"MerkelMain.cpp"

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

// START OF MY CODE

#include "sstream"
#include <iomanip>

// END OF MY CODE

MerkelMain::MerkelMain()
{

}

// START OF MY CODE

/// @brief Creates a bar plot for the trading volume of a product selected by the user
/// @param product
void MerkelMain::plot_volume(std::string product){
    std::vector<double> volumes;
    std::string timeframe = orderBook.getEarliestTime();
    std::vector<std::string> x_axis;
    double min_value, max_value;
    char plot[30][115];
    int i,j,k;

    // Calculates the volume of the product at each timeframe
    do{
        std::vector<OrderBookEntry> prodOrdersAsk = orderBook.getOrders(OrderBookType::ask, product,
timeframe);
        std::vector<OrderBookEntry> prodOrdersBid = orderBook.getOrders(OrderBookType::bid, product, ti
meframe);
        if(prodOrdersAsk.size() > 0 || prodOrdersBid.size() > 0){
            double volume=0;
            for(const auto& order: prodOrdersAsk){
                volume += order.amount;
            }
            for(const auto& order: prodOrdersBid){
                volume += order.amount;
            }
            volumes.push_back(volume);
            x_axis.push_back(timeframe);
        }
        timeframe = orderBook.getNextTime(timeframe);
    }while(timeframe != orderBook.getNextTime(currentTime));

    // Some instructions for creation of y-axis numeric range
    min_value = volumes[0];
    max_value = volumes[0];
```

```
for (i=0; i<volumes.size(); i++){
    if(volumes[i] < min_value){
        min_value = volumes[i];
    }
    if(volumes[i] > max_value){
        max_value = volumes[i];
    }
}

double diff = max_value - min_value;
max_value = max_value + 0.25*diff;
min_value = min_value - 0.25*diff;

double step = (max_value - min_value)/30;
double range[30];

range[0] = max_value;

for(i=1; i<30; i++){
    range[i] = range[i-1] - step;
}

for(i=0; i<30; i++){
    for(j=0; j<115; j++){
        plot[i][j] = ' ';
    }
}

// Takes a batch of trading volumes each iteration and prints them as char array (bar plot)
while(!volumes.empty()){

    int plot_size = 15;
    if(plot_size > volumes.size()){
        plot_size = volumes.size();
    }

    for(k=0; k<plot_size; k++){
        int pos=-1;
        for(i=1; i<30; i++){
            if(range[i-1] >= volumes[k] && volumes[k] > range[i]){
                if(range[i-1] - volumes[k] < volumes[k] - range[i]){
                    pos = i-1;
                }
                else{
                    pos = i;
                }
            }

        }

        for(i=28; i>=pos; i--){
            for(j=10; j<15; j++){
                plot[i][j+7*k] = '|';
            }
```

```
    }

    for(j=10; j<15; j++){
        plot[29][j+7*k] = x_axis[k][j+4];
    }

}

// Inserts numeric range to the left of the plot
for(i=0; i<30; i++){
    if(i==5 || i==11 || i==17 || i==23){
        std::string num = std::to_string(range[i]);
        if(range[i] > 0.001 && range[i] < 1000000){
            for(j=0; j<7; j++){
                plot[i][j] = num[j];
            }
        }
        else{
            std::ostringstream oss;
            oss << std::scientific << std::setprecision(1) << range[i];
            num = oss.str();
            for(j=0; j<7; j++){
                plot[i][j] = num[j];
            }
        }
    }
}

// Inserts the title of the product to the plot
for(j=50; j<50+product.length(); j++){
    plot[0][j] = product[j-50];
}

// Inserts the first timeframe of data in the plot
plot[0][0] = 'F'; plot[0][1] = 'R'; plot[0][2] = 'O'; plot[0][3] = 'M';
for(j=5; j<24; j++){
    plot[0][j] = x_axis[0][j-5];
}

// Inserts the last timeframe of data in the plot
plot[0][93] = 'T'; plot[0][94] = 'O';
for(j=96; j<115; j++){
    plot[0][j] = x_axis[plot_size-1][j-96];
}

// Deletes all contents that are going to be printed from the vectors
for(i=0; i<plot_size; i++){
    volumes.erase(volumes.begin());
    x_axis.erase(x_axis.begin());
}

for(i=0; i<115; i++){
    std::cout << "=";
}
std::cout << std::endl;
```

```
        // Prints final plot
        for(i=0; i<30; i++){
            for(j=0; j<115; j++){
                std::cout << plot[i][j];
            }
            std::cout << std::endl;
        }

