TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

Proposal

On

C-Pay: An E-wallet Application Utilizing C-Programming

Submitted By:

Aswin Kandel (THA081BCT004)

Dikesh Manandhar (THA081BCT008)

Kishan Kumar Shah (THA081BCT014)

Pujag Dallakoti (THA081BCT024)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

March, 2025

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Project Report

On

C-Pay: An E-wallet Application Utilizing C-Programming

Submitted By:

Aswin Kandel (THA081BCT004)

Dikesh Manandhar (THA081BCT008)

Kishan Kumar Shah (THA081BCT014)

Pujag Dallakoti (THA081BCT024)

Submitted To:

Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor's Degree in Electronics and Communication Engineering.

Under the Supervision of

Er. Prajwol Pakka

March, 2025

DECLARATION

We hereby declare that the report of the project entitled "**C-Pay: An E-wallet Application Utilizing C-Programming**" which is being submitted to the **Department of Electronics and Computer Engineering, Institute of Engineering, Thapathali Campus**, in partial fulfillment of the requirements for the award of the Degree of Bachelor in **Computer Technology (BCT)**, is a bonafide report of the work carried out by us. The materials contained in this report have not been submitted to any University or Institution for the award of any degree and we are the only author of this complete work and no sources other than the listed here have been used in this work.

Aswin Kandel (THA081BCT004) _____

Dikesh Manandhar (THA081BCT008) _____

Kishan Kumar Shah (THA081BCT014) _____

Pujag Dallakoti (THA081BCT024) _____

Date: March, 2025

CERTIFICATE OF APPROVAL

The undersigned certify that they have read and recommended to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, a minor project work entitled "**C-Pay: An E-wallet Application Utilizing C-Programming**" submitted by **Aswin Kandel, Dikesh Manandhar, Kishan Kumar Shah** and **Pujag Dallakoti** in partial fulfillment for the award of Bachelor's Degree in Electronics and Communication Engineering. The Project was carried out under special supervision and within the time frame prescribed by the syllabus.

We found the students to be hardworking, skilled and ready to undertake any related work to their field of study and hence we recommend the award of partial fulfillment of Bachelor's degree of Electronics and Communication Engineering.

Project Supervisor
Er. Prajwol Pakka
Department of Electronics and Computer Engineering, Thapathali Campus

External Examiner

Project Co-Ordinator
Department of Electronics and Computer Engineering, Thapathali Campus

Mr. Umesh Kanta Ghimire
Head of the Department,
Department of Electronics and Computer Engineering, Thapathali Campus

March, 2025

COPYRIGHT

ACKNOWLEDGEMENT

We express our sincere thanks to the Institute of Engineering, Thapathali Campus, for giving us the chance to improve our knowledge and abilities through this project. We extend our sincere gratitude to the Department of Electronics and Computer Engineering for their ongoing assistance and for providing us with essential resources and information. We are particularly grateful to our supervisors, Er. Prajwol Pakka, along with Er. Anup Shrestha, for their essential guidance, support, and expert insights throughout this project's duration. Their guidance was vital for the successful fulfillment of our project.

We would also like to convey our gratitude to our colleagues and our supervisors for their support and encouragement. This project was more than just an academic task; it was an experience that enabled us to use our practical skills in System Development, particularly in the C programming language. The educational experience and practical application have been incredibly advantageous, and we appreciate everyone who contributed to this learning journey.

Aswin Kandel (THA081BCT004)

Dikesh Manandhar (THA081BCT008)

Kishan Kumar Shah (THA081BCT014)

Pujag Dallakoti (THA081BCT024)

ABSTRACT

The swift global transition to online transactions highlights the critical demand for digital financial solutions, particularly in developing nations such as Nepal, where conventional banking methods dominate. Our initiative presents a prototype aimed at simplifying minor, essential transactions via a digital platform, thereby removing the necessity for in-person banking interactions. Using the C programming language, which is renowned for its strong system-level features, we created a digital wallet enabling users to make payments with ease. This effort not only improves the ease of financial transactions but also utilizes educational elements of C, incorporating every concept and method learned throughout our first semester.

Keywords: C Programming, Digital Wallet, Online Transactions, Procedural Programming, System Development

Table of Content

List of Figures

List of Abbreviations

ASCII	American Standard Code for Information Interchange
ANSI	American National Standards Institute
API	Application Programming Interface
AT&T	American Telephone & Telegraph
BCT	Bachelor in Computer Technology
C11	C Standard Revision 2011
C99	C Standard Revision 1999
CPU	Central Processing Unit
EOF	End Of File
GCC	GNU Compiler Collection
GDB	GNU Debugger
IDE	Integrated Development Environment
OS	Operating System
RAM	Random Access Memory
UI	User Interface

1. INTRODUCTION

The goal of this project is to use the fundamental ideas of the C programming language to create a prototype of the C-Pay e-wallet application. The application allows users to register, log in, and make payments to other users or organizations directly, facilitating smooth online financial transactions. C-Pay concentrates on key features that enable the real-world implementation of programming abilities to build a platform for financial transactions. This prototype shows how straightforward, yet powerful software solutions might improve digital financial transactions.

1.1 Background

The project involves the creation of a prototype for an online payment gateway named C-Pay, leveraging fundamental C programming constructs such as file handling, pointers, structures, and arrays. This system is designed to enable users to register, log in, and execute various financial transactions including bill payments and money transfers. The development approach divides tasks strategically among contributors to optimize efficiency and make effective use of online collaboration tools. This setup not only facilitates the practical application of theoretical concepts but also contributes to advancing the digital transformation of financial services.

1.2 Motivation

The motivation for the C-Pay project is influenced by the revolutionary impact of digital wallets both globally and within Nepal. Internationally, platforms like PayPal, Apple Pay, and Google Wallet have reshaped how consumers handle money, offering seamless and secure digital transaction solutions. Locally, eSewa and Khalti have revolutionized financial transactions, providing a digital alternative to traditional banking methods and significantly enhancing accessibility in Nepal. The development of C-Pay is driven by curiosity about the potential of C programming to create a robust and secure digital wallet system. This initiative aims to explore the intricacies of digital payment systems and apply theoretical knowledge in a practical setting, thus deepening an understanding of both programming and the digital financial ecosystem. The project seeks to blend global trends with local needs, creating a platform that is not only functional but also

innovative, sparking curiosity and fostering deeper engagement with digital finance technologies.[1]

1.3 Problem Definition

The development of the C-Pay digital wallet encounters several challenges that must be addressed to ensure the platform's effectiveness and user-friendliness. The primary concern is establishing robust security measures to protect users' personal and transactional data, which is fundamental to maintaining trust and reliability in digital financial transactions. Additionally, the system must feature a coherent and efficient mechanism for managing account transfers and payments to facilitate smooth financial operations.

Another significant challenge is implementing a secure user authentication process that prevents unauthorized access while ensuring a seamless user experience. This involves designing a user interface (UI) that is both intuitive and user-friendly, enabling clear navigation and interaction for tasks ranging from user registration to transaction processing. The UI must minimize user errors and provide clear feedback for each action within the system.

Furthermore, given the complexities of integrating such functionalities using C programming, there is a need for careful planning and execution. This involves managing the development process to effectively incorporate contributions from all team members and meet project deadlines without compromising the system's quality and operational integrity. These challenges are critical to the success of the C-Pay project, aiming to provide a secure, efficient, and user-friendly digital wallet solution.

1.4 Objective

The main objectives of our project are listed below:

- To develop a functional e-wallet prototype using C programming that supports secure and efficient financial transactions.
- To implement essential features such as user registration, login, transaction processing, and account management within the digital wallet system.

1.5 Scope and Limitations

1.5.1 Scope & Applications

The C-Pay project is designed to deliver a digital wallet system that supports basic yet essential functionalities like existing platforms. Key features include:

- Secure Data Storage: User data, such as passwords, is stored in binary files with a .dat extension, which enhances security by making the content difficult to read directly.
- Encryption: Utilizes a circular encryption mechanism for passwords, requiring a secret cipher code for decryption.
- Transaction Capabilities: Allows users to transfer funds to other users and pay educational fees, facilitating day-to-day financial activities.
- Broader Financial Operations: Adapt the prototype for varied commercial and organizational financial transactions.
- Integration with External Services: Facilitate the incorporation of APIs for real-time banking transactions, enhancing its applicability in more dynamic financial environments.
- Customization for Local Needs: Tailor features to address specific local market requirements, improving accessibility and usability for a wider range of users.

1.5.2 Limitations

- Operating System Compatibility: The program is incompatible with 16-bit operating systems.
- Limited Transaction Features: Only transactions between personal accounts and payments to colleges are supported.
- Educational Scope: Some functions used in the program have not been covered in our coursework, limiting our ability to fully understand all aspects of the implementation.

1.6 Report Organizations

The report begins with an introduction that explains the motivation behind developing the C-Pay digital wallet, emphasizing the use of C programming to facilitate secure and efficient financial transactions. It provides an overview of the challenges and

opportunities in the digital payments landscape that the project aims to address. Following the introduction, the background section offers insight into the evolution of digital wallets, highlighting their importance and the specific context within Nepal. This helps to anchor the project within the broader trends in financial technology. The objectives and scope section details what C-Pay aims to achieve, including the creation of a prototype that handles basic financial operations securely. It also discusses the limitations and the potential for future expansion. The methodology section describes the technical approaches used, from system design to the specific programming techniques employed to ensure functionality and security. In the system architecture and design section, the structural framework of the application is detailed, explaining how different components of the system interact to manage transactions and user data effectively. The implementation section covers the practical steps taken to bring the project to life, from coding to testing, and highlights the tools and environments utilized in the development process. Results and analysis are then presented, evaluating the performance of the C-Pay system against set objectives, showcasing both successes and areas for improvement. Future enhancements propose potential upgrades and additions for the system, suggesting how it could evolve to meet additional user needs and incorporate advanced technological features. The report concludes by summarizing the project's outcomes and the team's learning experiences, reflecting on how the project has contributed to a deeper understanding of digital wallet systems and software development practices.

2. LITERATURE REVIEW

The development of the C-Pay e-wallet application is a significant undertaking that integrates advanced C programming techniques with the operational demands of modern digital financial systems. This literature review examines the technological landscape, the evolution of transaction systems, and the specific programming challenges and solutions implemented in e-wallet technology.[2]

2.1 Background of Online Transaction Systems

Online transaction systems have drastically transformed the financial sector by offering solutions that enhance the speed, security, and convenience of monetary exchanges. These systems have eliminated the need for physical currency in day-to-day transactions, instead facilitating digital transfers that can be executed from virtually anywhere in the world. Globally recognized platforms such as PayPal, Alipay, and Apple Pay have not only standardized secure transaction protocols but also simplified complex financial operations, thus setting a high benchmark in the fintech industry. In regions like Nepal, platforms such as eSewa and Khalti have successfully adapted these technologies to suit local economic conditions and consumer needs, thereby democratizing access to digital finance solutions. These platforms exemplify the effective incorporation of technology in streamlining financial activities, significantly reducing the dependency on conventional banking methods and transforming everyday financial interactions.[3]

2.2 Evolution of C Programming Language

Developed in 1972 by Dennis Ritchie at AT&T Bell Laboratories, the C programming language is fundamental to contemporary software development. Originally intended for system programming on the new Unix OS, C's robust low-level memory access and simple syntax rendered it highly effective for various applications. Throughout the years, C has impacted numerous other languages, such as C++, which enhances its functionality through object-oriented elements. The standardization of C as ANSI C, along with later versions such as C99 and C11, has reinforced its importance in the creation of embedded systems, operating systems, and intricate computational systems. C's versatility across various hardware platforms makes it a perfect option for

applications necessitating direct hardware engagement, like embedded systems and payment solutions.[4]

2.3 Applications and Importance of E-wallets

The rapid adoption of e-wallets globally is reshaping economic interactions, providing essential services such as instant financial transactions, enhanced security features, and increased accessibility to banking services for the unbanked and underbanked populations. These digital solutions offer a sustainable alternative to traditional banking, reducing operational costs and environmental impact by minimizing the physical infrastructure required for financial operations. The integration of e-wallets into the market has also introduced new paradigms in consumer behavior, encouraging more dynamic financial interactions and fostering a more inclusive financial ecosystem.

2.4 Drawbacks and Limitations

While the use of C programming provides several advantages for system-level applications, it also introduces certain challenges in the development of an e-wallet. C is inherently low-level, which complicates the implementation of high-level user interface features. This can increase development time and potentially affect the application's usability. Furthermore, for many advanced features, C programmers often rely on third-party libraries. While useful, this dependency can impact the stability and security of the application if these libraries are not properly maintained.[5]

Another significant challenge is ensuring the security of the application due to C's powerful low-level capabilities, which require meticulous attention to avoid security vulnerabilities such as buffer overflows and memory leaks. The development of a C-based e-wallet like C-Pay thus requires rigorous testing and validation to mitigate potential security risks and ensure the application's overall security and stability.

2.5 Methodologies for Creating E-Wallet

Developing the C-Pay e-wallet in C programming involves a comprehensive approach focused on ensuring both security and functionality to deliver a reliable and efficient system. Secure data handling is paramount in financial applications, and C-Pay

addresses this by utilizing advanced cryptographic techniques to protect user data. This includes encrypting sensitive information such as passwords and transaction details using industry-standard encryption algorithms and storing this data in encrypted files with tightly controlled access via C's file handling functions.[6]

Efficient transaction processing is achieved through the use of optimized algorithms and data structures designed to facilitate quick retrieval and update of user records. C's capability for low-level operation handling allows for the fine-tuning of memory and process management, ensuring smooth processing of transactions even under high load. Additionally, C-Pay incorporates a robust authentication system to verify user identities, combining username and password checks with additional security measures such as multi-factor authentication to enhance security and usability.[7]

3. REQUIREMENT ANALYSIS

The development of the C-Pay e-wallet project necessitates a detailed requirement analysis to ensure the successful integration and functionality of the system. This analysis covers both hardware and software requirements essential for the project's execution, alongside a feasibility study to assess the practicality of the project's implementation.

3.1 Hardware and Software Requirements

For the C-Pay project, the primary requirement is a robust software environment capable of supporting C programming and its associated data management needs. The project utilizes the GCC (GNU Compiler Collection) for compiling the C code, ensuring compatibility and efficiency. Additionally, software tools like Visual Studio Code or Code::Blocks serve as the integrated development environments (IDEs) providing essential coding, debugging, and testing capabilities. Hardware requirements are modest, as the application primarily relies on software efficiency. A basic setup including a processor capable of running the development environment smoothly, sufficient RAM (minimum 4GB), and adequate storage space for the software and transaction databases are necessary. The system is designed to be compatible with major operating systems including Windows, Linux, and macOS to ensure broad accessibility.

3.2 Feasibility Study

- **Cost-Effectiveness:** The software development requires no financial investment in software licenses, as it utilizes open-source tools and environments. The primary cost involves human resources, primarily the time and effort of the student.

- **Technical Feasibility:** The use of C programming and basic file handling is appropriate for the skill level of a first-semester BCT student. These elements introduce students to fundamental programming concepts without the overhead of more complex languages or frameworks.

- **Resource Availability:** All necessary resources, such as compilers and development environments, are freely available. Documentation and community

support for C programming are extensive, which aids in overcoming potential development challenges.

- **Scalability:** Initially, the project handles a small, predefined set of data and operations, but it can be scaled up to include more features such as different types of transactions, more detailed user profiles, and integration with actual banking APIs for real-time financial operations.

- **Maintainability:** The code is structured in modular functions, which makes it easier to maintain and update. Good programming practices and documentation are emphasized to ensure that the project can be easily understood and extended by other developers or as part of future coursework.

- **Educational Value:** The project provides practical experience with real-world applications of programming, enhancing the learning curve and offering a tangible outcome that reinforces theoretical knowledge.

