```
// main.cpp
#include "Wallet.h"
#include "MerkelMain.h"
#include "AccountManager.h"
#include <iostream>

int main()
{
    MerkelMain app{AccountManager::login()};
    app.init();
}
// MerkelMain.h
#pragma once

#include "OrderBookEntry.h"
#include "OrderBook.h"
#include "Wallet.h"
#include <vector>

class MerkelMain
{
    public:
        MerkelMain(Wallet _wallet);
        /** Call this to start the sim */
        void init();
    private:
        void printMenu();
        void cleanConsole();
        // main menu functions
        void printHelp();
        void printMarketStats();
        void enterAsk();
        void enterBid();
        // Made by myself - start
        void simulateTrade();
        void jumpToWallet();
        void jumpToCandlestick();
        // Made by myself - end
        void gotoNextTimeframe();
        void exitApp();
        // Made by myself - start
        // wallet menu funcitons
        void dopsiteToWallet();
        void withdrawFromWallet();
        void printCurrencies();
        void printStatistic();
        void printRecentHistory();
        void updateUserCSV();
        void exitWalletPage();
        // drawing menu functions
        std::string candlestickProduct = "ETH/BTC";
        std::string candlestickStartTimestamp;
        OrderBookType candlestickType = OrderBookType::ask;
        unsigned int candlestickInterval = 5;
        void printCandlestick();
        void switchCandlestickProduct();
        void switchCandlestickType();
        void switchCandlestickStartTimestamp();
        void switchCandlestickInterval();
        void exitDrawingPage();
        // Made by myself - end
        // process input functions
        int getUserOption();
        void processUserOption(int userOption);
```

```
        std::string getCurrentSystemTimestamp();

        std::string currentTime;
        bool exitFlag = false;
        unsigned int simulateTimes = 5;
        bool debug = false;

        // menu variables
        // Made by myself - start
        unsigned int indexOfMenus = 0;
        // Made by myself - end
        using voidFunc = void (MerkelMain::*)();
        // Made by myself - start
        std::vector<std::vector<std::pair<std::string, voidFunc>>> menus = {
            // main menu
            {
                {"Print help", printHelp},
                {"Print exchange stats", printMarketStats},
                {"Make an offer", enterAsk},
                {"Make a bid", enterBid},
                {"Simulate trades", simulateTrade},
                {"Check wallet", jumpToWallet},
                {"Show candle stick", jumpToCandlestick},
                {"Continue", gotoNextTimeframe},
                {"Exit", exitApp}
            },
            // wallet menu
            {
                {"Deposit", dopsiteToWallet},
                {"Withdraw", withdrawFromWallet},
                {"Print currencies", printCurrencies},
                {"Print statistic", printStatistic},
                {"Print recent activity", printRecentHistory},
                {"Update user history", updateUserCSV},
                {"Exit", exitWalletPage}
            },
            // drawing menu
            {
                {"Switch candle stick product", switchCandlestickProduct},
                {"Switch candle stick type", switchCandlestickType},
                {"Switch candle stick start timestamp",
switchCandlestickStartTimestamp},
                {"Switch candle stick interval", switchCandlestickInterval},
                {"Exit", exitDrawingPage}
            }
        };
        // Made by myself - end

        // OrderBook orderBook{"20200317.csv"};
          OrderBook orderBook{"20200601.csv"};
        Wallet wallet;

};
// MerkelMain.cpp
#include "MerkelMain.h"
#include "OrderBookEntry.h"
#include "CSVReader.h"
#include "AccountManager.h"
#include "Candlestick.h"
#include <ctime>
#include <iostream>
#include <random>
#include <vector>
```

```
MerkelMain::MerkelMain(Wallet _wallet)
: wallet(_wallet)
{

}

void MerkelMain::init()
{
    int input;
    currentTime = orderBook.getEarliestTime();
    candlestickStartTimestamp = currentTime.substr(0, 19);
    getCurrentSystemTimestamp();

    while(true)
    {
        printMenu();
        input = getUserOption();

        cleanConsole();
        processUserOption(input);

        // Made by myself - start
        if (exitFlag) break;

        std::cout << "" << std::endl;
        // Made by myself - end
    }
}

void MerkelMain::printMenu()
{
    // Made by myself - start
    // print menu based on the menu vector
    unsigned int index = 1;
    for (std::pair<std::string, voidFunc>& pair: menus[indexOfMenus])
    {
        std::cout << std::to_string(index) << ": " << pair.first << std::endl;
        ++index;
    }
    // Made by myself - end

    std::cout << "============== " << std::endl;

    std::cout << "Current time is: " << currentTime << std::endl;
}

// Main menu
void MerkelMain::printHelp()
{
    std::cout << "Help - your aim is to make money. Analyse the market and make
bids and offers. " << std::endl;
}
void MerkelMain::printMarketStats()
{
    for (std::string const& p : orderBook.getKnownProducts())
    {
        std::cout << "Product: " << p << std::endl;
        std::vector<OrderBookEntry> entries =
orderBook.getOrders(OrderBookType::ask,
                                                                p, currentTime);
        std::cout << "Asks seen: " << entries.size() << std::endl;
        std::cout << "Max ask: " << OrderBook::getHighPrice(entries) <<
std::endl;
```

```
        std::cout << "Min ask: " << OrderBook::getLowPrice(entries) <<
std::endl;
    }
}
void MerkelMain::enterAsk()
{
    std::cout << "Make an ask - enter the amount: product,price, amount, eg
ETH/BTC,200,0.5" << std::endl;
    std::string input;
    std::getline(std::cin, input);

    std::vector<std::string> tokens = CSVReader::tokenise(input, ',');
    if (tokens.size() != 3)
    {
        std::cout << "MerkelMain::enterAsk Bad input! " << input << std::endl;
    }
    else {
        try {
            OrderBookEntry obe = CSVReader::stringsToOBE(
                tokens[1],
                tokens[2],
                currentTime,
                tokens[0],
                OrderBookType::ask
            );
            obe.username = "simuser";
            if (wallet.canFulfillOrder(obe))
            {
                std::cout << "Wallet looks good. " << std::endl;
                orderBook.insertOrder(obe);
            }
            else {
                std::cout << "Wallet has insufficient funds . " << std::endl;
            }
        }catch (const std::exception& e)
        {
            std::cout << " MerkelMain::enterAsk Bad input " << std::endl;
        }
    }
}
void MerkelMain::enterBid()
{
    std::cout << "Make an bid - enter the amount: product,price, amount, eg
ETH/BTC,200,0.5" << std::endl;
    std::string input;
    std::getline(std::cin, input);

    std::vector<std::string> tokens = CSVReader::tokenise(input, ',');
    if (tokens.size() != 3)
    {
        std::cout << "MerkelMain::enterBid Bad input! " << input << std::endl;
    }
    else {
        try {
            OrderBookEntry obe = CSVReader::stringsToOBE(
                tokens[1],
                tokens[2],
                currentTime,
                tokens[0],
                OrderBookType::bid
            );
            obe.username = "simuser";

            if (wallet.canFulfillOrder(obe))
```

```
            {
                std::cout << "Wallet looks good. " << std::endl;
                orderBook.insertOrder(obe);
            }
            else {
                std::cout << "Wallet has insufficient funds . " << std::endl;
            }
        }catch (const std::exception& e)
        {
            std::cout << " MerkelMain::enterBid Bad input " << std::endl;
        }
    }
}
// Made by myself - start
void MerkelMain::simulateTrade()
{
    std::cout << "Simulating trade..." << std::endl;
    std::string systemTimestamp = getCurrentSystemTimestamp();
    std::vector<std::string> products = orderBook.getKnownProducts();
    std::vector<OrderBookEntry> simulateOBES;

