

Cryptocurrency Wallet Application Report

Table of Contents

- Introduction
 - Overview of the project
 - Purpose and goals of the project
 - Scope of the cryptocurrency wallet application
- Project Requirements
 - Functional requirements
 - Non-functional requirements
 - Tools, technologies, and libraries used
- CLI Code Explanation
- GUI Code Explanation
- Conclusion
 - Summary of accomplishments and learning outcomes
 - Reflection on project success and areas for improvement
- References
 - List of referenced books, websites, or libraries
- Appendices
 - Appendix A
 - Appendix B
 - Appendix C
 - Appendix D
 - Appendix E
 - Appendix F
 - Appendix G

1. Introduction

Overview of the Project

This cryptocurrency wallet application allows users to manage digital assets by creating wallets, logging in, depositing, withdrawing, and checking balances. The system includes both a command-line interface (CLI) and a graphical user interface (GUI).

Purpose and Goals of the Project

The project aims to provide a basic cryptocurrency wallet that supports multiple cryptocurrencies. The goal is to develop secure user authentication, wallet management, and ensure data integrity using file handling.

Scope of the Cryptocurrency Wallet Application

The wallet supports a limited number of cryptocurrencies and uses a simple password hashing mechanism to ensure user security. It can be expanded to include more features, such as multi-user support and integration with a database.

2. Project Requirements

Functional Requirements

- Create and manage user wallets.
- Secure password authentication and verification.
- Cryptocurrency deposit and withdrawal.
- Check wallet balance.

Non-Functional Requirements

- Performance: Fast response times for user interactions.
- Security: Secure password handling and file storage.
- Usability: Easy to use with both CLI and GUI interfaces.

Tools, Technologies, and Libraries Used

- **C++** for core application logic.
- **unordered_map** for managing wallet data.
- **fstream** for file I/O (storing user data and wallet balances).

CLI Code Breakdown

1. Included libraries for core functionality and utilities

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <unordered_map>
#include <string>
#include <vector>
#include <iomanip>
#include <algorithm>
```

Figure 1.1

Explanation

1. **#include <fstream>:** It enables file stream operations, such as reading from and writing to files using `std::ifstream` and `std::ofstream`.
2. **#include <sstream>:** It facilitates string stream operations, allowing you to work with strings as streams using `std::stringstream`.
3. **#include <unordered_map>:** It offers a hash table-based associative container for storing key-value pairs with fast lookup, insertion, and deletion.
4. **#include <vector>:** It provides a dynamic array implementation.
5. **#include <iomanip>:** It supplies utilities for input/output manipulation, like setting precision or controlling formatting.
6. **#include <algorithm>:** It contains a collection of algorithms for operations like searching, sorting, and manipulating containers.

2. Hash Function for password

```

string hashPassword(const string& password)
{
    unsigned long hash = 5381;
    for (char c : password)
    {
        hash = ((hash << 5) + hash) + c;
    }
    return to_string(hash);
}

```

Figure 1.2

Explanation

string hashPassword defines a function named *hashPassword* that returns a string and *const string& password* accepts a constant reference to a string called password as input.

A variable **hash** of type **unsigned long** is initialized to **5381**. This value is often used in hash functions as a starting "seed" for hashing.

Loop iterates over the password characters through each character **c** in the input string password.

hash = ((hash << 5) + hash) + c;

This line updates the hash variable using a specific formula:

Left shifts hash by 5 bits, equivalent to multiplying it by 32 and then adds the original hash, making it effectively $hash * 33$ and $+c$ adds the ASCII value of the current character **c**. It is the core step of the hashing process.

return to_string(hash);

Converts the final numeric hash value to a `std::string` using `std::to_string` and returns the hashed string to the caller.

3. Function to check password is in breach

```

bool isPasswordBreached(const string& password)
{
    ifstream breachFile("pwd.txt");
    string breachedPassword;
    while (getline(breachFile, breachedPassword))
    {
        if (breachedPassword == password)
        {
            return true;
        }
    }
    return false;
}

```

Figure 1.3

Explanation

bool isPasswordBreached(const string& password)

- This function checks if a given password exists in a list of breached passwords. `bool`—returns true if the password is found in the breached list, otherwise false.
- `const string& password`: A constant reference to a string representing the password to check. Using a reference avoids copying the string, and `const` ensures the function doesn't modify it.

ifstream breachFile("pwd.txt");

- Opens a file named `pwd.txt` in read mode using an input file stream (**ifstream**).
- **breachFile** is an object of type `ifstream` used to read from the file.

The file **pwd.txt** contains a list of breached passwords, each on a new line.

string breachedPassword;

Declares a variable `breachedPassword` to store each password read from the file.

while (getline(breachFile, breachedPassword))

- Loop checks through the file line by line.
- `getline(breachFile, breachedPassword)` reads the next line from `breachFile` and stores it in `breachedPassword`.
- The loop continues until there are no more lines to read (i.e., until `getline` returns false).

if (breachedPassword == password)

```
{
    return true;
}
```

- It compares the breachedPassword (from the file) with the input password.
- If both match, the function immediately returns true, indicating the password is found in the breached list.

```
return false;
```

- If the loop completes without finding a match, the function returns false, indicating the password is not in the breached list.

4. Function to convert a string to uppercase

```
// Function to convert a string to uppercase
string toUpperCase(const string& str)
{
    string result = str;
    transform(result.begin(), result.end(), result.begin(), ::toupper);
    return result;
}
```

Figure 1.4

Explanation:

string toUpperCase(const string& str)

- It converts all the characters in the input string `str` to uppercase and returns the resulting string.
- `const string& str` is a constant reference to the input string.
- **const**: Ensures the input string `str` is not modified within the function.
- **&**: Indicates the string is passed by reference, avoiding a costly copy of the input.

string result = str;

- It creates a mutable copy of the input string `str` named **result**. This copy will be modified to store the uppercase version.
- The input `str` is `const`, so we cannot modify it directly. The copy allows us to safely make changes without altering the original string.

transform(result.begin(), result.end(), result.begin(), ::toupper);

- Its purpose is to convert all characters in result to uppercase using the transform function.
- **transform**: A standard library algorithm that applies a transformation function to a range of elements.

Syntax: transform(start, end, destination_start, function).

- **Arguments:**
 - **result.begin()**: The starting iterator of the string result.
 - **result.end()**: The ending iterator of the string result.
 - **result.begin()**: The destination where the transformed characters are stored (overwrites the original string result in this case).
 - **::toupper**: The transformation function. ::toupper converts a single character to its uppercase equivalent.

return result;

- It returns the modified string result containing all uppercase characters to the caller.

5. Function to create a wallet

```
void createWallet()
{
    string email, password, hashedPassword, walletFile;
    cout << "Enter your email: ";
    cin >> email;
    cout << "Enter your password: ";
    cin >> password;

    // Check if the password is breached
    if (isPasswordBreached(password))
    {
        cout << "Password found in breach. Please use another password.\n";
        return;
    }

    hashedPassword = hashPassword(password);
    walletFile = email + "_wallet.txt";

    // Store user details in accounts.txt
    ofstream accountsFile("accounts.txt", ios::app);
    accountsFile << email << " " << hashedPassword << " " << walletFile << endl;
    accountsFile.close();

    // Create an empty wallet file
    ofstream wallet(walletFile);
    wallet.close();

    cout << "Wallet created successfully!\n";
}
```

Figure 1.5

Explanation

void createWallet()

- This function allows a user to create a new wallet by providing an email and password. It stores user credentials and creates an associated wallet file.
- void—this function does not return any value.

string email, password, hashedPassword, walletFile;

- Declares variables are **email**, **password**, **hashedPassword** and **walletFile**.
- It prompts the user to enter their email and password.
- The user inputs their email and password, which are stored in email and password.

if (isPasswordBreached(password))