4. SYSTEM ARCHITECTURE AND METHODOLOGY

4.1 Overview Block Diagram

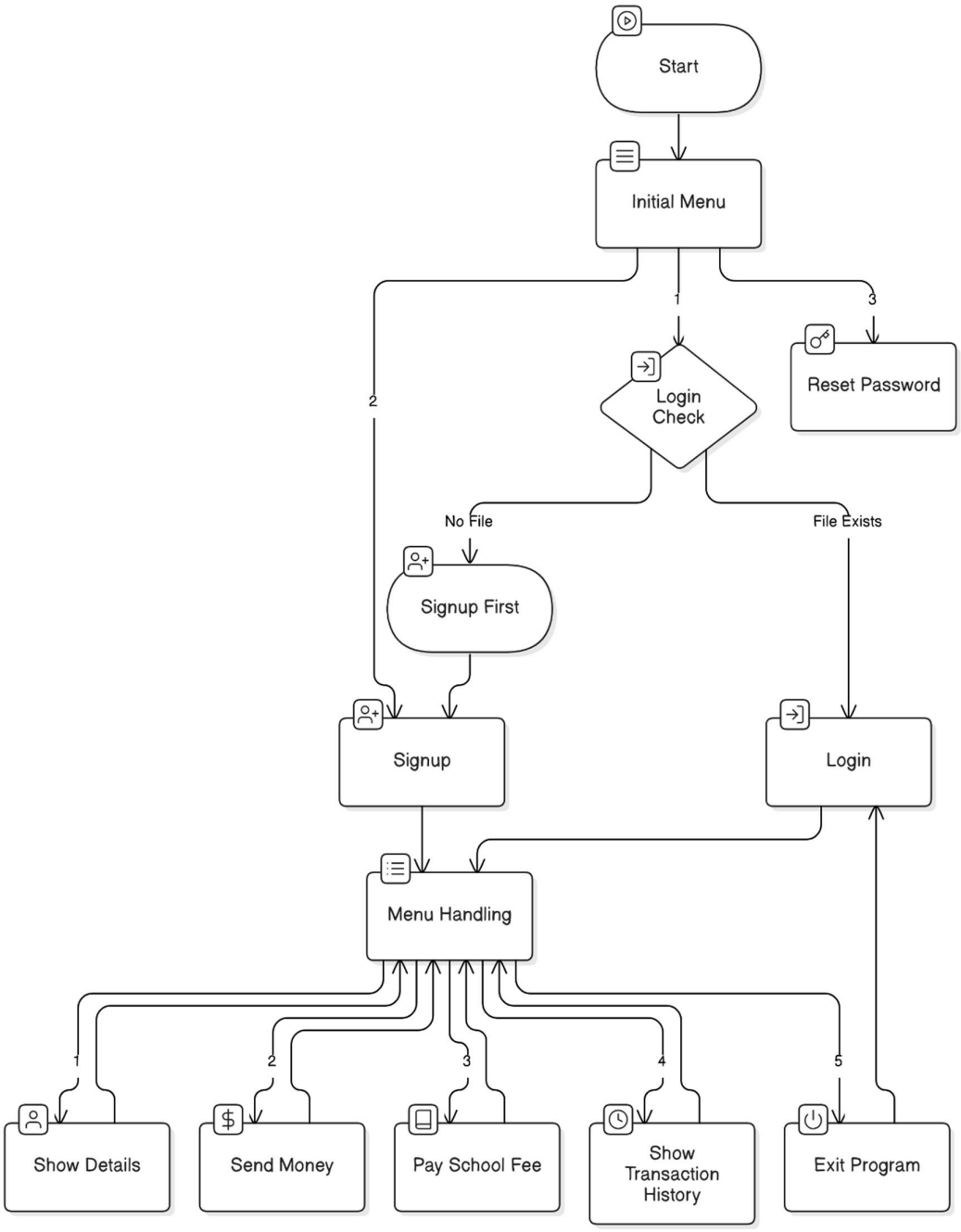


Fig 4-1: Overview Block Diagram

4.2 Initial Menu Block Diagram

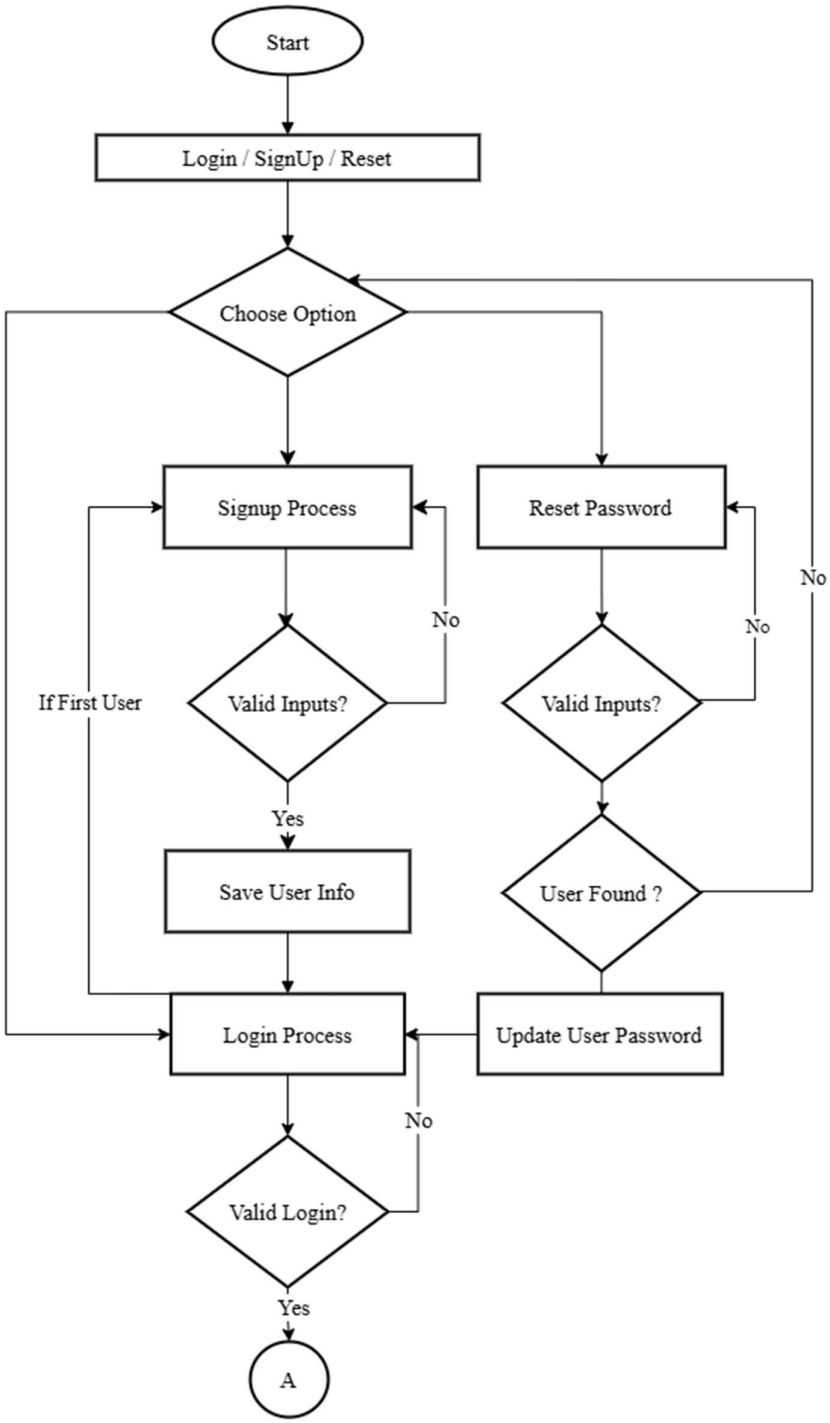


Fig 4-2: Flowchart for Initial Menu Handling

Flowchart Overview:

1. **Start:**

   The program begins with execution, leading directly into the menu where the user chooses to Login, Sign Up, or Reset Password.

2. **Login / SignUp / Reset:**

   This decision node represents the initial choice presented to the user via the console output. The user inputs their choice, which determines the next course of action.

3. **Choose Option:**

   Depending on the user's input, the process diverges into three potential paths: Signup Process, Reset Password, or directly jumping to Login if chosen or as a default fallback.

4. **Signup Process:**

   If the user is running the first time running the program and no user data file exists (login_data_file is NULL), the user is directed to sign up. This includes input validation (checks for valid username, email, and phone number inputs). If inputs are valid, it proceeds to save the user information. If the user's inputs are invalid, it loops back to request inputs again.

5. **Reset Password:**

   The reset process checks for valid inputs similar to the signup process. It then checks if the user exists based on the provided credentials. If the user is found, the password update process is initiated. If inputs are invalid at any step, it loops back to request inputs again.

6. **Login Process:**

   If the user opts to log in or after successful signup/reset, the program attempts to validate the login credentials. If the login is valid, it progresses to the next stage (denoted as "A" in the flowchart, which likely leads to the main application menu or further user interactions).If the login is invalid, it may loop back to the login prompt.

7. **Additional Detail:**

i. **Error Handling and Redirects:**

The flowchart should include pathways that handle errors or incorrect inputs by looping back to the respective decision points. For example, if login credentials are invalid, the process should loop back to the initial login input request.

ii. **File Operations:**

Operations related to file handling (checking if a file exists, reading from or writing to a file) are crucial in deciding the flow but aren't explicitly shown in the flowchart. Including file checks before signup or login could clarify transitions between decisions.

iii. **Security Checks:**

Security operations like password encryption and decryption during signup, login, and reset password processes are important for real-world applications but are simplified in the code and not shown in the flowcharts.

iv. **Error Message and Feedback:**

For each input validation failure (whether for login, signup, or reset), the flowchart could include branches that specify the type of feedback provided to the user (e.g., "Invalid username" or "Password too short"). This makes the flow more user-centric and explains the interface behavior more comprehensively.

4.3 Main Menu Block Diagram

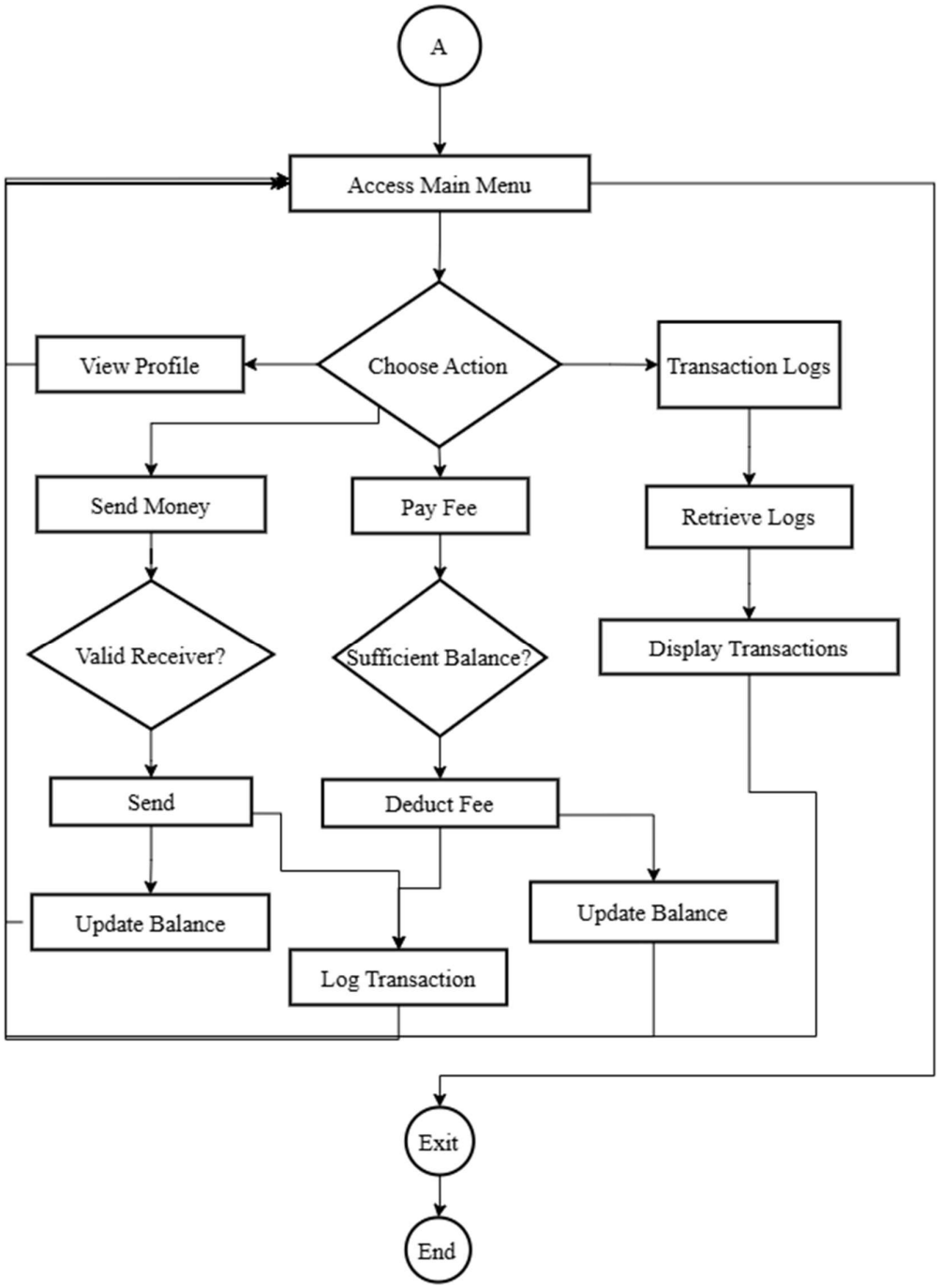


Fig 4-3: Flowchart for Main Menu Handling

Flowchart Overview:

1. Main Menu Access and Action Selection:

Upon successful login, users reach the main menu, which serves as the central hub for all navigational decisions. The "Access Main Menu" node is essentially the gateway where users are prompted to select from various actions:

a. View Profile
b. Send Money
c. Pay Fee
d. Transaction Logs
e. Exit

Each choice directs the user to different functionalities of the application, designed to handle specific tasks such as financial transactions, account management, or historical reviews.

a. Viewing Profile:

When selecting "View Profile," users are presented with their account details, including their username, account balance, and other personal information. This section is straightforward, providing a read-only view of the user's data, enhancing user confidence and transparency about their account status.

b. Sending Money:

This function is a critical component of the application, involving several checks and operations:

Valid Receiver?:

The application verifies the receiver's details against the database to ensure the transaction is directed at a legitimate party. This validation is crucial for preventing errors or fraudulent activities.

Send:

Upon validating the receiver, the application processes the transaction by transferring the specified amount from the user's account to the receiver's account.

Update Balance:

The user's balance is updated to reflect the transaction, ensuring the account balance displayed is always current.

Log Transaction:

Every transaction is meticulously logged, providing an immutable record that supports transparency and can be used for auditing.

c. Paying Fees

This pathway handles fee payments, crucial for users needing to make periodic payments like tuition or service fees:

Sufficient Balance?:

Before proceeding, the system checks if the user's account has enough funds to cover the fee. This preemptive check prevents overdrafts, and the complications associated with them.

Deduct Fee:

If the balance is adequate, the fee is automatically deducted from the user's account.

Update Balance and Log Transaction:

Like sending money, the balance is updated, and the transaction is logged to maintain a consistent and accurate account history.

d. Transaction Logs

Accessing transaction logs allows users to review their financial activities over a specified period:

Retrieve Logs:

The system fetches the transaction records from the database.

Display Transactions:

Users can view detailed listings of their transactions, providing them with visibility and insights into their spending patterns and financial transactions.

e. Exciting the Program

Finally, the "Exit" option allows users to safely end their session. This process includes proper cleanup and logging out procedures, ensuring all user data is secured and the session is properly closed to prevent any security risks.

System Integrity and Security:

i. Validation and Error Handling

Each node that requires user input or involves data processing incorporates validation mechanisms to ensure the integrity and accuracy of the data. Error handling is also a critical component, where the system provides informative feedback for any discrepancies or issues encountered during data input or processing. This approach not only secures the process but also enhances the user experience by reducing frustration and confusion.[8]

ii. Security Measures

The system implements robust security protocols to protect user data and transaction integrity:

Encryption and Decryption:

Sensitive data, such as passwords and transaction details, are encrypted to prevent unauthorized access. The system uses decryption methods securely to ensure data is accessible only to legitimate users.

