CandleStick.cpp

```cpp
/* ------------------------ Code written by myself ------------------------*/

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

Candlestick::Candlestick(
std::string candleTimestamp,
double candleOpenPrice,
double candleHighPrice,
double candleLowPrice,
double candleClosingPrice)
: candleTimestamp(candleTimestamp),
candleOpenPrice(candleOpenPrice),
candleHighPrice(candleHighPrice),
candleLowPrice(candleLowPrice),
candleClosingPrice(candleClosingPrice)
{
}

/* ------------------------ end ------------------------*/
```

Candlestick.h

```
/* ---------------------- Code written by myself ----------------------*/

#pragma oncE
#include <string>
#include "OrderBookEntry.h"
#include <vector>
#include <map>

class Candlestick
{
public:
Candlestick(
std::string candleTimestamp,
double candleOpenPrice,
double candleHighPrice,
double candleLowPrice,
double candleClosingPrice);

std::string candleTimestamp;
double candleOpenPrice;
double candleHighPrice;
double candleLowPrice;
double candleClosingPrice;
};

/* ---------------------- end ----------------------*/
```

CSVReader.cpp

```
#include "CSVReader.h"
#include <fstream>
#include <iostream>

CSVReader::CSVReader() {}

std::vector<OrderBookEntry> CSVReader::readCSV(std::string csvFilename)
{
std::vector<OrderBookEntry> entries; // Vector to store the order book entries

std::ifstream csvFile{csvFilename}; // Open the CSV file
std::string line;

if (csvFile.is_open())
{
while (std::getline(csvFile, line))
{
try
{
OrderBookEntry obe = stringsToOBE(tokenise(line, ',')); // Convert CSV line to OrderBookEntry
entries.push_back(obe); // Add the entry to the vector
}
catch (const std::exception &e)
{
std::cout << "CSVReader:: readCSV red " << entries.size() << " entries" << std::endl;
}
}
}

std::cout << "CSVReader:: readCSV red " << entries.size() << " entries" << std::endl;
return entries; // Return the vector of order book entries
}

std::vector<std::string> CSVReader::tokenise(std::string csvLine, char separator)
{
std::vector<std::string> tokens; // Vector to store the tokens

signed int start, end; // Variables to keep track of token start and end positions

std::string token; // Variable to store each token

start = csvLine.find_first_not_of(separator, 0); // Find the first non-separator character in the line

// Loop until the end of the line is reached
do
```

```
{
end = csvLine.find_first_of(separator, start); // Find the next separator character from the current
start position

// If no separator is found or the start and end positions are the same, break the loop
if (start == csvLine.length() || start == end)
break;

// If a separator is found, extract the token between start and end positions
if (end >= 0)
token = csvLine.substr(start, end - start);
else
token = csvLine.substr(start, csvLine.length() - start);

tokens.push_back(token); // Add the token to the vector

start = end + 1; // Update the start position for the next iteration
} while (end != std::string::npos); // Continue until the end of the line is reached

return tokens; // Return the vector of tokens
}

OrderBookEntry CSVReader::stringsToOBE(std::vector<std::string> tokens)
{
double price, amount;

if (tokens.size() != 5)
{
std::cout << "Bad Line" << tokens[0] << std::endl;
throw std::exception{};
}

try
{
// Convert data types
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; // Return the OrderBookEntry object
}

OrderBookEntry CSVReader::stringsToOBE(std::string priceString, std::string amountString,
std::string timestamp, std::string product,
OrderBookType orderType)
{
double price, amount;

try
{
// Convert data types
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; // Return the OrderBookEntry object
}
```