```
{
    cout << "Password found in breach. Please use another password.\n";
    return;
}
```

- It checks if the entered password is in a list of breached passwords.
- Calls the function `isPasswordBreached(password)`
 - If true, it means the password is compromised.
- If the password is breached, the user is notified with a message.
- The function exits early using `return`, preventing the wallet creation process.

hashedPassword = hashPassword(password);

- It converts the user's plaintext password into a hashed format for secure storage.
- Calls `hashPassword(password)`
 - This function hashes the password using a secure hashing algorithm.
 - `hashedPassword` holds the resulting hash.

walletFile = email + "_wallet.txt";

- This generates a unique wallet file name for the user based on their email.

ofstream accountsFile("accounts.txt", ios::app);

```
accountsFile << email << " " << hashedPassword << " " << walletFile << endl;
```


accountsFile.close();

- It appends the user's information to a file named accounts.txt
- accounts.txt contains lines with email, hashed password, and wallet file name.

<email> <hashedPassword> <walletFile>

- **Steps:**

1. Opens *accounts.txt* in append mode (ios::app), so new entries don't overwrite existing ones.
2. Writes the user's details (email, hashed password, and wallet file name).
3. Closes the file to ensure the data is saved.

ofstream wallet(walletFile);

wallet.close();

- It creates an empty file for the user's wallet.

- **Steps:**

1. Opens a file with the name stored in walletFile (e.g., user@example.com_wallet.txt).
2. Immediately closes the file, leaving it empty and ready for use.

cout << "Wallet created successfully!\n";

- It informs the user that the wallet has been successfully created.

6. Function to log in

```

// Function to log in
bool login(string& walletFile)
{
    string email, password, hashedPassword;
    cout << "Enter your email: ";
    cin >> email;
    cout << "Enter your password: ";
    cin >> password;

    hashedPassword = hashPassword(password);

    ifstream accountsFile("accounts.txt");
    string storedEmail, storedPassword, storedWalletFile;
    while (accountsFile >> storedEmail >> storedPassword >> storedWalletFile)
    {
        if (email == storedEmail && hashedPassword == storedPassword)
        {
            walletFile = storedWalletFile;
            cout << "Logged in successfully!\n";
            return true;
        }
    }
    cout << "Invalid credentials. Please try again.\n";
    return false;
}

```

Figure 1.6

Explanation

bool login(string& walletFile)

- It verifies the user's email and password to authenticate them. If successful, it sets the walletFile for further use.
- **bool**—returns true if login is successful, otherwise false.
- **string& walletFile**: A reference to a string that will store the user's wallet file name upon successful login. This allows the caller to access the wallet file.

string email, password, hashedPassword;

- Declares variables such as **email**, **password**, **hashedPssword**

hashedPassword = hashPassword(password);

- Hashes the input password using a secure hashing function for comparison with stored hashed passwords.
- **Function Call**: Calls `hashPassword(password)`
 - Converts the plaintext password to a secure hashed version.
 - Stores the result in `hashedPassword`.

```
ifstream accountsFile("accounts.txt");
```

- It opens the file accounts.txt, which stores user credentials and wallet file names.
 - Each line contains: <email> <hashedPassword> <walletFile>.

Variable Declarations for Reading File Data

```
string storedEmail, storedPassword, storedWalletFile;
```

- Declares variables to temporarily hold data read from each line of accounts.txt:
 - Stores the email, hashed password, wallet file name from the file.

```
while (accountsFile >> storedEmail >> storedPassword >> storedWalletFile)
```

- It reads the file line by line, extracting the storedEmail, storedPassword, and storedWalletFile values.

```
if (email == storedEmail && hashedPassword == storedPassword)
```

```
{
```

```
    walletFile = storedWalletFile;
```

```
    cout << "Logged in successfully!\n";
```

```
    return true;
```

```
}
```

- Its purpose is to compare the user-provided email and hashed password with the values read from the file.
- Steps:
 1. Condition:
 - Checks if the input email matches storedEmail.
 - Checks if the hashed version of the input password matches storedPassword.
 2. If Match Found:
 - Assigns the corresponding wallet file name (storedWalletFile) to the reference variable walletFile.
 - Displays a success message and returns true to indicate successful login.
 3. If No Match is Found

```
cout << "Invalid credentials. Please try again.\n";
```

```
return false;
```

- If the loop finishes without finding a matching email and password it displays an error message and Returns false to indicate login failure.

7. Function to deposit currency

```
void deposit(string& walletFile)
{
    string coinName;
    double amount;

    cout << "Enter the cryptocurrency you want to deposit (BTC, ETH, SOL, TRX, USDT): ";
    cin >> coinName;
    coinName = toUpperCase(coinName); // Convert to uppercase
    cout << "Enter the amount to deposit: ";
    cin >> amount;

    unordered_map<string, double> wallet;
    ifstream walletFileIn(walletFile);
    string coin;
    double balance;
    while (walletFileIn >> coin >> balance)
    {
        wallet[coin] = balance;
    }
    walletFileIn.close();

    wallet[coinName] += amount;

    ofstream walletFileOut(walletFile);
    for (const auto& entry : wallet)
    {
        walletFileOut << entry.first << " " << fixed << setprecision(2) << entry.second << endl;
    }
    walletFileOut.close();

    cout << "Deposit successful!\n";
}
```

Figure 1.7

Explanation

void deposit(string& walletFile)

- This function allows the user to deposit a specified amount of cryptocurrency (such as BTC, ETH, etc.) into their wallet. It updates the wallet file to reflect the new balance.
- **string& walletFile:** A reference to a string that holds the path of the user's wallet file. The wallet file stores the user's cryptocurrency balances.

Declares:

- **coinName:** A string to store the name of the cryptocurrency the user wants to deposit (e.g., BTC, ETH).
- **amount:** A double to store the amount of cryptocurrency to deposit.
- Prompts the user to input the name of the cryptocurrency they wish to deposit.
- **Input:** The input is stored in coinName.

- **toUpperCase Function:**
 - Converts the entered cryptocurrency name to uppercase to standardize it (e.g., "btc" becomes "BTC"). This ensures uniformity in the wallet file (which expects uppercase currency codes).

```
cout << "Enter the amount to deposit: ";
```

```
cin >> amount;
```

Prompts the user to input the amount of cryptocurrency they want to deposit. The amount is stored in the amount variable.

```
unordered_map<string, double> wallet;
```

```
ifstream walletFileIn(walletFile);
```

- It creates an unordered map wallet to store the current balances of cryptocurrencies.
 - The map will use the cryptocurrency name (coin) as the key and the balance (balance) as the value.
- *ifstream walletFileIn(walletFile)* opens the wallet file for reading.
- *walletFileIn*: An ifstream object to read from the specified wallet file.