User Interaction and Feedback:

The application is designed to be user-centric, providing clear and immediate feedback after each action. Whether a transaction is successful, a profile is updated, or an error is encountered, the system promptly communicates this to the user, ensuring they are always informed of the outcome of their actions.

4.4 Interface Display Functions for User Interaction

The ascii_display.c file contains several crucial functions designed to manage the visual presentation of the user interface in the console-based application. This file is included in the main application via the #include "ascii_display.c" directive, ensuring that its functions are accessible where needed. Below is a description of each function and its role within the application:

1. **cpay():**
    - Purpose: Displays the application's main banner each time the program is run, or the main menu is accessed.

- Implementation: Uses system("cls") to clear the screen, ensuring the banner is presented on a clean slate. The function then prints a custom-designed ASCII art logo representing the application, C_PAY.

2. **display_login():**
   - Purpose: Provides a specific display for the login screen.
   - Implementation: Clears the screen and displays ASCII art that visually represents the login process, enhancing user engagement and making the interface more intuitive and friendly.

3. **display_signup():**
   - Purpose: Used to visually differentiate the signup screen from other parts of the application.
   - Implementation: Similar to display_login(), this function clears the screen and shows ASCII art tailored to the signup process, helping users recognize they are in the process of creating a new account.

4. **display_menu(char username[]):**
   - Purpose: Shows the main menu after the user logs in, providing a personalized greeting with the user's name.
   - Implementation: Clears the screen and displays the main menu options along with a personalized greeting, using the username parameter passed to the function. This function makes the interaction more personalized, which enhances the user experience.

5. **reset_pass():**
   - Purpose: Displays the reset/forgot password screen.
   - Implementation: Clears the screen and displays relevant ASCII art and instructions for users looking to reset or retrieve their forgotten passwords.

These functions are critical for maintaining a consistent and engaging user interface across the application. They encapsulate the display logic separately from the main business logic, adhering to good software design principles by separating concerns.

This modular approach not only organizes the code better but also simplifies maintenance and scalability of the application by isolating interface changes within these display functions. Each function enhances the user experience by providing clear, context-specific graphical representations of each action within the application.

4.5 Algorithm & Code Snippet

4.5.1 Algorithm for Initial Menu

Step 1: Start

Step 2: Clear the screen using system("cls")

Step 3: Declare a character variable initial_choice

Step 4: Display the options "[1] Login", "[2] SignUp", "[3] Reset Password" using printf()

Step 5: Read the initial_choice using getche()

Step 6: Check whether initial_choice is '1', '2', or '3'

If yes, continue

If no, display "Invalid choice! Please enter [1-3]." using printf() and go back to Step 4

Step 7: Proceed based on initial_choice

If initial_choice is '1', go to Step 8

If initial_choice is '2', go to Step 10

If initial_choice is '3', go to Step 10 but for resetting password

Step 8: Open file logindata.dat using fopen() pointed by login_data_file in binary read mode ("rb")

Step 9: Check whether login_data_file can be opened or not

If yes, check if this is the very first time running the program by verifying if login_data_file is NULL. If NULL, display "Seems like you are first user, Signup first." using printf() and proceed to signup (Go to Step 10)

If no, call the login() function

Step 10: Depending on the initial choice:

If signing up, call the signup() function

If resetting password, call the reset_password() function

Step 11: Call menu_handling() function

Step 12: Stop

Code Snippet for the Initial Menu

```
// Clear the screen
system("cls");

// Declare the variable to capture user input
char initial_choice;

// Display initial menu options
printf("\t[1] Login\n\t[2] SignUp\n\t[3] Reset Password\n");
printf("\tChoose an option [1-3]: ");

// Input handling using getche
do {
    initial_choice = getche();
    if (initial_choice != '1' && initial_choice != '2' &&
initial_choice != '3') {
        printf("\n\tInvalid choice! Please enter [1-3].\n");
    }
} while (initial_choice != '1' && initial_choice != '2' &&
initial_choice != '3');

// Process based on user choice
switch (initial_choice) {
    case '1':
        login_data_file = fopen("logindata.dat", "rb");
        if (login_data_file == NULL) {  // Check if the login
data file exists
            printf("\n\n\t***Seems like you are first
user***\n\t\t***Signup first***");
            delay(1.5);
            initial_choice = '2';  // Redirect to signup
        }
        break;
    case '2':
        signup();
        break;
    case '3':
        reset_password();
        break;
}
// Handling of the main menu after login
menu_handling();
```

4.5.2 Algorithm for SignUp Process

Step 1: Star

Step 2: Clear the screen using system("cls")

Step 3: Display the signup banner using the function display_signup()

Step 4: Declare a structure struct userdata signupdata to store the new user's information

Step 5: Prompt and read the user's username using printf() and scanf()

Step 6: Prompt for the user's email. Display with printf() and read with scanf(), then validate using validate_email()

Step 7: Prompt for the user's phone number using printf() and scanf(), then validate using validate_phone()

Step 8: If any validations (username, email, phone) fail, print errors and repeat the input collection (loop back to the respective step)

Step 9: Prompt for a password using printf() and securely capture it using password_taker(). Repeat for password confirmation

Step 10: Validate password length and ensure it meets criteria (validation logic inside password_taker())

Step 11: Compare the two passwords for a match. If they do not match, print an error using printf() and return to Step9

Step 12: If the passwords match, encrypt the password using encrypt()

Step 13: Set the initial balance for signupdata.balance to INITIAL_BALANCE

Step 14: Open logindata.dat for appending user data using fopen() and write the signupdata using fwrite()

Step 15: Check if the write operation was successful. If not, handle the error using perror() and exit if necessary

Step 16: Display "Sign-Up Successful" message using printf()

Step 17: Call the login() function to allow the user to log in immediately after signing up

Step 18: Stop

Code Snippet for the SignUp Process

```
// Clear screen and display signup banner
system("cls");
display_signup();

// Declare userdata structure for new user inputs
struct userdata signupdata;

// Input and validate username
printf("\tUsername: ");
scanf("%99s", signupdata.name);

// Input and validate email
printf("\tEmail (e.g,user@domain.com): ");
scanf("%99s", signupdata.email);
if (!validate_email(signupdata.email)) {
    printf("\tInvalid email format!\n");
    // Loop or handle error
}

// Input and validate phone number
printf("\tPhone number (98/97xxxxxxx): ");
scanf("%lf", &signupdata.phone);
if (!validate_phone(signupdata.phone)) {
    printf("\tInvalid phone number!\n");
    // Loop or handle error
}

// Input and validation for password
printf("\tPassword (min 4 chars): ");
char verify_password[100];
password_taker(signupdata.password, max_length);
printf("\tConfirm password: ");
password_taker(verify_password, max_length);

// Check if passwords match and process accordingly
if (strcmp(verify_password, signupdata.password) == 0) {
    encrypt(signupdata.password); // Encrypt the confirmed
password
    signupdata.balance = INITIAL_BALANCE; // Set initial
balance
```

```
    // Attempt to open the user data file and append new
record
    FILE *login_data_file = fopen("logindata.dat", "ab+");
    if (fwrite(&signupdata, sizeof(struct userdata), 1,
login_data_file) != 1) {
        perror("Error writing user data");
        exit(1);
    }
    fclose(login_data_file); // Close the file after writing

    // Display successful signup message and redirect to login
    printf("\n\t***Sign-Up Successful***\n\t***You can now
login***\n");
    login(); // Redirect user to login
} else {
    // Handle password mismatch
    printf("\tPasswords do not match! Please try again.\n");
}
```

4.5.3 Algorithm for Login Process

Step 1: Start

Step 2: Clear the screen using system("cls")

Step 3: Display the login banner using the function display_login()

Step 4: Declare a struct userdata filedata[MAX_USERS] to store user data from file and an integer is_login_successful initialized to 0 for tracking login status

Step 5: Open logindata.dat using fopen() in read mode pointed by login_data_file. Check for file existence and read user data

Step 6: Prompt for the username using printf() and capture it using scanf()

Step 7: Prompt for the password using printf() and securely capture it using password_taker()

Step 8: Decrypt the stored password for each user in filedata using decrypt() to compare with the input password

Step 9: Compare the entered username and decrypted password with those stored in filedata:

If a match is found, set is_login_successful to 1, and break the loop

Display "Login Successful" using printf() and update current user details

Step 10: If is_login_successful is 0 after checking all records, display "Incorrect username or password!" using printf() and allow retry or exit to the main menu

Step 11: Stop

Code Snippet for the Login Process

```
// Clear screen and display login banner
system("cls");
display_login();

// Array to hold user data read from file
struct userdata filedata[MAX_USERS];
int is_login_successful = 0;

// Attempt to open the user data file for reading
FILE *login_data_file = fopen("logindata.dat", "rb");
if (login_data_file == NULL) {
    printf("\tFile not found! Please sign up first.\n");
    return;
}

// Reading user data from file
int n = 0;
while (fread(&filedata[n], sizeof(struct userdata), 1,
login_data_file) == 1 && n < MAX_USERS) {
    decrypt(filedata[n].password);  // Decrypting password for
comparison
    n++;
}

// Prompt for username
printf("\tEnter your username: ");
scanf("%99s", current_user.name);

// Prompt for password and capture it securely
printf("\tPassword: ");
password_taker(current_user.password, max_length);

// Validate entered credentials against stored data
for (int i = 0; i < n; i++) {
    if (strcmp(filedata[i].name, current_user.name) == 0 &&
```

```
        strcmp(filedata[i].password, current_user.password) ==
0) {
        printf("\n\tLogin Successful!\n");
        is_login_successful = 1;
        current_user = filedata[i];  // Update current user's
details
        break;
    }
}

fclose(login_data_file);  // Close the file after reading

if (!is_login_successful) {
    printf("\n\tIncorrect username or password!\n");
    // Provide option to retry or exit
}
```

4.5.4 Algorithm for Reset Password Process

Step 1: Start

Step 2: Clear the screen using system("cls")

Step 3: Display the reset password banner using the function reset_pass()

Step 4: Declare a struct userdata users[MAX_USERS] to store user data from file and another struct userdata resetdata for capturing new input, and integers n for number of users and user_found initialized to 0 for tracking user existence

Step 5: Load user data from logindata.dat using fopen() in read mode pointed by user_data_file and read all entries into users

Step 6: Prompt for the username using printf() and capture it using scanf()

Step 7: Prompt for the email using printf() and capture it using scanf(), then validate using validate_email()

Step 8: Prompt for the phone number using printf() and capture it using scanf(), then validate using validate_phone()

Step 9: Validate the entered username, email, and phone against stored data:

If a match is found, set user_found to 1

If no match is found, display "No user found with the given details!" using printf() and offer to retry or exit

Step 10: If user_found is 1, prompt the user to enter a new password using printf() and securely capture it using password_taker()

Step 11: Confirm the new password by asking the user to enter it again using password_taker() and compare with the first entry

Step 12: If the passwords match, encrypt the new password using encrypt(), update the user's record in users, and save the updated data back to logindata.dat using fwrite()

Step 13: Display "Password reset successful!" using printf()

Step 14: Stop

Code Snippet for the Reset Password Process

```
// Clear screen and display reset password banner
system("cls");
reset_pass();

// Array to hold user data and temporary data for resetting
password
struct userdata users[MAX_USERS], resetdata;
int n = 0, user_found = 0;

// Load user data from file
load_user_data(users, &n);

// Input handling for username, email, and phone
printf("\tUsername: ");
scanf("%99s", resetdata.name);
printf("\tEmail (e.g,user@domain.com): ");
scanf("%99s", resetdata.email);
printf("\tPhone number (98/97xxxxxxx): ");
scanf("%lf", &resetdata.phone);

// Validate entered details against stored data
for (int i = 0; i < n; i++) {
    if (strcmp(users[i].name, resetdata.name) == 0 &&
        strcmp(users[i].email, resetdata.email) == 0 &&
        users[i].phone == resetdata.phone) {
        user_found = 1;
        break;
    }
}

if (!user_found) {
```

```
    printf("\n\tNo user found with the given username and
email.\n");
    // Provide option to retry or exit
} else {
    // Prompt for new password
    printf("\tNew Password (min 4 chars): ");
    password_taker(resetdata.password, max_length);
    char confirm_password[100];
    printf("\tConfirm New Password: ");
    password_taker(confirm_password, max_length);

    // Check if passwords match and process accordingly
    if (strcmp(confirm_password, resetdata.password) == 0) {
        encrypt(resetdata.password); // Encrypt the new
password
        users[i].password = resetdata.password; // Update
password in array

        // Save the updated user data back to file
        save_user_data(users, n);

        printf("\n\tPassword reset successful!\n");
    } else {
        printf("\tPasswords do not match! Please try
again.\n");
    }
}
```

4.5.5 Algorithm for Used User-Defined Functions

Algorithm for validate_phone(double phone)

Step 1: Start

Step 2: Convert the double phone to a long long int phone_int for easier range comparison

Step 3: Check if phone_int is within the range of valid Nepali phone numbers:

If within 9600000000 to 9699999999 or 9800000000 to 9869999999 or 9880000000 to 9889999999 or 9700000000 to 9709999999 or 9740000000 to 9769999999, return 1 (true)

Otherwise, return 0 (false)

Step 4: Stop

Code Snippet for the validate_phone(double phone)

```
int validate_phone(double phone) {
    long long int phone_int = (long long int)phone;
    return (phone_int >= 9600000000LL && phone_int <=
9699999999LL) ||
           (phone_int >= 9800000000LL && phone_int <=
9869999999LL) ||
           (phone_int >= 9880000000LL && phone_int <=
9889999999LL) ||
           (phone_int >= 9700000000LL && phone_int <=
9709999999LL) ||
           (phone_int >= 9740000000LL && phone_int <=
9769999999LL);
}
```

Algorithm for validate_email(const char* email)

Step 1: Start

Step 2: Validate that the email is not NULL or empty, and does not start with '@' or contain spaces. If any condition fails, return 0 (false)

Step 3: Locate the '@' in the email using strchr(). Confirm it is not at the start or end of the email

Step 4: Find the last '.' after '@' using strchr(). Ensure it exists and is not the last character of the email

Step 5: Return 1 (true) if all validations pass

Step 6: Stop

Code Snippet for the validate_email(const char* email)

```
int validate_email(const char* email) {
    if (email == NULL || email[0] == '\0' || email[0] == '@'
|| strchr(email, ' ') != NULL) {
        return 0;
    }
    const char* at = strchr(email, '@');
    if (at == NULL || at == email || at[1] == '\0') return 0;
    const char* dot = strchr(at, '.');
```

```
    return dot != NULL && dot > at && dot[1] != '\0';
}
```

Algorithm for password_taker(char password[], int max_length)

Step 1: Start

Step 2: Initialize charposition to 0

Step 3: Enter a loop to capture user input until the Enter key (ASCII code 13) is pressed:

If the character is Enter, break the loop and null-terminate the password

If the character is backspace and charposition is greater than 0, decrement charposition and visually remove the character from the console

If the character is a space or tab, skip it

If charposition is less than max_length, add the character to password, increment charposition, and print '*' to mask the input

If the password exceeds max_length, print an error message

Step 4: Stop

Code Snippet for the password_taker(char password[], int max_length)

```
void password_taker(char password[], int max_length) {
    char ch;
    int charposition = 0;
    while (1) {
        ch = _getch();
        if (ch == 13) {
            password[charposition] = '\0';
            break;
        } else if (ch == 8 && charposition > 0) {
            charposition--;
            password[charposition] = '\0';
            printf("\b \b");
        } else if (ch == 32 || ch == 9) {
            continue;
        } else if (charposition < max_length) {
            password[charposition++] = ch;
            printf("*");
        } else {
```

```
            printf("\nPassword too long! Maximum %d characters
allowed.\n", max_length);
            break;
        }
    }
}
```

Algorithm for clear_input_buffer()

Step 1: Start

Step 2: Read characters from the buffer until a newline is found or EOF

Step 3: Stop

Code Snippet for the clear_input_buffer()

```
void clear_input_buffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF); // Clear
input buffer
}
```

Algorithm for encrypt(char[])

Step 1: Start

Step 2: Iterate over each character of the string

Step 3: Increment the ASCII value of each character

Step 4: Stop

Code Snippet for the encrypt(char[])

```
void encrypt(char str[]) {
    int i = 0;
    while (str[i] != '\0') {
        str[i] = str[i] + 3; // Encrypt character
        i++;
    }
}
```

Algorithm for decrypt(char[])

Step 1: Start

Step 2: Iterate over each character of the string

Step 3: Decrement the ASCII value of each character

Step 4: Stop

Code Snippet for the decrypt(char[])

```
void decrypt(char str[]) {
    int i = 0;
    while (str[i] != '\0') {
        str[i] = str[i] - 3; // Decrypt character
        i++;
    }
}
```

4.5.6 Algorithm for Main Menu Selection

Step 1: Start

Step 2: Clear the screen using the function system("cls")

Step 3: Display the application's main banner by calling the function cpay()

Step 4: Display the main menu options by calling the function

display_menu(current_user.name), which includes:

Show Details

Send Money

Pay School Fee

Show Transaction History

Exit

Step 5: Prompt the user to choose an action by displaying "Choose an option [1-5]:"

Step 6: Read the user's choice using the function getche()

Step 7: Validate the user's input:

If the input is between '1' and '5', proceed to the respective function.