    // Add randomness
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> fluctuation(-0.05, 0.05);

    struct amountRange
    {
        double min = 0.001;
        double max = 0.001;
    };


    for (std::string& p: products)
    {
        std::vector<OrderBookEntry> askHistory =
orderBook.getOrders(OrderBookType::ask, p, currentTime);
        std::vector<OrderBookEntry> bidHistory =
orderBook.getOrders(OrderBookType::bid, p, currentTime);
        amountRange askRange;
        amountRange bidRange;

        // get an avarage price as the base price and the range of amount
        double askPrice = 10;
        if (!askHistory.empty())
        {
            double askSum = 0;
            askRange.max = askHistory[0].amount;
            askRange.min = askHistory[0].amount;

            for (OrderBookEntry& obe: askHistory)
            {
                if (obe.amount > askRange.max) askRange.max = obe.amount;
                if (obe.amount < askRange.min) askRange.min = obe.amount;

                askSum += obe.price;
            }
            askPrice = askSum / askHistory.size();
        }
        double bidPrice = 10;
        if (!bidHistory.empty())
        {
            double bidSum = 0;
            bidRange.max = bidHistory[0].amount;
```

```
            bidRange.min = bidHistory[0].amount;

            for (OrderBookEntry& obe: bidHistory)
            {
                if (obe.amount > bidRange.max) bidRange.max = obe.amount;
                if (obe.amount < bidRange.min) bidRange.min = obe.amount;

                bidSum += obe.price;
            }
            bidPrice = bidSum / bidHistory.size();
        }

        // simulate n number of ask
        for (unsigned int i = 0; i < simulateTimes; ++i)
        {
            double price = askPrice * (1 + fluctuation(gen));

            std::uniform_real_distribution<> randomAmount(askRange.min,
askRange.max);
            double amount = randomAmount(gen);

            OrderBookEntry obe {
                price,
                amount,
                systemTimestamp,
                p,
                OrderBookType::ask,
                "simuser"
            };

            simulateOBES.push_back(obe);
            orderBook.appendOrder(obe);
        }

        // simulate n number of bid
        for (unsigned int i = 0; i < simulateTimes; ++i)
        {
            double price = bidPrice * (1 + fluctuation(gen));

            std::uniform_real_distribution<> randomAmount(bidRange.min,
bidRange.max);
            double amount = randomAmount(gen);

            OrderBookEntry obe {
                price,
                amount,
                systemTimestamp,
                p,
                OrderBookType::bid,
                "simuser"
            };

            simulateOBES.push_back(obe);
            orderBook.appendOrder(obe);
        }
    }
    orderBook.sortOrder();
    std::cout << "Simulate successfully." << std::endl;
    if (debug)
    {
        for (OrderBookEntry& obe: simulateOBES)
        {
            std::cout << "Timestamp: " << obe.timestamp << std::endl;
            if (obe.orderType == OrderBookType::bid) std::cout << "Type: bid" <<
```

```
std::endl;
            if (obe.orderType == OrderBookType::ask) std::cout << "Type: ask" <<
std::endl;
            std::cout << "Product" << obe.product << std::endl;
            std::cout << "Amount: " << obe.amount << std::endl;
            std::cout << "Price: " << obe.price << std::endl;
        }
    }

}
void MerkelMain::jumpToWallet()
{
    std::cout << "Switch to wallet menu." << std::endl;
    indexOfMenus = 1;
}
void MerkelMain::jumpToCandlestick()
{
    std::cout << "Switch to candle stick page.\n" << std::endl;
    printCandlestick();
    indexOfMenus = 2;
}
// Made by myself - end
void MerkelMain::gotoNextTimeframe()
{
    std::cout << "Going to next time frame. " << std::endl;
    for (std::string p : orderBook.getKnownProducts())
    {
        std::cout << "matching " << p << std::endl;
        std::vector<OrderBookEntry> sales =  orderBook.matchAsksToBids(p,
currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;
        for (OrderBookEntry& sale : sales)
        {
            std::cout << "Sale price: " << sale.price << " amount " <<
sale.amount << std::endl;
            if (sale.username == "simuser")
            {
                // update the wallet
                wallet.processSale(sale);
            }
        }

    }

    currentTime = orderBook.getNextTime(currentTime);
}
void MerkelMain::exitApp()
{
    std::cout << "See you next time." << std::endl;
    wallet.logInCSV();
    wallet.updateUserWalletCSV();
    exitFlag = true;
}
// Made by myself - start
// Wallet menu
void MerkelMain::dopsiteToWallet()
{
    std::string currency;
    std::string amountString;
    double amount;
    std::cout << "Deposite currency" << std::endl;

    while (true)
    {
```

```
        std::cout << "Currency: " << std::flush;
        std::cin >> currency;
        std::cout << "Amount: " << std::flush;
        std::cin >> amountString;
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

        try
        {
            bool find = false;
            std::vector<std::string> existProducts =
orderBook.getKnownProducts();
            for (std::string p: existProducts)
            {
                std::vector<std::string> tokens = CSVReader::tokenise(p, '/');
                if (currency == tokens[0] || currency == tokens[1])
                {
                    find = true;
                    break;
                }
            }
            amount = std::stod(amountString);
            if (find) break;
        }
        catch(std::exception& e)
        {
            std::cerr << e.what() << '\n';
        }

        std::cout << "Bad input. Please try again." << std::endl;
    }

    wallet.insertCurrency(currency, amount);
    std::cout << "Deposite successfuly" << std::endl;
}
void MerkelMain::withdrawFromWallet()
{
    std::string currency;
    std::string amountString;
    double amount;
    std::cout << "Withdraw currency" << std::endl;
    while (true)
    {
        std::cout << "Currency: " << std::flush;
        std::cin >> currency;
        std::cout << "Amount: " << std::flush;
        std::cin >> amountString;
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

        try
        {
            bool find = false;
            std::vector<std::string> existProducts =
orderBook.getKnownProducts();
            for (std::string p: existProducts)
            {
                std::vector<std::string> tokens = CSVReader::tokenise(p, '/');
                if (currency == tokens[0] || currency == tokens[1])
                {
                    find = true;
                    break;
                }
            }
            amount = std::stod(amountString);
            if (find) break;
```

```
        }
        catch(std::exception& e)
        {
            std::cerr << e.what() << '\n';
        }

        std::cout << "Bad input. Please try again." << std::endl;
    }

    wallet.removeCurrency(currency, amount);
    std::cout << "Withdraw successfuly" << std::endl;
}
void MerkelMain::printCurrencies()
{
    std::cout << wallet.toString();
}
void MerkelMain::printStatistic()
{
    wallet.statisticsUserActivity();
}
void MerkelMain::printRecentHistory()
{
    std::string input;
    unsigned int num;
    std::cout << "Please input how many to display." << std::endl;
    while (true)
    {
        std::cin >> input;
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

        // input check
        try
        {
            int n = std::stoi(input);
            if (n > 0)
            {
                num = n;
                break;
            }
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << '\n';
        }
        std::cout << "Wrong input. Please input a number again." << std::endl;
    }

    std::cout << "" << std::endl;
    wallet.showTansitionOrTradingHistory(num);
}
void MerkelMain::updateUserCSV()
{
    wallet.updateUserWalletCSV();
    wallet.logInCSV();
    std::cout << "Successfully update." << std::endl;
}
void MerkelMain::exitWalletPage()