```
string coin;
```

```
double balance;
```

```
while (walletFileIn >> coin >> balance)
```

```
{
```

```
wallet[coin] = balance;
```

```
}
```

```
walletFileIn.close();
```

- It reads the existing balances from the wallet file and stores them in the wallet unordered map.
 - Reads each line from the file, extracting the cryptocurrency name (coin) and its corresponding balance (balance).
 - The *wallet[coin] = balance* line inserts or updates the balance for each cryptocurrency in the map.
- **Closing the File:** After reading the data, the file is closed using *walletFileIn.close()*.

wallet[coinName] += amount;

- It adds the deposit amount (amount) to the existing balance of the specified cryptocurrency (coinName).
 - If the cryptocurrency (coinName) is not already in the wallet, it initializes the balance with the deposit amount.
 - If the cryptocurrency is already in the wallet, the deposit is added to the existing balance.

ofstream walletFileOut(walletFile);

- It opens the wallet file for writing. The updated balances will be written back to the file.
- *ofstream walletFileOut(walletFile)* opens the file in write mode, overwriting its contents.

for (const auto& entry : wallet)

{

walletFileOut << entry.first << " " << fixed << setprecision(2) << entry.second << endl;

}

- Writes the updated balances back to the wallet file.
- Loop iterates through each key-value pair in the wallet map:
 - **entry.first:** The cryptocurrency name.
 - **entry.second:** The balance of that cryptocurrency.
- **Formatting:**
 - The fixed and setprecision(2) manipulators ensure the balance is written with two decimal places, making it look like a monetary value (e.g., 1.50 instead of 1.5).

walletFileOut.close();

- It closes the wallet file after the updated balances are written.

cout << "Deposit successful!\n";

- Displays a message confirming that the deposit has been successfully processed.

8. Function to withdraw cryptocurrency

```

void withdraw(string& walletFile)
{
    string coinName;
    double amount;

    cout << "Enter the cryptocurrency you want to withdraw (BTC, ETH, SOL, TRX, USDT): ";
    cin >> coinName;
    coinName = toUpperCase(coinName); // Convert to uppercase
    cout << "Enter the amount to withdraw: ";
    cin >> amount;

    unordered_map<string, double> wallet;
    ifstream walletFileIn(walletFile);
    string coin;
    double balance;
    while (walletFileIn >> coin >> balance)
    {
        wallet[coin] = balance;
    }
    walletFileIn.close();

    if (wallet[coinName] < amount)
    {
        cout << "Insufficient balance!\n";
        return;
    }

    wallet[coinName] -= amount;

    ofstream walletFileOut(walletFile);
    for (const auto& entry : wallet)
    {
        walletFileOut << entry.first << " " << fixed << setprecision(2) << entry.second << endl;
    }
    walletFileOut.close();

    cout << "Withdrawal successful!\n";
}

```

Figure 1.8

Explanation

void withdraw(string& walletFile)

- This function allows the user to withdraw a specified amount of cryptocurrency (like BTC, ETH, etc.) from their wallet. It updates the wallet file to reflect the new balance.
- `string& walletFile`: A reference to a string that holds the file name (path) of the user's wallet. The file stores the user's cryptocurrency balances.
- `coinName` and `amount` are declared.

coinName = toUpperCase(coinName);

It prompts the user to input the name of the cryptocurrency they want to withdraw.

- The input is stored in `coinName`.

- **toUpperCase Function:** Converts the input cryptocurrency name to uppercase to standardize it (e.g., "btc" becomes "BTC"). This ensures uniformity, as wallet files expect uppercase cryptocurrency codes.
- Prompts the user to input the amount of cryptocurrency they want to withdraw. The input is stored in amount.

```
unordered_map<string, double> wallet;
```

```
ifstream walletFileIn(walletFile);
```

- This creates an unordered map wallet to store the current balances of cryptocurrencies.
- *ifstream walletFileIn(walletFile)* opens the wallet file for reading.
 - walletFileIn: An ifstream object to read from the specified wallet file.

```
string coin;
```

```
double balance;
```

```
while (walletFileIn >> coin >> balance)
```

```
{  
    wallet[coin] = balance;  
}
```

```
walletFileIn.close();
```

- Reads the existing balances from the wallet file and stores them in the wallet unordered map.
- The >> operator reads each line from the file and splits it into the coin (cryptocurrency name) and balance (cryptocurrency balance).
- The wallet[coin] = balance line inserts or updates the balance for each cryptocurrency in the map.
- After reading the data, the file is closed with walletFileIn.close().

```
if (wallet[coinName] < amount)
```

```
{  
    cout << "Insufficient balance!\n";  
    return;  
}
```


- This checks if the user has enough balance of the selected cryptocurrency to complete the withdrawal.
- **Condition:**
 - If the current balance for the specified cryptocurrency (`wallet[coinName]`) is less than the requested withdrawal amount (`amount`), the function displays an error message: Insufficient balance!.
 - The function returns early without making any changes to the wallet file if the balance is insufficient.

wallet[coinName] -= amount;

- Deducts the withdrawal amount (`amount`) from the balance of the specified cryptocurrency (`coinName`).
 - If the withdrawal amount is valid and there are sufficient funds, this line subtracts the amount from the existing balance.
 - If the cryptocurrency doesn't exist in the wallet, this line will create a new entry for it with a negative balance (though this shouldn't happen unless a mistake occurs).

ofstream walletFileOut(walletFile);

- It opens the wallet file for writing, to save the updated balances after the withdrawal.
- *ofstream walletFileOut(walletFile)* opens the file in write mode, overwriting its contents.

for (const auto& entry : wallet)

{

walletFileOut << entry.first << " " << fixed << setprecision(2) << entry.second << endl;

}

- Writes the updated balances to the wallet file.
- Iterates through each key-value pair in the wallet map:
- **entry.first:** The cryptocurrency name (e.g., BTC, ETH).
- **entry.second:** The balance of the cryptocurrency
- `fixed` and `setprecision(2)` ensure that the balance is displayed with two decimal places, making it look like a monetary value (e.g., 1.50 instead of 1.5).

walletFileOut.close();

- It closes the wallet file after the updated balances have been written to it.
- Displays a message confirming that the withdrawal has been successfully processed.

9. Function to check balance

```
void checkBalance(string& walletFile)
{
    ifstream walletFileIn(walletFile);
    string coin;
    double balance;

    cout << "Your wallet balance:\n";
    while (walletFileIn >> coin >> balance)
    {
        cout << coin << ": " << fixed << setprecision(2) << balance << endl;
    }
    walletFileIn.close();
}
```

Figure 1.9

Explanation

void checkBalance(string& walletFile)

- This function displays the user's cryptocurrency wallet balance. It reads from the wallet file and prints the balances for each cryptocurrency listed in the file.
- **void**—the function does not return any value.
- **string& walletFile**: A reference to a string that holds the file name (path) of the user's wallet file, which stores the cryptocurrency balances.

ifstream walletFileIn(walletFile);

- It opens the wallet file for reading using an ifstream (input file stream).
- **walletFileIn**: An ifstream object is created to handle reading the file contents. It takes the walletFile string (which contains the file path) as the file name.
- If the file cannot be opened, an error may occur (though it's not explicitly handled here).
- **string coin and double balance are declared**

cout << "Your wallet balance:\n";

- Prints a header message to indicate the beginning of the wallet balance output.

while (walletFileIn >> coin >> balance)

```
{
```

```
cout << coin << ": " << fixed << setprecision(2) << balance << endl;  
}
```

- Reads the cryptocurrency names and their balances from the wallet file and prints them.
- The while loop reads each line from the wallet file:
- *walletFileIn >> coin* reads the cryptocurrency name from the file (e.g., BTC, ETH) and stores it in *coin*.
- *walletFileIn >> balance* reads the balance of that cryptocurrency and stores it in *balance*.

For each cryptocurrency, the loop prints the name and balance

- The loop will continue until the end of the file is reached.

