CM2005 Object Oriented Programming - Midterm Report

Introduction

The goal of this assignment is to develop a comprehensive technical analysis toolkit for a cryptocurrency exchange platform. The application is built using C++ and follows Object-Oriented Programming (OOP) principles to ensure modularity and extensibility. Key features include parsing historical market data, visualizing market trends via candlestick charts, managing secure user accounts with persistent wallets, and simulating real-time trading activities.

TASK 1: Loading Trading Data and Computing Candlestick Data

(i) Explanation

The application reads trading data from a CSV file (20200601.csv) and processes it into an OrderBook. Therefore, to compute the candlestick from the orders, I implemented a function in OrderBook which will return a vector of candlestick entries based on the given composition. So, in the console user can go to the candlestick page and the system will print out those candlesticks. At the beginning, the default candlestick calculated ETH/BTC with five-second intervals from the earliest timestamp until the current timestamp. Users can change the composition of the candlestick and the start timestamp on the console. And to visualize the up and down, I used the ASCII color (Green & Red) when printing the information on the console.

(ii) Implementation Logic

The logic is split to the data processing part in OrderBook and the visualization part in CandleStick.

Data Aggregation (OrderBook::generateCnadleSticks):

1. The function iterates through the orders.
2. It filters orders by the parameters passed when calling (product & type).
3. It separates orders by the time block.

4. For each block, it computes the High and Low using helper functions (getHighPrice, getLowPrice) in OrderBookEntry and identifies the Open (first order) and Close (last order).

Candlestick Visualization (CandleStick::printCandleStick):

1. The function iterates through the passed candlestick entries vector.
2. It compares close and open prices.
3. Print the information green if the close value is larger or equal to the open value. Otherwise print red.

Pseudocode (Visualization Logic):

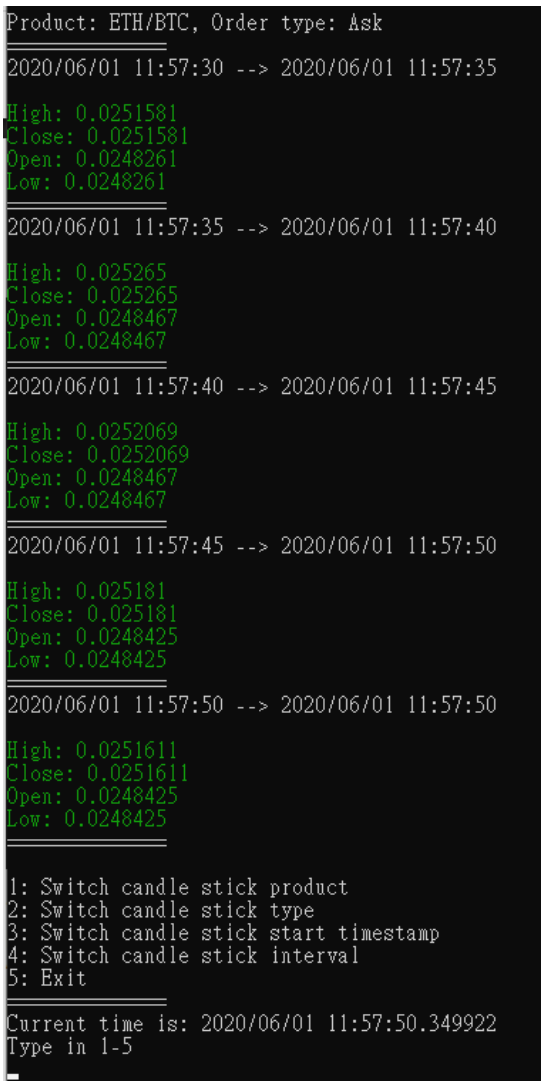
```
FUNCTION printCandleStick(candleSticks):
    FOR EACH candle IN candleSticks:
        PRINT candle.timestamp
        IF candle.close >= candle.open:
            SET_TEXT_COLOR(GREEN)
            PRINT "High: " + candle.high
            PRINT "Close: " + candle.close
            PRINT "Open: " + candle.open
            PRINT "Low: " + candle.low
        ELSE:
            SET_TEXT_COLOR(RED)
            PRINT "High: " + candle.high
            PRINT "Open: " + candle.open
            PRINT "Close: " + candle.close
            PRINT "Low: " + candle.low
        RESET_TEXT_COLOR()
    END FOR
END FUNCTION
```

(iii) Evidence

On the right-hand side is a screenshot from the console.

So, the first line will show the product and type. And each time block will show the start and end timestamp first then print the High, Close, Open, and Low with color.