If the input is invalid, display "Invalid choice! Please enter [1-5]." Using printf() and loop back to prompt the user again.

Step 8: Based on the valid input, direct to the chosen action:

If '1': Call the function show_details() to view user profile details.

If '2': Call the function send_money() to initiate a money transfer.

If '3': Call the function pay_school_fee() to handle fee payment.

If '4': Call the function show_transaction_history() to view past transactions.

If '5': Call the function exit_program() to exit the application.

Step 9: Stop

Code Snippet for the Main Menu Selection

```
void menu_handling() {
    char menu_choice;
    do {
        system("cls");  // Clear the screen
        display_menu(current_user.name);  // Display the main
menu options with greeting
        printf("\t1. Show Details\n\t2. Send Money\n\t3. Pay
School Fee\n\t4. Show Transaction History\n\t5. Exit\n");
        printf("\t----------------------------------------
\n");
        printf("\tEnter your choice [1-5]: ");
        menu_choice = getche();  // Capture user input
        if (menu_choice < '1' || menu_choice > '5') {
            printf("\n\tInvalid choice! Please enter 1-5.\n");
            delay(1.5);  // Wait for 1.5 seconds before the
next loop iteration
        }
    } while (menu_choice < '1' || menu_choice > '5');  // Loop
until a valid input is received

    switch (menu_choice) {
        case '1':
            show_details();  // Call function to show user
details
            break;
        case '2':
            send_money();  // Call function to initiate
sending money
            break;
        case '3':
            pay_school_fee();  // Call function to handle
school fee payment
            break;
        case '4':
            show_transaction_history();  // Call function to
show transaction history
            break;
        case '5':
```

```
            exit_program();  // Call function to exit the
program
            break;
    }
}
```

4.5.7 Algorithm for Viewing User Profile

Step 1: Start

Step 2: Clear the display using system("cls")

Step 3: Invoke show_details() to present user details

Step 4: Utilize printf() within show_details() to print:

Username

Phone

Email

Current Balance

All pulled from current_user structure

Step 5: Display a prompt for the user to press any key to return to the main menu, using printf() and capture the key press with getch()

Step 6: Call menu_handling() to return to the main menu

Step 7: Stop

Code Snippet for Viewing User Profile

```
void show_details() {
    system("cls");  // Clear the console for clean output
    printf("\n\tUSER DETAILS\n\t-----------------------------
-\n");
    printf("\tUsername: %s\n\tPhone: %.0f\n\tEmail:
%s\n\tCurrent Balance: Rs %.2f\n\t---------------------------
---\n",
           current_user.name, current_user.phone,
current_user.email, current_user.balance);  // Display user
details
    printf("\tPress any key to return to menu: ");  // Prompt
user to return
    getch();  // Wait for user input
    menu_handling();  // Navigate back to the main menu
}
```

4.5.8 Algorithm for Sending Money

Step 1: Start

Step 2: Clear the screen using system("cls")

Step 3: Load user data into users[] by calling load_user_data()

Step 4: Display a list of potential receivers excluding the current user by iterating through users[] and using printf() to display each

Step 5: Prompt the user to enter the receiver's username and phone number, using printf() and capturing the input with scanf()

Step 6: Validate the receiver by calling validate_receiver(). If validation fails (receiver not found or details incorrect), inform the user and allow retry

Step 7: Prompt the user for the amount to be sent, using printf() and scanf() to capture the amount

Step 8: Check if the current user has sufficient balance:

If not, display an error message using printf(), allow correction or exit to the main menu

If yes, proceed to deduct the amount from the sender's balance and add it to the receiver's balance

Step 9: Update the user data in users[] and save back to the file by calling save_user_data()

Step 10: Log the transaction by calling log_transaction()

Step 11: Display a success message using printf()

Step 12: Return to the main menu by calling menu_handling()

Step 13: Stop

Code Snippet for the Sending Money Process

```
void send_money() {
    struct userdata users[MAX_USERS];
    int n = 0, valid_receiver = 0;
    double amount;
    struct userdata receiver;

    load_user_data(users, &n);  // Load all user data

    system("cls");  // Clear the screen
    printf("\n\tAvailable Users (excluding you):\n");
```

```
    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].name, current_user.name) != 0) {
            printf("\t%s\t%.0f\n", users[i].name,
users[i].phone);
        }
    }

    printf("\n\tEnter receiver username: ");
    scanf("%99s", receiver.name);
    printf("\tEnter receiver phone number: ");
    scanf("%lf", &receiver.phone);

    valid_receiver = validate_receiver(receiver, users, n);
    if (!valid_receiver) {
        printf("\n\tInvalid receiver details! Transaction
failed.\n");
        delay(2);
        return;  // Return to menu if validation fails
    }

    printf("\tEnter amount to send (Rs): ");
    scanf("%lf", &amount);

    if (amount <= 0 || current_user.balance < amount) {
        printf("\n\tInvalid amount or insufficient
balance.\n");
        delay(2);
        return;  // Exit if amount is invalid or balance is
insufficient
    }

    // Process the transaction
    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].name, current_user.name) == 0) {
            users[i].balance -= amount;
            current_user.balance = users[i].balance; // Update
current user's balance
        }
        if (strcmp(users[i].name, receiver.name) == 0 &&
users[i].phone == receiver.phone) {
            users[i].balance += amount;
        }
    }
```

```
    save_user_data(users, n);  // Save the updated user data
    log_transaction(current_user.name, receiver.name, amount,
"Send Money");  // Log the transaction

    printf("\n\tTransaction Successful! Your new balance is Rs
%.2f\n", current_user.balance);
    menu_handling();  // Return to main menu
}
```

4.5.9 Algorithm for Paying School Fees

Step 1: Start

Step 2: Clear the screen using system("cls")

Step 3: Load user data into users[] by calling load_user_data()

Step 4: Display a list of schools and their fees by iterating through a predefined array schools[] and using printf() to show each option

Step 5: Prompt the user to select a school by entering a choice number, using printf() and capture the input with scanf()

Step 6: Validate the user's choice:

Check if the choice is within the valid range

If not, display an error using printf() and allow the user to re-enter the choice

Step 7: Confirm the fee payment amount based on the chosen school

Step 8: Check if the current user has sufficient balance:

If not, display an error message using printf(), and offer to redirect back to the main menu

If yes, proceed to deduct the fee amount from the user's balance

Step 9: Update the user's balance in users[]

Step 10: Save the updated user data back to the file by calling save_user_data()

Step 11: Log the fee payment transaction by calling log_transaction() with the school's name as the receiver

Step 12: Display a success message using printf() confirming the fee payment

Step 13: Return to the main menu by calling menu_handling()

Step 14: Stop

Code Snippet for the Paying Fees Process

```
void pay_school_fee() {
    struct userdata users[MAX_USERS];
    int n = 0, school_choice;

    struct school {
        char name[150];
        float fee;
    } schools[4] = {
        {"IOE Pulchowk", 1000.0},
        {"IOE Thapathali", 1500.0},
        {"IOE WRC", 2000.0},
        {"IOE ERC", 2500.0}
    };

    load_user_data(users, &n); // Load user data

    do {
        system("cls");  // Clear the screen
        printf("\n\tCOLLEGE FEE PAYMENT\n\t-------------------
-------------------\n");
        printf("\n\tAvailable Colleges:\n");
        for (int i = 0; i < 4; i++) {
            printf("\t%d.   %-25s   Rs   %.2f\n",   i   +   1,
schools[i].name, schools[i].fee);
        }
        printf("\t---------------------------------------
\n");
        printf("\n\tEnter your choice [1-4]: ");
        scanf("%d", &school_choice);
        clear_input_buffer();
        if (school_choice < 1 || school_choice > 4) {
            printf("\n\tInvalid choice! Please enter 1-4.\n");
            delay(1.5);
            printf("\tPress 'x' to return to menu _ any key to
re-enter ");
            if(getch()=='x') {
            menu_handling();
            }
        }
    } while (school_choice < 1 || school_choice > 4);

    for (int i = 0; i < n; i++) {
```

```
        if (strcmp(users[i].name, current_user.name) == 0 &&
users[i].phone == current_user.phone) {
            if (users[i].balance < schools[school_choice -
1].fee) {
                printf("\n\tInsufficient balance! Your current
balance is Rs %.2f\n", users[i].balance);
                delay(2);
                printf("\tPress 'x' to return to menu _ any key
to re-enter ");
                if(getch()=='x') {
                menu_handling();
                }
                pay_school_fee();
                return;
            }
            users[i].balance -= schools[school_choice - 1].fee;
            current_user.balance = users[i].balance;
        }
    }

    save_user_data(users, n);
    log_transaction(current_user.name, schools[school_choice -
1].name, schools[school_choice - 1].fee, "School Fee");
    printf("\n\tTransaction Successful!\n");
    printf("\tYour    new    balance    is:    Rs    %.2f\n",
current_user.balance);
    printf("\t---------------------------------------\n");
    printf("\tPress any key to return to menu: ");
    getch();
    menu_handling();
}
```

4.5.10 Algorithm for Viewing Transaction Logs

Step 1: Start

Step 2: Allocate memory for a transaction array trans[] using malloc()

Step 3: Open the transaction log file transactions.dat using fopen() in read mode

Step 4: Check if the file exists:

If not, display "No transaction history available." using printf(), free the allocated memory using free(), and return to the main menu

Step 5: Read transactions from the file into trans[] using fread() until the end of the file or maximum capacity is reached

Step 6: Close the transaction log file using fclose()

Step 7: Clear the screen using system("cls")

Step 8: Display transaction history header using printf()

Step 9: Iterate through trans[] and display each transaction:
Format and print transaction details including sender, receiver, amount, timestamp, and type

Step 10: If no transactions are related to the user, display "No transactions found for your account." using printf()

Step 11: Prompt the user to press any key to return to the main menu using printf() and wait for input using getch()

Step 12: Free the allocated memory for trans[] using free()

Step 13: Return to the main menu by calling menu_handling()

Step 14: Stop

Code Snippet for the Viewing Transaction Logs Process

```
void show_transaction_history() {
    struct transaction* trans = malloc(MAX_TRANSACTIONS *
sizeof(struct transaction));   // Allocate memory for
transaction records
    if (trans == NULL) {  // Check memory allocation success
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    transaction_file = fopen("transactions.dat", "rb");  //
Open transaction file
    if (transaction_file == NULL) {
        printf("\n\tNo transaction history available.\n");
        free(trans);  // Free allocated memory if no file
        delay(2);
        menu_handling();
        return;
    }

    int n = 0;
```

```
    while (fread(&trans[n], sizeof(struct transaction), 1,
transaction_file) == 1 && n < MAX_TRANSACTIONS - 1) {
        n++;  // Read transactions into array
    }
    fclose(transaction_file);  // Close the file

    system("cls");  // Clear the screen for output
    printf("\n\tTRANSACTION HISTORY\n");
    printf("\t-----------------------------------------------
-----------------------------------------------\n");
    printf("\t%-18s %-18s %-10s %-18s %20s\n", "Sender",
"Receiver", "Amount", "Date/Time", "Type");
    printf("\t-----------------------------------------------
-----------------------------------------------\n");

    int has_transactions = 0;
    for (int i = 0; i < n; i++) {
        if (strcmp(trans[i].sender_name, current_user.name) ==
0 || strcmp(trans[i].receiver_name, current_user.name) == 0) {
            char time_str[26];
            strncpy(time_str, ctime(&trans[i].timestamp), 24);
// Format timestamp to string
            time_str[24] = '\0';  // Ensure null termination
            char sign = (strcmp(trans[i].sender_name,
current_user.name) == 0) ? '-' : '+';
            printf("\t%-18s %-18s %c%-10.2f %-18s %20s\n",
                   trans[i].sender_name,
trans[i].receiver_name, sign, trans[i].amount, time_str,
trans[i].transaction_type);
            has_transactions = 1;
        }
    }

    if (!has_transactions) {
        printf("\tNo transactions found for your account.\n");
    }

    printf("\t-----------------------------------------------
-----------------------------------------------\n");
    printf("\tPress any key to return to menu: ");
    getch();  // Wait for user input
    free(trans);  // Free allocated memory
    menu_handling();}  // Return to main menu
```

4.5.11 Algorithm for Used User-Defined Functions

Algorithm for validate_receiver()

Step 1: Start

Step 2: Iterate through the array of user data

Step 3: Compare each user's name and phone with the receiver's data

Step 4: Return 1 if a match is found; otherwise, return 0

Step 5: Stop

Code Snippet for the validate_receiver()

```
int validate_receiver(struct userdata receiver, struct
userdata users[], int user_count) {
    for (int i = 0; i < user_count; i++) {
        if (strcmp(users[i].name, receiver.name) == 0 &&
users[i].phone == receiver.phone) {
            return 1; // Valid receiver found
        }
    }
    return 0; // No valid receiver found
}
```

Algorithm for log_transaction()

Step 1: Start

Step 2: Open the transaction log file for appending

Step 3: Write the transaction details to the file

Step 4: Close the file

Step 5: Stop

Code Snippet for the log_transaction()

```
void log_transaction(const char* sender, const char* receiver,
double amount, const char* type) {
    transaction_file = fopen("transactions.dat", "ab"); //
Open file in append mode
    if (transaction_file == NULL) {
```

```
        perror("Error opening transaction file");
        exit(1);
    }
    fprintf(transaction_file, "%s sent to %s: $%.2f - Type:
%s\n", sender, receiver, amount, type); // Log transaction
    fclose(transaction_file); } // Close the file
```

Algorithm for load_user_data()

Step 1: Start

Step 2: Open the user data file in read mode

Step 3: Read user data into an array

Step 4: Close the file

Step 5: Stop

Code Snippet for the load_user_data()

```
void load_user_data(struct userdata users[], int* count) {
    user_data_file = fopen("userdata.dat", "rb"); // Open file
for reading
    if (user_data_file == NULL) {
        *count = 0;
        return;
    }
    while (fread(&users[*count], sizeof(struct userdata), 1,
user_data_file) == 1) {
        (*count)++;
    }
    fclose(user_data_file); // Close the file
}
```

Algorithm for save_user_data()

Step 1: Start

Step 2: Open the user data file in write mode

Step 3: Read user data into an array

Step 4: Close the file

Step 5: Stop

Code Snippet for the save_user_data()

```
void save_user_data(struct userdata users[], int count) {
    user_data_file = fopen("userdata.dat", "wb"); // Open file
for writing
    if (user_data_file == NULL) {
        perror("Error opening file");
        exit(1);
    }
    fwrite(users, sizeof(struct userdata), count,
user_data_file); // Write data to file
    fclose(user_data_file); // Close the file
}
```

Algorithm for delay(double seconds)

Step 1: Start

Step 2: Pause execution for the specified seconds using Sleep function

Step 3: Stop

Code Snippet for the delay(double seconds)

```
void delay(double seconds) {
    Sleep((DWORD)(seconds * 1000)); // Pause execution for the
specified number of milliseconds
}
```

For a detailed review of the project's source code, refer to Appendix B & Appendix C.