{
    std::cout << "Back to main menu." << std::endl;
    indexOfMenus = 0;
}

// Candle stick menu
void MerkelMain::printCandlestick()
```

```
{
    std::vector<candlestickEntry> candlesticks = orderBook.generateCandlesticks(
        candlestickStartTimestamp,
        currentTime.substr(0, 19),
        candlestickInterval,
        candlestickProduct,
        candlestickType
    );

    std::string typeString;
    switch (candlestickType)
    {
        case OrderBookType::ask:
            typeString = "Ask";
            break;
        case OrderBookType::bid:
            typeString = "Bid";
            break;
        default:
            typeString = "Unknown";
            break;
    }
    std::cout << "Product: " << candlestickProduct + ", Order type: " <<
typeString << std::endl;
    std::cout << "==============" << std::endl;

    // generate a candlestick vector by orderbook
    Candlestick::printCandlestick(candlesticks);
}
void MerkelMain::switchCandlestickProduct()
{
    std::string product;
    std::cout << "Enter product. (ETH/BTC)" << std::endl;
    while (true)
    {
        std::cout << "Product: ";
        std::cin >> product;

        std::vector<std::string> existProducts = orderBook.getKnownProducts();
        for (std::string p: existProducts)
        {
            if (p == product)
            {
                candlestickProduct = product;
                std::cout << "Change successfully. Current product is: " <<
product << "\n" << std::endl;
                printCandlestick();
                std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
                return;
            }
        }

        std::cout << "Bad input. Please try again." << std::endl;
    }
}
void MerkelMain::switchCandlestickType()
{
    std::string type;
    OrderBookType obType;
    std::cout << "Enter a order type to set type of the candle stick. (ask or
bid)" << std::endl;
    while (true)
    {
```

```
        std::cin >> type;


        if (type == "ask")
        {
            obType = OrderBookType::ask;
            break;
        }
        if (type == "bid")
        {
            obType = OrderBookType::bid;
            break;
        }

        std::cout << "Bad input. Please try again. (Format: YYYY/MM/DD
HH:MM:SS)" << std::endl;
    }

    candlestickType = obType;
    std::cout << "Change successfuly. Current type is: " << type << "\n" <<
std::endl;
    printCandlestick();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
void MerkelMain::switchCandlestickStartTimestamp()
{
    std::string timestamp;
    std::cout << "Enter a timestamp to set it as the start timestamp of the
candle stick. (Format: YYYY/MM/DD HH:MM:SS)" << std::endl;
    while (true)
    {
        std::cin >> timestamp;

        try
        {
            timestamp = timestamp.substr(0, 19);
            std::string day = timestamp.substr(0, 10);
            std::string time = timestamp.substr(11, 8);

            std::vector<std::string> tokens = CSVReader::tokenise(day, '/');
            if (tokens[0].length() != 4) throw;
            if (tokens[1].length() != 2) throw;
            if (tokens[2].length() != 2) throw;
            tokens.clear();

            tokens = CSVReader::tokenise(time, ':');
            if (tokens.size() != 3) throw;
            if (tokens[0].length() != 2 || tokens[1].length() != 2 ||
tokens[2].length() != 2) throw;

            break;
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << '\n';
        }

        std::cout << "Bad input. Please try again. (Format: YYYY/MM/DD
HH:MM:SS)" << std::endl;
    }

    candlestickStartTimestamp = timestamp;
    std::cout << "Change successfuly. Current start timestamp is: " << timestamp
<< "\n" << std::endl;
```

```
    printCandlestick();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
void MerkelMain::switchCandlestickInterval()
{
    std::string input;
    int num;
    std::cout << "Enter a new timestamp interval of the candlestick. (Will floor
to multiples of five)" << std::endl;
    while (true)
    {
        std::cin >> input;

        try
        {
            num = std::stoi(input);
            break;
        }
        catch(std::exception& e)
        {
            std::cerr << e.what() << '\n';
        }

        std::cout << "Please enter a positive integer." << std::endl;
    }

    candlestickInterval = num - (num % 5);
    std::cout << "Change successfuly. Current interval is: " <<
candlestickInterval << "\n" << std::endl;
    printCandlestick();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
void MerkelMain::exitDrawingPage()
{
    std::cout << "Back to main menu." << std::endl;
    indexOfMenus = 0;
}
// Made by myself - end
int MerkelMain::getUserOption()
{
    int userOption = 0;
    std::string line;
    std::cout << "Type in 1-" << menus[indexOfMenus].size() << std::endl;
    std::getline(std::cin, line);
    try{
        userOption = std::stoi(line);
    }catch(const std::exception& e)
    {
        //
    }
    std::cout << "You chose: " << userOption << std::endl;
    return userOption;
}

void MerkelMain::processUserOption(int userOption)
{
    // bad input
    if (userOption <= 0 || userOption > menus[indexOfMenus].size())
    {
        std::cout << "Invalid choice. Choose 1-" << menus[indexOfMenus].size()
<< std::endl;
        return;
    }
```

```
    // Wrote by AI
    // Prompt: How to store a function into a pair and call it
    voidFunc functionPointer = menus[indexOfMenus][userOption - 1].second;
    (this->*functionPointer)();
}

void MerkelMain::cleanConsole()
{
    // code from https://stackoverflow.com/questions/31201631/execute-cmd-
commands-using-c
    std::system("cls");
}

std::string MerkelMain::getCurrentSystemTimestamp()
{
    std::time_t t = std::time(0);
    std::tm* now = std::localtime(&t);
    // Made by myself - start
    std::string timestamp = std::to_string(now->tm_year + 1900) + '/' +
                            std::to_string(now->tm_mon + 1) + '/' +
                            std::to_string(now->tm_mday) + ' ';

    std::string hour = std::to_string(now->tm_hour);
    std::string min = std::to_string(now->tm_min);
    std::string sec = std::to_string(now->tm_sec);
    if (hour.length() < 2) hour = "0" + hour;
    if (min.length() < 2) min = "0" + min;
    if (sec.length() < 2) sec = "0" + sec;

    timestamp += hour + ':' + min + ':' + sec;

    return timestamp;
    // Made by myself - end
}
// OrderBookEntry.h
#pragma once

#include <string>

enum class OrderBookType{bid, ask, unknown, asksale, bidsale};

class OrderBookEntry
{
    public:


        OrderBookEntry( double _price,
                        double _amount,
                        std::string _timestamp,
                        std::string _product,
                        OrderBookType _orderType,
                        std::string username = "dataset");

        static OrderBookType stringToOrderBookType(std::string s);

        static bool compareByTimestamp(OrderBookEntry& e1, OrderBookEntry& e2)
        {
            return e1.timestamp < e2.timestamp;
        }
        static bool compareByPriceAsc(OrderBookEntry& e1, OrderBookEntry& e2)
        {
            return e1.price < e2.price;
        }
        static bool compareByPriceDesc(OrderBookEntry& e1, OrderBookEntry& e2)
```

```
        {
            return e1.price > e2.price;
        }
        // Made by myself - start
        static unsigned int calcIntervalByTimestamp(OrderBookEntry& e1,
OrderBookEntry&e2);
        /** give a timestamp and an interval to calculate the next timestamp
after the interval */
        static std::string calcNextTimestamp(std::string timestamp, unsigned int
interval);
        // Made by myself - end

        double price;
        double amount;
        std::string timestamp;
        std::string product;
        OrderBookType orderType;
        std::string username;
};
// OrderBookEntry.cpp
#include "OrderBookEntry.h"
#include "CSVReader.h"
#include <vector>

OrderBookEntry::OrderBookEntry( double _price,
                        double _amount,
                        std::string _timestamp,
                        std::string _product,