CSVReader.h

```
#pragma once
#include "OrderBookEntry.h"
#include <string>
#include <vector>

/** The CSVReader class is responsible for reading a CSV file and converting it into a vector of
OrderBookEntry objects. */
class CSVReader
{

public:
CSVReader();

/** Read the CSV file and convert it into a vector of OrderBookEntry objects.
* @param csvFilename The filename of the CSV file to be read.
* @return A vector of OrderBookEntry objects representing the data from the CSV file.
*/
static std::vector<OrderBookEntry> readCSV(std::string csvFilename);

/** Tokenize a CSV line into individual tokens.
* @param csvLine The CSV line to be tokenized.
* @param separator The character used as a separator in the CSV line.
* @return A vector of strings representing the tokens extracted from the CSV line.
*/
static std::vector<std::string> tokenise(std::string csvLine, char separator);

/** Convert a vector of strings to an OrderBookEntry object.
* @param tokens The vector of strings representing the tokens of an OrderBookEntry.
* @return An OrderBookEntry object created from the provided tokens.
*/
static OrderBookEntry stringsToOBE(std::vector<std::string> tokens);

/** Convert strings representing individual attributes of an OrderBookEntry to an OrderBookEntry
object.
* @param priceString The string representing the price of the OrderBookEntry.
* @param amountString The string representing the amount of the OrderBookEntry.
* @param timestamp The string representing the timestamp of the OrderBookEntry.
* @param product The string representing the product of the OrderBookEntry.
* @param orderType The OrderBookType of the OrderBookEntry.
* @return An OrderBookEntry object created from the provided attribute strings.
*/
static OrderBookEntry stringsToOBE(std::string priceString, std::string amountString,
std::string timestamp, std::string product,
OrderBookType orderType);
};
```

main.cpp

```
/*
main.cpp

This is the main entry point of the program. It initializes the MerkelMain
application and starts the program execution.

*/

#include <iostream>
#include <string>
#include <vector>
#include "OrderBookEntry.h"
#include "MerkelMain.h"
#include "CSVReader.h"
#include "Wallet.h"
#include "CandleStick.h"

int main()
{
MerkelMain app{}; // Create an instance of the MerkelMain application
app.init(); // Initialize the application

return 0;
}
```

