.  \* If no user is logged in, it outputs an error message.  \*/  **void** TimeTracker::displayTime() {  **if** (!currentUser) {  // Check if a user is logged in before displaying time  cout << "No user logged in.**\n**";  **return**;  }  **int** displayHours = currentUser->hours;  **int** displayMinutes = currentUser->minutes;  **int** displaySeconds = currentUser->seconds;  **if** (tracking) {  // Calculate additional time if the timer is currently running  **time\_t** currentTime = time(nullptr);  **double** elapsed\_seconds = difftime(currentTime, startTime);  displaySeconds += **static\_cast**<**int**>(elapsed\_seconds) % **60**;  displayMinutes += (**static\_cast**<**int**>(elapsed\_seconds) / **60**) % **60**;  displayHours += **static\_cast**<**int**>(elapsed\_seconds) / **3600**;  // Normalize time to handle overflow from seconds to minutes and minutes to hours  normalizeTime(displayHours, displayMinutes, displaySeconds);  }  // Output the total time logged for the user, including the current session if applicable  cout << "Logged Time: " << displayHours << "h " << displayMinutes << "m " << displaySeconds << "s**\n**";  }  /\*\*  \* @brief Normalizes the time values, rolling over seconds to minutes and minutes to hours.  \*  \* Ensures that seconds and minutes do not exceed their maximum values by converting excess into the next highest unit.  \* @param hours Reference to the hours to be normalized.  \* @param minutes Reference to the minutes to be normalized.  \* @param seconds Reference to the seconds to be normalized.  \*/  **void** TimeTracker::normalizeTime(**int**& hours, **int**& minutes, **int**& seconds) {  minutes += seconds / **60**; // Convert excess seconds into minutes  seconds %= **60**; // Keep the remainder of seconds after dividing by 60  hours += minutes / **60**; // Convert excess minutes into hours  minutes %= **60**; // Keep the remainder of minutes after dividing by 60  } |

**UserManager.cpp**

|  |
| --- |
| /\*  \* @file UserManager.cpp  \* @brief Implementation of the UserManager class that handles user management tasks like adding,  \* deleting, and editing users, as well as loading and saving user data to a file.  \* @author alex  \* @date March 30 , 2024, 7:23 AM  \*/  #include "UserManager.h"  #include <iostream>  #include <fstream>  #include <sstream>  #include <algorithm> // Include for remove\_if  #include <cstring> // for strlen, memset  **using** **namespace** std;  /\*\*  \* @brief Destructor for UserManager.  \*  \* Ensures all user data is saved before the object is destroyed.  \*/  UserManager::~UserManager() {  saveUserData(); // Save any remaining data.  }  /\*\*  \* @brief Constructor for UserManager.  \*  \* Opens the users.dat file or creates it if it does not exist, then loads existing user data.  \*/  UserManager::UserManager() {  userFile.open("users.dat", ios::binary | ios::in | ios::out | ios::ate); // Attempt to open an existing file.  **if** (!userFile.is\_open()) {  std::cerr << "Failed to open users.dat. Attempting to create a new file." << std::endl;  userFile.clear(); // Clear any error flags.  userFile.open("users.dat", ios::binary | ios::out | ios::trunc); // Create a new file.  userFile.close(); // Close the newly created file.  userFile.open("users.dat", ios::binary | ios::in | ios::out); // Reopen with read/write permissions.  }  **if** (!userFile) {  std::cerr << "Error: Unable to open or create the users file!" << std::endl;  exit(**1**); // Exit if still unable to open or create the file.  }  loadUsers(); // Load users from the file.  }  /\*\*  \* @brief Save user data to the file.  \*  \* This method encrypts each user's data and writes it to the binary file.  \* If the file is not open, it outputs an error message.  \*/  **void** UserManager::saveUserData() {  **if** (userFile.is\_open()) {  userFile.seekp(**0**); // Start writing from the beginning of the file.  userFile.clear(); // Clear any error flags that might be set.  **for** (User& user : users) {  encryptUser(user); // Encrypt the user data.  userFile.write(**reinterpret\_cast**<**const** **char**\*>(&user), **sizeof**(User)); // Write the encrypted data to the file.  decryptUser(user); // Decrypt the user data to restore original values.  }  userFile.flush(); // Ensure all data is written to the disk.  } **else** {  cerr << "File not open for writing." << endl; // Error handling if file isn't open.  }  }  /\*\*  \* @brief Encrypts user data using XOR encryption.  \*  \* @param user Reference to user object to be encrypted.  \*/  **void** UserManager::encryptUser(User& user) {  **char** key = 'K'; // Encryption key, simple for demonstration.  **for** (**size\_t** i = **0**; i < **sizeof**(User); ++i) {  **reinterpret\_cast**<**char**\*>(&user)[i] ^= key; // Apply XOR for each byte of user data.  }  }  /\*\*  \* @brief Decrypts user data.  \*  \* This function uses the same encryptUser function to decrypt because XOR is its own inverse.  \* @param user Reference to user object to be decrypted.  \*/  **void** UserManager::decryptUser(User& user) {  encryptUser(user); // Decrypt by re-applying the XOR encryption.  }  /\*\*  \* @brief Writes encrypted users data to a file.  \*  \* This method handles the file operations required to write all user data after encrypting it.  \*/  **void** UserManager::writeUsersToFile() {  userFile.close(); // Ensure the file is not already open.  userFile.open("users.dat", ios::binary | ios::out | ios::trunc); // Open file in binary mode to write from scratch.  **if** (!userFile.is\_open()) {  std::cerr << "Failed to open file for writing." << std::endl;  **return**;  }  **for** (**auto**& user : users) {  encryptUser(user); // Encrypt each user's data.  userFile.write(**reinterpret\_cast**<**const** **char**\*>(&user), **sizeof**(User)); // Write the encrypted data.  decryptUser(user); // Decrypt the data to retain the original state in memory.  }  userFile.flush(); // Flush the stream to ensure all data is written.  userFile.close(); // Close the file after writing.  userFile.open("users.dat", ios::binary | ios::in | ios::out); // Reopen file for both reading and writing.  }  /\*\*  \* @brief Loads users data from a file.  \*  \* This function reads encrypted user data from a file, decrypts it, and adds it to the users vector.  \* If the file fails to open, it prints an error message.  \*/  **void** UserManager::loadUsers() {  std::ifstream file("users.dat", ios::binary); // Open the file in binary read mode.  **if** (!file) {  std::cerr << "Failed to open file for reading." << std::endl; // Error handling if file isn't open.  **return**;  }  User user;  **while** (file.read(**reinterpret\_cast**<**char**\*>(&user), **sizeof**(User))) {  decryptUser(user); // Decrypt user data after reading from the file.  users.push\_back(user); // Add the decrypted user to the vector of users.  }  file.close(); // Close the file after all users have been read.  }  /\*\*  \* @brief Retrieves a user object by username.  \*  \* Searches for a user by username in the list of users. Returns a pointer to the User object if found,  \* otherwise returns nullptr.  \* @param username The username of the user to find.  \* @return A pointer to the User object or nullptr if not found.  \*/  User\* UserManager::getUser(**const** string& username) {  **for** (**auto**& user : users) {  **if** (strcmp(user.username, username.c\_str()) == **0**) {  **return** &user; // Return the address of the user if found  }  }  **return** nullptr; // Return nullptr if no user is found  }  /\*\*  \* @brief Edits the time logged for a specific user.  \*  \* First normalizes the provided time values (hours, minutes, and seconds) and then updates the  \* specified user's logged time if they exist in the system.  \* @param username The username of the user whose time needs updating.  \* @param hours The number of hours to set.  \* @param minutes The number of minutes to set.  \* @param seconds The number of seconds to set.  \*/  **void** UserManager::editUserTime(**const** string& username, **int** hours, **int** minutes, **int** seconds) {  // Normalize the entered time first  minutes += seconds / **60**;  seconds %= **60**;  hours += minutes / **60**;  minutes %= **60**;  // Check if the user exists and update their time  **for** (**auto**& user : users) {  **if** (user.username == username) {  user.hours = hours;  user.minutes = minutes;  user.seconds = seconds;  writeUsersToFile(); // Save changes to file  cout << "User time updated successfully.**\n**";  **return**; // Exit the function after successful update  }  }  cout << "User not found. No time updated.**\n**"; // Notify if user not found  }  /\*\*  \* @brief Authenticates a user with a specific role.  \*  \* Checks if there exists a user with the given username, password, and role.  \* @param username The username of the user.  \* @param password The password of the user.  \* @param role The role of the user.  \* @return True if such a user exists, otherwise false.  \*/  **bool** UserManager::authenticateWithRole(**const** string& username, **const** string& password, **const** string& role) {  **for** (**const** **auto**& user : users) {  **if** (strcmp(user.username, username.c\_str()) == **0** &&  strcmp(user.password, password.c\_str()) == **0** &&  strcmp(user.role, role.c\_str()) == **0**) {  **return** true; // User found and matches all credentials  }  }  **return** false; // No matching user found  }  /\*\*  \* @brief Checks if a user exists based on username.  \*  \* Searches for a user in the list by the username.  \* @param username The username to check against the user list.  \* @return True if the user exists, otherwise false.  \*/  **bool** UserManager::userExists(**const** string& username) {  **return** any\_of(users.begin(), users.end(), [&username](**const** User& user) {  **return** strcmp(user.username, username.c\_str()) == **0**; // Compare the current user's username with the given one  });  }  /\*\*  \* @brief Displays all registered users along with their details.  \*  \* Lists every user's username, role, class (if applicable), and time logged. It only displays  \* class and time for students.  \*/  **void** UserManager::viewUsers() {  **if** (users.empty()) {  cout << "No users available.**\n**"; // Inform if no users are registered  **return**;  }  **for** (**const** **auto**& user : users) {  cout << "Username: " << user.username << ", Role: " << user.role;  **if** (strcmp(user.role, "student") == **0**) {  cout << ", Class: " << user.userClass; // Display class information if the user is a student  cout << ", Time: " << user.hours << "h " << user.minutes << "m " << user.seconds << "s";  }  cout << "**\n**";  }  }  /\*\*  \* @brief Filters and displays users by their role.  \*  \* Allows viewing of users grouped by a specified role such as 'student' or 'instructor'.  \* @param inputRole The role to filter the users by.  \*/  **void** UserManager::viewUsersByRole(**const** string& inputRole) {  string role;  // Normalize input to handle different case inputs and partial inputs  **if** (inputRole == "s" || inputRole == "S" || inputRole == "student" || inputRole == "Student" || inputRole == "STUDENT") {  role = "student";  } **else** **if** (inputRole == "i" || inputRole == "I" || inputRole == "instructor" || inputRole == "Instructor" || inputRole == "INSTRUCTOR") {  role = "instructor";  } **else** {  cout << "Invalid role input. Please enter 's' for Student or 'i' for Instructor.**\n**";  **return**;  }  **bool** found = false;  **for** (**const** **auto**& user : users) {  **if** (user.role == role) {  found = true;  cout << "Username: " << user.username << ", Role: " << user.role;  **if** (role == "student") {  cout << ", Class: " << user.userClass; // Display class if student  }  cout << ", Time: " << user.hours << "h " << user.minutes << "m " << user.seconds << "s**\n**";  }  }  **if** (!found) {  cout << "No users found with role " << role << ".**\n**"; // Inform if no users are found with the specified role  }  }  /\*\*  \* @brief Filters and displays users by their class.  \*  \* Allows viewing of users grouped by a specified class, typically applicable to students.  \* @param userClass The class to filter the users by.  \*/  **void** UserManager::viewUsersByClass(**const** string& userClass) {  **bool** found = false;  **for** (**const** **auto**& user : users) {  **if** (user.userClass == userClass) {  found = true;  cout << "Username: " << user.username << ", Class: " << user.userClass  << ", Role: " << user.role << ", Time: "  << user.hours << "h " << user.minutes << "m " << user.seconds << "s**\n**";  }  }  **if** (!found) {  cout << "No users found in class " << userClass << ".**\n**"; // Notify if no users are found in the specified class  }  }  /\*\*  \* @brief Adds a new user to the system if they do not already exist.  \*  \* Creates a new user with specified details and saves them to the system, ensuring no username duplication.  \* @param username The username for the new user.  \* @param password The password for the new user.  \* @param role The role of the new user (e.g., student, instructor).  \* @param userClass The class of the new user, relevant for students.  \* @param hours Initial hours logged (optional).  \* @param minutes Initial minutes logged (optional).  \* @param seconds Initial seconds logged (optional).  \*/  **void** UserManager::addUser(string username, string password, string role, string userClass, **int** hours, **int** minutes, **int** seconds) {  **if** (!userExists(username)) {  User newUser(username, password, role, userClass, hours, minutes, seconds);  users.push\_back(newUser);  writeUsersToFile(); // Persist the new user data  } **else** {  cout << "An account with that username already exists.**\n**"; // Notify if the username is already taken  }  }  /\*\*  \* @brief Interactively creates a new user based on console input.  \*  \* Prompts for and receives user details from the console, then adds the user to the system if the username is not taken.  \* @return True if the user was created successfully, otherwise false.  \*/  **bool** UserManager::createUser() {  string username, password, role, userClass, input;  cout << "Create a new account.