                        OrderBookType _orderType,
                        std::string _username)
: price(_price),
  amount(_amount),
  timestamp(_timestamp),
  product(_product),
  orderType(_orderType),
  username(_username)
{


}
// Made by myself - start
unsigned int OrderBookEntry::calcIntervalByTimestamp(OrderBookEntry& e1,
OrderBookEntry&e2)
{
  std::string timestamp1;
  std::string timestamp2;
  if (compareByTimestamp(e2, e1))
  {
    timestamp1 = e1.timestamp;
    timestamp2 = e2.timestamp;
  }
  else
  {
    timestamp1 = e2.timestamp;
    timestamp2 = e1.timestamp;
  }

  unsigned int interval = 0;
  // std::string day1 = timestamp1.substr(0, 10);
  std::string time1 = timestamp1.substr(11, 8);

  // std::string day2 = timestamp2.substr(0, 10);
  std::string time2 = timestamp2.substr(11, 8);
```

```
  if (time1 != time2)
  {
    std::vector<std::string> tokens1 = CSVReader::tokenise(time1, ':');
    std::vector<std::string> tokens2 = CSVReader::tokenise(time2, ':');
    int sec1 = std::stoi(tokens1[2]);
    int sec2 = std::stoi(tokens2[2]);
    int min1 = std::stoi(tokens1[1]);
    int min2 = std::stoi(tokens2[1]);
    int hour1 = std::stoi(tokens1[0]);
    int hour2 = std::stoi(tokens2[0]);

    interval += (hour1 - hour2) * 3600 + (min1 - min2) * 60 + (sec1 - sec2);
  }

  return interval;
}

std::string OrderBookEntry::calcNextTimestamp(std::string timestamp, unsigned
int interval)
{
  std::string day = timestamp.substr(0, 10);
  std::string time = timestamp.substr(11, 8);

  std::vector<std::string> tokens = CSVReader::tokenise(time, ':');
  int sec = std::stoi(tokens[2]);
  int min = std::stoi(tokens[1]);
  int hour = std::stoi(tokens[0]);
  int totalSeconds = hour * 3600 + min * 60 + sec + interval;

  std::string newHour = std::to_string((totalSeconds / 3600) % 24);
  if (newHour.length() < 2) newHour = "0" + newHour;
  std::string newMin = std::to_string((totalSeconds % 3600) / 60);
  if (newMin.length() < 2) newMin = "0" + newMin;
  std::string newSec = std::to_string(totalSeconds % 60);
  if (newSec.length() < 2) newSec = "0" + newSec;
  std::string newTime = newHour + ":" + newMin + ":" + newSec;

  return day + " " + newTime;
}
// Made by myself - end

OrderBookType OrderBookEntry::stringToOrderBookType(std::string s)
{
  if (s == "ask")
  {
    return OrderBookType::ask;
  }
  if (s == "bid")
  {
    return OrderBookType::bid;
  }
  return OrderBookType::unknown;
}
// OrderBook.h
#pragma once

#include "OrderBookEntry.h"
#include "candlestick.h"
#include "CSVReader.h"
#include <random>
#include <string>
#include <vector>

class OrderBook
```

```
{
    public:
    /** construct, reading a csv data file */
        OrderBook(std::string filename);
    /** return vector of all know products in the dataset*/
        std::vector<std::string> getKnownProducts();
    /** return vector of Orders according to the sent filters*/
        std::vector<OrderBookEntry> getOrders(OrderBookType type,
                                              std::string product,
                                              std::string timestamp);

        /** returns the earliest time in the orderbook*/
        std::string getEarliestTime();
        /** returns the next time after the
         * sent time in the orderbook
         * If there is no next timestamp, wraps around to the start
         * */
        std::string getNextTime(std::string timestamp);

        void insertOrder(OrderBookEntry& order);

        // Made by myself - start
        void appendOrder(OrderBookEntry& order);
        void sortOrder();
        // Made by myself - end

        std::vector<OrderBookEntry> matchAsksToBids(std::string product,
std::string timestamp);

        static double getHighPrice(std::vector<OrderBookEntry>& orders);
        static double getLowPrice(std::vector<OrderBookEntry>& orders);

        // Made by myself - start
        /** generate a vector includes candlestickEntry between two gave
timestamp */
        std::vector<candlestickEntry> generateCandlesticks(
            std::string startTimestamp,
            std::string endTimestamp,
            unsigned int timeInterval,
            std::string product,
            OrderBookType candlestickType
        );

        /** calculate the interval seconds between two time stamp */
        static unsigned int calcTimeInterval(std::string& timeStamp1,
std::string& timeStamp2);
        // Made by myself - end

    private:
        std::vector<OrderBookEntry> orders;


};
// OrderBook.cpp
#include "OrderBook.h"
#include "CSVReader.h"
#include <algorithm>
#include <iostream>
#include <cmath>
#include <fstream>
#include <map>

/** construct, reading a csv data file */
OrderBook::OrderBook(std::string filename)
```

```
{
    orders = CSVReader::readCSV(filename);
}

/** return vector of all know products in the dataset*/
std::vector<std::string> OrderBook::getKnownProducts()
{
    std::vector<std::string> products;

    std::map<std::string,bool> prodMap;

    for (OrderBookEntry& e : orders)
    {
        prodMap[e.product] = true;
    }

    // now flatten the map to a vector of strings
    for (auto const& e : prodMap)
    {
        products.push_back(e.first);
    }

    return products;
}
/** return vector of Orders according to the sent filters*/
std::vector<OrderBookEntry> OrderBook::getOrders(OrderBookType type,
                                          std::string product,
                                          std::string timestamp)
{
    std::vector<OrderBookEntry> orders_sub;
    for (OrderBookEntry& e : orders)
    {
        if (e.orderType == type &&
            e.product == product &&
            e.timestamp == timestamp )
            {
                orders_sub.push_back(e);
            }
    }
    return orders_sub;
}


double OrderBook::getHighPrice(std::vector<OrderBookEntry>& orders)
{
    double max = orders[0].price;
    for (OrderBookEntry& e : orders)
    {
        if (e.price > max)max = e.price;
    }
    return max;
}


double OrderBook::getLowPrice(std::vector<OrderBookEntry>& orders)
{
    double min = orders[0].price;
    for (OrderBookEntry& e : orders)
    {
        if (e.price < min)min = e.price;
    }
    return min;
}
```

```
std::string OrderBook::getEarliestTime()
{
    return orders[0].timestamp;
}

std::string OrderBook::getNextTime(std::string timestamp)
{
    std::string next_timestamp = "";
    for (OrderBookEntry& e : orders)
    {
        if (e.timestamp > timestamp)
        {
            next_timestamp = e.timestamp;
            break;
        }
    }
    if (next_timestamp == "")
    {
        next_timestamp = orders[0].timestamp;
    }
    return next_timestamp;
}

void OrderBook::insertOrder(OrderBookEntry& order)
{
    orders.push_back(order);
    std::sort(orders.begin(), orders.end(), OrderBookEntry::compareByTimestamp);
}
// Made by myself - start
void OrderBook::appendOrder(OrderBookEntry& order)
{
    orders.push_back(order);
}
void OrderBook::sortOrder()
{
    std::sort(orders.begin(), orders.end(), OrderBookEntry::compareByTimestamp);
}

std::vector<candlestickEntry> OrderBook::generateCandlesticks(
    std::string startTimestamp,
    std::string endTimestamp,
    unsigned int timeInterval,
    std::string product,
    OrderBookType candlestickType
)
{
    std::vector<candlestickEntry> candlesticks;
    std::vector<OrderBookEntry> orders_sub;
    std::string nextTimestamp =
OrderBookEntry::calcNextTimestamp(startTimestamp, timeInterval);

    for (OrderBookEntry& obe: orders)
    {
        std::string timestamp = obe.timestamp.substr(0, 19);
        // filter
        if (obe.product != product) continue;
        if (obe.orderType != candlestickType) continue;
        if (timestamp < startTimestamp) continue;
        if (timestamp > endTimestamp) break;

        // Time block check
        while (timestamp >= nextTimestamp)
        {
            if (!orders_sub.empty())
```

```
            {
                candlestickEntry cse {
                    startTimestamp,
                    nextTimestamp,
                    orders_sub[0].price,
                    OrderBook::getHighPrice(orders_sub),