MerkelMain.cpp

```
/*
MerkelMain.cpp

This file contains the implementation of the MerkelMain class, which represents
the application itself. It handles user input, menu printing, and the execution
of various functionalities.
*/

#include "MerkelMain.h"
#include "CSVReader.h"
#include "OrderBookEntry.h"
#include "CandleStick.h"
#include <iostream>
#include <vector>
#include <limits>

MerkelMain::MerkelMain()
{
// Constructor implementation
}

void MerkelMain::init()
{
int input;
currentTime = orderBook.getEarliestTime();

wallet.insertCurrency("BTC", 10);

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

void MerkelMain::printMenu()
{
// Print the menu options
std::cout << "1: Print help" << std::endl;
std::cout << "2: Print exchange stats" << std::endl;
std::cout << "3: Print candlestck stats" << std::endl;
std::cout << "4: Make an offer" << std::endl;
std::cout << "5: Make a bid" << std::endl;
std::cout << "6: Print wallet" << std::endl;
```

```
std::cout << "7: Continue" << std::endl;
std::cout << "================ " << std::endl;
std::cout << "Current Time is: " << currentTime << std::endl;
}

void MerkelMain::printHelp()
{
// Print the help information
std::cout << "Help - your aim is to make money. Analyze the market and make bids and offers." <<
std::endl;
}

void MerkelMain::printMarketStats()
{
// Print the market statistics
for (std::string const &p : orderBook.getKnownProducts())
{
std::cout << "Product: " << p << std::endl;

std::vector<OrderBookEntry> entriesask = orderBook.getOrders(OrderBookType::ask, p,
currentTime);
std::cout << "Asks for product seen: " << entriesask.size() << std::endl;
std::cout << "Max ask: " << OrderBook::getHighPrice(entriesask) << std::endl;
std::cout << "Min ask: " << OrderBook::getLowPrice(entriesask) << std::endl;
std::cout << " " << std::endl;

std::vector<OrderBookEntry> entriesbid = orderBook.getOrders(OrderBookType::bid, p,
currentTime);
std::cout << "Bids for product seen: " << entriesbid.size() << std::endl;
std::cout << "Max bid: " << OrderBook::getHighPrice(entriesbid) << std::endl;
std::cout << "Min bid: " << OrderBook::getLowPrice(entriesbid) << std::endl;
std::cout << " " << std::endl;
}
}

/* ---------------------- Code written by myself ----------------------*/

void MerkelMain::displayKnownProducts()
{
// Display the known products
std::vector<std::string> products = orderBook.getKnownProducts();
for (std::string const &p : products)
{
std::cout << p << std::endl;
}
}

std::vector<Candlestick> MerkelMain::candleStickData()
{
// Get candlestick data for a specific product and order type
```

```
std::string orderTypeStr;
std::string product;

// Prompt user for order type
std::cout << "Enter order type (ask or bid): " << std::endl;
std::getline(std::cin, orderTypeStr);

// Display the known products and prompt user for product selection
displayKnownProducts();
std::cout << "Enter product from the options above: " << std::endl;
std::getline(std::cin, product);

// Determine the order type based on user input
OrderBookType order = orderTypeStr == "ask" ? OrderBookType::ask : OrderBookType::bid;

// Get the unique timestamps from the order book
std::vector<std::string> uniqueTimestamps;
std::string currentTimestamp = orderBook.getEarliestTime();
uniqueTimestamps.push_back(currentTimestamp);

while (orderBook.getNextTime(currentTimestamp) > currentTimestamp)
{
currentTimestamp = orderBook.getNextTime(currentTimestamp);
uniqueTimestamps.push_back(currentTimestamp);
}

// Create a vector to store the candlestick data
std::vector<Candlestick> candlesticks;
double openPrice = 0.0;
double highestPrice = 0.0;