(print candlestick code is at line 1825)



TASK 2: Login or Register a Trading User Profile

(i) Explanation

Because there are no familiar classes dealing with accounts, I created a new class to handle all the functionality of accounts. It supports login, creates accounts (stores password via hash and generates an UUID), and password recovery (simulate an OTP process). And store the data in several csv. There is one CSV file containing all UUID and each user's information. And for each user, it will generate a CSV file that is named by the uuid and will log user's activity inside the CSV.

(ii) Implementation Logic

The logic can split to Registration, Password, and PasswordReset

Registration (createAccount):

1. Get username and email, checking cache to prevent duplicates.
2. Generates a unique 10-digit UUID using a random number generator. And check whether there is a same UUID exist or not.
3. Hashes the password for security.
4. Initializes a new wallet for the user with free starting capital.

Authentication (login):

1. Loads existing accounts from CSV.
2. Hashes the input password and compares it with the stored hash.
3. If user tries too many times, ask user does he/she wants to reset the password.

Password Reset (resetPassword):

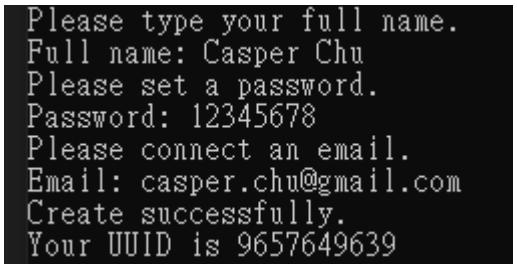
1. Simulates an email verification process by generating a fixed OTP ("1234").
2. Allow the user to set a new password upon correct OTP entry.

Pseudocode (Registration Flow):

```
FUNCTION createAccount():
    WHILE:
    INPUT username
        IF username not exists in cache:
        BREAK
        PRINT "Duplicate"
    END WHILE:
    INPUT password
    hashed_pw = HASH(password)
    uuid = generateUUID()
    WRITE uuid, username, hashed_pw, email TO "accounts.csv"
    INITIALIZE wallet FOR uuid WITH 10 BTC, 100 USDT
    RETURN true
END FUNCTION
```

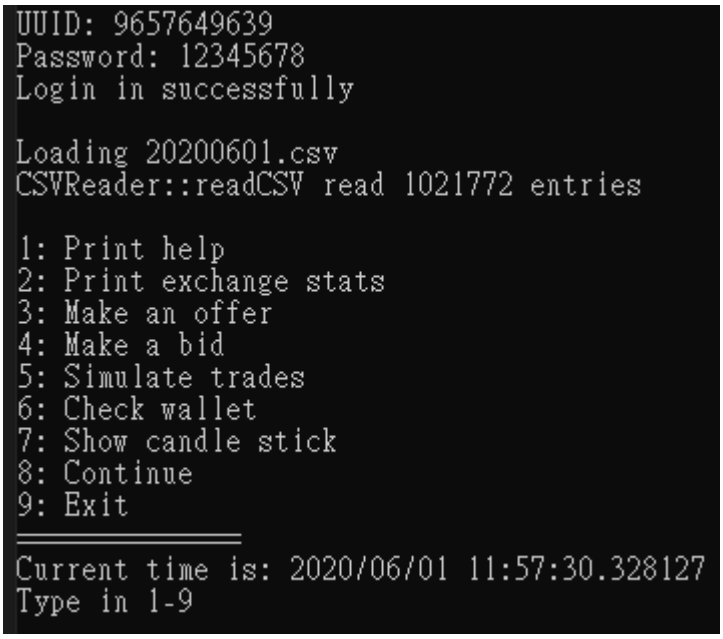
(iii) Evidence

Account registration:



(Account registration code is at line 2007)

Authentication:



(Authentication code is at line 1909)

TASK 3: User Wallet & Trading Transactions History

(i) Explanation

The original wallet class is executable; therefore, I add the log function inside the class, so once the function is called, it can directly know which wallet is logging. And I also wrote two functions which will generate statistics or show history

activities based on the log.

(ii) Implementation Logic

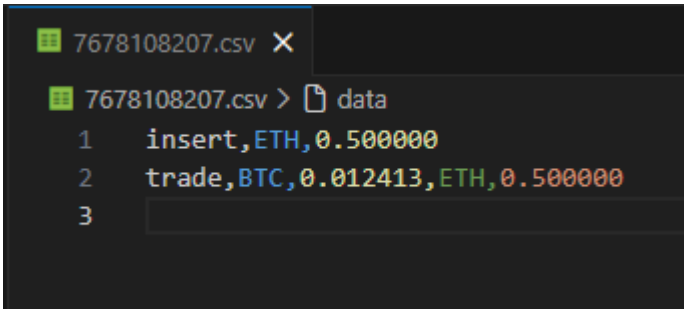
There are three parts of my work, log, display trading history, and statistic trading history. And under every operate, will generate a line of log then push it into the cache. If the operate is a transition, it will append a line with 3 column action, currency, and amount. If it is a trade, it will append a line with 5 column action, outgoing currency, outgoing amount, incoming currency, incoming currency.