                    OrderBook::getLowPrice(orders_sub),
                    orders_sub.back().price
                };
                candlesticks.push_back(cse);
                orders_sub.clear();
            }

            startTimestamp = nextTimestamp;
            nextTimestamp = OrderBookEntry::calcNextTimestamp(startTimestamp,
timeInterval);

            if (timestamp > endTimestamp) break;
        }

        // push the obe to cache
        if (timestamp >= startTimestamp && timestamp < nextTimestamp)
        {
            orders_sub.push_back(obe);
        }
    }

    // process remain obes
    if (!orders_sub.empty())
    {
        candlestickEntry cse {
            startTimestamp,
            orders_sub[orders_sub.size()-1].timestamp.substr(0, 19),
            orders_sub[0].price,
            OrderBook::getHighPrice(orders_sub),
            OrderBook::getLowPrice(orders_sub),
            orders_sub[orders_sub.size()-1].price
        };
        candlesticks.push_back(cse);
    }

    return candlesticks;
}

unsigned int OrderBook::calcTimeInterval(std::string& timeStamp1, std::string&
timeStamp2)
{
    std::string timeStamp1_p = timeStamp1.substr(11, 8);
    std::string timeStamp2_p = timeStamp2.substr(11, 8);

    int hour1 = std::stoi(timeStamp1_p.substr(0, 2));
    int minute1 = std::stoi(timeStamp1_p.substr(3, 2));
    int second1 = std::stoi(timeStamp1_p.substr(7, 2));
    int hour2 = std::stoi(timeStamp2_p.substr(0, 2));
    int minute2 = std::stoi(timeStamp2_p.substr(3, 2));
    int second2 = std::stoi(timeStamp2_p.substr(7, 2));

    unsigned int hour = std::abs(hour1 - hour2) * 3600;
    unsigned int minute = std::abs(minute1 - minute2) * 60;
    unsigned int second = std::abs(second1 - second2);

    return hour + minute + second;
}
// Made by myself - end
```

```
std::vector<OrderBookEntry> OrderBook::matchAsksToBids(std::string product,
std::string timestamp)
{
// asks = orderbook.asks
    std::vector<OrderBookEntry> asks = getOrders(OrderBookType::ask,
                                                 product,
                                                 timestamp);
// bids = orderbook.bids
    std::vector<OrderBookEntry> bids = getOrders(OrderBookType::bid,
                                                 product,
                                                    timestamp);

    // sales = []
    std::vector<OrderBookEntry> sales;

    // I put in a little check to ensure we have bids and asks
    // to process.
    if (asks.size() == 0 || bids.size() == 0)
    {
        std::cout << " OrderBook::matchAsksToBids no bids or asks" << std::endl;
        return sales;
    }

    // sort asks lowest first
    std::sort(asks.begin(), asks.end(), OrderBookEntry::compareByPriceAsc);
    // sort bids highest first
    std::sort(bids.begin(), bids.end(), OrderBookEntry::compareByPriceDesc);
    // for ask in asks:
    std::cout << "max ask " << asks[asks.size()-1].price << std::endl;
    std::cout << "min ask " << asks[0].price << std::endl;
    std::cout << "max bid " << bids[0].price << std::endl;
    std::cout << "min bid " << bids[bids.size()-1].price << std::endl;

    for (OrderBookEntry& ask : asks)
    {
    //     for bid in bids:
        for (OrderBookEntry& bid : bids)
        {
    //         if bid.price >= ask.price # we have a match
            if (bid.price >= ask.price)
            {
    //             sale = new order()
    //             sale.price = ask.price
            OrderBookEntry sale{ask.price, 0, timestamp,
                product,
                OrderBookType::asksale};

                if (bid.username == "simuser")
                {
                    sale.username = "simuser";
                    sale.orderType = OrderBookType::bidsale;
                }
                if (ask.username == "simuser")
                {
                    sale.username = "simuser";
                    sale.orderType =  OrderBookType::asksale;
                }

    //             # now work out how much was sold and
    //             # create new bids and asks covering
    //             # anything that was not sold
    //             if bid.amount == ask.amount: # bid completely clears ask
                if (bid.amount == ask.amount)
                {
```

```
    //                sale.amount = ask.amount
                    sale.amount = ask.amount;
    //                sales.append(sale)
                    sales.push_back(sale);
    //                bid.amount = 0 # make sure the bid is not processed again
                    bid.amount = 0;
    //                # can do no more with this ask
    //                # go onto the next ask
    //                break
                    break;
                }
    //           if bid.amount > ask.amount:  # ask is completely gone slice the
bid
                if (bid.amount > ask.amount)
                {
    //                sale.amount = ask.amount
                    sale.amount = ask.amount;
    //                sales.append(sale)
                    sales.push_back(sale);
    //                # we adjust the bid in place
    //                # so it can be used to process the next ask
    //                bid.amount = bid.amount - ask.amount
                    bid.amount =  bid.amount - ask.amount;
    //                # ask is completely gone, so go to next ask

    //                break
                    break;
                }


    //             if bid.amount < ask.amount # bid is completely gone, slice
the ask
                if (bid.amount < ask.amount &&
                   bid.amount > 0)
                {
    //                sale.amount = bid.amount
                    sale.amount = bid.amount;
    //                sales.append(sale)
                    sales.push_back(sale);
    //                # update the ask
    //                # and allow further bids to process the remaining amount
    //                ask.amount = ask.amount - bid.amount
                    ask.amount = ask.amount - bid.amount;
    //                bid.amount = 0 # make sure the bid is not processed again
                    bid.amount = 0;
    //                # some ask remains so go to the next bid
    //                continue
                    continue;
                }
            }
        }
    }
    return sales;
}
// Wallet.h
#pragma once

#include "OrderBookEntry.h"
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <random>
```

```
class Wallet
{
    public:
        // Made by myself - start
        Wallet(std::string _uuid, std::string walletString);
        // Made by myself - end
        /** insert currency to the wallet */
        void insertCurrency(std::string type, double amount);
        /** remove currency from the wallet */
        bool removeCurrency(std::string type, double amount);

        /** check if the wallet contains this much currency or more */
        bool containsCurrency(std::string type, double amount);
        /** checks if the wallet can cope with this ask or bid.*/
        bool canFulfillOrder(OrderBookEntry order);
        /** update the contents of the wallet
         * assumes the order was made by the owner of the wallet
        */
        void processSale(OrderBookEntry& sale);
        // Made by myself - start
        /** log user transition and the balance */
        void logInCSV();
        // Made by myself - end

        /** generate a string representation of the wallet */
        std::string toString();
        friend std::ostream& operator<<(std::ostream& os, Wallet& wallet);
        // Made by myself - start
        /** convert to a format that can store in CSV and read by the code */
        std::string storeInString();

        /** this funciton will read the history csv and display a specific
pieces by the arugment */
        void showTansitionOrTradingHistory(unsigned int pieces);
        /** this function will read the history csv and statistic all of the
activity */
        void statisticsUserActivity();

        void updateUserWalletCSV();

        std::string uuid;

    private:
        // there are two format of the operate
