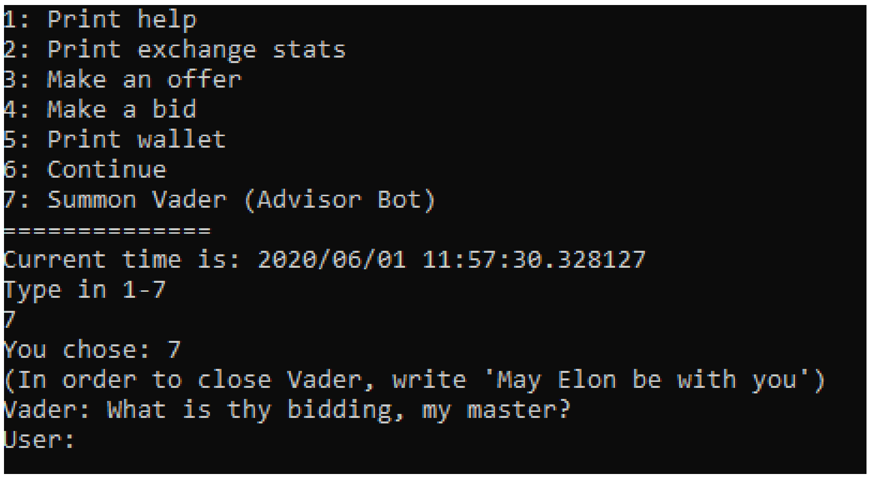
INTRODUCTION

General Information

In this report, I have introduced you to the usage and analysis of my advisor bot for the MerkelRex program.

The advisor bot is called ‘Vader’. It is called by selecting ‘7’ from the main menu, where the program will enter the ‘advisor bot mode’.



```
1: Print help
2: Print exchange stats
3: Make an offer
4: Make a bid
5: Print wallet
6: Continue
7: Summon Vader (Advisor Bot)
==============
Current time is: 2020/06/01 11:57:30.328127
Type in 1-7
7
You chose: 7
(In order to close Vader, write 'May Elon be with you')
Vader: What is thy bidding, my master?
User:
```

After starting the ‘advisor bot mode’, the user can enter several commands to execute different functions. The bot is able to execute 10 different commands in total. 9 of them are the ones from the ‘Task 1’ of the midterm, and the last one is my own command, as ‘Task 2’ of the midterm demands. Below, the table for all the commands are presented.

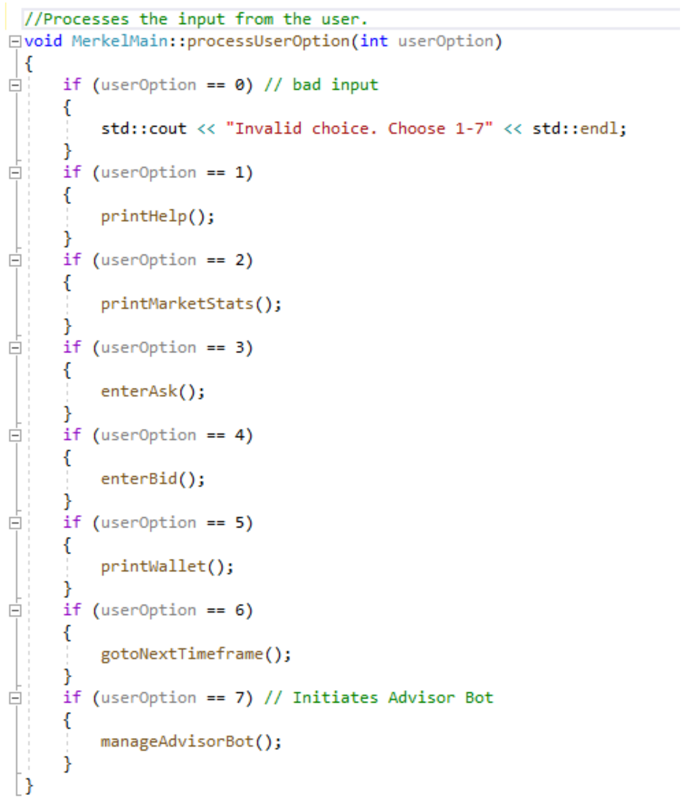
help	implemented
help avg	implemented
prod	implemented
min	implemented
max	implemented
avg	implemented
predict	implemented
time	implemented
step	implemented
change (my own command)	implemented

All of the commands are called by user input. How to process and call the commands are explained under the following two sections, “‘manageAdvisorBot’ Function” and “‘inputProcess’ Function”. General information about the commands given under the “Commands” section.