```
walletFileIn.close();
```

- It closes the wallet file after reading its contents.

10. Main function

```

int main()
{
    int choice;
    string walletFile;

    while (true)
    {
        cout << "Cryptocurrency Wallet Application\n";
        cout << "1. Create Wallet\n";
        cout << "2. Login\n";
        cout << "3. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice)
        {
            case 1:
                createWallet();
                break;
            case 2:
                if (login(walletFile))
                {
                    int walletChoice;
                    do
                    {
                        cout << "1. Deposit Cryptocurrency\n";
                        cout << "2. Withdraw Cryptocurrency\n";
                        cout << "3. Check Balance\n";
                        cout << "4. Logout\n";
                        cout << "Enter your choice: ";
                        cin >> walletChoice;

                        switch (walletChoice)
                        {
                            case 1:
                                deposit(walletFile);
                                break;
                            case 2:
                                withdraw(walletFile);
                                break;
                            case 3:
                                checkBalance(walletFile);
                                break;
                            case 4:
                                cout << "Logged out successfully!\n";
                                break;
                            default:
                                cout << "Invalid choice. Please try again.\n";
                        }
                    } while (walletChoice != 4);
                }
                break;
            case 3:
                cout << "Exiting the application. Goodbye!\n";
                return 0;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }

    return 0;
}

```

Figure 1.10

Explanation

- This function provides the user interface for a cryptocurrency wallet application, allowing users to create wallets, log in, deposit, withdraw, check balances, and exit.

int choice, string walletFile are declared in the start of main function.

- An infinite loop that continuously displays the main menu until the user chooses to exit the application.
- Displays the main menu options to the user. Reads the user's choice and stores it in choice.
- Executes different actions based on the user's choice using a switch statement.
- If the user selects option 1, the createWallet function is called to create a new wallet.
- Exits the switch block to return to the main menu.
- If the user selects option 2, the login function is called. If the login is successful:
 - The walletFile is set to the logged-in user's wallet file name.
 - A nested menu (wallet menu) is displayed, allowing the user to perform wallet-related actions.
- An integer to store the user's choice from the wallet menu.
- Calls the withdraw function, allowing the user to withdraw cryptocurrency from their wallet.
- Calls the checkBalance function, displaying the current balances of cryptocurrencies in the user's wallet.
- Logs the user out and exits the wallet menu. Returns to the main menu.
- If the user enters an invalid option in the wallet menu, displays an error message and prompts them to try again.
- Keeps the wallet menu running until the user chooses option 4 (Logout)
- Displays a goodbye message or exits the program by returning 0 in the main menu.
- If the user enters an invalid option in the main menu, displays an error message and prompts them to try again.

11. Sample Output

```
Cryptocurrency Wallet Application
1. Create Wallet
2. Login
3. Exit
Enter your choice: 1
Enter your email: goodreads101@gmail.com
Enter your password: 12#$56
Wallet created successfully!
Cryptocurrency Wallet Application
1. Create Wallet
2. Login
3. Exit
Enter your choice: 2
Enter your email: goodreads101@gmail.com
Enter your password: 12#$56
Logged in successfully!
1. Deposit Cryptocurrency
2. Withdraw Cryptocurrency
3. Check Balance
4. Logout
Enter your choice: 3
Your wallet balance:
1. Deposit Cryptocurrency
2. Withdraw Cryptocurrency
3. Check Balance
4. Logout
Enter your choice: 4
Logged out successfully!
Cryptocurrency Wallet Application
1. Create Wallet
2. Login
3. Exit
Enter your choice: 3
Exiting the application. Goodbye!
```

Figure 1.11 (Output of CLI Code)

GUI Code Breakdown

1. Main form page

Includes and Directives in the Code

```
#include <vcl.h>
#pragma hdrstop
#include "MainFormPage.h"
#include "CreateWalletPage.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
#include "LoginPage.h"
```

Figure 2.1

Explanation

#include <vcl.h>: This line includes the VCL (Visual Component Library) header file, which provides classes and functions for creating GUI applications in C++ Builder.

#pragma hdrstop: This is a compiler directive specific to the Borland/Embarcadero compiler used with VCL applications.

#include "MainFormPage.h": This line includes the header file for the **main form** of the application.

#include "CreateWalletPage.h": This line includes the header file for the **Create Wallet** form.

#pragma package(smart_init): This is another compiler directive that automates certain initialization tasks.

#pragma resource "*.dfm": This directive links the DFM (Delphi Form File) resource file with the code.

#include "LoginPage.h": This line includes the header file for the Login form.

Global Pointer

```
TMainForm *MainForm;
```

Figure 2.2

It defines a global pointer for the TMainForm object, which is the main form of the application.

Constructor

```
// Constructor
__fastcall TMainForm::TMainForm(TComponent* Owner)
: TForm(Owner)
{
}
```

Figure 2.3

Explanation

- A constructor for the TMainForm class.
- It uses the __fastcall calling convention, which is standard in VCL applications for performance optimization.
- The TComponent* Owner parameter specifies the component that owns this form, managing its lifetime.
- The body of the constructor is empty, implying no additional initialization is done.

Event handlers

```
// Create Wallet Button Click
void __fastcall TMainForm::btnCreateWalletClick(TObject *Sender)
{
    TCreateWalletForm *createWalletForm = new TCreateWalletForm(this);
    createWalletForm->ShowModal(); // Use ShowModal to display it as a dialog
}
```

Figure 2.4

Explanation

- Opens a new form for creating a wallet.
- **new TCreateWalletForm(this)**: Creates an instance of the TCreateWalletForm form.
- The current form (this) is passed as the owner, ensuring proper cleanup when the main form is destroyed.
- **ShowModal**: Displays the form as a modal dialog, meaning the user must close this dialog before interacting with the main form again.

Login buttons


```
// Login Button Click
void __fastcall TMainForm::btnLoginClick(TObject *Sender)
{
    TLoginForm *loginForm = new TLoginForm(this);
    loginForm->ShowModal();
}
```

Figure 2.5

Explanation

- This function opens a new form for user login.
- new TLoginForm(this): Creates an instance of the TLoginForm form.
- ShowModal: Displays the login form as a modal dialog.

Exit Button

```
// Exit Button Click
void __fastcall TMainForm::btnExitClick(TObject *Sender)
{
    Application->Terminate(); // Close the application
}
```

Figure 2.6

Explanation

- Terminates the application.
- Application->Terminate(): Gracefully shuts down the application and releases resources.

Output :

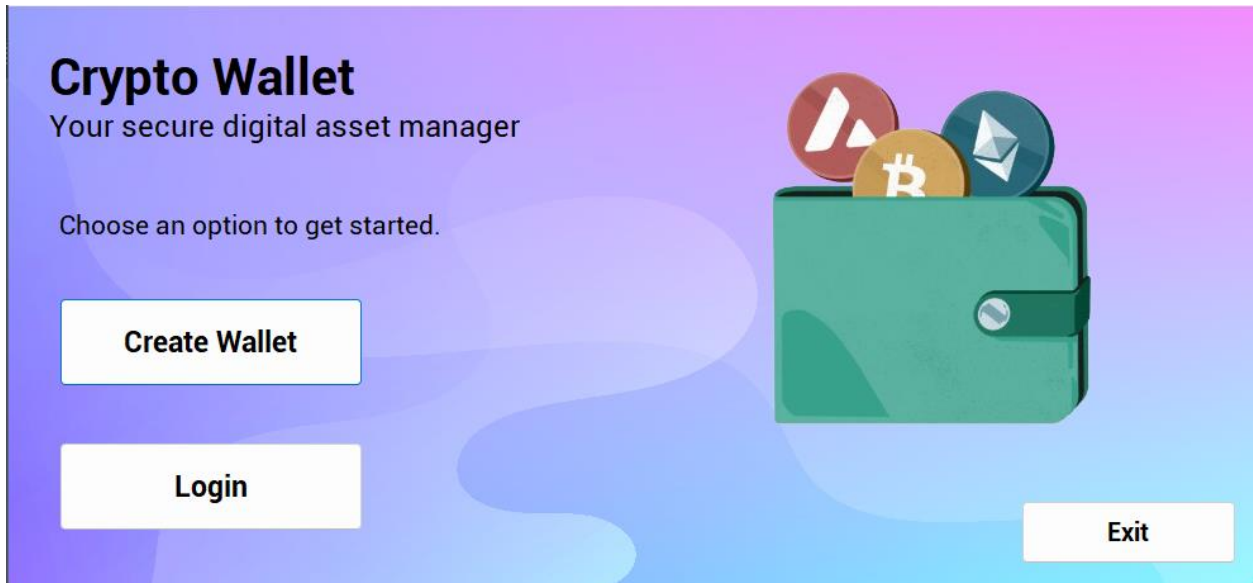


Figure 2.7 (Output of main page)

2. Login Page

Header Files