        // if only have input or output: {action},{currency},{amount}
        // if have both input and output: {action},{outgoingCurrency},
{outgoingAmount},{incomingCurrency},{incomingAmount}
        std::vector<std::string> operatesCache;
        // Made by myself - end
        std::map<std::string,double> currencies;
};
// Wallet.cpp
#include "Wallet.h"
#include "CSVReader.h"
#include <iostream>
#include <fstream>
#include <algorithm>

// Made by myself - start
Wallet::Wallet(std::string _uuid, std::string walletString)
: uuid(_uuid)
{
    // convert the data in the string and add to this wallet
    std::vector<std::string> tokens = CSVReader::tokenise(walletString, '|');
```

```
    for (std::string& token: tokens)
    {
        std::vector<std::string> elements = CSVReader::tokenise(token, ':');
        std::string currency = elements[0];
        double amount = std::stod(elements[1]);
        currencies[currency] = amount;
    }
}
// Made by myself - end
void Wallet::insertCurrency(std::string type, double amount)
{
    double balance;
    if (amount < 0)
    {
        throw std::exception{};
    }
    if (currencies.count(type) == 0) // not there yet
    {
        balance = 0;
    }
    else { // is there
        balance = currencies[type];
    }
    balance += amount;
    currencies[type] = balance;

    // Made by myself - start
    // Add log line in the cache
    std::string logLine = "insert," + type + "," + std::to_string(amount);
    operatesCache.push_back(logLine);
    // Made by myself - end
}

bool Wallet::removeCurrency(std::string type, double amount)
{
    if (amount < 0)
    {
        return false;
    }
    if (currencies.count(type) == 0) // not there yet
    {
        //std::cout << "No currency for " << type << std::endl;
        return false;
    }
    else { // is there - do  we have enough
        if (containsCurrency(type, amount))// we have enough
        {
            //std::cout << "Removing " << type << ": " << amount << std::endl;
            currencies[type] -= amount;

            // Made by myself - start
            std::string logLine = "remove," + type + "," +
std::to_string(amount);
            operatesCache.push_back(logLine);
            // Made by myself - end
            return true;
        }
        else // they have it but not enough.
            return false;
    }
}

bool Wallet::containsCurrency(std::string type, double amount)
{
```

```
    if (currencies.count(type) == 0) // not there yet
        return false;
    else
        return currencies[type] >= amount;

}

void Wallet::processSale(OrderBookEntry& sale)
{
    std::vector<std::string> currs = CSVReader::tokenise(sale.product, '/');
    double outgoingAmount;
    std::string outgoingCurrency;
    double incomingAmount;
    std::string incomingCurrency;
    // Made by myself - start
    // ask
    if (sale.orderType == OrderBookType::asksale)
    {
        outgoingAmount = sale.amount;
        outgoingCurrency = currs[0];
        incomingAmount = sale.amount * sale.price;
        incomingCurrency = currs[1];
    }
    // bid
    if (sale.orderType == OrderBookType::bidsale)
    {
        incomingAmount = sale.amount;
        incomingCurrency = currs[0];
        outgoingAmount = sale.amount * sale.price;
        outgoingCurrency = currs[1];
    }

    currencies[incomingCurrency] += incomingAmount;
    currencies[outgoingCurrency] -= outgoingAmount;

    // Add log line in the cache
    std::string logLine = "trade," + outgoingCurrency + "," +
std::to_string(outgoingAmount) + "," + incomingCurrency + "," +
std::to_string(incomingAmount);
    operatesCache.push_back(logLine);
    // Made by myself - end
}

bool Wallet::canFulfillOrder(OrderBookEntry order)
{
    std::vector<std::string> currs = CSVReader::tokenise(order.product, '/');
    // ask
    if (order.orderType == OrderBookType::ask)
    {
        double amount = order.amount;
        std::string currency = currs[0];
        std::cout << "Wallet::canFulfillOrder " << currency << " : " << amount
<< std::endl;

        return containsCurrency(currency, amount);
    }
    // bid
    if (order.orderType == OrderBookType::bid)
    {
        double amount = order.amount * order.price;
        std::string currency = currs[1];
        std::cout << "Wallet::canFulfillOrder " << currency << " : " << amount
<< std::endl;
        return containsCurrency(currency, amount);
```

```
    }


    return false;
}
// Made by myself - start
void Wallet::logInCSV()
{
    std::string filename = uuid + ".csv";
    std::ofstream writeFile(filename, std::ios::app);
    if (writeFile.is_open())
    {

        for (std::string operate: operatesCache)
        {
            writeFile << operate + '\n';
        }
        writeFile.close();
    }
}

void Wallet::updateUserWalletCSV()
{
    std::ifstream readWalletTable("walletTable.csv");
    std::map<std::string, std::string> table;
    if (readWalletTable.is_open())
    {
        std::string line;
        while(std::getline(readWalletTable, line))
        {
            std::vector<std::string> tokens = CSVReader::tokenise(line, ',');
            table[tokens[0]] = tokens[1];
        }
        table[uuid] = storeInString();
        readWalletTable.close();
    }

    std::ofstream writeWalletTable("walletTable.csv", std::ios::trunc);
    if (writeWalletTable.is_open())
    {
        for (std::pair<const std::string, std::string>& pair: table)
        {
            std::string newLine = pair.first + ',';
            if (pair.first == uuid)
            {
                newLine += storeInString();
            }
            else
            {
                newLine += pair.second + '\n';
            }
            writeWalletTable << newLine;
        }
        writeWalletTable.close();
    }
}

void Wallet::showTansitionOrTradingHistory(unsigned int pieces)
{
    std::ifstream readFile(uuid + ".csv");
    if (readFile.is_open())
    {
        std::vector<std::string> lines;
        std::string line;
```

```
        while (std::getline(readFile, line))
        {
            lines.push_back(line);
        }

        // set the start index at 0 if there are not enough pieces
        unsigned int startIndex;
        if (lines.size() < pieces + 1) startIndex = 0;
        else startIndex = lines.size() - 1 - pieces;

        for (unsigned int i = startIndex; i < lines.size(); ++i)
        {
            std::vector<std::string> tokens = CSVReader::tokenise(lines[i],
','');
            std::string action = tokens[0];
            if (action == "insert" || action == "remove")
            {
                line = "You inserted " + tokens[2] + " " + tokens[1] + ".";
            }
            if (action == "trade")
            {
                line = "You sold " + tokens[2] + " " + tokens[1] + " to get " +
tokens[4] + " " + tokens[3] + ".";
            }
            std::cout << line << std::endl;
        }

        readFile.close();
    }
}

void Wallet::statisticsUserActivity()
{
    std::ifstream readFile(uuid + ".csv");
    if (readFile.is_open())
    {
        // statistic the history
        std::map<std::string, double> statisticMap;
        std::string line;
        while (std::getline(readFile, line))
        {
            std::vector<std::string> tokens = CSVReader::tokenise(line, ',');
            std::string action = tokens[0];

            statisticMap[action] += 1;
            if (action == "insert" || action == "remove")
            {
                std::string currency = tokens[1];
                double amount = std::stod(tokens[2]);
                statisticMap[currency] += amount;
            }
            if (action == "trade")
            {
                std::string outgoingCurrency = tokens[1];
                double outgoingAmount = std::stod(tokens[2]);
                std::string incomingCurrency = tokens[3];
                double incomingAmount = std::stod(tokens[4]);

                statisticMap[outgoingCurrency] -= outgoingAmount;
                statisticMap[incomingCurrency] += incomingAmount;
            }
        }

        // print out the statistic
```

```
        for (std::pair<const std::string, double>& pair: statisticMap)
        {