double lowestPrice = std::numeric_limits<double>::max();
double closePrice = 0.0;

// Iterate over the unique timestamps and compute candlestick data
for (const std::string &timestamp : uniqueTimestamps)
{
// Get the orders for the current timestamp, product, and order type
std::vector<OrderBookEntry> entries = orderBook.getOrders(order, product, timestamp);

double amount = 0.0;
double price = 0.0;

// Iterate over the entries for the current timestamp
for (const OrderBookEntry &entry : entries)
{
// Update the highest and lowest prices
if (entry.price > highestPrice)
{
highestPrice = entry.price;
}
```

```
if (entry.price < lowestPrice)
{
lowestPrice = entry.price;
}

// Compute the total amount and total price
amount += entry.amount;
price += entry.price;
}

// Update the open and close prices for the current timestamp
openPrice = closePrice;
double totalAmount = amount * price;
closePrice = totalAmount / amount;

// Create a Candlestick object and add it to the vector
Candlestick candlestick(timestamp, openPrice, highestPrice, lowestPrice, closePrice);
candlesticks.push_back(candlestick);
}

// Print the candlestick data
for (const Candlestick &candlestick : candlesticks)
{
std::cout << "[" << candlestick.candleTimestamp << " " << candlestick.candleOpenPrice << " " <<
candlestick.candleHighPrice
<< " " << candlestick.candleLowPrice << " " << candlestick.candleClosingPrice << "]" <<
std::endl;
}

return candlesticks;
}

/* ---------------------- end ----------------------*/

void MerkelMain::enterAsk()
{
// Handle entering an ask
std::cout << "Mark and ask - enter the amount: product,price,amount, e.g., 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 << "Bad input!" << input << "Please make sure you don't use spaces between the 3
values" << 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 << "Insufficient funds." << std::endl;
}
}
catch (const std::exception &e)
{
std::cout << "MerkelMain::enterAsk Bad input" << std::endl;
}
}

std::cout << "You typed: " << input << std::endl;
}

void MerkelMain::enterBid()
{
// Handle entering a bid
std::cout << "Make a bid - enter the amount: product,price,amount, e.g., 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 << "Bad input!" << input << "Please make sure you don't use spaces between the 3
values" << 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 << "Insufficient funds." << std::endl;
}
}
catch (const std::exception &e)
{
std::cout << "MerkelMain::enterBid Bad input" << std::endl;
}
}

std::cout << "You typed: " << input << std::endl;
}

void MerkelMain::printWallet()
{
// Print the wallet contents
std::cout << wallet.toString() << std::endl;
}

void MerkelMain::gotoNextTimeframe()
{
// Go to the next timeframe
std::cout << "Going to next time frame." << std::endl;
std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids("ETH/BTC", 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")
{
wallet.processSale(sale);
}
}

currentTime = orderBook.getNextTime(currentTime);
}

int MerkelMain::getUserOption()
{
// Get the user's menu option
```

```
int userOption = 0;
std::string line;

std::cout << "Type in 1-6" << 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)
{
// Process the user's menu option
if (userOption == 0)
{
std::cout << "Invalid choice. Choose 1-6" << std::endl;
}
else if (userOption == 1)
{
printHelp();
}
else if (userOption == 2)
{
printMarketStats();
}
else if (userOption == 3)
{
candleStickData();
}
else if (userOption == 4)
{
enterAsk();
}
else if (userOption == 5)
{
enterBid();
}
else if (userOption == 6)
{
printWallet();
}
else if (userOption == 7)
```

```
{
gotoNextTimeframe();
}
}
```