In order to close the advisor bot, the user needs to write ‘May Elon be with you’. Then, the program will return to the main menu.

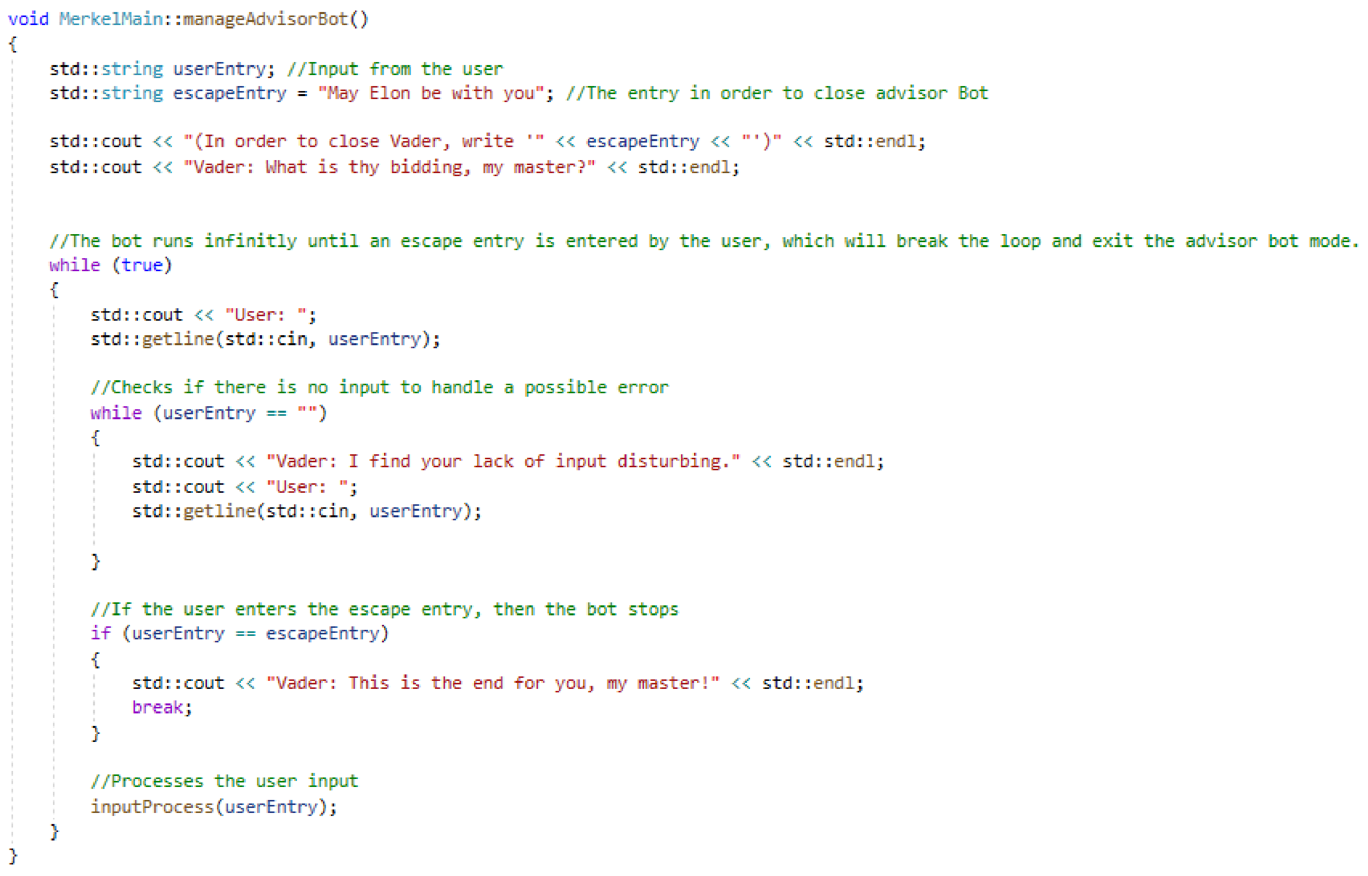
“manageAdvisorBot” FUNCTION

As stated before, the advisor bot mode is entered by selecting ‘7’ from the main menu, where MerkelMain::manageAdvisorBot function will be called.



```cpp
//Processes the input from the user.
void MerkelMain::processUserOption(int userOption)
{
    if (userOption == 0) // bad input
    {
        std::cout << "Invalid choice. Choose 1-7" << 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();
    }
    if (userOption == 7) // Initiates Advisor Bot
    {
        manageAdvisorBot();
    }
}
```