        // Clear all contents of the plot array
        for(i=0; i<30; i++){
            for(j=0; j<115; j++){
                plot[i][j] = ' ';
            }
        }

        for(i=0; i<115; i++){
            std::cout << "=";
        }
        std::cout << std::endl;
    }

}

/// @brief the function that searches the vector candleSticks for the candlestick that has as fields its argu
ments
/// @param timeframe
/// @param type
/// @param product
/// @return position integer of candleSticks vector
int MerkelMain::findCandleStick(std::string timeframe, OrderBookType type, std::string product){
    for (int i=0; i<candleSticks.size(); i++){
        if(timeframe == candleSticks[i].getTime() && type == candleSticks[i].getOrderType() && product == c
andleSticks[i].getProduct()){
            return i;
        }
    }
    // returns -1 if no candlestick was found
    return -1;
}


/// @brief prints all candlesticks of the product and type received as arguments in the time interval from th
e first to the current timeframe
/// @param product
/// @param type
void MerkelMain::plot_data(std::vector<Candlestick> plot_candlesticks){
    char plot[30][112];
    double min_value, max_value;

    // Calculate lowest and highest values of all candles
    // (This help to create numeric range that is the same for whole plot)
    min_value = plot_candlesticks[0].getLow();
    max_value = plot_candlesticks[0].getHigh();
    for (auto candlestick : plot_candlesticks){
```

```
    if(candlestick.getLow() < min_value){
        min_value = candlestick.getLow();
    }
    if(candlestick.getHigh() > max_value){
        max_value = candlestick.getHigh();
    }
}

double diff = max_value - min_value;
max_value = max_value + 0.25*diff;
min_value = min_value - 0.25*diff;

// Fixed step for y axis
double step = (max_value - min_value)/30;

double range[30];
range[0] = max_value;
for(int i=1; i<30; i++){
    range[i] = range[i-1] - step;
}

// Initialize plot array with space(empty) character
for(int i=0; i<30; i++){
    for(int j=0; j<112; j++){
        plot[i][j] = ' ';
    }
}

char** result;
int i=0, j=0, k = 0;

for(i=0; i<112; i++){
    std::cout << "=";
}
std::cout << std::endl;

// Loop that prints 5 candlesticks each time till all candlesticks are printed
while(!plot_candlesticks.empty()){

    int plot_size = 5;
    if(plot_size > plot_candlesticks.size()){
        plot_size = plot_candlesticks.size();
    }
    for(k=0; k<plot_size; k++){
        // Calls class method toText to visualise each candlestick
        result = plot_candlesticks[0].toText(range);
        plot_candlesticks.erase(plot_candlesticks.begin());
        for(i=0; i<30; i++){
            for(j=7; j<28; j++){
                plot[i][j+21*k] = result[i][j-7];
            }
            delete[] result[i];
        }
        delete[] result;
```

```
        }

        // Inserts numeric range to the left of the plot
        for(i=0; i<30; i++){
            if(i==5 || i==11 || i==17 || i==23){
                std::string num = std::to_string(range[i]);
                if(range[i] > 0.001){
                    for(j=0; j<7; j++){
                        plot[i][j] = num[j];
                    }
                }
                else{
                    std::ostringstream oss;
                    oss << std::scientific << std::setprecision(1) << range[i];
                    num = oss.str();
                    for(j=0; j<7; j++){
                        plot[i][j] = num[j];
                    }
                }
            }
        }

        // Inserts the title of the product to the plot
        for(int j=50; j<50+plot_candlesticks[0].getProduct().length(); j++){
            plot[0][j] = plot_candlesticks[0].getProduct()[j-50];
        }

        // Prints final plot
        for(i=0; i<30; i++){
            for(j=0; j<112; j++){
                std::cout << plot[i][j];
            }
            std::cout << std::endl;
        }

        // Clear all contents of the plot array
        for(i=0; i<30; i++){
            for(j=0; j<112; j++){
                plot[i][j] = ' ';
            }
        }

        for(i=0; i<112; i++){
            std::cout << "=";
        }
        std::cout << std::endl;
    }

}

/// @brief Each time we move to the next timeframe,
///        we call this function for calculating all the candlesticks of the
///        known products for each of their types.
/// @param timeframe, prevTimeFrame
/// @return all timeframe's candlesticks
```

```
std::vector<Candlestick> MerkelMain::calculateCandlesticks(std::string timeframe, std::string prevTimeFra
me){
    std::vector<Candlestick> currTimeVector;