MerkelMain.h

```
/*
This file contains the declaration of the MerkelMain class, which represents
the application itself. It handles user input, menu printing, and the execution
of various functionalities.
*/

#pragma once

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

class MerkelMain
{
public:
MerkelMain();

/*This function will initiatize the constructor function */
void init();

private:
/*This function will display all the menu options*/
void printMenu();

/*This function will handle the help menu*/
void printHelp();

/* ---------------------- Code written by myself ----------------------*/

/*This function will handle the candlestick stats*/
std::vector<Candlestick> candleStickData();

/*This function will handle the known products*/
void displayKnownProducts();

/* ---------------------- end ----------------------*/

/*This function will handle the market stats*/
void printMarketStats();

/*This function will handle the offers*/
void enterAsk();
```

```
/*This function will handle the bids*/
void enterBid();

/*This function will handle the wallet*/
void printWallet();

/*This function will handle the time frame*/
void gotoNextTimeframe();

/*This function will receive and return the user input in response to the menu
* option selected*/
int getUserOption();

/*This function will handle userOption and execute the desired functionality*/
void processUserOption(int userOption);

std::string currentTime;

OrderBook orderBook{"20200601.csv"};

Wallet wallet;
};
```

OrderBook.cpp

```cpp
#include "OrderBook.h"
#include "CSVReader.h"
#include <map>
#include <algorithm>

// Construct the OrderBook by reading a CSV data file
OrderBook::OrderBook(std::string filename)
{
orders = CSVReader::readCSV(filename);
}

// Return a vector of all known products in the dataset
std::vector<std::string> OrderBook::getKnownProducts()
{
std::vector<std::string> products;
std::map<std::string, bool> prodMap;

// Iterate through each OrderBookEntry and add unique products to the map
for (OrderBookEntry &e : orders)
{
prodMap[e.product] = true;
}

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

return products;
}

// Return a vector of Orders according to the specified filters (type, product, timestamp)
std::vector<OrderBookEntry> OrderBook::getOrders(OrderBookType type, std::string product,
std::string timestamp)
{
std::vector<OrderBookEntry> orders_sub;

// Iterate through each OrderBookEntry and add matching orders to the vector
for (OrderBookEntry &e : orders)
{
if (e.orderType == type && e.product == product && e.timestamp == timestamp)
{
orders_sub.push_back(e);
}
```

```
}

return orders_sub;
}

// Get the highest price from a vector of OrderBookEntry objects
double OrderBook::getHighPrice(std::vector<OrderBookEntry> &orders)
{
double max = orders[0].price;

// Iterate through each OrderBookEntry and update the maximum price if found
for (OrderBookEntry &e : orders)
{
if (e.price > max)
{
max = e.price;
}
}

return max;
}

// Get the lowest price from a vector of OrderBookEntry objects
double OrderBook::getLowPrice(std::vector<OrderBookEntry> &orders)
{
double min = orders[0].price;

// Iterate through each OrderBookEntry and update the minimum price if found
for (OrderBookEntry &e : orders)
{
if (e.price < min)
{
min = e.price;
}
}

return min;
}

// Get the earliest timestamp in the OrderBook
std::string OrderBook::getEarliestTime()
{
return orders[0].timestamp;
}

// Get the next timestamp after the given timestamp in the OrderBook
std::string OrderBook::getNextTime(std::string timestamp)
{
std::string next_timestamp = "";
```

```
// Iterate through each OrderBookEntry and find the next timestamp
for (OrderBookEntry &e : orders)
{
if (e.timestamp > timestamp)
{
next_timestamp = e.timestamp;
break;
}
}

// If no next timestamp is found, wrap around to the start by using the earliest timestamp
if (next_timestamp == "")
{
next_timestamp = orders[0].timestamp;
}

return next_timestamp;
}

// Insert an OrderBookEntry into the OrderBook
void OrderBook::insertOrder(OrderBookEntry &order)
{
orders.push_back(order);
std::sort(orders.begin(), orders.end(), OrderBookEntry::compareByTimeStamp);
}

// Match asks to bids and generate sales based on the given product and timestamp
std::vector<OrderBookEntry> OrderBook::matchAsksToBids(std::string product, std::string
timestamp)
{
std::vector<OrderBookEntry> asks = getOrders(OrderBookType::ask, product, timestamp);
std::vector<OrderBookEntry> bids = getOrders(OrderBookType::bid, product, timestamp);
std::vector<OrderBookEntry> sales;

// Sort asks in ascending order of price
std::sort(asks.begin(), asks.end(), OrderBookEntry::compareByPriceAsc);

// Sort bids in descending order of price
std::sort(bids.begin(), bids.end(), OrderBookEntry::compareByPriceDesc);

// Iterate through asks and bids to match and generate sales
for (OrderBookEntry &ask : asks)
{
for (OrderBookEntry &bid : bids)
{
if (bid.price >= ask.price)
{
OrderBookEntry sale{ask.price, 0, timestamp, product, OrderBookType::asksale};

// Adjust sale parameters based on bid and ask types
```

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

// Match bids and asks based on their amounts
if (bid.amount == ask.amount)
{
sale.amount = ask.amount;
sales.push_back(sale);
bid.amount = 0;
break;
}
if (bid.amount > ask.amount)
{
sale.amount = ask.amount;
sales.push_back(sale);
bid.amount -= ask.amount;
break;
}
if (bid.amount < ask.amount && bid.amount > 0)
{
sale.amount = bid.amount;
sales.push_back(sale);
ask.amount -= bid.amount;
bid.amount = 0;
continue;
}
}
}
}

return sales;
}
```

OrderBook.h

```cpp
#pragma once
#include "CSVReader.h"
#include "OrderBookEntry.h"
#include <string>
#include <vector>

/** The OrderBook class presents a high-level interface for working with the orders in the order book dataset.
* It provides functions to retrieve orders based on filters, get information about known products, and perform calculations on the orders.
*/
class OrderBook
{
public:
/** Construct the OrderBook by reading a CSV data file */
OrderBook(std::string filename);

/** Return a vector of all known products in the dataset */
std::vector<std::string> getKnownProducts();

/** Return a vector of orders filtered by type, product, and timestamp */
std::vector<OrderBookEntry> getOrders(OrderBookType type, std::string product, std::string timestamp);

/** Get the highest price from a vector of OrderBookEntry objects */
static double getHighPrice(std::vector<OrderBookEntry> &orders);

/** Get the lowest price from a vector of OrderBookEntry objects */
static double getLowPrice(std::vector<OrderBookEntry> &orders);

/** Get the earliest timestamp in the OrderBook */
std::string getEarliestTime();