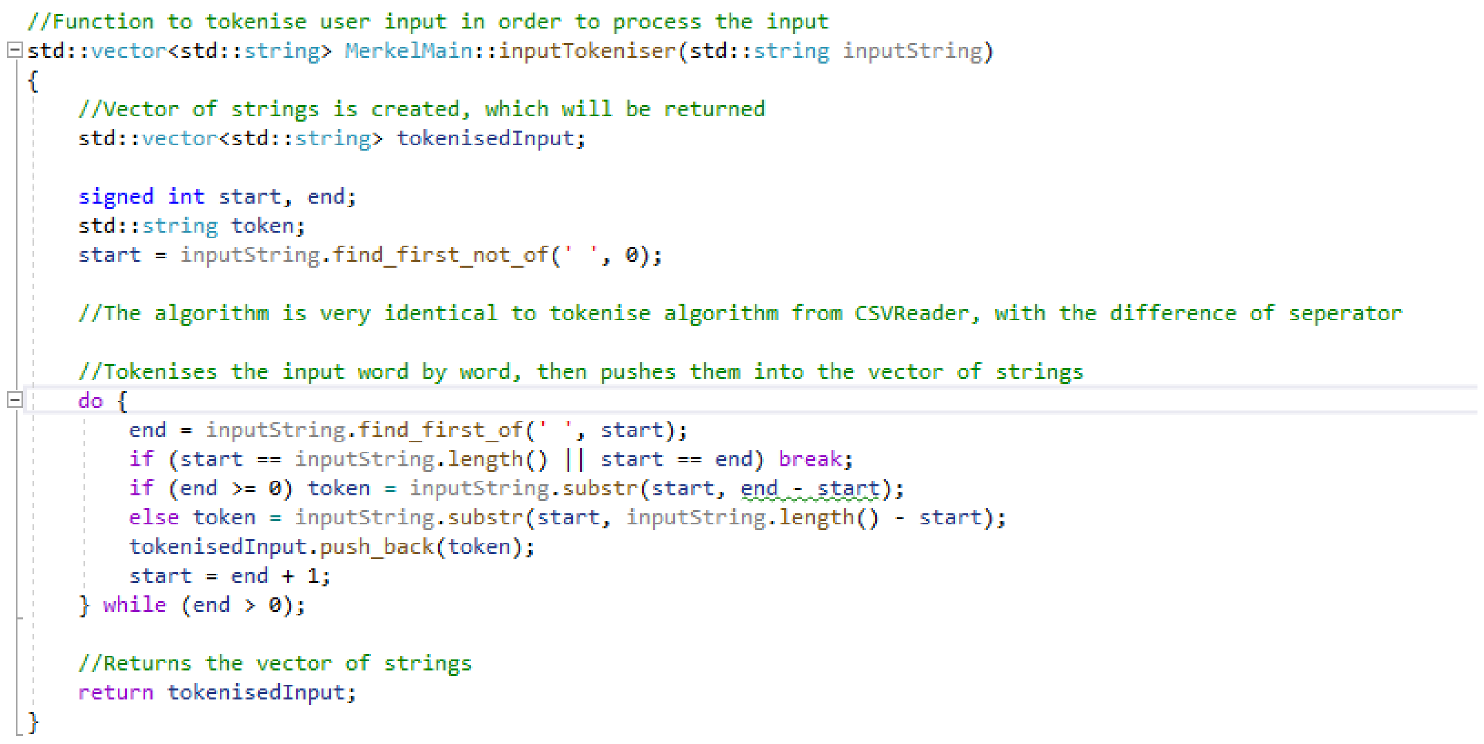
MerkelMain::manageAdvisorBot function is the main function to handle the advisor bot. It asks for the user to enter an input, then either sends this input to MerkelMain::inputProcess function or closes the advisor bot mode. The function also checks for a NULL input. The algorithm for MerkelMain::manageAdvisorBot is presented below:



```cpp
void MerkelMain::manageAdvisorBot()
{
    std::string userEntry; //Input from the user
    std::string escapeEntry = "May Elon be with you"; //The entry in order to close advisor Bot

    std::cout << "(In order to close Vader, write '" << escapeEntry << "')" << std::endl;
    std::cout << "Vader: What is thy bidding, my master?" << std::endl;


    //The bot runs infinitly until an escape entry is entered by the user, which will break the loop and exit the advisor bot mode.
    while (true)
    {
        std::cout << "User: ";
        std::getline(std::cin, userEntry);

        //Checks if there is no input to handle a possible error
        while (userEntry == "")
        {
            std::cout << "Vader: I find your lack of input disturbing." << std::endl;
            std::cout << "User: ";
            std::getline(std::cin, userEntry);

        }

        //If the user enters the escape entry, then the bot stops
        if (userEntry == escapeEntry)
        {
            std::cout << "Vader: This is the end for you, my master!" << std::endl;
            break;
        }

        //Processes the user input
        inputProcess(userEntry);
    }
}
```