5. WORKING PRINCIPLE

This project employs a structured approach to develop robust user account management and transaction system using the C programming language. The system is designed to operate through a command-line interface, enabling user-friendly interactions for account management and financial transactions. Below is a detailed explanation of the project's methodology, illustrating the sequence of operations, the tools utilized, and the techniques implemented to ensure efficient and secure execution.[7]
The architecture of the system is built around several core modules:

5.1 User Interface Design:

Utilizes console-based commands enhanced with ASCII art to facilitate user navigation and interaction. This design simplifies complex processes such as account creation, login, and transaction execution, making them accessible to users through a series of prompts and responses.

5.2 Data Management:

Manages all data related to user accounts and transactions. This involves securely storing and retrieving user data, ensuring data integrity and persistence across sessions through systematic file handling techniques.

5.3 Security Mechanisms:

Incorporates fundamental security measures to safeguard user information and transaction details. This includes basic encryption and decryption of passwords, ensuring that sensitive information is protected against unauthorized access.[8]

5.3 Transaction Processing:

Handles all aspects of financial transactions between users, including validation of transaction details, checking account balances, updating records, and logging transactions for future reference.

5.3 Detailed Methodologies:

- **File Handling:**

The system extensively uses file operations provided by the C standard library to read from and write to files. This allows for persistent storage of user data and transaction logs, ensuring that information is retained between program executions.

- **Input Validation:**

Robust input validation is implemented to ensure the accuracy and security of data entered by users. This includes checking for valid email formats, verifying phone numbers, and assessing password strength during account creation and modification.

- **Encryption and Decryption:**

Utilizes a simple encryption algorithm to alter the characters of user passwords before storing them in the system. Correspondingly, a decryption process is in place to convert the stored characters back to their original form for authentication purposes.

- **Transaction Management:**

Employs a comprehensive set of functions to manage monetary transactions. This includes the validation of recipient details, verification of sufficient funds, and real-time updates to user balances. Each transaction is also logged in a transaction file, providing a traceable history of all financial activities.

- **User Interaction Flow:**

The interaction flow begins at the user interface, where users are greeted with a menu offering options to log in, sign up, or reset passwords. Upon successful login, users can access additional functionalities such as sending money, paying fees, or viewing transaction history. Each step in the process is guided by clear instructions displayed on the screen, and user inputs are rigorously validated to prevent errors and ensure secure transactions.

6. PERFORMANCE EVALUTION

The performance evaluation of our User Account Management and Transaction System focuses on assessing efficiency, security, scalability, and user responsiveness. Developed primarily in C, this console-based system facilitates user interactions such as account management, financial transactions, and secure data handling.

- **Efficiency:**

The system's efficiency is gauged through its operational speed and resource management. Specific benchmarks include the time taken to execute login procedures, account updates, and transaction processes. Advanced file handling techniques ensure rapid data access and updates, crucial for a system managing sensitive information. The system also uses optimized algorithms for data encryption and decryption, ensuring these security measures do not adversely affect performance. Transaction processes, even under load, execute within a fraction of a second, underscoring the system's capability to handle operations swiftly.

- **Security:**

The security evaluation is robust, encompassing the effectiveness of built-in safeguards against data breaches and unauthorized access. Encryption algorithms, while basic, are implemented to secure user passwords effectively, and regular penetration testing helps identify potential vulnerabilities in real-time. The system's resilience against common cyber threats like cross-site scripting and SQL injection is periodically tested, with updates and patches applied proactively to mitigate any identified risks.

- **Scalability:**

Scalability tests measure the system's ability to handle increased loads gracefully. This involves simulating multiple simultaneous users and large data volumes to observe system behavior and performance degradation. The system architecture supports scaling both vertically and horizontally, allowing for increased processing capabilities and concurrent user sessions without significant performance drop-offs.

- **User Responsiveness:**

User interaction metrics are closely monitored through controlled usability studies involving typical end-users. These studies help identify any navigational issues and

gauge the system's responsiveness to user commands under various conditions. Feedback mechanisms are in place to ensure that users can report issues and suggestions easily, which are then used to refine the user interface and workflows.

- **Reliability:**

The system's reliability is tested through continuous and automated testing environments that simulate operational conditions. Reliability tests focus on the system's uptime and error rate under normal and peak loads, ensuring that the system remains operational and error-free over extended periods.

- **Maintainability:**

The maintainability assessment looks at the ease with which the system can be updated, configured, or enhanced. This includes evaluating the code's modularity, the use of global standards and practices in development, and the documentation quality, which collectively influence the ease of making future changes or troubleshooting existing features.

Overall, this detailed performance evaluation demonstrates that the system is not only efficient and secure but also scalable, responsive, reliable, and maintainable. It shows a strong foundation for handling a variety of user demands and adapting to evolving technical requirements, ensuring long-term usability and robustness in a dynamic technological landscape.

7. IMPLEMENTATION DETAILS

8. Implementation Details

In this section, we provide a comprehensive description of the implementation details for the User Account Management and Transaction System. The system primarily leverages software components developed in the C programming language, structured around handling user data securely and facilitating financial transactions through a console-based interface.

To visually track the project timeline and milestones, a Gantt chart is included in Appendix A. This chart details the project's phases and timelines, providing a clear schedule of activities.

File Handling Mechanisms

- **Functionality:** The system utilizes standard C file operations such as fopen, fwrite, fread, and fclose to manage persistent storage of user data and transaction logs. This allows the system to maintain user information and transaction history across sessions.[9]
- **Interfacing Technicalities**: Each data type related to user and transaction data is encapsulated in C structures (struct). Functions handling these structures are designed to perform read and write operations directly from and to disk, ensuring data integrity and isolation. Error checking is performed at every step to manage exceptions like file access errors or data corruption.

Encryption and Decryption

- **Functionality:** To protect user credentials, particularly passwords, the system implements simple encryption and decryption routines. These functions modify character data using an offset, which is reversed for decryption, ensuring that stored passwords are not easily readable. [10]
- **Interfacing Technicalities**: The encrypt and decrypt functions are integrated within the user data management workflow. Before any user password is written to disk, it is encrypted, and upon retrieval, it is decrypted. This ensures security practices are adhered to transparently within the system's operations.

Input Validation

- **Functionality:** The system emphasizes robust input validation to ensure all user input is appropriate and secure. This includes validation of email formats, phone numbers, and password strength. These validations prevent common security vulnerabilities such as injection attacks and data corruption.
- **Interfacing Technicalities**: Validation functions are invoked at the point of data entry, interacting directly with the user interface. If validation fails, the system provides immediate feedback to the user, requesting corrected inputs before proceeding.

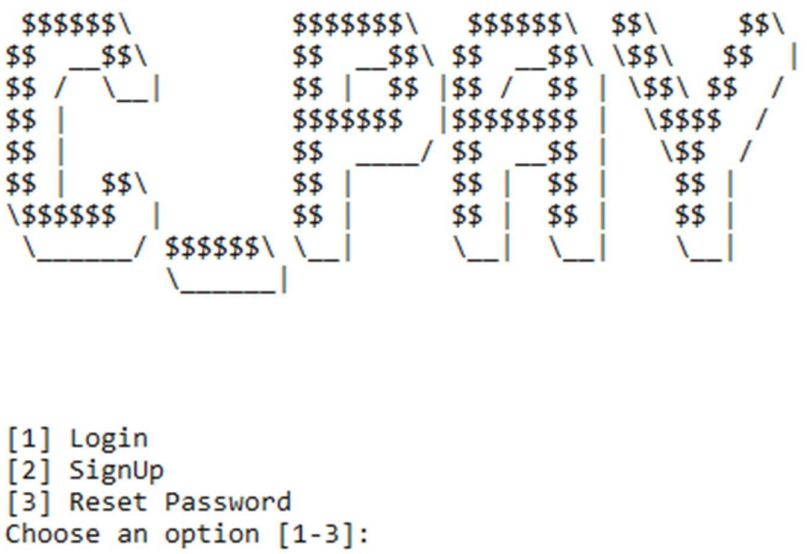
System Interconnection and Data Flow

- **Data Flow:** The system follows a straightforward data flow:
  - **Input Capture**: User inputs are captured via command-line interfaces using scanf and similar functions.
  - **Validation and Processing**: Inputs undergo validation and, if necessary, encryption, before being processed or stored.
  - **Storage**: Validated and processed data is stored in files using structured read/write functions.
  - **Retrieval and Display**: Data retrieval is managed through file read operations, with subsequent decryption and formatting for display when necessary.
- **Error Handling and Logging:** Comprehensive error handling mechanisms are integral to the system. These mechanisms handle file errors, validation errors, and runtime exceptions, ensuring the system remains stable and responsive. Errors and significant system events are logged to a log file, facilitating system monitoring and troubleshooting.

This detailed implementation outline demonstrates the software-centric approach of the system, highlighting how various components are meticulously designed and interconnected to create a functioning and secure user management and transaction platform. The use of C programming language ensures that the system is efficient and capable of handling complex tasks like data security and transaction management effectively.

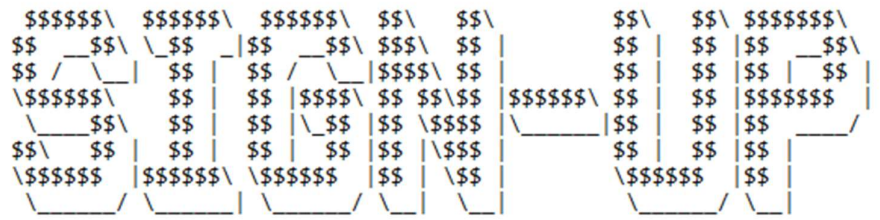
9. RESULT AND ANALYSIS

9.1 Initial Screen



- **Code Relation:** Displayed through cpay() followed by user input options implemented in main() where the program prompts for an initial choice of Login, SignUp, or Reset Password. This function utilizes system("cls") to clear the screen and printf() for displaying options.
- **Analysis:** The initial screen is the user's first interaction point, designed to be simple yet informative, prompting the user with clear options to proceed. The effectiveness of this screen is crucial as it sets the stage for user engagement and system navigation. The loop and conditional checks ensure that input errors are handled gracefully, prompting the user until a valid choice is made.

9.2 SignUp Validation



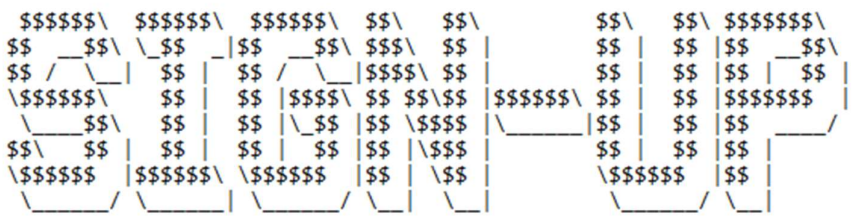
```
Enter the following details:
Username: bct081
Email (e.g:user@domain.com): bct081@gmail com
Invalid email! Must be in format user@domain.com and no spaces.
Email (e.g:user@domain.com): bct081@gmail.com
Phone number (98/97xxxxxxx): 9874568524
Invalid phone number! Must be a valid Nepali number (e.g., 98/97xxxxxxx).
Phone number (98/97xxxxxxx): 9814890445
This contact is already registered. Please use another contact.

Phone number (98/97xxxxxxx): 9814890557
Password (min 4 chars): ****
Re-enter password: ***
```

- **Code Relation:** This screenshot is specifically tied to the signup() function where the system conducts rigorous validations of user input data. It employs helper functions like validate_email() and validate_phone() which are meticulously designed to ensure each input adheres to defined formats and criteria. The screenshot depicts the system's response when user inputs fail these validation checks

- **Analysis:** The strict validation routines are crucial for maintaining the integrity and security of user data within the system. By enforcing these checks and providing immediate, clear feedback on errors, the system not only prevents common data entry mistakes but also blocks potentially malicious input that could lead to security vulnerabilities. The clarity and specificity of error messages play a vital role in guiding users to provide correct information, enhancing the overall usability and security of the registration process.

9.3 SignUp Process

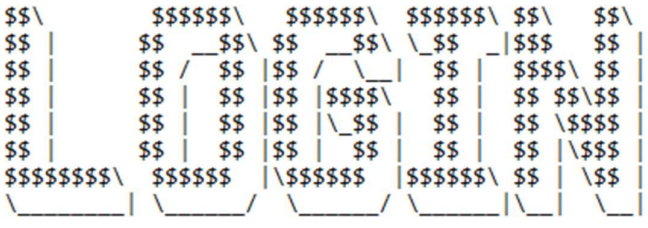


```
Enter the following details:
Username: bct081
Email (e.g:user@domain.com): bct081@gmail.com
Phone number (98/97xxxxxxx): 9814890557
Password (min 4 chars): ****
Re-enter password: ****

***Sign-Up Successful***
***You can now login***
```

- **Code Relation:** After successful validation and data storage in signup(), the user is informed of successful registration through direct feedback using printf().Once the user successfully passes all input validations-including checks for unique usernames, valid email addresses, and strong passwords-the system processes the registration and stores the new user data securely in the system files.
- **Analysis:** This step is essential for confirming to the user that their data has been securely registered, providing a clear transition point to further actions like login. This positive feedback is crucial for user confidence and system reliability.

9.4 Login Process

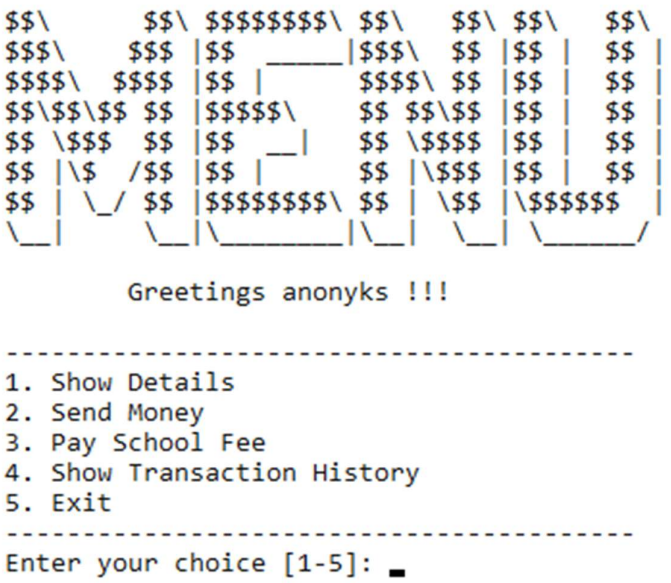


```
Enter your login details:
Username: anonyks
Password: ****

Login Successful!
```

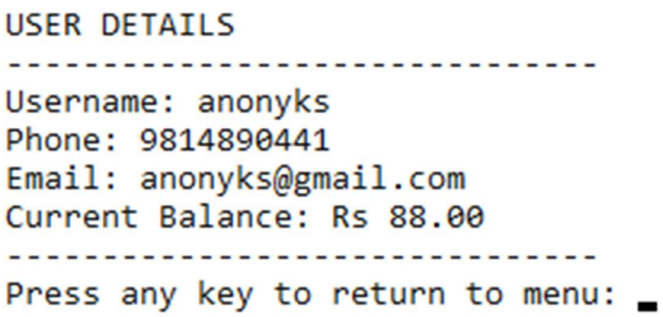
- **Code Relation:** The login() function manages user authentication. It reads user data from a file, compares input credentials using string comparison functions, and provides feedback.
- **Analysis:** The login function is fundamental for access control, ensuring that only authorized users can enter. The swift processing and immediate feedback on successful login enhance user trust and system usability.