/** Get the next timestamp after the given timestamp in the OrderBook */
std::string getNextTime(std::string timestamp);

/** Insert an OrderBookEntry into the OrderBook */
void insertOrder(OrderBookEntry &order);

/** Match asks to bids and generate sales based on the given product and timestamp */
std::vector<OrderBookEntry> matchAsksToBids(std::string product, std::string timestamp);

private:
std::vector<OrderBookEntry> orders;
};
```

OrderBookEntry.cpp

```cpp
#include "OrderBookEntry.h"

/* Constructor: Constructs an OrderBookEntry object with the given parameters. */
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)
{
}

/* Converts a string to an OrderBookType enum value. */
OrderBookType OrderBookEntry::stringToOrderBookType(const std::string &s)
{
if (s == "ask")
{
return OrderBookType::ask;
}
if (s == "bid")
{
return OrderBookType::bid;
}
return OrderBookType::unknown;
}
```

OrderBookEntry.h

```
#pragma once
#include <string>

/*This is a class that will store the two options of OrderBookType vector*/
enum class OrderBookType
{
bid,
ask,
unknown,
asksale,
bidsale
};

/** The OrderBookEntry represents a row in the order book data set (i.e. a
* single order in the order book). It can be a bid or an ask order.*/
class OrderBookEntry
{

public:
/*Define constructor function and define data types of the class as arguments
* of the constructor function, additionally insert the initialization list
* (end of the argument parenthesis)*/
OrderBookEntry(double _price,
double _amount,
std::string _timestamp,
std::string _product,
OrderBookType _orderType,
std::string username = "dataset");

/* Converts a string to an OrderBookType enum value. */
static OrderBookType stringToOrderBookType(const std::string &s);

/* Comparison function for sorting OrderBookEntry objects by timestamp in ascending order. */
static bool compareByTimeStamp(OrderBookEntry &e1, OrderBookEntry &e2)
{

return e1.timestamp < e2.timestamp;
}

/* Comparison function for sorting OrderBookEntry objects by price in ascending order. */
static bool compareByPriceAsc(OrderBookEntry &e1, OrderBookEntry &e2)
{

return e1.price < e2.price;
}
```

```
/* Comparison function for sorting OrderBookEntry objects by price in descending order. */
static bool compareByPriceDesc(OrderBookEntry &e1, OrderBookEntry &e2)
{

return e1.price > e2.price;
}

/*Data members*/
double price;
double amount;
std::string timestamp;
std::string product;
OrderBookType orderType;
std::string username;
};
```

Wallet.cpp

```cpp
#include "Wallet.h"

Wallet::Wallet()
{
// Default constructor
}

/** Inserts currency into the wallet.
* @param type The type of currency to insert.
* @param amount The amount of currency to insert.
*/
void Wallet::insertCurrency(std::string type, double amount)
{
double balance;

if (amount < 0)
{
throw std::exception{};
}

// Check if the currency already exists in the wallet
if (currencies.count(type) == 0)
{
balance = 0;
}
else
{
balance = currencies[type];
}

// Update the balance by adding the amount
balance += amount;

// Store the updated balance in the wallet
currencies[type] = balance;
}

/** Removes currency from the wallet.
* @param type The type of currency to remove.
* @param amount The amount of currency to remove.
* @return True if the currency was successfully removed, false otherwise.
*/
bool Wallet::removeCurrency(std::string type, double amount)
{
if (amount < 0)
```

```
{
return false;
}

// Check if the currency exists in the wallet
if (currencies.count(type) == 0)
{
return false;
}
else
{
// Check if the wallet contains enough currency to remove
if (containsCurrency(type, amount))
{
currencies[type] -= amount;
return true;
}
else
{
return false; // Not enough currency in the wallet
}
}
}