“InputProcess” FUNCTION

The purpose of the MerkelMain::inputProcess function is to parse the user input in order to be able to understand the user commands and then call the command functions. It first parses the user command input by MerkelMain::inputTokeniser function. inputTokeniser function parses the user entry into words and returns a vector of strings. The algorithm is almost identical with the CSVReader::tokenise function, except that the separator is blank spaces for inputTokeniser. The algorithm of the inputTokeniser function is given below:

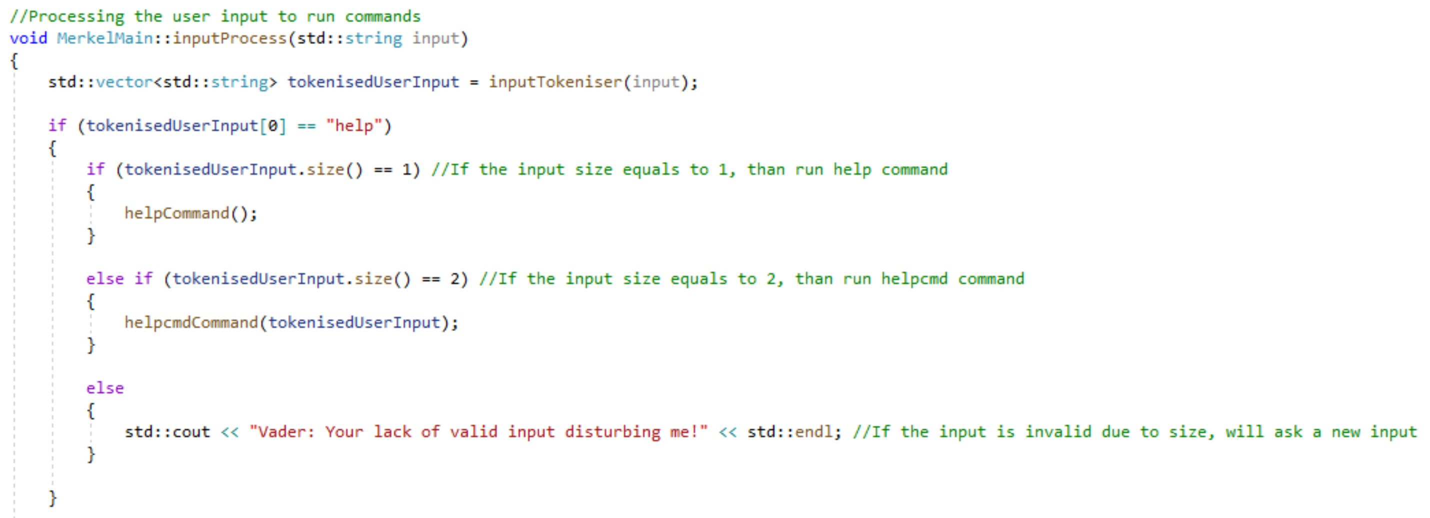


```
//Function to tokenise user input in order to process the input
std::vector<std::string> MerkelMain::inputTokeniser(std::string inputString)
{
    //Vector of strings is created, which will be returned
    std::vector<std::string> tokenisedInput;

    signed int start, end;
    std::string token;
    start = inputString.find_first_not_of(' ', 0);

    //The algorithm is very identical to tokenise algorithm from CSVReader, with the difference of seperator

    //Tokenises the input word by word, then pushes them into the vector of strings
    do {
        end = inputString.find_first_of(' ', start);
        if (start == inputString.length() || start == end) break;
        if (end >= 0) token = inputString.substr(start, end - start);
        else token = inputString.substr(start, inputString.length() - start);
        tokenisedInput.push_back(token);
        start = end + 1;
    } while (end > 0);

    //Returns the vector of strings
    return tokenisedInput;
}
```