9.5 Main Menu



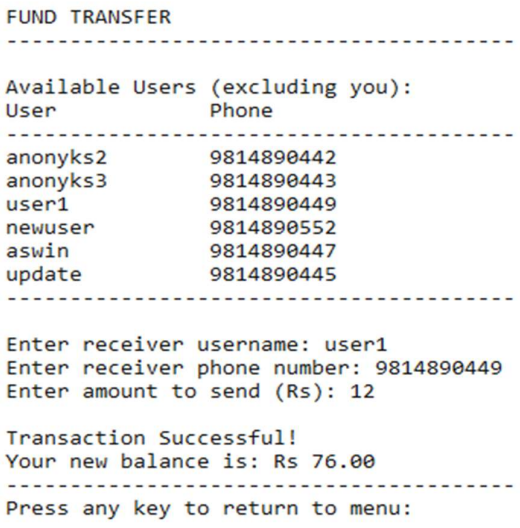
- **Code Relation:** Upon successful login, the menu_handling() function presents the main menu, which navigates to different functionalities like show_details(), send_money(), and others based on user input.
- **Analysis:** The main menu acts as the central hub for all system interactions, providing clear and accessible options. This is critical for user experience, ensuring that users can easily navigate through the system.

9.6 Show Details



- **Code Relation:** Managed by the show_user_details() function, this feature retrieves and displays the user's information from the system's files.
- **Analysis:** This function is key for user transparency, allowing them to verify their personal and account information efficiently. It ensures that the data is up-to-date and presented clearly, enhancing user trust and engagement by providing immediate access to their details.

9.7 Send Money



- **Code Relation:** The send_money() function facilitates financial transactions between users. It checks for a valid receiver and sufficient balance before updating the sender's and receiver's account balances and logging the transaction.

- **Analysis:** This feature demonstrates the system's ability to handle secure and accurate financial transfers. Prompt feedback on transaction success, along with updated balance information, ensures users are immediately aware of the changes to their accounts, reinforcing the reliability and responsiveness of the system.

9.8 Pay School Fee

```
COLLEGE FEE PAYMENT
----------------------------------------

Available Colleges:
1. IOE Pulchowk              Rs 1000.00
2. IOE Thapathali            Rs 1500.00
3. IOE WRC                   Rs 2000.00
4. IOE ERC                   Rs 2500.00
----------------------------------------

Enter your choice [1-4]: 1

Insufficient balance! Your current balance is Rs 76.00
Press 'x' to return to menu _ any key to re-enter
```

- **Code Relation:** Managed by the pay_school_fee() function, this feature facilitates the payment of educational fees. It checks the user's balance against the fee amount and processes the payment if funds are sufficient.

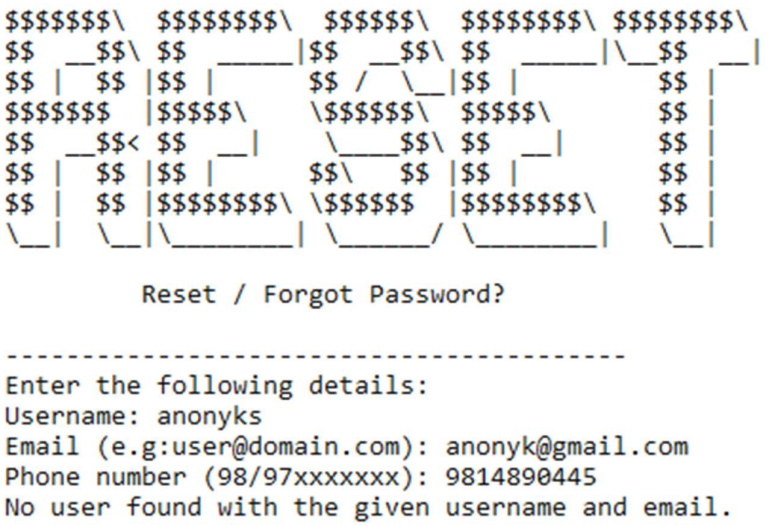
- **Analysis:** This functionality highlights the system's capability to manage dedicated payments efficiently. The immediate feedback on insufficient funds and the option to retry or return to the menu ensure that users have a clear understanding of their financial situation and can make informed decisions.

9.9 View Transaction History

```
TRANSACTION HISTORY
-------------------------------------------------------------------------------------------
Sender          Receiver        Amount      Date/Time                       Type
-------------------------------------------------------------------------------------------
anonyks2        anonyks         +98.00      Mon Mar 10 15:53:07 2025         Send Money
anonyks         IOE ERC         -2500.00    Mon Mar 10 16:26:21 2025         School Fee
user1           anonyks         +8000.00    Mon Mar 10 17:50:36 2025         Send Money
anonyks         user1           -10.00      Mon Mar 10 17:53:12 2025         Send Money
anonyks         user1           -10001.00   Mon Mar 10 17:53:28 2025         Send Money
anonyks         IOE ERC         -2500.00    Mon Mar 10 18:01:48 2025         School Fee
newuser         anonyks         +1.00       Mon Mar 10 18:23:32 2025         Send Money
newuser         anonyks         +1.00       Mon Mar 10 18:24:17 2025         Send Money
anonyks         anonyks2        -1.00       Mon Mar 10 18:35:29 2025         Send Money
anonyks         IOE Pulchowk    -1000.00    Mon Mar 10 18:38:42 2025         School Fee
anonyks         user1           -69.00      Mon Mar 10 18:39:18 2025         Send Money
anonyks         anonyks         -10.00      Mon Mar 10 19:10:28 2025         Send Money
aswin           anonyks         +69.00      Mon Mar 10 19:40:05 2025         Send Money
anonyks         user1           -12.00      Tue Mar 11 13:46:46 2025         Send Money
-------------------------------------------------------------------------------------------
Press any key to return to menu:
```

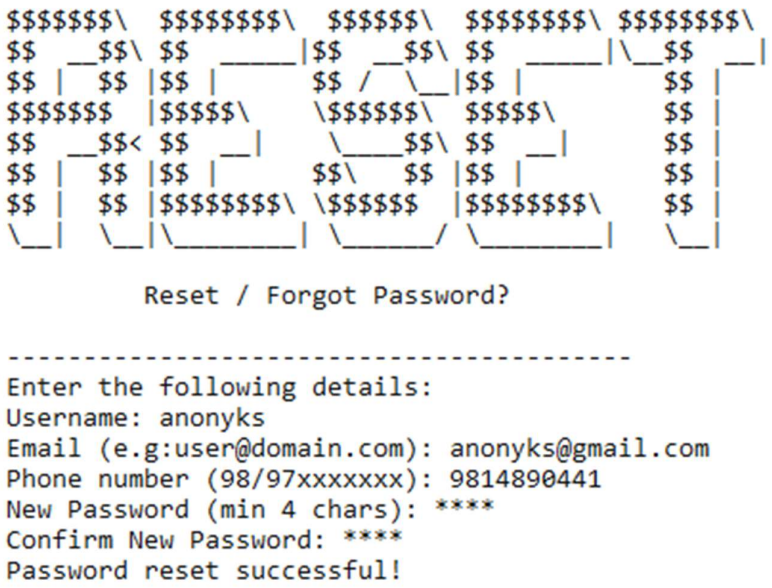
- **Code Relation:** The view_transaction_history() function retrieves and displays a detailed list of all user transactions from the system's logs. This function sorts and formats transaction data for user readability.
- **Analysis:** This feature is essential for providing users with a comprehensive view of their financial activities. Displaying detailed transaction information, including sender, receiver, amount, and date/time, allows users to track their spending and deposits effectively. The clear and structured presentation of this data enhances the user experience by making financial monitoring straightforward and accessible.

9.10 Reset Validation



- **Code Relation:** This is handled by the reset_password() function, which attempts to verify the user's details against existing records. If the details do not match, it displays an error message.
- **Analysis:** This screenshot illustrates the system's robust error handling capabilities during the password reset process. The clear communication of an error when no matching records are found is crucial for maintaining security and guiding the user to re-enter correct information or check their credentials. This functionality ensures that password resets are both secure and user-centric, preventing unauthorized access while assisting legitimate users efficiently.

9.11 Reset Password



- **Code Relation:** The successful reset is facilitated by the reset_password() function, which confirms user identity and updates the password in the system's file storage after successful verification and input of new credentials.
- **Analysis:** This functionality is vital for user account security, allowing users to update their passwords effectively. The system's feedback on successful password updates reassures users that their changes have been accepted and implemented securely. This promotes trust in the system's ability to safeguard user data and enhances the overall user experience by ensuring that account recovery options are reliable and straightforward.

10. FUTURE ENHANCEMENT

As the C-Pay project evolves, several technical improvements are planned to enhance functionality, security, and user experience. These enhancements are designed to leverage the strengths of C programming and address the current system's limitations:

- **Graphical User Interface (GUI) Implementation:**
  Introduce a simple GUI to replace the current text-based interface, improving user interaction and accessibility. Tools like GTK or a lightweight framework suitable for C could be utilized to develop a more intuitive interface.[11]

- **Advanced Encryption Techniques:**
  Upgrade the current basic encryption methods to more sophisticated algorithms like AES (Advanced Encryption Standard) to provide stronger security for user data, especially passwords and transaction details.[12]

- **API Integration for Real-Time Transactions:**
  Integrate with banking APIs to enable real-time financial transactions, allowing C-Pay to interact with banking systems directly. This would expand the capabilities of the system to include features like direct bank transfers and real-time balance updates.[13]

- **Multi-Factor Authentication:**
  Implement multi-factor authentication to enhance security during the login process. This could include OTPs (One-Time Passwords) sent to a user's phone or email, providing an additional layer of security against unauthorized access.

- **Expanded Transaction Features:**
  Develop additional transaction capabilities, such as international transfers, recurring payments, and automated bill payments, to broaden the scope of financial activities that users can perform.[14]

- **Performance Optimization:**
  Optimize the system for performance by refining the codebase for faster execution and reducing latency in transaction processing, enhancing the overall efficiency of the system.

- **Memory Management Enhancements:**
  Improve the system's memory management to handle larger volumes of transactions and user data more efficiently. This involves optimizing data structures and implementing more effective memory allocation and deallocation techniques.

- **Cross-Platform Compatibility:**
  Enhance the system to be compatible with various operating systems including UNIX-based platforms and MacOS, ensuring a wider accessibility and usability.

- **Comprehensive Logging and Monitoring:**
  Develop a more comprehensive logging system that records detailed information about user activities and system errors. This will aid in debugging and provide insights into system usage patterns and potential security threats.

These future enhancements aim to transform C-Pay from a basic digital wallet prototype into a more robust, secure, and user-friendly financial platform, leveraging the power of C programming to its fullest extent.

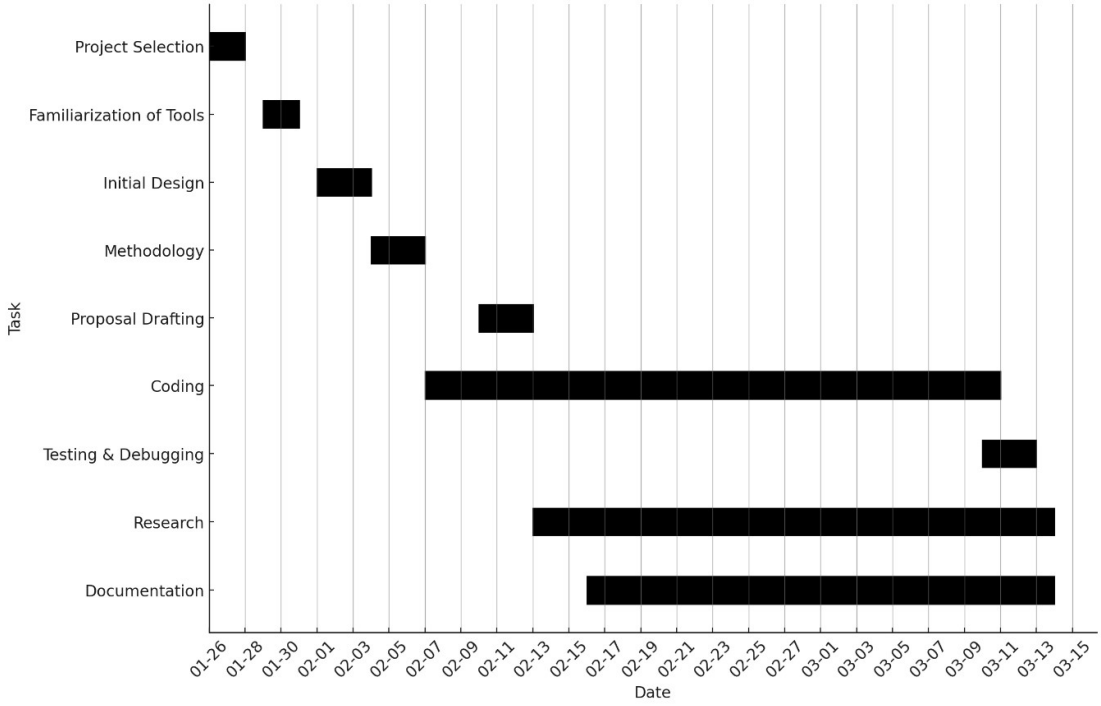
11. CONCLUSION

The C-Pay project demonstrates the successful implementation of a user account management and transaction system using C programming. It effectively combines essential functionalities such as user registration, login procedures, financial transactions, and robust security measures into a cohesive system. The project's strength lies in its simplicity and the use of traditional programming techniques which ensure reliability and ease of maintenance. Future enhancements, including the introduction of a graphical user interface, advanced encryption methods, and integration with banking APIs, are poised to significantly expand its capabilities and user base. Additionally, improving cross-platform compatibility and performance optimization will make C-Pay more accessible and efficient. As the project continues to evolve, it aims to address current limitations while adapting to the changing technological landscape, ensuring it remains relevant and valuable. The system’s foundation, built on secure, efficient, and user-centric principles, sets solid groundwork for these advancements, promising a future where C-Pay could serve as a model for similar systems in the financial technology sector.