```
#include <vcl.h>
#pragma hdrstop
#include <fstream>
//#include "CommonFunctions.h"
#include "LoginPage.h"
#include "CreateWalletPage.h" // Include the header where hashPassword is declared
#include "DashBoardPage.h"

//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
```

Figure 2.8

- **#include <vcl.h>**: Includes the Visual Component Library for GUI components.
- **#include <fstream>**: Allows file input/output operations.
- **#include "LoginPage.h", CreateWalletPage.h, DashBoardPage.h**: Includes specific components related to the Login Page, Wallet creation, and Dashboard functionality.
- **#pragma hdrstop** and **#pragma package(smart_init)**: Compiler directives for efficient compilation.
- **#pragma resource "*.dfm"**: Links the .dfm (form description file) to the code.

Global Variable

```
AnsiString currentLoggedInEmail;
```

Figure 2.8

AnsiString currentLoggedInEmail; Stores the email of the currently logged-in user for use in other parts of the application.

Constructor

```
__fastcall TLoginForm::TLoginForm(TComponent* Owner)
: TForm(Owner)
{
}
```

Figure 2.9

This constructor initializes the login form when the application starts. The TForm constructor is invoked with the owner component.

Login Button Event

a. Input Validation

```
void __fastcall TLoginForm::btnLoginClick(TObject *Sender)
{
    String email = edtEmail->Text.Trim();
    String password = edtPassword->Text;

    if (email.IsEmpty() || password.IsEmpty())
    {
        ShowMessage("Email and Password cannot be empty!");
        return;
    }
}
```

Figure 2.10(a)

- Reads and trims the user-entered email and password.
- Displays a warning message if any field is empty and exits the function.

b. Open Accounts File

```
// Check if email exists in the accounts file
std::ifstream accountsFile("D:\\Crypto Wallet\\GUI Proj\\Accounts\\accounts.txt");
if (!accountsFile.is_open())
{
    ShowMessage("Accounts file could not be opened. Please check the file path.");
    return; ↵
}
```

Figure 2.10(b)

- Opens a file (accounts.txt) where account information is stored.
- If the file cannot be opened, a message is shown, and the function exits.

c. Process File Data

```
std::string storedEmail, storedPassword, storedWalletFile;
bool found = false;

while (std::getline(accountsFile, storedEmail, ' ') &&
        std::getline(accountsFile, storedPassword, ' ') &&
        std::getline(accountsFile, storedWalletFile))
```

Figure 2.10 (c)

- Reads each line of the file. Each account is stored in the format:
- Fields are delimited by a space (' ').

d. Validate Credentials

```
if (storedEmail == AnsiString(email).c_str())
{
    found = true;
    // If email matches, check if the hashed password matches
    String hashedPassword = hashPassword(password);
    if (storedPassword == AnsiString(hashedPassword).c_str())
    {
        currentLoggedInEmail = edtEmail->Text;
        ShowMessage("Login successful!");

        // Proceed to the next form or action
        Close(); // Close the login form
        DashBoardForm = new TDashBoardForm(Application); // Create the dashboard form
        DashBoardForm->Show();
    }
    else
    {
        ShowMessage("Invalid password.");
        return; ↵
    }
}
```

Figure 2.10 (d)

- Compares the input email to the storedEmail.If matched, hashes the input password using hashPassword.
- Compares the hashed password to the storedPassword.
 - On success, sets currentLoggedInEmail and shows a success message otherwise, displays "Invalid password" and exits.

e. Email Not Found

```
if (!found)
{
    ShowMessage("Email not registered.");
}
```

Figure 2.10 (e)

If the loop completes without finding a matching email, displays "Email not registered."

f. Open Dashboard on Success

```
// Proceed to the next form or action
Close(); // Close the login form
DashBoardForm = new TDashBoardForm(Application); // Create the dashboard form
DashBoardForm->Show();
```

Figure 2.10 (f)

Cancel Button Event

```
// optionally clear the fields before closing (if required)
edtEmail->Clear();
edtPassword->Clear();

// Close the login form
Close();
```

Figure 2.11

Output:

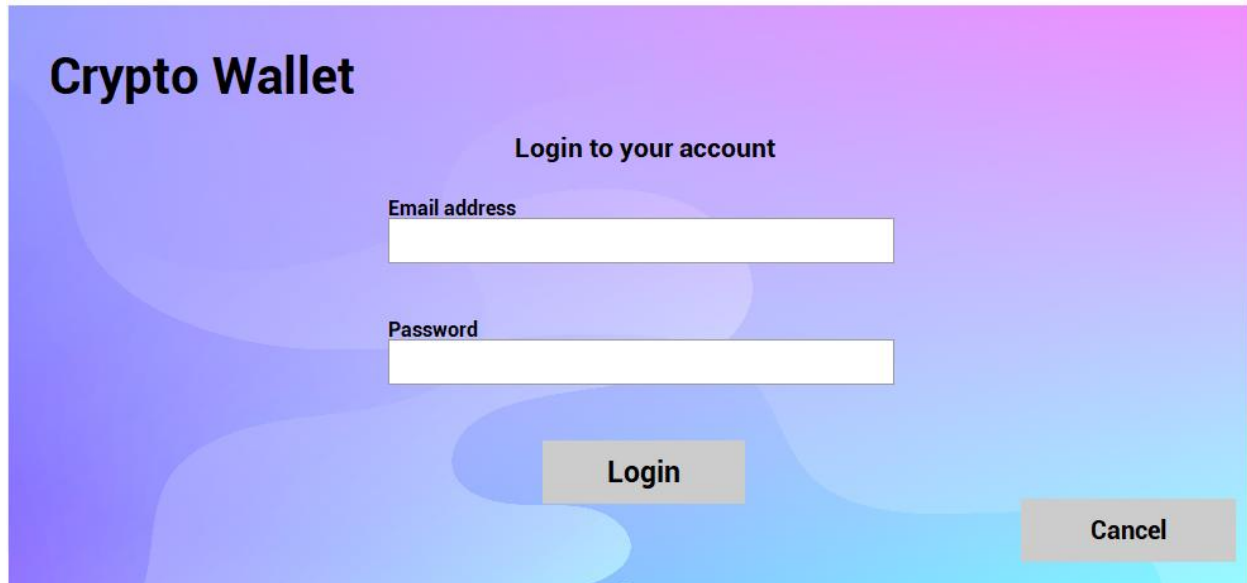


Figure 2.12 (Output of login page)

3. Create Wallet Page

Header Files

```
#include <vcl.h>
#include <fstream>
#include <direct.h> // For _mkdir function (to create directory)
#include <sys/stat.h> // For stat() to check directory existence
#include <regex> // For email validation
#pragma hdrstop
#pragma package(smart_init)
#pragma resource "*.dfm"
#include "CreateWalletPage.h"
```

Figure 2.13

Explanation

VCL Headers

- **#include <vcl.h>**: Provides GUI components and forms.
 - **#pragma hdrstop and #pragma package(smart_init)**: Compiler directives for efficiency.
 - **#pragma resource "*.dfm"**: Links the .dfm (form description file) to the code.

Standard Library Headers

- **#include <fstream>**: For file input/output operations.

- **#include <direct.h>**: For creating directories.
- **#include <sys/stat.h>**: To check if a directory exists.
- **#include <regex>**: For email validation.

Helper Functions