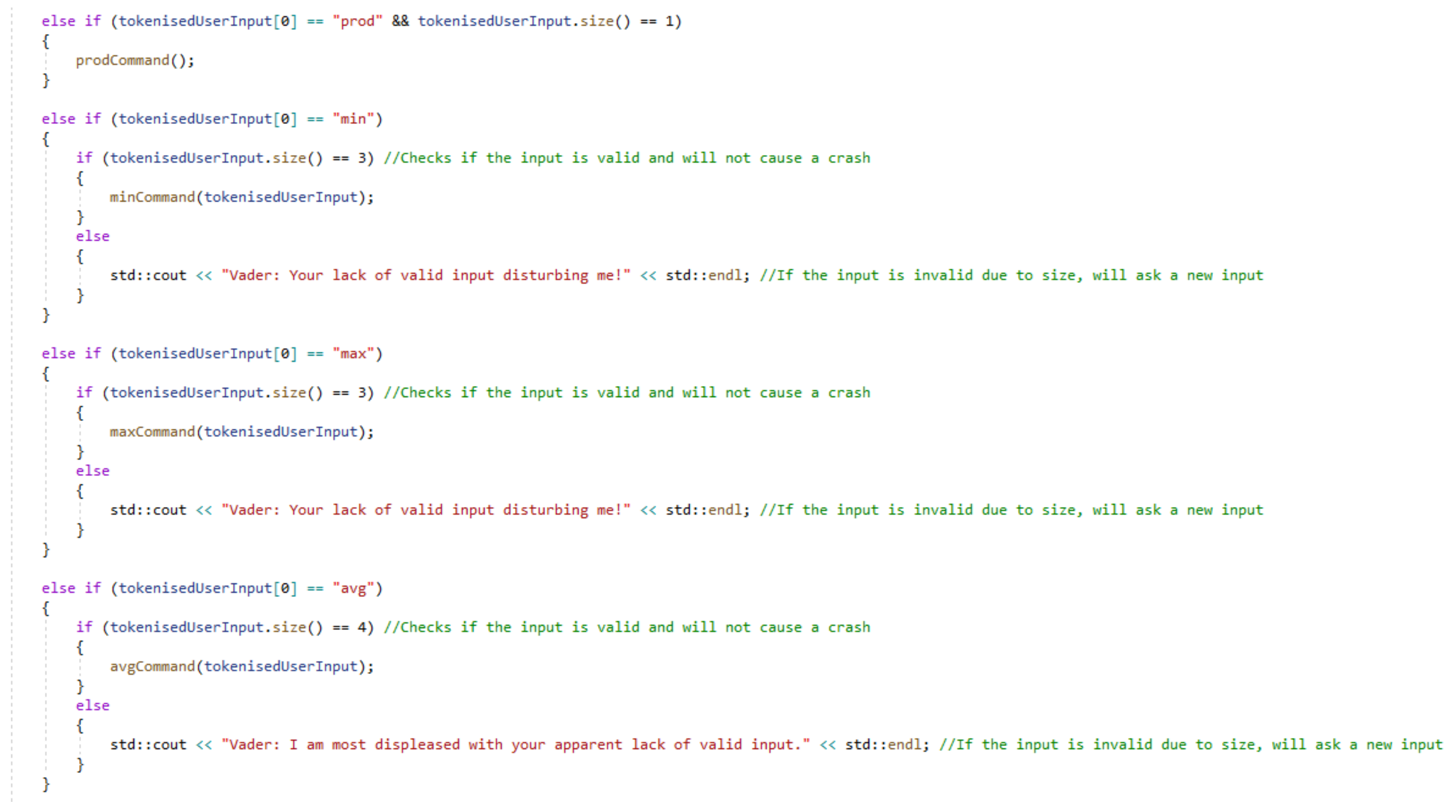
After the parsing, inputProcess function looks for the first element of the tokenized input vector, where the name of the command is expected. If the first element of the vector matches with one of the implemented commands, then it checks the size of the vector, in order to look if the input size matches with the command’s expected input size (for example, ‘prod’ command expects 1 word, which is ‘prod’, while avg command expects 4 different words). If either the command name or the expected sizes do not match, it prompts an error message and asks for another input.