/** Checks if the wallet contains a specific amount of currency or more.
* @param type The type of currency to check.
* @param amount The amount of currency to check.
* @return True if the wallet contains the specified amount of currency or more, false otherwise.
*/
bool Wallet::containsCurrency(std::string type, double amount)
{
if (currencies.count(type) == 0)
{
return false; // Currency does not exist in the wallet
}
else
{
return currencies[type] >= amount; // Check if the balance is greater than or equal to the requested
amount
}
}

/** Checks if the wallet can fulfill an order.
* @param order The OrderBookEntry representing the order to be fulfilled.
* @return True if the wallet can fulfill the order, false otherwise.
*/
bool Wallet::canFulfillOrder(OrderBookEntry order)
{
std::vector<std::string> currs = CSVReader::tokenise(order.product, '/');
```

```
// Check if the order is an ask
if (order.orderType == OrderBookType::ask)
{
double amount = order.amount;
std::string currency = currs[0];

// Check if the wallet contains enough currency to fulfill the ask order
return containsCurrency(currency, amount);
}

// Check if the order is a bid
if (order.orderType == OrderBookType::bid)
{
double amount = order.amount * order.price;
std::string currency = currs[1];

// Check if the wallet contains enough currency to fulfill the bid order
return containsCurrency(currency, amount);
}

return false;
}

/** Updates the contents of the wallet after a sale.
* @param sale The OrderBookEntry representing the sale.
*/
void Wallet::processSale(OrderBookEntry &sale)
{
std::vector<std::string> currs = CSVReader::tokenise(sale.product, '/');

// Check if the sale is an ask sale
if (sale.orderType == OrderBookType::asksale)
{
double outgoingAmount = sale.amount;
std::string outgoingCurrency = currs[0];

double incomingAmount = sale.amount * sale.price;
std::string incomingCurrency = currs[1];

// Update the wallet balances
currencies[incomingCurrency] += incomingAmount;
currencies[outgoingCurrency] -= outgoingAmount;
}

// Check if the sale is a bid sale
if (sale.orderType == OrderBookType::bidsale)
{
double incomingAmount = sale.amount;
std::string incomingCurrency = currs[0];
```

```
double outgoingAmount = sale.amount * sale.price;
std::string outgoingCurrency = currs[1];

// Update the wallet balances
currencies[incomingCurrency] += incomingAmount;
currencies[outgoingCurrency] -= outgoingAmount;
}
}

/** Generates a string representation of the wallet, showing the amount of each currency.
* @return The string representation of the wallet.
*/
std::string Wallet::toString()
{
std::string s;

// Iterate over each currency in the wallet
for (std::pair<std::string, double> pair : currencies)
{
std::string currency = pair.first;
double amount = pair.second;

// Append the currency and its amount to the string
s += currency + " : " + std::to_string(amount) + "\n";
}

return s;
}
```

Wallet.h

```
#pragma once

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

class Wallet
{
public:
Wallet();

/**
* Insert currency to the wallet.
*
* @param type The type of currency to insert.
* @param amount The amount of currency to insert.
*/
void insertCurrency(std::string type, double amount);

/**
* Remove currency from the wallet.
*
* @param type The type of currency to remove.
* @param amount The amount of currency to remove.
* @return True if the removal was successful, false otherwise.
*/
bool removeCurrency(std::string type, double amount);

/**
* Check if the wallet contains at least the specified amount of currency.
*
* @param type The type of currency to check.
* @param amount The minimum amount of currency to check for.
* @return True if the wallet contains the specified amount of currency or more, false otherwise.
*/
bool containsCurrency(std::string type, double amount);

/**
* Check if the wallet has sufficient funds to fulfill the given order.
*
* @param order The order to fulfill.
* @return True if the wallet has sufficient funds, false otherwise.
```

```
*/
bool canFulfillOrder(OrderBookEntry order);

/**
* Process a sale and update the contents of the wallet.
*
* @param sale The sale to process.
*/
void processSale(OrderBookEntry &sale);

/**
* Generate a string representation of the wallet, showing the amount of each type of currency.
*
* @return The string representation of the wallet.
*/
std::string toString();

private:
std::map<std::string, double> currencies;
};
```