            std::cout << pair.first << ": " << std::to_string(pair.second) <<
std::endl;
        }

        readFile.close();
    }
}
// Made by myself - end
std::string Wallet::toString()
{
    std::string s;
    for (std::pair<const std::string,double>& pair : currencies)
    {
        std::string currency = pair.first;
        double amount = pair.second;
        s += currency + " : " + std::to_string(amount) + "\n";
    }
    return s;
}
std::ostream& operator<<(std::ostream& os,  Wallet& wallet)
{
    os << wallet.toString();
    return os;
}
// Made by myself - start
std::string Wallet::storeInString()
{
    std::string s;
    for (std::pair<const std::string,double>& pair: currencies)
    {
        std::string currency = pair.first;
        double amount = pair.second;
        s += currency + ":" + std::to_string(amount) + "|";
    }

    s[s.length() - 1] = '\n';
    return s;
}
// Made by myself - end
// CSVReader.h
#pragma once

#include "OrderBookEntry.h"
#include <vector>
#include <string>


class CSVReader
{
    public:
     CSVReader();

     static std::vector<OrderBookEntry> readCSV(std::string csvFile);
     static std::vector<std::string> tokenise(std::string csvLine, char
separator);

     static OrderBookEntry stringsToOBE(std::string price,
                                        std::string amount,
                                        std::string timestamp,
                                        std::string product,
                                        OrderBookType OrderBookType);
```

```
    private:
     static OrderBookEntry stringsToOBE(std::vector<std::string> strings);

};
// CSVReader.cpp
#include "CSVReader.h"
#include <iostream>
#include <fstream>


CSVReader::CSVReader()
{

}

std::vector<OrderBookEntry> CSVReader::readCSV(std::string csvFilename)
{
    std::vector<OrderBookEntry> entries;

    std::ifstream csvFile{csvFilename};
    std::string line;
    if (csvFile.is_open())
    {
        std::cout << "Loading " << csvFilename << std::endl;
        while(std::getline(csvFile, line))
        {
            try
            {
                OrderBookEntry obe = stringsToOBE(tokenise(line, ','));
                entries.push_back(obe);
            }
            catch(const std::exception& e)
            {
                std::cout << "CSVReader::readCSV bad data"  << std::endl;
            }
        }// end of while
    }

    std::cout << "CSVReader::readCSV read " << entries.size() << " entries\n"
<< std::endl;
    return entries;
}

std::vector<std::string> CSVReader::tokenise(std::string csvLine, char
separator)
{
   std::vector<std::string> tokens;
   signed int start, end;
   std::string token;
    start = csvLine.find_first_not_of(separator, 0);
    do{
        end = csvLine.find_first_of(separator, start);
        if (start == csvLine.length() || start == end) break;
        if (end >= 0) token = csvLine.substr(start, end - start);
        else token = csvLine.substr(start, csvLine.length() - start);
        tokens.push_back(token);
    start = end + 1;
    }while(end > 0);

   return tokens;
}

OrderBookEntry CSVReader::stringsToOBE(std::vector<std::string> tokens)
{
```

```
    double price, amount;

    if (tokens.size() != 5) // bad
    {
        std::cout << "Bad line " << std::endl;
        throw std::exception{};
    }
    // we have 5 tokens
    try {
         price = std::stod(tokens[3]);
         amount = std::stod(tokens[4]);
    }catch(const std::exception& e){
        std::cout << "CSVReader::stringsToOBE Bad float! " << tokens[3]<<
std::endl;
        std::cout << "CSVReader::stringsToOBE Bad float! " << tokens[4]<<
std::endl;
        throw;
    }

    OrderBookEntry obe{price,
                       amount,
                       tokens[0],
                       tokens[1],
                       OrderBookEntry::stringToOrderBookType(tokens[2])};

    return obe;
}


OrderBookEntry CSVReader::stringsToOBE(std::string priceString,
                                       std::string amountString,
                                       std::string timestamp,
                                       std::string product,
                                       OrderBookType orderType)
{
    double price, amount;
    try {
         price = std::stod(priceString);
         amount = std::stod(amountString);
    }catch(const std::exception& e){
        std::cout << "CSVReader::stringsToOBE Bad float! " << priceString<<
std::endl;
        std::cout << "CSVReader::stringsToOBE Bad float! " << amountString<<
std::endl;
        throw;
    }
    OrderBookEntry obe{price,
                    amount,
                    timestamp,
                    product,
                    orderType};

    return obe;
}
// Candlestick.h
// Made by myself - start
#pragma once

#include <string>
#include <vector>

struct candlestickEntry
{
    std::string startTimestamp;
```

```
    std::string endTimestamp;
    double open;
    double high;
    double low;
    double close;
};

class Candlestick
{
    public:
        Candlestick();
        /** print out a list of candle stick information by the input vector */
        static void printCandlestick(std::vector<candlestickEntry>
candlesticks);

        std::string date;
        double open;
        double high;
        double low;
        double close;
};
// Made by myself - end
// Candlestick.cpp
// Made by myself - start
#include "Candlestick.h"
#include "CSVReader.h"
#include <iostream>
#include <vector>

Candlestick::Candlestick()
{
}

void Candlestick::printCandlestick(std::vector<candlestickEntry> candlesticks)
{
    if (candlesticks.empty()) return;

    for (candlestickEntry& cse: candlesticks)
    {
        std::cout << cse.startTimestamp << " --> " << cse.endTimestamp << "\n"
<< std::endl;

        // up day
        if (cse.close >= cse.open)
        {
            // Made by myself - end
            std::cout << "\033[32m";
            // Made by myself - start
            std::cout << "High: " << cse.high << std::endl;
            std::cout << "Close: " << cse.close << std::endl;
            std::cout << "Open: " << cse.open << std::endl;
            std::cout << "Low: " << cse.low << std::endl;
        }
        // down day
        if (cse.close < cse.open)
        {
            // Made by myself - end
            std::cout << "\033[31m";
            // Made by myself - start
            std::cout << "High: " << cse.high << std::endl;
            std::cout << "Open: " << cse.open << std::endl;
            std::cout << "Close: " << cse.close << std::endl;
            std::cout << "Low: " << cse.low << std::endl;
        }
```

```
        // stop color text
        // Made by myself - end
        std::cout << "\033[0m";
        // Made by myself - start
        std::cout << "==============" << std::endl;
    }
}
// Made by myself - end
// AccountManager.h
// Made by myself - start
#pragma once

#include "Wallet.h"
#include <string>
#include <map>
#include <set>

struct UserInfo
{
    std::string username;
    std::size_t password_h;
    std::string email;
};

class AccountManager
{
    public:
        AccountManager();
        static Wallet login();
    private:
        static bool createAccount();
        static void loadAccounts(std::string filename);
        static void updateUserCSV();
        static Wallet getWallet(std::string uuid);
        static bool findUsername(std::string username);
        static bool resetPassword(std::string uuid);
        static std::string generateUUID(int length);
        static std::map<std::string, UserInfo> cache;
        static std::set<std::string> existUUID;
        static std::hash<std::string> hasher;
};
// Made by myself - end
// AccountManager.cpp
// Made by myself - start
#include "AccountManager.h"
#include "CSVReader.h"
#include <iostream>
#include <fstream>
#include <random>
#include <vector>

AccountManager::AccountManager() {}

Wallet AccountManager::login()
{

    loadAccounts("accounts.csv");
    std::string uuid;
    std::string password;
    std::string mode;
    std::cout << "Please choose Login or Create a new account. (Type login or
create)" << std::endl;