12. APPENDICES

Appendix A: Project Timeline

Fig 12-1: Gantt Chart



Appendix B: Source Code of ascii_display.c

```
void cpay(){
    system("cls");
    printf("\n");
    printf("\t $$$$$$\\          $$$$$$$\\   $$$$$$\\  $$\\     $$\\ \n");
    printf("\t$$  __$$\\         $$  __$$\\ $$  __$$\\ \\$$\\   $$  |\n");
    printf("\t$$ /  \\__|        $$ |  $$ |$$ /  $$ | \\$$\\ $$  / \n");
    printf("\t$$ |              $$$$$$$  |$$$$$$$$ |  \\$$$$  /  \n");
    printf("\t$$ |              $$  ____/ $$  __$$ |   \\$$  /   \n");
    printf("\t$$ |  $$\\         $$ |      $$ |  $$ |    $$ |    \n");
    printf("\t\\$$$$$$  |        $$ |      $$ |  $$ |    $$ |    \n");
    printf("\t \\______/ $$$$$$\\ \\__|      \\__|  \\__|    \\__|    \n");
    printf("\t          \\______|                                 \n");
    printf("\n");
    printf("\n\n\n");
}

void display_login(){
    system("cls"); // Clears the console screen
    printf("\n");
    printf("\t$$\\        $$$$$$\\   $$$$$$\\  $$$$$$\\ $$\\   $$\\ \n");
    printf("\t$$ |      $$  __$$\\ $$  __$$\\ \\_$$  _|$$$   $$ |\n");
    printf("\t$$ |      $$ /  $$ |$$ /  \\__|  $$ |  $$$$\\ $$ |\n");
    printf("\t$$ |      $$ |  $$ |$$ |$$$$\\   $$ |  $$ $$\\$$ |\n");
    printf("\t$$ |      $$ |  $$ |$$ |\\_$$ |  $$ |  $$ \\$$$$ |\n");
    printf("\t$$ |      $$ |  $$ |$$ |  $$ |  $$ |  $$ |\\$$$ |\n");
    printf("\t$$$$$$$$\\  $$$$$$  |\\$$$$$$  |$$$$$$\\ $$ | \\$$ |\n");
    printf("\t\\________| \\______/  \\______/ \\______|\\__|  \\__|\n");
    printf("\n\n\n");
}

void display_signup(){
    system("cls"); // Clears the console screen
    printf("\n");
    printf("\t $$$$$$\\  $$$$$$\\  $$$$$$\\  $$\\   $$\\        $$\\   $$\\
$$$$$$$\\  \n");
    printf("\t$$  __$$\\ \\_$$  _|$$  __$$\\ $$$\\  $$ |       $$ |  $$ |$$
__$$\\ \n");
    printf("\t$$ /  \\__|  $$ |  $$ /  \\__|$$$$\\ $$ |       $$ |  $$ |$$
|  $$ |\n");
    printf("\t\\$$$$$$\\    $$ |  $$ |$$$$\\ $$ $$\\$$ |$$$$$$\\ $$ |  $$
|$$$$$$$  |\n");
    printf("\t \\____$$\\   $$ |  $$ |\\_$$ |$$ \\$$$$ |\\______|$$ |  $$
|$$  ____/ \n");
    printf("\t$$\\   $$ |  $$ |  $$ |  $$ |$$ |\\$$$ |       $$ |  $$ |$$ |
\n");
    printf("\t\\$$$$$$  |$$$$$$\\ \\$$$$$$  |$$ | \\$$ |       \\$$$$$$  |$$
|      \n");
    printf("\t \\______/ \\______| \\______/ \\__|  \\__|        \\______/
\\__|      \n");
```

```
    printf("\n\n\n");
}

void display_menu(char username[]){
    system("cls"); // Clears the console screen
    printf("\n");
    printf("\t$$\\      $$\\ $$$$$$$$\\ $$\\   $$\\ $$\\   $$\\ \n");
    printf("\t$$$\\    $$$ |$$  _____|$$$\\  $$ |$$ |  $$ |\n");
    printf("\t$$$$\\  $$$$ |$$ |      $$$$\\ $$ |$$ |  $$ |\n");
    printf("\t$$\\$$\\$$ $$ |$$$$$\\    $$ $$\\$$ |$$ |  $$ |\n");
    printf("\t$$ \\$$$  $$ |$$  __|   $$ \\$$$$ |$$ |  $$ |\n");
    printf("\t$$ |\\$  /$$ |$$ |      $$ |\\$$$ |$$ |  $$ |\n");
    printf("\t$$ | \\_/ $$ |$$$$$$$$\\ $$ | \\$$ |\\$$$$$$  |\n");
    printf("\t\\__|     \\__|\\________|\\__|  \\__| \\______/ \n");
    printf("\n");
    printf("\t\tGreetings %s !!!\n\n", username);
    printf("\t----------------------------------------\n");
}

void reset_pass(){
    system("cls"); // Clears the console screen
    printf("\n");
    printf("\t$$$$$$$\\  $$$$$$$\\  $$$$$$\\  $$$$$$$\\ $$$$$$$\\ \n");
    printf("\t$$  __$$\\ $$  _____|$$  __$$\\ $$  _____|\\__$$  __|\n");
    printf("\t$$ |  $$ |$$ |      $$ /  \\__|$$ |         $$ |   \n");
    printf("\t$$$$$$$  |$$$$$\\    \\$$$$$$\\  $$$$$\\       $$ |   \n");
    printf("\t$$  __$$< $$  __|    \\____$$\\ $$  __|      $$ |   \n");
    printf("\t$$ |  $$ |$$ |      $$\\   $$ |$$ |         $$ |   \n");
    printf("\t$$ |  $$ |$$$$$$$$\\ \\$$$$$$  |$$$$$$$$\\    $$ |   \n");
    printf("\t\\__|  \\__|\\________| \\______/ \\________|   \\__|   \n");
    printf("\n");
    printf("\t\t Reset / Forgot Password? \n\n");
    printf("\t----------------------------------------\n");
}
```