Pseudocode:

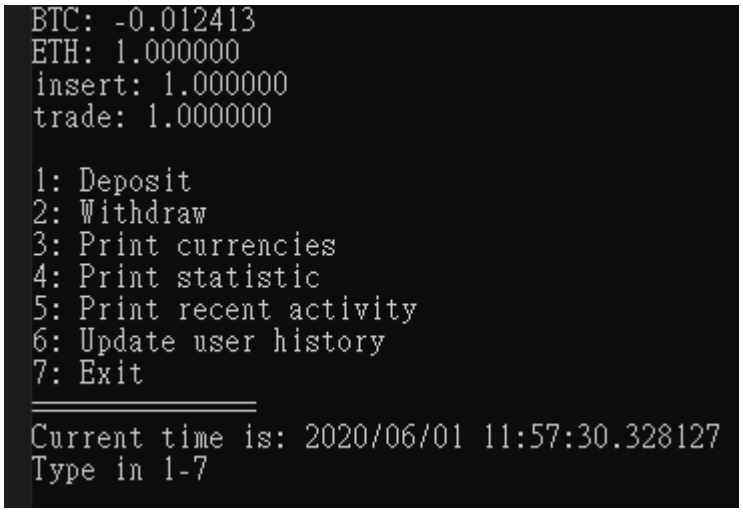
```
VECTOR cache
CREATE_TRADE()
cache.PUSH("trade,BTC,0.1,ETH,0.5")
INSERT_CURRENCY()
cache.PUSH("insert,ETH,0.5")

FUNCTION LogInCSV():
    FOR EACH line IN cache:
        APPEND line TO "UUID.csv"
    END FOR
END FUNCTION
```

(iii) Evidence

This is the screenshot of the history csv from an account which UUID is equal to 7678108207.



And this is the screenshot of display statistics.



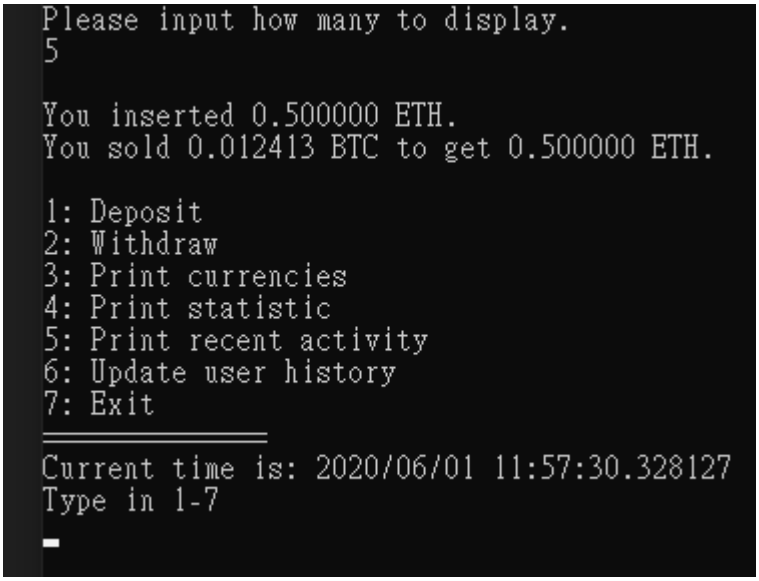
So based on the previous csv history, ETH amount is 0.5 + 0.5 = 1. Therefore, here will show “ETH: 1”.

(Statistic code is at line 1576)

And for the recent activities.

You can select how many lines. And if there aren't enough lines, it will show all the lines.

(Insert currency log code is at line 1388)



(History display code is at line1540)

TASK 4: Simulating User Trading Activities

(i) Explanation

In my application, it will fetch the data from the orders in OrderBook and calculate the average price to become the based price. And to make it look more real, I add a fluctuation generator to make sure Asks and Bids won't always be the same value.

(ii) Implementation Logic and Justification

Price Generation Strategy:

To ensure generated prices are realistic, the system fetches the average price of existing orders for each product from the OrderBook.

It applies to a random fluctuation (+/- 5%) simulates natural market and adds some uncertainty .

Performance Optimization:

Instead of using the insert order function, I split it into two small helper functions. Therefore, instead of appending and sorting immediately after finishing one simulation of ask of bid, it will append first and sort after whole simulation is complete.