```

• // Convert UnicodeString (or String) to std::string using AnsiString
• std::string StringToStdString(const String &str)...
•
• // Function to hash the password using a simple hashing algorithm
• String hashPassword(const String &password)
• {
•     unsigned long hash = 5381;
•     for (int i = 1; i <= password.Length(); ++i)
•     {
•         hash = ((hash << 5) + hash) + password[i];
•     }
•     return IntToStr(static_cast<int>(hash));
• }
•
• // Check if password is in the breached list
• bool isPasswordBreached(const String &password)
• {
•     std::ifstream breachFile("D:\\Crypto Wallet\\GUI Proj\\Accounts\\pwd.txt");
•     if (!breachFile.is_open())
•     {
•         ShowMessage("Password breach file could not be opened. Please check the file path.");
•         return false; // Assume not breached if the file cannot be read
•     }
•     std::string inputPassword = StringToStdString(password);
•     std::string breachedPassword;
•     while (std::getline(breachFile, breachedPassword))
•     {
•         breachedPassword.erase(breachedPassword.find_last_not_of(" \n\r\t") + 1);
•         if (breachedPassword == inputPassword)
•         {
•             breachFile.close();
•             return true; // Password is in the breached list
•         }
•     }
•     breachFile.close();
•     return false; // Password is not found in the breached list
• }

```

Figure 2.13

Explanation

- Converts a String (used by VCL) to a standard C++ string (std::string) using an intermediate AnsiString.
- Implements a simple hashing algorithm (DJB2) to generate a hash from the password.
- Iterates through each character in the password.
- Updates a hash value using bitwise operations.
- Returns the hash as a String.

- Checks if the entered password exists in a file of breached passwords (pwd.txt).
- Reads the breached password file line by line.
- Compares each line with the entered password.
- Returns true if a match is found, indicating the password is compromised.

Function to check directory and email

```
// Function to check if a directory exists
bool directoryExists(const char* directory)
{
    struct stat info;
    return (stat(directory, &info) == 0 && (info.st_mode & S_IFDIR));
}

// Check if the email is already registered
bool isEmailRegistered(const String &email)
{
    std::ifstream accountsFile("D:\\Crypto Wallet\\GUI Proj\\Accounts\\accounts.txt");
    if (!accountsFile.is_open())
    {
        ShowMessage("Accounts file could not be opened. Please check the file path.");
        return false; // Assume not registered if file cannot be opened
    }
    std::string storedEmail, storedPassword, storedWalletFile;
    while (std::getline(accountsFile, storedEmail, ' ') &&
           std::getline(accountsFile, storedPassword, ' ') &&
           std::getline(accountsFile, storedWalletFile))
    {
        if (storedEmail == AnsiString(email).c_str())
        {
            return true; // Email found
        }
    }
    return false; // Email not found
}
```

Figure 2.13

Explanation

Function checks if a directory exists using stat, then returns true if the directory exists and is valid.

isEmailRegistered checks if the provided email already exists in the accounts.txt file, reads the file line by line and compares the input email with stored email entries. It returns true if a match is found.

Checks wallet file for email


```

// Create the wallet file for the user
void createWalletFile(const String &email)
{
    String walletFile = email + "_wallet.txt";
    std::ofstream wallet(AnsiString("D:\\Crypto Wallet\\GUI Proj\\Accounts\\" + walletFile).c_str());
    if (wallet.is_open())
    {
        wallet << "BTC 0.00\n"
                << "ETH 0.00\n"
                << "SOL 0.00\n"
                << "TRX 0.00\n"
                << "USDT 0.00\n";
        wallet.close();
    }
    else
    {
        ShowMessage("Failed to create wallet file. Please check file permissions.");
    }
}

// Validate email format using a regular expression
bool isValidEmail(const String &email)
{
    std::string emailStr = StringToStdString(email);
    const std::regex emailPattern(R"([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})");
    return std::regex_match(emailStr, emailPattern);
}

```

Figure 2.14

Explanation

Creates a wallet file for the user (<email>_wallet.txt). Writes default wallet balances (e.g., BTC, ETH) to the file. If the file cannot be created, displays an error message.

Validates the email format using a regular expression:

- Matches common email patterns (e.g., example@domain.com) and returns true if the format is valid.

Button Event Handlers

```

// On Create Wallet Button Click
void __fastcall TCreateWalletForm::btnCreateClick(TObject *Sender)
{
    String email = edtEmail->Text.Trim();
    String password = edtPassword->Text;
    // Input validation
    if (email.IsEmpty() || password.IsEmpty())
    {
        ShowMessage("Email and Password cannot be empty!");
        return;
    }
    // Validate email format
    if (!isValidEmail(email))
    {
        ShowMessage("Please enter a valid email address (e.g., example@domain.com).");
        return;
    }

    // Check if the email is already registered
    if (isEmailRegistered(email))
    {
        ShowMessage("Email is already registered. Please use a different email.");
        return;
    }
}

```

Figure 2.15

Explanation

- These functions ensures that both email and password fields are filled. Displays a warning if either is empty. Ensures the email format is valid. Displays a warning if invalid.
- Prevents duplicate registrations by checking if the email already exists.

Checking for breached password

```
// Check if the password is breached
if (isPasswordBreached(password))
{
    ShowMessage("Password found in a breached! Please use a different password.");
    return; ↵
}
// Hash the password
String hashedPassword = hashPassword(password);
// Ensure the directory exists or create it
const char* directory = "D:\\Crypto Wallet\\GUI Proj\\Accounts";
if (!directoryExists(directory))
{
    if (_mkdir(directory) != 0)
    {
        ShowMessage("Failed to create directory. Check if you have write permissions.");
        return; ↵
    }
}
```

Figure 2.16

Explanation

- Verifies that the password is not in the list of compromised passwords.
- Hash the password and checks if the target directory (Accounts) exists. If not, attempts to create it using _mkdir

Saving user details

```

// Save user details to accounts.txt
std::ofstream accountsFile("D:\\Crypto Wallet\\GUI Proj\\Accounts\\accounts.txt", std::ios::app);
if (accountsFile.is_open())
{
    String walletFile = email + "_wallet.txt";
    accountsFile << AnsiString(email).c_str() << " "
                << AnsiString(hashPassword).c_str() << " "
                << AnsiString(walletFile).c_str() << std::endl;
    accountsFile.close();
}
else
{
    ShowMessage("Failed to write to accounts file.");
    return;
}
// Create the wallet file
createWalletFile(email);
// Display success message
ShowMessage("Account created successfully! Wallet file created and details saved.");
Close();

```

Figure 2.17

Explanation

- Appends the user's email, hashed password, and wallet file name to the accounts.txt file.
- It creates the wallet file then displays the success message and at last closes the form.

Closing form