    std::cin >> mode;
```

```
    while (true)
    {
        if (mode == "login")
        {
            std::cout << "UUID: " << std::flush;
            std::cin >> uuid;
            if (findUsername(uuid))
            {
                int attempts = 0;
                while (true)
                {
                    std::cout << "Password: " << std::flush;
                    std::cin >> password;
                    if (hasher(password) == cache[uuid].password_h)
                    {
                        std::cout << "Login in successfully\n" << std::endl;

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                        return getWallet(uuid);
                    }
                    else
                    {
                        if (attempts > 1)
                        {
                            std::string answer;
                            std::cout << "Forget password?. (Type retry or
reset)" << std::endl;
                            std::cin >> answer;

                            if (answer == "retry")
                            {
                                continue;
                            }
                            else
                            {
                                resetPassword(uuid);
                                break;
                            }
                        }
                        else
                        {
                            std::cout << "Wrong password. Please try again." <<
std::endl;
                        }
                        attempts++;
                        continue;
                    }
                }
            }
            else
            {
                std::string answer;
                std::cout << "This UUID is not exist.\nDo u want to create a
account or try again. (Input create or retry)" << std::endl;
                while (true)
                {
                    std::cin >> answer;
                    if (answer != "retry" && answer != "create")
                    {
                        std::cout << "Unknown answer, please try again." <<
std::endl;
                    }
                    else
                    {
```

```
                        break;
                    }
                }

                if (answer == "create")
                {
                    createAccount();
                }
                else
                {
                    continue;
                }
            }

        }
        else if (mode == "create")
        {
            if (createAccount())
            {
                mode = "login";
            }
        }
        else
        {
            std::cout << "Unknow input. Please Try again. (Type login or
create)" << std::endl;
            std::cin >> mode;
        }
    }
}

bool AccountManager::createAccount()
{
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    std::string username;
    std::string password;
    std::string email;
    std::cout << "Please type your full name." << std::endl;
    while (true)
    {
        bool duplicates = false;
        std::cout << "Full name: " << std::flush;
        std::getline(std::cin, username);

        for (auto& data: cache)
        {
            if (data.second.username == username)
            {
                std::cout << "This name is already have an account. Please use
other name." <<std::endl;
                duplicates = true;
            }
        }

        if (!duplicates)
        {
            break;
        }
    }

    std::cout << "Please set a password." << std::endl;
    std::cout << "Password: " << std::flush;
    std::cin >> password;
```

```
    std::cout << "Please connect an email." << std::endl;
    while (true)
    {
        bool duplicates = false;
        std::cout << "Email: " << std::flush;
        std::cin >> email;

        for (auto& data: cache)
        {
            if (data.second.username == username)
            {
                std::cout << "This email is already been use. Please use other
email." <<std::endl;
                duplicates = true;
            }
        }

        if (!duplicates)
        {
            break;
        }
    }

    UserInfo info = {username, hasher(password), email};
    std::string uuid = generateUUID(10);
    existUUID.insert(uuid);
    cache[uuid] = info;
    updateUserCSV();

    Wallet wallet{uuid, ""};
    wallet.insertCurrency("BTC", 10);
    wallet.insertCurrency("USDT", 100);
    wallet.updateUserWalletCSV();
    std::cout << "Create successfully.\nYour UUID is " << uuid << std::endl;
    return true;
}

void AccountManager::loadAccounts(std::string filename)
{
    cache.clear();
    std::fstream accounts(filename);
    if (accounts.is_open())
    {
        std::string line;
        while (std::getline(accounts, line))
        {
            std::vector<std::string> tokens = CSVReader::tokenise(line, ',');
            UserInfo info = { tokens[1], std::stoull(tokens[2]), tokens[3] };
            cache[tokens[0]] = info;
            existUUID.insert(tokens[0]);
        }

        accounts.close();
    }
}

void AccountManager::updateUserCSV()
{
    std::ofstream accounts("accounts.csv", std::ios::trunc);
    if (accounts.is_open())
    {
        for (std::pair<const std::string, UserInfo>& pair: cache)
        {
```

```
            std::string uuid = pair.first;
            std::string username = pair.second.username;
            std::string password = std::to_string(pair.second.password_h);
            std::string email = pair.second.email;
            std::string newLine = uuid + ',' + username + ',' + password + ',' +
email + '\n';
            accounts << newLine;
        }

        accounts.close();
    }
}

Wallet AccountManager::getWallet(std::string uuid)
{
    std::ifstream table("walletTable.csv");
    std::string walletSetting;
    if (table.is_open())
    {
        std::string line;
        while (std::getline(table, line))
        {
            std::vector<std::string> tokens = CSVReader::tokenise(line, ',');
            if (tokens[0] == uuid)
            {
                walletSetting = tokens[1];
                break;
            }
        }

        table.close();
    }
    Wallet wallet{uuid, walletSetting};
    return wallet;
}

bool AccountManager::findUsername(std::string username)
{
    return cache.find(username) != cache.end();
}

bool AccountManager::resetPassword(std::string uuid)
{
    std::string OTP;
    while (true)
    {
        // Generate OTP
        OTP = "1234";
        // send OTP to email
        while (true)
        {
            std::string input;
            std::cout << "OTP: " << std::flush;
            std::cin >> input;

            if (input == OTP)
            {
                std::string newPassword;
                std::cout << "New Password: " << std::flush;
                std::cin >> newPassword;

                cache[uuid].password_h = hasher(newPassword);
                updateUserCSV();
                return true;
```

```
            }
            else
            {
                std::string answer;
                std::cout << "Wrong OTP. You want to try again or resent a new
one. (Type retry or send)" << std::endl;
                std::cin >> answer;

                if (answer == "send")
                {
                    break;
                }
            }
        }
    }

    return false;
}

std::string AccountManager::generateUUID(int length)
{
    const std::string characters = "0123456789";
    std::string uuid;
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<> distribution(0, characters.size() - 1);

    while (true)
    {
        uuid = "";
        for (int i = 0; i < length; i++)
        {
            int index = distribution(generator);
            uuid += characters[index];
        }

        if (existUUID.find(uuid) != existUUID.end())
        {
            continue;
        }
        else
        {
            return uuid;
        }
    }
}

std::map<std::string, UserInfo> AccountManager::cache;
std::set<std::string> AccountManager::existUUID;
std::hash<std::string> AccountManager::hasher;
// Made by myself - end
```