**\n**";  cout << "Enter username: ";  getline(cin, username);  **if** (userExists(username)) {  cout << "An account with that username already exists. Please choose a different username.**\n**";  **return** false;  }  cout << "Enter password: ";  getline(cin, password);  cout << "Enter role (S for Student, I for Instructor): ";  getline(cin, input);  **char** roleChoice = input.length() > **0** ? input[**0**] : ' ';  **switch** (roleChoice) {  **case** 'S':  **case** 's':  role = "student";  cout << "Enter class: ";  getline(cin, userClass); // Prompt for class if the role is student  **break**;  **case** 'I':  **case** 'i':  role = "instructor";  **break**;  **default:**  cout << "Invalid role. Only 'S' for Student or 'I' for Instructor are allowed.**\n**";  **return** false;  }  addUser(username, password, role, userClass); // Add the new user  cout << "Account successfully created as " << role << ".**\n**";  **return** true;  }  /\*\*  \* @brief Edits an existing user's details.  \*  \* Allows modification of username, password, role, and class based on provided inputs. Ensures the new username is not already taken.  \* @param oldUsername The current username of the user to be edited.  \* @param newUsername The new username to update to, if provided.  \* @param newPassword The new password to update to, if provided.  \* @ant'semail.comwRole The new role to update to, if provided.  \* @param newUserClass The new class to update to, if applicable.  \*/  **void** UserManager::editUser(string oldUsername, string newUsername, string newPassword, string newRole, string newUserClass) {  **for** (**auto**& user : users) {  **if** (strcmp(user.username, oldUsername.c\_str()) == **0**) {  **if** (!newUsername.empty() && !userExists(newUsername)) {  strncpy(user.username, newUsername.c\_str(), **20**); // Update username if new one is not taken  user.username[**20**] = '\0';  }  **if** (!newPassword.empty()) {  strncpy(user.password, newPassword.c\_str(), **20**); // Update password  user.password[**20**] = '\0';  }  **if** (!newRole.empty()) {  strncpy(user.role, newRole.c\_str(), **10**); // Update role  user.role[**10**] = '\0';  }  **if** (!newUserClass.empty()) {  strncpy(user.userClass, newUserClass.c\_str(), **20**); // Update class for students  user.userClass[**20**] = '\0';  }  writeUsersToFile(); // Persist changes to disk  cout << "User details updated successfully.**\n**";  **return**;  }  }  cout << "User not found. No changes made.**\n**"; // Notify if the specified user does not exist  }  /\*\*  \* @brief Deletes a user from the system.  \*  \* Removes a user with the specified username from the list and updates the user file.  \* @param username The username of the user to delete.  \*/  **void** UserManager::deleteUser(string username) {  **auto** it = remove\_if(users.begin(), users.end(), [&username](**const** User& user) {  **return** username == user.username; // Find the user to delete  });  **if** (it != users.end()) {  users.erase(it, users.end()); // Remove the user from the list  writeUsersToFile(); // Update the file after removal  cout << "User deleted successfully." << endl;  } **else** {  cout << "User not found. No user deleted." << endl;  }  } |

**Main.cpp**

|  |
| --- |
| /\*\*  \* @file main.cpp  \* @brief Entry point for the Time Tracker application.  \*  \* Initializes the main components of the application including UserManager, TimeTracker, and Menu.  \* It sets up the relationships between these components and starts the user interaction process through the main menu.  \* @author Aleksandar Videv  \* @date March 12, 2024  \*/  #include "Menu.h"  #include "UserManager.h"  #include "TimeTracker.h"  /\*\*  \* @brief The main function that serves as the entry point of the application.  \*  \* Sets up the user manager and time tracker, links them together, and launches the main menu.  \* @return Returns 0 upon successful completion.  \*/  **int** **main**() {  UserManager userManager; // Instantiate the UserManager to manage user data.  TimeTracker tracker; // Create a TimeTracker to handle timing functionality.  tracker.setUserManager(&userManager); // Associate the UserManager with the TimeTracker.  Menu menu(userManager, tracker); // Create the main Menu with references to userManager and tracker.  menu.mainMenu(); // Display the main menu and handle user interactions.  **return** **0**; // Return 0 to indicate successful completion of the program.  } |

**7.2 Testing Results**

|  |
| --- |