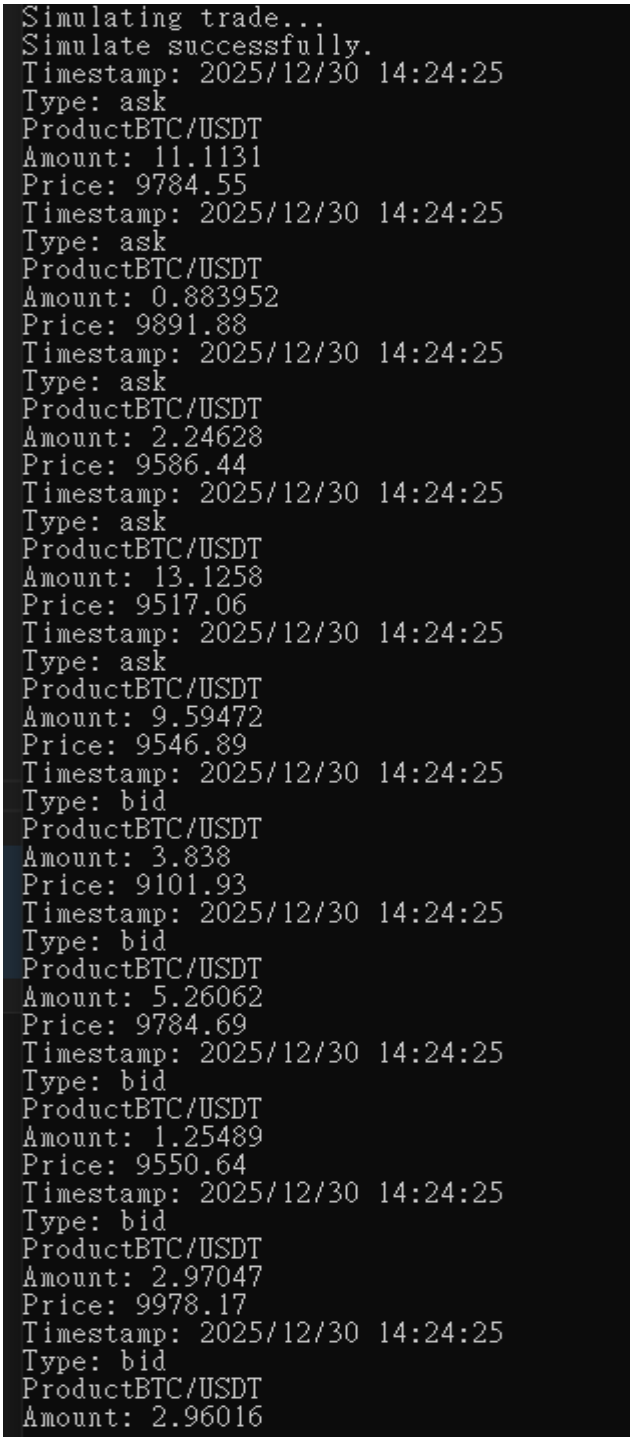
Pseudocode (Simulation):

```
FUNCTION simulateTrade():
    time = GET_SYSTEM_TIME()
    products = GET_KNOWN_PRODUCTS()

    FOR EACH product IN products:
        avg_price = GET_AVERAGE_PRICE(product)
        FOR i FROM 1 TO 5:
            price = avg_price * (1 + RANDOM(-0.05, 0.05))
            order = ORDER(price, amount, time, product)
            INSERT_TO_ORDERS(order)
        END FOR

        SORT_ORDERS()
    END FOR
END FUNCTION
```

(iii) Evidence

On the right-hand side is the screenshot after executed simulateTrade. However, this is a debug version; therefore, those orders will show. But in the normal mode user will only see “Simulating trade...” and “Simulate successfully”.

(Code is at line 264)



TASK 5: User Interaction and Input Validation

(i) Explanation

In my application, I sperate the menu to different pages. There are three pages, main menu, wallet menu, candlestick menu. The wallet menu and candlestick menu are inside menus of main menu. Therefore, users can input specific numbers to jump into those menus. And add input check for all the places that user need to enter something.

(ii) Implementation Logic

I categorize all functions into three different pages. The wallet page deals with deposits, withdrawals, currencies check, etc. And the candlestick handles all the variables which control the candlesticks.

To make expansion and adjusting easier, each option is stored as a pair (text & function pointer) within a vector. Therefore, all pages can represent as a vector. Hence, these three pages are stored in a menu vector and control the pages by an index variable.

Pseudocode (Menu structure):

```
1.  option1 = PAIR("Option 1", OPTION1FUNCTION)
2.  option2 = PAIR("Option 2", OPTION2FUNCTION)
3.  page = VECTOR(option1, option2)
4.  menus = VECTOR(page)
5.  index = 0
6.
7.  FUNCTION menu_loop():
8.      PRINT_MENU()
9.      GET_USER_INPUT()
10.     CLEAN_CONSOLE()
11.     HANDLE_USER_INPUT()
12. END FUNCTION
```

(iii) Evidence

Here are the three pages.

And the arrows mean user can jump from A to B.

(Struct is at line 79)

(Menu loop is at line 144)