```

// On Cancel Button Click
void __fastcall TCreateWalletForm::btnCancelClick(TObject *Sender)
{
    Close();
}

```

Figure 2.18

Explanation

This simply closes the form without taking any action.

Output:

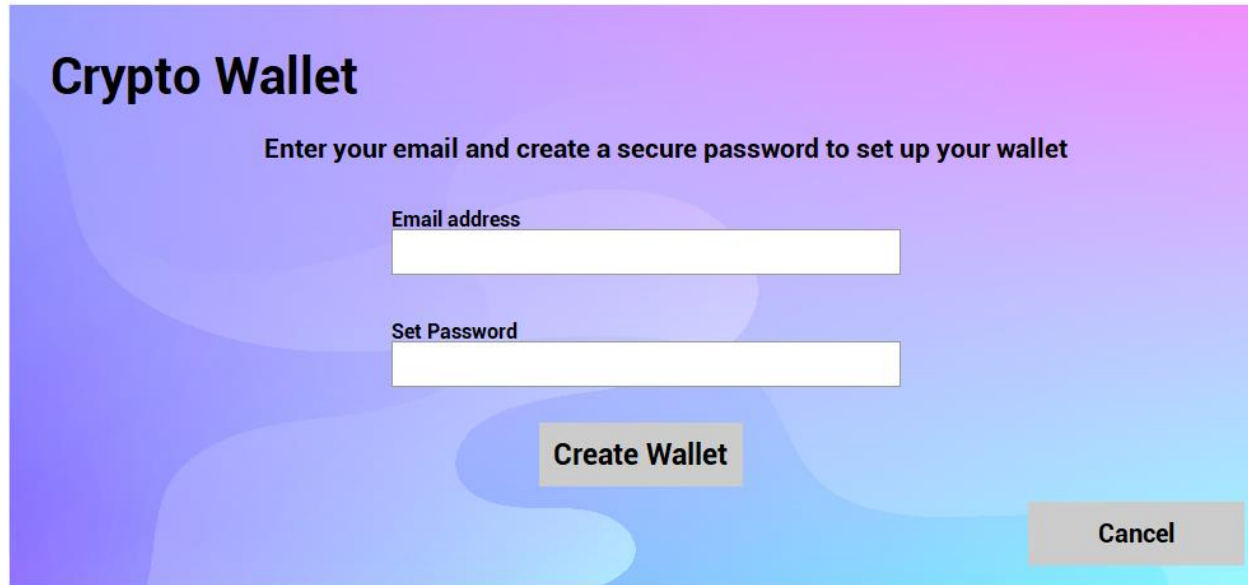


Figure 2.19 (Output of create wallet page)

4. Dashboard page

Header and Form Initialization

```
#include <vcl.h>
#pragma resource "*.dfm"
#pragma hdrstop
#include <iomanip>
#include <System.hpp> // For TFileStream, TMemoryStream
#include <unordered_map>
#include "DashBoardPage.h"
#include "LoginPage.h"
#pragma package(smart_init)
```

Figure 2.20

Explanation

#include <vcl.h>: Includes the Visual Component Library (VCL) header, which provides classes and functions for building GUI applications in C++ Builder.

#pragma resource "*.dfm": Links the .dfm file (the design file for the GUI form) to the source code. The .dfm file contains the layout and properties of the GUI components defined in the IDE.

#pragma hdrstop: Optimizes the compilation process by signaling the end of precompiled header files.

#include <iomanip>: Provides tools for formatting output, such as controlling decimal precision, alignment, and other text-based formatting.

#include <System.hpp>: A Borland-specific header file that includes various system utilities and classes. It is likely included for working with streams such as TFileStream and TMemoryStream.

#include "DashBoardPage.h": This file contains the declarations of the dashboard's GUI components and event handlers.

#include "LoginPage.h": Includes the header file for the login page (TLoginForm).

#pragma package(smart_init): It manages smart linking for the package and ensures that unused code is not linked, reducing the binary size.

Wallet file management

```
std::string GetWalletFilePath(const std::string& email)
{
    std::string baseDir = "D:\\Crypto Wallet\\GUI Proj\\Accounts\\";
    return baseDir + email + "_wallet.txt";
}
```

Figure 2.21

Explanation

- Constructs the file path for a wallet file based on the user's email then stores wallet data in text files located in a predefined directory.

ReadWallet:

- Reads wallet data from a file into an unordered map.Each line of the file contains a cryptocurrency name and its balance.

WriteWallet:

- Writes the updated wallet data to a file and formats the cryptocurrency balance to two decimal places.

Global Variables and Constructor

```
TDashBoardForm *DashBoardForm;
__fastcall TDashBoardForm::TDashBoardForm(TComponent* Owner)
    : TForm(Owner)
{
}
std::string GetWalletFilePath(const std::string& email)
{
    std::string baseDir = "D:\\Crypto Wallet\\GUI Proj\\Accounts\\";
    return baseDir + email + "_wallet.txt";
}
```

Figure 2.22

Explanation

- Declares a global pointer to an instance of TDashBoardForm.
- Constructor for the TDashBoardForm class, inheriting from TForm. It initializes the form with its owner component.

Helper Function to Get Wallet File Path

- Constructs the file path for a user's wallet based on their email.

Read wallet data

```
std::unordered_map<std::string, double> ReadWallet(const std::string& walletFile)
{
    std::unordered_map<std::string, double> wallet;
    TFileStream* walletFileIn = nullptr;
    try
    {
        walletFileIn = new TFileStream(walletFile.c_str(), fmOpenRead);
        if (walletFileIn->Size > 0)
        {
            AnsiString fileContent;
            fileContent.SetLength(walletFileIn->Size);
            walletFileIn->Read(fileContent.c_str(), fileContent.Length());
            TStringList* lines = new TStringList();
            lines->Text = fileContent;
            for (int i = 0; i < lines->Count; ++i)
            {
                AnsiString line = lines->Strings[i].Trim();
                if (line.IsEmpty()) continue;
                int spacePos = line.Pos(" ");
                if (spacePos > 0)
                {
                    std::string coin = line.SubString(1, spacePos - 1).Trim().UpperCase().c_str();
                    double balance = StrToFloatDef(line.SubString(spacePos + 1, line.Length() - spacePos).Trim().c_str(), 0);
                    if (balance > 0)
                    {
                        wallet[coin] = balance;
                    }
                }
            }
            delete lines;
        }
    }
    catch (...)
    {
        if (walletFileIn) delete walletFileIn;
        ShowMessage("Error reading wallet file.");
    }
    if (walletFileIn) delete walletFileIn;
    return wallet;
}
```

Figure 2.23

Explanation

- Reads wallet data from a text file into an unordered map:
 - Opens the file for reading. Reads each line to extract the coin name and balance. Skips empty lines.
 - Handles exceptions gracefully with error messages.

Write Wallet Data

```

void WriteWallet(const std::string& walletFile, const std::unordered_map<std::string, double>& wallet)
{
    TFileStream* walletFileOut = nullptr;
    try
    {
        walletFileOut = new TFileStream(walletFile.c_str(), fmCreate);
        AnsiString updatedContent;
        for (const auto& entry : wallet)
        {
            updatedContent += AnsiString(entry.first.c_str()) + " " +
                               FloatToStrF(entry.second, ffFixed, 15, 2) + "\r\n";
        }
        walletFileOut->Write(updatedContent.c_str(), updatedContent.Length());
    }
    catch (...)
    {
        if (walletFileOut) delete walletFileOut;
        ShowMessage("Error writing to wallet file.");
    }
    if (walletFileOut) delete walletFileOut;
}

```

Figure 2.24

Explanation

- Writes the wallet data (coin and balance) into a file.
- Replaces the existing file with new data.
- Formats balances as fixed-point numbers with two decimal places.

Form Initialization

```

void __fastcall TDashBoardForm::FormCreate(TObject *Sender)
{
    cmbCoinType->Items->Clear(); // Optional, to clear any previous items
    cmbCoinType->Items->Add("BTC");
    cmbCoinType->Items->Add("ETH");
    cmbCoinType->Items->Add("SOL");
    cmbCoinType->Items->Add("USDT");
}

```

Figure 2.25

Explanation

- Initializes the form by populating a dropdown (cmbCoinType) with supported cryptocurrency types.

Dropdown Change Event and withdraw funds