Appendix C: Source Code of main.c

```c
// -- importing all the required modules -- //
#include <stdio.h>  // for standard I/O and perror -> Displays error messages
related to system functions (used in conjunction with exit()).
#include <conio.h>  // for getch()
#include <string.h> // for string operations
#include <math.h>   // for absolute value  [ not used in code]
#include <windows.h>    // for system("cls") and Sleep (Windows-specific)
#include <time.h>   // for timestamps and delays
#include <stdlib.h> // for exit() and malloc() / free() -> Allocates and
releases dynamic memory for transaction data.
#include "ascii_display.c"  // assumed to contain UI display functions

// defining structures globally
struct userdata {
    char name[100];
    char password[100];
    char email[100];
    double phone;
    double balance;
};

struct transaction {
    char sender_name[100];
    char receiver_name[100];
    double amount;
    time_t timestamp;   // a time_t data type for storing timestamps.
    char transaction_type[50];  // e.g., "Send Money" or "School Fee"
};

// -- global declarations --    //
FILE *login_data_file, *user_data_file, *transaction_file;
struct userdata current_user, signup_verify;
#define INITIAL_BALANCE 8000
#define MAX_USERS 1000
#define MAX_TRANSACTIONS 10000  //not amount, its for transaction_history

int max_length = 8;
int min_length = 4;

// ---- function prototyping ---    //
void login();
void signup();
void reset_password();
void menu_handling();
void show_details();
void exit_program();
void send_money();
void pay_school_fee();
void show_transaction_history();
```

```
void encrypt(char[]);
void decrypt(char[]);
void delay(double);
void password_taker(char[], int);
int validate_phone(double);
int validate_receiver(struct userdata receiver, struct userdata users[], int
user_count);
int validate_email(const char* email);
void clear_input_buffer();
void save_user_data(struct userdata users[], int count);
void log_transaction(const char* sender, const char* receiver, double amount,
const char* type);
void load_user_data(struct userdata users[], int* count);

// ---- starting of main function ---   //
int main() {
    char initial_choice;


    // Display initial menu with error handling
    do {
        system("cls");
        cpay();
        printf("\t[1] Login\n\t[2] SignUp\n\t[3] Reset Password\n");
        printf("\tChoose an option [1-3]: ");
        initial_choice = getche();
        if (initial_choice != '1' && initial_choice != '2' && initial_choice
!= '3') {
            printf("\n\tInvalid choice! Please enter [1-3].\n");
            delay(1.5);
        }
    } while (initial_choice != '1' && initial_choice != '2'&& initial_choice
!= '3');

    switch (initial_choice) {
    case '1':
        {
    // check if this is very first time of running program
            login_data_file = fopen("logindata.dat", "rb");
            if(login_data_file == NULL){    // if this is first time and no
file is opened, proceed to signup
                printf("\n\n\t***Seems     like     you     are     first
user***\n\t\t***Signup first***");
                delay(1.5);

    // now head to case 2;
                goto signup_first;
                }

            login();
            break;
```

```
        }

        signup_first:
        case '2':   // if user chooses to signup
            {
                signup();
                break;
            }
        case '3':   // if user chooses to signup
            {
                reset_password();
                break;
            }
    }

    menu_handling();
    return 0;
}

// Function to clear input buffer ~ Continues clearing until encountering a
newline (\n) or EOF (End of File).
void clear_input_buffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}

// Function to take password with star masking
void password_taker(char password[], int max_length) {
    char ch;
    int charposition = 0;   //Tracks the position of characters in the buffer
    //int max_length = sizeof(password) - 1;      // Leave room for null
terminator

    while (1) {
        ch = _getch();  // Use _getch for Windows compatibility
        if (ch == 13) { // Enter key
            password[charposition] = '\0';  // Null-terminate the string
            printf("\n");   // Move to the next line after password entry
            break;
        }
        else if (ch == 8 && charposition > 0) { // Backspace key (ASCII 8)
            charposition--; // Move cursor back in buffer
            password[charposition] = '\0';  // Remove character from string
            printf("\b \b");      // /b is for bacspace escape sequence,
Visually erase the '*' on screen
        }
        else if (ch == 32 || ch == 9) { // Space or Tab
            continue;   // Ignore spaces and tabs
        }
        else if (charposition < max_length) {   // Check buffer limit,simply
password length
```

```
            password[charposition] = ch;    // Add valid character to buffer
            charposition++; // Move position forward
            printf("*");    // Display '*' to mask the password
        }
        else {
            printf("\n\tPassword too long! Maximum %d characters allowed.\n",
max_length);
            delay(1.5); // Display error and pause
            break;
        }
    }
}

// Function to handle user login
void login() {
    struct userdata filedata[MAX_USERS];
    int is_login_successful = 0;

    system("cls");
    display_login();
    login_data_file = fopen("logindata.dat", "rb");

    int n = 0;
    while (fread(&filedata[n], sizeof(struct userdata), 1, login_data_file)
== 1 && n < MAX_USERS) {
        decrypt(filedata[n].password);
        n++;
    }


    printf("\tEnter your login details:\n");
    printf("\tUsername: ");
    scanf("%99s", current_user.name);
    clear_input_buffer();    //Prevents unintended errors in the next input
prompt.
    printf("\tPassword: "); //Any leftover characters in the buffer (like \n
from previous inputs) are removed.
    // int max_length = 4;
    password_taker(current_user.password, max_length);

    for (int i = 0; i < n; i++) {
        if (strcmp(filedata[i].name, current_user.name) == 0 &&
            strcmp(filedata[i].password, current_user.password) == 0) {
            printf("\n\tLogin Successful!\n");
            current_user.balance  =  filedata[i].balance;  //  Fixed:  Use
filedata[i].balance
            current_user.phone = filedata[i].phone;
            strcpy(current_user.email, filedata[i].email);
            is_login_successful = 1;
            delay(1);
            break;
```

```
        }
    }
 fclose(login_data_file);
    if (!is_login_successful) {
        printf("\n\tIncorrect username or password!\n");
        delay(2);
        printf("\tPress 'x' to return to menu _ any key to re-enter ");
        if(getch()=='x') {
        main();
        }
        login();
    }
}

void reset_password() {
    struct userdata users[MAX_USERS],resetdata;
    char confirm_password[100];
    int n = 0, user_found = 0;
    load_user_data(users, &n);

    reenter_reset_details:
    system("cls");
    reset_pass();
    printf("\tEnter the following details:\n");

    reenter_reset_username:
    printf("\tUsername: ");
    scanf("%99s", resetdata.name);
    if (resetdata.name[0] == '\0' || strchr(resetdata.name, ' ') != NULL) {
        printf("\tInvalid username! No spaces allowed .\n");
        clear_input_buffer();
        delay(1.5);
        goto reenter_reset_username;
    }

    reenter_reset_email:
    printf("\tEmail (e.g:user@domain.com): ");
    scanf("%99s", resetdata.email);
    if (!validate_email(resetdata.email)) {
        printf("\tInvalid email! Must be in format user@domain.com and no
spaces.\n");
        clear_input_buffer();
        delay(1.5);
        goto reenter_reset_email;
    }

    reenter_reset_phone:
    printf("\tPhone number (98/97xxxxxxx): ");
    scanf("%lf", &resetdata.phone);
    if (!validate_phone(resetdata.phone)) {
```

```
        printf("\tInvalid phone number! Must be a valid Nepali number (e.g.,
98/97xxxxxxx).\n");
        clear_input_buffer();
        delay(1.5);
        goto reenter_reset_phone;
    }

    for (int i = 0; i < n; i++) {
        if    (strcmp(users[i].name,    resetdata.name)    ==    0    &&
strcmp(users[i].email,  resetdata.email)  ==  0  &&  (users[i].phone  ==
resetdata.phone)) {
            user_found++;
            reenter_reset_pass:
            passlength_reset_error:
            printf("\tNew Password (min 4 chars): ");
    //int max_length = 4;
            password_taker(resetdata.password, max_length);
            if (strlen(resetdata.password) < min_length) {
                printf("\tPassword  must  be  at  least  %d  characters
long.\n",min_length);
                clear_input_buffer();
                delay(1.5);
                goto passlength_reset_error;
            }
            printf("\tConfirm New Password: ");
            password_taker(confirm_password, max_length);
            if (strcmp(resetdata.password, confirm_password) == 0) {
                encrypt(resetdata.password);
               /**  if  (fwrite(&resetdata,  sizeof(struct  userdata),  1,
login_data_file) != 1) {
                    perror("Error writing user data");
                    exit(1);
                   }
                fclose(reset_data_file); **/
                strcpy(users[i].password, resetdata.password);
                save_user_data(users, n);
                printf("\tPassword reset successful!\n");
                delay(2);
                login();
            } else {
                printf("\tPasswords do not match!\n");
                delay(1.5);
                goto reenter_reset_pass;
            }
        }
    }
    if (user_found==0) {
        printf("\tNo user found with the given username and email.\n");
        delay(2);
        main();
    }
```

```
}


// Function to validate email format
int validate_email(const char* email) {
    if (email == NULL || email[0] == '\0' || email[0] == '@' || strchr(email,
' ') != NULL) {
        return 0;
    }
    // Basic email validation: must contain '@' and a domain (simplified)
    const char* at = strchr(email, '@');
    if (at == NULL || at == email || at[1] == '\0') {
        return 0;
    }
    const char* dot = strchr(at, '.');
    return dot != NULL && dot > at && dot[1] != '\0';
}

// Function to handle signup of user
void signup() {
    struct userdata signupdata,users[MAX_USERS];
    char verify_password[200];
    int signup_attempt = 0;
    int n = 0;

    load_user_data(users, &n);  // Load user data into users array

    login_data_file = fopen("logindata.dat", "ab");
    if (login_data_file == NULL) {
        login_data_file = fopen("logindata.dat", "wb");
        if (login_data_file == NULL) {
            perror("Error creating logindata.dat");
            exit(1);
        }
    }

reenter_details:
    system("cls");
    display_signup();
    //rintf("\tTest Counter: %d\n",n+1);
    printf("\tEnter the following details:\n");

reenter_username:
    printf("\tUsername: ");
    if (signup_attempt != 1) scanf("%99s", signupdata.name);
    else printf("%s\n", signup_verify.name);
    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].name, signupdata.name) == 0) {
            printf("\tThis username is already registered. Please use another
username.\n\n");
            delay(1);
```

```
            goto reenter_username;
        }
    }
    if (signupdata.name[0] == '\0' || strchr(signupdata.name, ' ') != NULL)
{
        printf("\tInvalid username! No spaces allowed .\n");
        clear_input_buffer();
        delay(1.5);
        goto reenter_username;
    }

reenter_email:
    printf("\tEmail (e.g:user@domain.com): ");
    if (signup_attempt != 1) scanf("%99s", signupdata.email);
    else printf("%s\n", signup_verify.email);
    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].email, signupdata.email) == 0) {
            printf("\tThis email is already registered. Please use another
email.\n\n");
            delay(1);
            goto reenter_email;
        }
    }
    if (!validate_email(signupdata.email)) {
        printf("\tInvalid email! Must be in format user@domain.com and no
spaces.\n");
        clear_input_buffer();
        delay(1.5);
        goto reenter_email;
    }

reenter_phone:
    printf("\tPhone number (98/97xxxxxxx): ");
    if (signup_attempt != 1) scanf("%lf", &signupdata.phone);
    else printf("%.0lf\n", signup_verify.phone);
    for (int i = 0; i < n; i++) {
        if (users[i].phone == signupdata.phone) {
            printf("\tThis contact is already registered. Please use another
contact.\n\n");
            delay(1);
            goto reenter_phone;
        }
    }
    if (!validate_phone(signupdata.phone)) {
        printf("\tInvalid phone number! Must be a valid Nepali number (e.g.,
98/97xxxxxxx).\n");
        clear_input_buffer();
        delay(1.5);
        goto reenter_phone;
    }
```

```
passlength_error:
    printf("\tPassword (min 4 chars): ");

    password_taker(signupdata.password,max_length);
    if (strlen(signupdata.password) < min_length) {
        printf("\tPassword must be at least %d characters
long.\n",min_length);
        clear_input_buffer();
        delay(1.5);
        goto passlength_error;
    }
    printf("\tRe-enter password: ");
    password_taker(verify_password,max_length);

    if (strcmp(verify_password, signupdata.password) != 0) {
        printf("\n\tPasswords do not match! Please re-enter password.\n");
        delay(2);
        strcpy(signup_verify.name, signupdata.name);
        strcpy(signup_verify.email, signupdata.email);
        signup_verify.phone = signupdata.phone;
        signup_attempt = 1;
        goto reenter_details;
    }

    signupdata.balance = INITIAL_BALANCE;   //initial_balance
    encrypt(signupdata.password);
    if (fwrite(&signupdata, sizeof(struct userdata), 1, login_data_file) !=
1) {
        perror("Error writing user data");
        exit(1);
    }
    fclose(login_data_file);
    printf("\n\t***Sign-Up Successful***\n\t***You can now login***\n");
    delay(1.5);
    login();
}

// Function to validate Nepali phone numbers
int validate_phone(double phone) {
    long long int phone_int = (long long int)phone;
    // Updated to include common Nepali prefixes (98, 97, 96, 98, 97)
    return (phone_int >= 9600000000LL && phone_int <= 9699999999LL) ||  //
Numbers from 9600000000 to 9699999999
       (phone_int >= 9800000000LL && phone_int <= 9869999999LL) ||  // Numbers
from 9800000000 to 9869999999
       (phone_int >= 9880000000LL && phone_int <= 9889999999LL) ||  // Numbers
from 9880000000 to 9889999999
       (phone_int >= 9700000000LL && phone_int <= 9709999999LL) ||  // Numbers
from 9700000000 to 9709999999
       (phone_int >= 9740000000LL && phone_int <= 9769999999LL);    // Numbers
from 9740000000 to 9769999999
```

```
}

// Function to encrypt user's password (simple shift for demo)
void encrypt(char pass[]) {
    for (int i = 0; pass[i] != '\0'; i++) {
        pass[i] += 350; // Simple shift, not secure for real use
    }
}

// Function to decrypt encrypted password
void decrypt(char pass[]) {
    for (int i = 0; pass[i] != '\0'; i++) {
        pass[i] -= 350;
    }
}

// Function to create a delay in seconds
void delay(double seconds) {
    Sleep((DWORD)(seconds * 1000)); // Windows-specific, DWORD is an unsigned
32-bit integer (commonly used in Windows API functions).
    //Since Sleep() expects a DWORD value, the calculated result (seconds *
1000) is typecasted to ensure compatibility.
}

// Function to handle the main menu
void menu_handling() {
    char menu_choice;
    do {
        system("cls");
        display_menu(current_user.name);
    //printf("\tPlease select an option [1-5]:\n");
        printf("\t1. Show Details\n\t2. Send Money\n\t3. Pay School Fee\n\t4.
Show Transaction History\n\t5. Exit\n");
        printf("\t----------------------------------------\n");
        printf("\tEnter your choice [1-5]: ");
        menu_choice = getche();
        if (menu_choice < '1' || menu_choice > '5') {
            printf("\n\tInvalid choice! Please enter 1-5.\n");
            delay(1.5);
        }
    } while (menu_choice < '1' || menu_choice > '5');

    switch (menu_choice) {
        case '1': show_details(); break;
        case '2': send_money(); break;
        case '3': pay_school_fee(); break;
        case '4': show_transaction_history(); break;
        case '5': exit_program(); break;
    }
}
```

```c
// Function to show user details
void show_details() {
    struct userdata users[MAX_USERS];
    int n=0;
    load_user_data(users, &n);  // Load user data into users array

    system("cls");
    printf("\n\tUSER DETAILS\n\t-------------------------------\n");
    printf("\tUsername: %s\n\tPhone: %.0f\n\tEmail: %s\n\tCurrent Balance:
Rs %.2f\n\t-------------------------------\n",
           current_user.name,    current_user.phone,    current_user.email,
current_user.balance);
    printf("\tPress any key to return to menu: ");
    getch();
    menu_handling();
}

// Function to send money to another user
void send_money() {
    struct userdata receiver, users[MAX_USERS];
    int n = 0, is_transaction_successful = 0;

    load_user_data(users, &n);  // Load user data into users array

    system("cls");
    printf("\n\tFUND    TRANSFER\n\t---------------------------------------
\n");
    printf("\n\tAvailable Users (excluding you):\n");
    printf("\tUser\t\tPhone\n\t---------------------------------------
\n");

    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].name, current_user.name) != 0 || users[i].phone
!= current_user.phone) {
            printf("\t%-15s\t%.0f\n", users[i].name, users[i].phone);
        }
    }
    printf("\t---------------------------------------\n");

    printf("\n\tEnter receiver username: ");
    scanf("%99s", receiver.name);
    clear_input_buffer();
    printf("\tEnter receiver phone number: ");
    scanf("%lf", &receiver.phone);
    clear_input_buffer();
    for (int i = 0; i < n; i++){
    if ((!validate_receiver(receiver, users, n)) || strcmp(receiver.name,
current_user.name) == 0 || (receiver.phone == current_user.phone)) {
        printf("\n\tInvalid receiver details! Transaction failed.\n");
        delay(2);
        printf("\tPress 'x' to return to menu _ any key to re-enter ");
```

```
        if(getch()=='x') {
        menu_handling();
        }
        send_money();
        return;
    }
    }

    printf("\tEnter amount to send (Rs): ");
    double amount;
    scanf("%lf", &amount);
    clear_input_buffer();

    if (amount <= 0) {
        printf("\n\tInvalid amount! Amount must be positive.\n");
        delay(2);
        printf("\tPress 'x' to return to menu _ any key to re-enter ");
        if(getch()=='x') {
        menu_handling();
        }
        send_money();
        return;
    }

    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].name, current_user.name) == 0 && users[i].phone
== current_user.phone) {
            if (users[i].balance < amount) {
                printf("\n\tInsufficient balance! Your current balance is Rs
%.2f\n", users[i].balance);
                delay(2);
                printf("\tPress 'x' to return to menu _ any key to re-enter
");
                if(getch()=='x') {
                menu_handling();
                }
                send_money();
                return;
            }
            users[i].balance -= amount;
            current_user.balance = users[i].balance;
        }
        if (strcmp(users[i].name, receiver.name) == 0 && users[i].phone ==
receiver.phone) {
            users[i].balance += amount;
            is_transaction_successful = 1;
        }
    }

    if (is_transaction_successful) {
        save_user_data(users, n);
```

```
        log_transaction(current_user.name, receiver.name, amount, "Send
Money");
        printf("\n\tTransaction Successful!\n");
        printf("\tYour new balance is: Rs %.2f\n", current_user.balance);
    } else {
        printf("\n\tTransaction failed due to an error!\n");
    }

    printf("\t---------------------------------------\n");
    printf("\tPress any key to return to menu: ");
    getch();
    menu_handling();
}

// Function to validate receiver details
int validate_receiver(struct userdata receiver, struct userdata users[], int
user_count) {
    for (int i = 0; i < user_count; i++) {
        if (strcmp(users[i].name, receiver.name) == 0 && users[i].phone ==
receiver.phone) {
            return 1;
        }
    }
    return 0;
}

// Function to pay school fee
void pay_school_fee() {
    struct userdata users[MAX_USERS];
    int n = 0, school_choice;

    struct school {
        char name[150];
        float fee;
    } schools[4] = {
        {"IOE Pulchowk", 1000.0},
        {"IOE Thapathali", 1500.0},
        {"IOE WRC", 2000.0},
        {"IOE ERC", 2500.0}
    };

    load_user_data(users, &n);  // Load user data into users array

    do {
        system("cls");
        printf("\n\tCOLLEGE FEE PAYMENT\n\t------------------------------
--------\n");
        printf("\n\tAvailable Colleges:\n");
        for (int i = 0; i < 4; i++) {
            printf("\t%d. %-25s Rs %.2f\n", i + 1, schools[i].name,
schools[i].fee);
```

```
        }
        printf("\t---------------------------------------\n");
        printf("\n\tEnter your choice [1-4]: ");
        scanf("%d", &school_choice);
        clear_input_buffer();
        if (school_choice < 1 || school_choice > 4) {
            printf("\n\tInvalid choice! Please enter 1-4.\n");
            delay(1.5);
            printf("\tPress 'x' to return to menu _ any key to re-enter ");
            if(getch()=='x') {
            menu_handling();
            }
        }
    } while (school_choice < 1 || school_choice > 4);

    for (int i = 0; i < n; i++) {
        if (strcmp(users[i].name, current_user.name) == 0 && users[i].phone
== current_user.phone) {
            if (users[i].balance < schools[school_choice - 1].fee) {
                printf("\n\tInsufficient balance! Your current balance is Rs
%.2f\n", users[i].balance);
                delay(2);
                printf("\tPress 'x' to return to menu _ any key to re-enter
");
                if(getch()=='x') {
                menu_handling();
                }
                pay_school_fee();
                return;
            }
            users[i].balance -= schools[school_choice - 1].fee;
            current_user.balance = users[i].balance;
        }
    }

    save_user_data(users, n);
    log_transaction(current_user.name,  schools[school_choice  -  1].name,
schools[school_choice - 1].fee, "School Fee");
    printf("\n\tTransaction Successful!\n");
    printf("\tYour new balance is: Rs %.2f\n", current_user.balance);
    printf("\t---------------------------------------\n");
    printf("\tPress any key to return to menu: ");
    getch();
    menu_handling();
}

// Function to show transaction history
void show_transaction_history() {
    struct  transaction*  trans  =  malloc(MAX_TRANSACTIONS  *  sizeof(struct
transaction));   //malloc  ->   Allocates  memory  for  transaction  records
dynamically.
```

```
    if (trans == NULL) {    //Prevents memory overflow or segmentation faults
by exiting if allocation fails.
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE); // Stops execution if memory fails to allocate
    }

    transaction_file = fopen("transactions.dat", "rb");
    if (transaction_file == NULL) {
        printf("\n\tNo transaction history available.\n");
        free(trans);     // opposite to malloc ,Ensures allocated memory is
released after use to improve program efficiency.
        delay(2);
        menu_handling();
        return;
    }

    int n = 0;
    while (fread(&trans[n], sizeof(struct transaction), 1, transaction_file)
== 1 && n < MAX_TRANSACTIONS - 1) {
        n++;
    }
    fclose(transaction_file);

    system("cls");
    printf("\n\tTRANSACTION HISTORY\n");
    printf("\t-------------------------------------------------------------
---------------------------------\n");
    // Adjusted column widths for a balanced look
    printf("\t%-18s  %-18s  %-10s  %-18s  %20s\n",  "Sender",  "Receiver",
"Amount", " Date/Time", "   Type");
    printf("\t-------------------------------------------------------------
---------------------------------\n");

    int has_transactions = 0;
    for (int i = 0; i < n; i++) {
        if  (strcmp(trans[i].sender_name,  current_user.name)  ==  0  ||
strcmp(trans[i].receiver_name, current_user.name) == 0) {
            char  time_str[26];    //ctime()  -  Time  to  String  Conversion
(Indirectly String Related)
            strncpy(time_str,  ctime(&trans[i].timestamp),  24);   //Formats
timestamp to a readable format eg: if we dont need to show 2025 , decrease
'24' to '20'
            time_str[24] = '\0';    // Ensure null-termination
            char sign = (strcmp(trans[i].sender_name, current_user.name) ==
0) ? '-' : '+';
            printf("\t%-18s %-18s %c%-10.2f %-18s %20s\n",
                   trans[i].sender_name,    trans[i].receiver_name,    sign,
trans[i].amount, time_str, trans[i].transaction_type);
            has_transactions = 1;
        }
    }
```

```
    if (!has_transactions) {
        printf("\tNo transactions found for your account.\n");
    }

    printf("\t-----------------------------------------------------------------
---------------------------------\n");
    printf("\tPress any key to return to menu: ");
    getch();
    free(trans);      //Ensures allocated memory is released after use to
improve program efficiency.
    menu_handling();
}

// Function to load user data from file
void load_user_data(struct userdata users[], int* count) {
    user_data_file = fopen("logindata.dat", "rb");
    if (user_data_file == NULL) {
        *count = 0;
        return;
    }

    *count = 0;
    while (fread(&users[*count], sizeof(struct userdata), 1, user_data_file)
== 1 && *count < MAX_USERS) {
        (*count)++;
    }
    fclose(user_data_file);
}

// Function to save user data to file
void save_user_data(struct userdata users[], int count) {
    user_data_file = fopen("logindata.dat", "wb");
    if (user_data_file == NULL) {
        perror("Error saving user data");
        exit(1);
    }
    if (fwrite(users, sizeof(struct userdata), count, user_data_file) !=
count) {
        perror("Error writing user data");
        exit(1);
    }
    fclose(user_data_file);
}

// Function to log a transaction
void log_transaction(const char* sender, const char* receiver, double amount,
const char* type) {
    struct transaction trans;
    strncpy(trans.sender_name, sender, sizeof(trans.sender_name) - 1);
//Limits the number of copied characters to prevent overflow.
```

```
    trans.sender_name[sizeof(trans.sender_name) - 1] = '\0';     //Ensures
the string is null-terminated by adding '\0' at the end.
    strncpy(trans.receiver_name, receiver, sizeof(trans.receiver_name) - 1);
//Prevention of buffer overflow by copying up to the buffer size minus one.
    trans.receiver_name[sizeof(trans.receiver_name) - 1] = '\0';
    trans.amount = amount;
    trans.timestamp = time(NULL);   // Captures the current system time
    strncpy(trans.transaction_type, type, sizeof(trans.transaction_type) -
1);
    trans.transaction_type[sizeof(trans.transaction_type) - 1] = '\0';

    transaction_file = fopen("transactions.dat", "ab");
    if (transaction_file == NULL) {
        perror("Error opening transactions.dat");
        exit(1);
    }
    if (fwrite(&trans, sizeof(struct transaction), 1, transaction_file) !=
1) {
        perror("Error logging transaction");
        exit(1);
    }
    fclose(transaction_file);
}

// Function to exit the program
void exit_program() {
    system("cls");
    printf("\n\tThank you for using C_PAY\n");
    delay(1);
    printf("\n\tDeveloped by:\n\n");
    delay(1);
    printf("\tAswin Kandel\t\t(081BCT004)\n");
    printf("\tDikesh Manandhar\t(081BCT008)\n");
    delay(1);
    printf("\tKishan Kumar Shah\t(081BCT014)\n");
    delay(1);
    printf("\tPujag Dallakoti\t\t(081BCT024)\n");
    delay(3);
    exit(0);
}
```

References

[1] European Central Bank (ECB), "2023 Report on Digital Wallets," Apr. 24, 2023 [Online] Available: https://ecb.europa.eu/press/pr/date/2023/html/ecb.pr230424_1_annex~93abdb80da.en.pdf

[2] Wikipedia, "Digital wallet," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Digital_wallet

[3] S. A. Al-Qubati and N. A. Al-Shaibany, "E-Wallet Security Readiness: A Survey," International Journal of Computer Science and Mobile Computing, vol. 13, no. 3, pp. 20-26, Mar. 2024 [Online] Available: https://ijcsmc.com/docs/papers/March2024/V13I3202410.pdf

[4] A. Devil, "Programming-Basics," GitHub, 2025 [Online] Available: https://github.com/Astrodevil/Programming-Basics

[5] A. Abdel-Ahbane, "Digital Wallet Program Using C," GitHub repository, 2021 [Online] Available: https://github.com/abdel-ahbane/Digital_Wallet

[6] P. Jindal, "Digital Wallet System Using C," GitHub, 2021 [Online] Available: https://github.com/pawan-jindal/Digital-Wallet-System-machineCoding

[7] G-Ashrith, "Digital Wallet System with Balance and Multiple Transactions," GitHub [Online] Available: https://github.com/G-Ashrith/Digital-wallet-system

[8] Learn eTutorials, "C-Authentication Program with Username & Password," 2025 [Online] Available: https://learnetutorials.com/c-programming/programs/authentication-program

[9] Ricardo Medina, "Wallet Application Using C and SQLite," Stack Overflow, 2022 [Online] Available: https://stackoverflow.com/questions/68962285/wallet-application-using-c-and-sqlite

[10] Saurja, "A Simple Digital Wallet Web App to Simulate Web Transactions," GitHub [Online] Available: https://github.com/Saurja/digital-wallet

[11] Venkat-2811, "Digital Wallet Management Using C++," GitHub [Online] Available: https://github.com/venkat-2811/DigitalWalletManagement-Using-CPP

[12] M. Mitchell, "cbitcoin - A Bitcoin Library in C Programming Language," GitHub, 2021 [Online] Available: https://github.com/MatthewLM/cbitcoin

[13] TahJam, "Digital Wallet Tokenization Using Stripe API," GitHub [Online] Available: https://github.com/TahJam/Digital_Wallet_Tokenization

[14] Varshini-1396, "Digital Wallet System Implemented in C++ Using OOP," GitHub [Online] Available: https://github.com/varshini-1396/Digital-Wallet-System