    // With this loop, two candlesticks are created for each product (bid and ask)
    for (std::string const& p : orderBook.getKnownProducts()){

        // we separate according to type and collect all the orders of the product for the current timeframe
        std::vector<OrderBookEntry> prodOrdersAsk = orderBook.getOrders(OrderBookType::ask, p, curren
tTime);
        std::vector<OrderBookEntry> prodOrdersBid = orderBook.getOrders(OrderBookType::bid, p, current
Time);

        // this case covers the very first timeframe in the dataset where there is no data for the calculation of
the open value of the candlestick
        // the case of not finding an order with these filters is also being checked
        if(prevTimeFrame == ""){
            if(prodOrdersAsk.size() != 0){
                Candlestick candleAsk = {prodOrdersAsk, orderBook.getLowPrice(prodOrdersAsk), orderBook.
getHighPrice(prodOrdersAsk), prodOrdersAsk[0].price};
                currTimeVector.push_back(candleAsk);
            }
            if(prodOrdersBid.size() != 0){
                Candlestick candleBid = {prodOrdersBid, orderBook.getLowPrice(prodOrdersBid), orderBook.g
etHighPrice(prodOrdersBid), prodOrdersBid[0].price};
                currTimeVector.push_back(candleBid);
            }

        }
        // this case covers all the other timeframes where there is data for the calculation of the open value o
f the candlestick
        // the case of not finding an order with these filters is also being checked
        else{
            double open;
            if(prodOrdersAsk.size() != 0){
                int pos = findCandleStick(prevTimeFrame, prodOrdersAsk[0].orderType, prodOrdersAsk[0].prod
uct);
                open = candleSticks[pos].getClose();
                Candlestick candleAsk = {prodOrdersAsk, orderBook.getLowPrice(prodOrdersAsk), orderBook.
getHighPrice(prodOrdersAsk), open};
                currTimeVector.push_back(candleAsk);
            }
            if(prodOrdersBid.size() != 0){
                int pos = findCandleStick(prevTimeFrame, prodOrdersBid[0].orderType, prodOrdersBid[0].prod
uct);
                open = candleSticks[pos].getClose();
                Candlestick candleBid = {prodOrdersBid, orderBook.getLowPrice(prodOrdersBid), orderBook.g
etHighPrice(prodOrdersBid), open};
                currTimeVector.push_back(candleBid);
            }

        }
    }

    return currTimeVector;
```

```
}

// END OF MY CODE

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


    // START OF MY CODE
    // Calculate and store candlesticks of the very first timeframe
    std::vector<Candlestick> subvector = calculateCandlesticks(currentTime, "");
    candleSticks.insert(candleSticks.end(), subvector.begin(), subvector.end());
    // END OF MY CODE

    wallet.insertCurrency("BTC", 10);

    while(true)
    {
        printMenu();
        input = getUserOption(8);
        processUserOption(input);
    }
}


void MerkelMain::printMenu()
{
    // 1 print help
    std::cout << "1: Print help " << std::endl;
    // 2 print exchange stats
    std::cout << "2: Print exchange stats" << std::endl;
    // 3 make an offer
    std::cout << "3: Make an offer " << std::endl;
    // 4 make a bid
    std::cout << "4: Make a bid " << std::endl;
    // 5 print wallet
    std::cout << "5: Print wallet " << std::endl;
    // 6 continue
    std::cout << "6: Continue " << std::endl;

    // START OF MY CODE
    // 7 plot candlestick data of a product
    std::cout << "7: Plot exchange data " << std::endl;
    // 8 plot trading volume of a product
    std::cout << "8: Plot trading volume " << std::endl;
    // END OF MY CODE

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

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

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;



    }
    // std::cout << "OrderBook contains :  " << orders.size() << " entries" << std::endl;
    // unsigned int bids = 0;
    // unsigned int asks = 0;
    // for (OrderBookEntry& e : orders)
    // {
    //     if (e.orderType == OrderBookType::ask)
    //     {
    //         asks ++;
    //     }
    //     if (e.orderType == OrderBookType::bid)
    //     {
    //         bids ++;
    //     }
    // }
    // std::cout << "OrderBook asks:  " << asks << " bids:" << bids << std::endl;

}

void MerkelMain::enterAsk()
{
    std::cout << "Make an ask - enter the amount: product,price, amount, eg  ETH/BTC,200,0.5" << std::en
dl;
    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::en
dl;
   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;
       }
   }
```