```

void __fastcall TDashBoardForm::cmbCoinTypeChange(TObject *Sender)
{
    String selectedCoin = cmbCoinType->Text;
}

void __fastcall TDashBoardForm::btnWithdrawClick(TObject *Sender)
{
    String selectedCoin = cmbCoinType->Text.Trim();
    double withdrawalAmount = StrToFloatDef(edtAmount->Text, 0);
    if (withdrawalAmount <= 0)
    {
        ShowMessage("Please enter a valid amount to withdraw.");
        return;
    }
    std::string walletFile = GetWalletFilePath(currentLoggedInEmail.c_str());
    auto wallet = ReadWallet(walletFile);
    std::string selectedCoinStr = AnsiString(selectedCoin).Trim().UpperCase().c_str();
    if (wallet.find(selectedCoinStr) == wallet.end())
    {
        ShowMessage("This coin is not found in your wallet.");
        return;
    }
    double currentBalance = wallet[selectedCoinStr];
    if (currentBalance < withdrawalAmount)
    {
        ShowMessage("Insufficient balance for withdrawal!");
        return;
    }
    wallet[selectedCoinStr] -= withdrawalAmount;
    WriteWallet(walletFile, wallet);
    ShowMessage("Withdrawal successful!");
}

```

Figure 2.26

Explanation

- Captures the selected cryptocurrency from the dropdown.
- Checks if the withdrawal amount is valid.
- Reads wallet data and validates if the selected coin exists and has sufficient balance.
- Updates the balance and saves the changes.

Function to check balance


```

void __fastcall TDashBoardForm::btnDepositClick(TObject *Sender)
{
    String selectedCoin = cmbCoinType->Text.Trim();
    double depositAmount = StrToFloatDef(edtAmount->Text, 0);
    if (depositAmount <= 0)
    {
        ShowMessage("Please enter a valid amount to deposit.");
        return;
    }
    std::string walletFile = GetWalletFilePath(currentLoggedInEmail.c_str());
    auto wallet = ReadWallet(walletFile);
    std::string selectedCoinStr = AnsiString(selectedCoin).Trim().UpperCase().c_str();
    if (wallet.find(selectedCoinStr) != wallet.end())
    {
        wallet[selectedCoinStr] += depositAmount;
    }
    else
    {
        wallet[selectedCoinStr] = depositAmount;
    }
    WriteWallet(walletFile, wallet);
    ShowMessage("Deposit successful!");
}

void __fastcall TDashBoardForm::btnCheckBalanceClick(TObject *Sender)
{
    String selectedCoin = cmbCoinType->Text.Trim();
    if (selectedCoin.IsEmpty())
    {
        ShowMessage("Please select a coin to check its balance.");
        return;
    }
    std::string walletFile = GetWalletFilePath(currentLoggedInEmail.c_str());
    auto wallet = ReadWallet(walletFile);
    std::string selectedCoinStr = AnsiString(selectedCoin).Trim().UpperCase().c_str();
    if (wallet.find(selectedCoinStr) != wallet.end())
    {
        double balance = wallet[selectedCoinStr];
        ShowMessage("Your balance for " + selectedCoin + " is: " +
            FloatToStrF(balance, ffFixed, 15, 2));
    }
    else
    {
        ShowMessage("This coin is not found in your wallet.");
    }
}

```

Figure 2.27

Explanation

- Validates the deposit amount.
- Adds the amount to the wallet or creates a new entry if the coin doesn't exist.
- Displays the balance of the selected cryptocurrency.

```

void __fastcall TDashBoardForm::btnLogoutClick(TObject *Sender)
{
    int response = MessageDlg("Are you sure you want to log out?", mtConfirmation, TMsgDlgButtons() << mbYes << mbNo, 0);
    if (response == mrYes)
    {
        currentLoggedInEmail = "";
        this->Hide();
        LoginForm->Show();
    }
}

```

Figure 2.28

Explanation

- Prompts for confirmation.
- Clears the current user and shows the login form.

Output:

Figure 2.29 (Output of dashboard page)

3. Conclusion

This project successfully created a basic cryptocurrency wallet application with secure user authentication and wallet management features. Although the application has some limitations, it provides a solid foundation for further development, including improved security and additional features.

Summary of the Code

Cryptocurrency Wallet Application written in C++ provides functionalities for creating a wallet, logging in, and managing cryptocurrency balances.

Key Features:

1. Password Hashing and Breach Check:

- Passwords are hashed using a custom hash function before being stored for security purposes. The program checks the entered password against a file of breached passwords (pwd.txt) and denies wallet creation if found.

2. Wallet Creation:

- Users can create a wallet by providing an email and password.
- User details (email, hashed password, wallet file name) are saved to an accounts.txt file.
- A wallet file is created for the user, where cryptocurrency balances are stored.

3. User Login:

- Users log in by providing their email and password.
- Credentials are verified using data stored in accounts.txt.
- Upon successful login, users can manage their wallet.

4. Wallet Management:

- Deposit Cryptocurrency:
 - Users can deposit specific cryptocurrencies (BTC, ETH, SOL, TRX, USDT) into their wallet.
 - Balances are stored in the user's wallet file and updated accordingly.
- Withdraw Cryptocurrency:
 - Users can withdraw cryptocurrencies if they have sufficient balance.
 - Insufficient balance results in an error message.
- Check Balance:
 - Users can view their current cryptocurrency balances.

5. Data Storage:

- User account information is stored in accounts.txt.
- Each user has a dedicated wallet file (<email>_wallet.txt) to store cryptocurrency balances.

6. User Interaction:

- Menu-driven interface with options for creating a wallet, logging in, and managing cryptocurrency.
- Options for deposit, withdrawal, and checking balance are presented after login.
- Users can log out and return to the main menu.

7. Error Handling:

- Invalid login credentials result in an error message.
- Insufficient balance during withdrawal is handled gracefully.
- The program ensures that cryptocurrencies are recorded in uppercase for consistency.

Workflow:

1. Main Menu Options:

- Create Wallet
- Login
- Exit

2. Post-Login Menu Options:

- Deposit Cryptocurrency
- Withdraw Cryptocurrency
- Check Balance
- Logout

3. File Handling:

- accounts.txt: Stores user credentials and wallet file names.
- <email>_wallet.txt: Stores cryptocurrency balances for individual users.

4. Exit Handling:

- Users can exit the application at any time via the main menu.

Security Measures:

- Hashing of passwords to avoid storing plain text.
- Password breach check against a known list (pwd.txt).

Potential Enhancements:

- Implement stronger password hashing.
- Encrypt user data for better security support for more cryptocurrencies or real-time pricing.
- Improve UI/UX by integrating with a GUI or web interface.

4. References

- C++ Standard Library Documentation
- Hashing Algorithms
- File Handling in C++
- [https://youtube.com/playlist?list=PL43pGnjiVwgQakzRxpt2amqN9f7tRtc &feature=shared](https://youtube.com/playlist?list=PL43pGnjiVwgQakzRxpt2amqN9f7tRtc&feature=shared)

5. Appendices

1. Appendix A: File Formats and Structures
 - accounts.txt.
 - <email> <hashed_password> <wallet_file>
 - <email>_wallet.txt.
2. Appendix B: List of Supported Cryptocurrencies
 - BTC (Bitcoin)
 - ETH (Ethereum)
 - SOL (Solana)
 - TRX (Tron)
 - USDT (Tether)
3. Appendix C: Hashing Algorithm Details
 - Hash Function
4. Appendix D: Menu Flow Diagram
 - Main Menu
 - Post-Login Menu
5. Appendix E: Error Handling and Messages

- Invalid Login Credentials
- Password Found in Breach
- Insufficient Balance for Withdrawal
- Invalid Menu Choice

6. Appendix F: Dependencies and Requirements

- Development Environment
- Input Files
- Output Files

7. Appendix G: Potential Enhancements

- Implement a graphical user interface (GUI).
- Add real-time cryptocurrency prices using APIs.
- Introduce two-factor authentication (2FA) for login.
- Use a database instead of text files for scalability.
- Improve the hashing algorithm for enhanced security.