| Main Menu  1. Admin Login  2. User Login  3. Create Account  4. Exit  Enter choice: 1  Admin Login  Enter username: alex  Enter password: alex  Incorrect credentials or access level.  Main Menu  1. Admin Login  2. User Login  3. Create Account  4. Exit  Enter choice: 3  Create a new account.  Enter username: alex  Enter password: alex  Enter role (S for Student, I for Instructor): i  Account successfully created as instructor.  Main Menu  1. Admin Login  2. User Login  3. Create Account  4. Exit  Enter choice: 3  Create a new account.  Enter username: jake  Enter password: 123  Enter role (S for Student, I for Instructor): s  Enter class: cis12  Account successfully created as student.  Main Menu  1. Admin Login  2. User Login  3. Create Account  4. Exit  Enter choice: 1  Admin Login  Enter username: alex  Enter password: alex  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 1  Username: z, Role: student, Class: 12, Time: 0h 0m 0s  Username: a, Role: instructor  Username: x, Role: student, Class: 1, Time: 0h 0m 0s  Username: alex, Role: instructor  Username: jake, Role: student, Class: cis12, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 2  Choose role (s = Student, i = Instructor): s  Username: z, Role: student, Class: 12, Time: 0h 0m 0s  Username: x, Role: student, Class: 1, Time: 0h 0m 0s  Username: jake, Role: student, Class: cis12, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 3  Enter username to delete: z  User deleted successfully.  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 1  Username: a, Role: instructor  Username: x, Role: student, Class: 1, Time: 0h 0m 0s  Username: alex, Role: instructor  Username: jake, Role: student, Class: cis12, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 4  Enter username to edit: x  Enter new username (leave empty if unchanged): Bob  Enter new password (leave empty if unchanged): bob123  Choose new role (1 for Student, 2 for Instructor, leave empty if unchanged): 1  Enter new class (leave empty if unchanged): cis3  User details updated successfully.  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 1  Username: a, Role: instructor  Username: Bob, Role: student, Class: cis3, Time: 0h 0m 0s  Username: alex, Role: instructor  Username: jake, Role: student, Class: cis12, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 3  Enter username to delete: a  User deleted successfully.  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 3  Enter class to filter by: cis3  Username: Bob, Class: cis3, Role: student, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 1  Username: Bob, Role: student, Class: cis3, Time: 0h 0m 0s  Username: alex, Role: instructor  Username: jake, Role: student, Class: cis12, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 5  Enter username of the user to edit time: bob  User not found.  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 5  Enter username of the user to edit time: Bob  Enter new hours (0-9999): 1323123  Invalid time. Please enter a valid number within the range.  Enter new hours (0-9999): 32  Enter new minutes (0-59): 23  Enter new seconds (0-59): 3  User time updated successfully.  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 1  1. View All Users  2. View by Role  3. View by Class  Enter choice for view: 1  Username: Bob, Role: student, Class: cis3, Time: 32h 23m 3s  Username: alex, Role: instructor  Username: jake, Role: student, Class: cis12, Time: 0h 0m 0s  Admin Menu  1. View Users  2. Add User  3. Delete User  4. Edit User  5. Edit User Time  0. Exit to Main Menu  Enter choice: 0  Main Menu  1. Admin Login  2. User Login  3. Create Account  4. Exit  Enter choice: 2  User Login  Enter username: jake  Enter password: 123  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 1  Timer started.  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 3  Logged Time: 0h 0m 2s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 3  Logged Time: 0h 0m 4s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 2  Timer stopped. Time logged: 0h 0m 6s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 3  Logged Time: 0h 0m 6s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 1  Timer started.  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 1  Error: Timer is already running.  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 3  Logged Time: 0h 0m 12s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 3  Logged Time: 0h 0m 13s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 2  Timer stopped. Time logged: 0h 0m 7s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 4  Enter new username (leave empty if unchanged): jake  Enter new password (leave empty if unchanged): jake  User details updated successfully.  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 3  Logged Time: 0h 0m 13s  User Menu  1. Start Timer  2. Stop Timer  3. Display Logged Time  4. Edit Details  0. Exit to Main Menu  Enter choice: 0  Main Menu  1. Admin Login  2. User Login  3. Create Account  4. Exit  Enter choice: 4  Exiting program.  RUN SUCCESSFUL (total time: 5m 50s) |