```
}

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

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);
            }
        }

    }

    // START OF MY CODE

    std::string prevTimeFrame = currentTime;

    // END OF MY CODE

    currentTime = orderBook.getNextTime(currentTime);

    // START OF MY CODE

    //Clear all stored candlesticks at the beginning of each data iteration
    if(currentTime == orderBook.getEarliestTime()){
        prevTimeFrame = "";
        candleSticks.clear();
    }
    // Calculate and store candlesticks of the next timeframe
    std::vector<Candlestick> subvector = calculateCandlesticks(currentTime, prevTimeFrame);
    candleSticks.insert(candleSticks.end(), subvector.begin(), subvector.end());

    // END OF MY CODE
}

//Function for printing option dialog
//Takes variable number of option as an argument

// START OF MY CODE
int MerkelMain::getUserOption(int num_options)
{
```

```
    int userOption = 0;
    std::string line;
    std::cout << "Type in 1-" << num_options << std::endl;
// END OF MY CODE
    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)
{
    if (userOption == 0) // bad input
    {
        std::cout << "Invalid choice. Choose 1-8" << std::endl;
    }
    if (userOption == 1)
    {
        printHelp();
    }
    if (userOption == 2)
    {
        printMarketStats();
    }
    if (userOption == 3)
    {
        enterAsk();
    }
    if (userOption == 4)
    {
        enterBid();
    }
    if (userOption == 5)
    {
        printWallet();
    }
    if (userOption == 6)
    {
        gotoNextTimeframe();
    }
    // START OF MY CODE

    if (userOption == 7){
        std::vector<std::string> products = orderBook.getKnownProducts();
        int i=1;

        std::cout << "============== " << std::endl;
        // Prints all products
        for(const auto& product: products){
            std::cout << i << ": "<< product << std::endl;
```

```
            i++;
        }
        std::cout << "============== " << std::endl;
        // Stores user's product selection
        int prod_choice = getUserOption(i-1);

        // Prints both order types
        std::cout << "============== " << std::endl;
        std::cout << "1: Bid orders" << std::endl;
        std::cout << "2: Ask orders" << std::endl;
        std::cout << "============== " << std::endl;

        // Stores user's product type
        int type_choice = getUserOption(2);

        OrderBookType type;

        if(type_choice == 1){
            type = OrderBookType::bid;
        }
        else if(type_choice == 2){
            type = OrderBookType::ask;
        }

        std::vector<Candlestick> plot_candlesticks;
        std::string timeframe = orderBook.getEarliestTime();
        // Collects all product's candlesticks till current timeframe
        do{
            int pos = findCandleStick(timeframe, type, products[prod_choice-1]);
            if(pos >= 0){
                plot_candlesticks.push_back(candleSticks[pos]);
            }
            timeframe = orderBook.getNextTime(timeframe);
        }while(timeframe != orderBook.getNextTime(currentTime));
        // Calls plot_data function passing all user's selections
        plot_data(plot_candlesticks);
    }
    if(userOption==8){
        std::vector<std::string> products = orderBook.getKnownProducts();
        int i=1;

        std::cout << "============== " << std::endl;
        // Prints all products
        for(const auto& product: products){
            std::cout << i << ": "<< product << std::endl;
            i++;
        }
        std::cout << "============== " << std::endl;
        // Stores user's product selection
        int prod_choice = getUserOption(i-1);
        plot_volume(products[prod_choice-1]);
    }

    // END OF MY CODE
}
```

"MerkelMain.h"

```
#pragma once

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

class MerkelMain
{
    public:
        MerkelMain();
        /** Call this to start the sim */
        void init();
    private:
        // START OF MY CODE

        int findCandleStick(std::string timeframe, OrderBookType type, std::string product);
        std::vector<Candlestick> calculateCandlesticks(std::string timeframe, std::string prevTimeFrame);
        void plot_data(std::vector<Candlestick> plot_candlesticks);
        void plot_volume(std::string product);

        // END OF MY CODE

        void printMenu();
        void printHelp();
        void printMarketStats();
        void enterAsk();
        void enterBid();
        void printWallet();
        void gotoNextTimeframe();
        int getUserOption(int num_options);
        void processUserOption(int userOption);

        std::string currentTime;

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

        // START OF MY CODE

        std::vector<Candlestick> candleSticks;

        // END OF MY CODE
};
```

"Candlestick.cpp"

```
// START OF MY CODE

#include <iostream>
#include <vector>
#include "Candlestick.h"
#include "OrderBook.h"
#include "OrderBookEntry.h"

Candlestick::Candlestick(){

}

// Main constructor that initializes all object fields and calculates close price
Candlestick::Candlestick(std::vector<OrderBookEntry> orders, double _low, double _high, double _open):
 low(_low), high(_high), open(_open){
    timeframe = orders[0].timestamp;
    orderType = orders[0].orderType;
    product = orders[0].product;

    double total_amount = 0;
    double total_value = 0;

    for (const auto& order : orders){
        total_amount = total_amount + order.amount;
        total_value = total_value + (order.price * order.amount);
    }

    close = total_value / total_amount;
}
std::string Candlestick::getTime(){
    return this->timeframe;
}

std::string Candlestick::getProduct(){
    return this->product;
}

OrderBookType Candlestick::getOrderType(){
    return this->orderType;
}

double Candlestick::getClose(){
    return this->close;
}

double Candlestick::getLow(){
    return this->low;
}
double Candlestick::getHigh(){
    return this->high;
}
double Candlestick::getOpen(){
    return this->open;
}
```

```
// Candlestick-to-ASCII function
char** Candlestick::toText(double* range){
    char** text_candlestick = new char*[30];
    int low_index, high_index, open_index, close_index;

    // Initializes array
    for(int i=0; i<30; i++){
        text_candlestick[i] = new char[21];
        for(int j=0; j<21; j++){
            text_candlestick[i][j] = ' ';
        }
    }


    // find the correspondence of the candlestick low, high, open, close values with the numeric range of th
e final plot
    for(int i=1; i<30; i++){
        if(range[i-1] >= this->high && this->high > range[i]){
            if(range[i-1] - this->high < this->high - range[i]){
                high_index = i-1;
            }
            else{
                high_index = i;
            }
        }
        if(range[i-1] >= this->low && this->low > range[i]){
            if(range[i-1] - this->low < this->low - range[i]){
                low_index = i-1;
            }
            else{
                low_index = i;
            }
        }
        if(range[i-1] >= this->open && this->open > range[i]){
            if(range[i-1] - this->open < this->open - range[i]){
                open_index = i-1;
            }
            else{
                open_index = i;
            }
        }
        if(range[i-1] >= this->close && this->close > range[i]){
            if(range[i-1] - this->close < this->close - range[i]){
                close_index = i-1;
            }
            else{
                close_index = i;
            }
        }
    }

    // open > close visualisation case
    if(open > close){
        for(int i=high_index; i<open_index; i++){
```

```
            text_candlestick[i][10] = '|';
        }
        for(int i=close_index; i<=low_index; i++){
            text_candlestick[i][10] = '|';
        }
        for(int i=open_index; i<=close_index; i++){
            text_candlestick[i][16] = '|';
            text_candlestick[i][4] = '|';
        }
        for(int i=open_index; i<close_index+1; i++){
            for(int j=5; j<16;  j++){
                text_candlestick[i][j] = 'v';
            }
        }
    }
    // open < close visualisation case
    else{
        for(int i=high_index; i<close_index; i++){
            text_candlestick[i][10] = '|';
        }
        for(int i=open_index; i<=low_index; i++){
            text_candlestick[i][10] = '|';
        }
        for(int i=close_index; i<=open_index; i++){
            text_candlestick[i][16] = '|';
            text_candlestick[i][4] = '|';
        }
        for(int i=close_index; i<open_index+1; i++){
            for(int j=5; j<16;  j++){
                text_candlestick[i][j] = '^';
            }
        }
    }

    // Time value in X-axis
    int k=11;
    for(int j=7; j<15; j++){
        text_candlestick[29][j] = timeframe[k];
        k++;
    }

    return text_candlestick;
}

// END OF MY CODE


"Candlestick.h"

// START OF MY CODE
#include <string>
#include "OrderBookEntry.h"

class Candlestick{
    public:
```

```
        // Default constructor
        Candlestick();
        // Initialization constructor
        Candlestick(std::vector<OrderBookEntry> orders, double _low, double _high, double _open);

        // Getter functions of the members
        std::string getTime();
        std::string getProduct();
        OrderBookType getOrderType();
        double getClose();
        double getLow();
        double getHigh();
        double getOpen();

        // Converts object to ascii representation
        char** toText(double* range);
    private:
        std::string timeframe;
        std::string product;
        OrderBookType orderType;
        double low;
        double high;
        double open;
        double close;
};

// END OF MY CODE
```

IN OTHER FILES OF ZIP FOLDER I DID NO CHANGES