If everything is ok, then inputProcess calls the commands own function. Further checks for invalid inputs are made from these functions, as the validity and the invalidity of the input changes from one command to another.

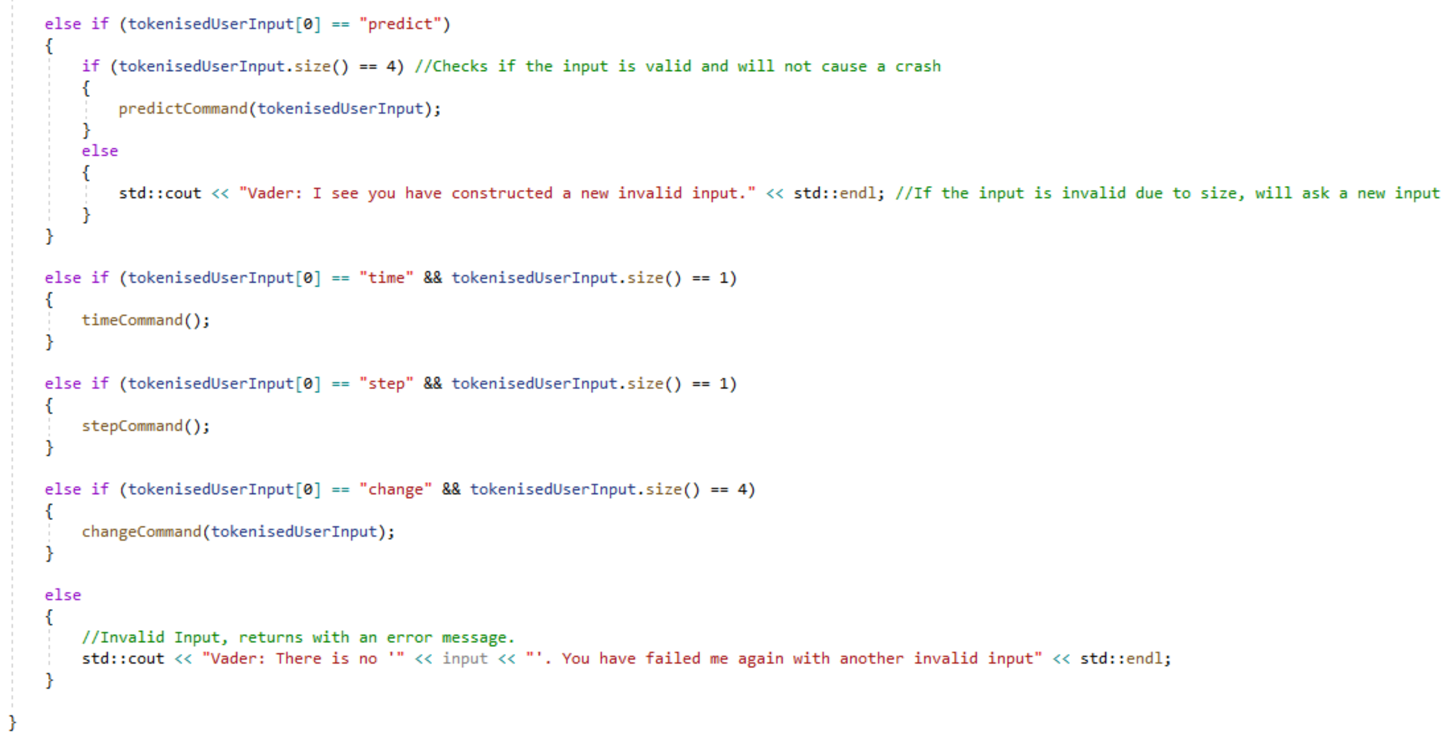
The algorithm for inputProcess function is shown at the next page.



```cpp
//Processing the user input to run commands
void MerkelMain::inputProcess(std::string input)
{
    std::vector<std::string> tokenisedUserInput = inputTokeniser(input);

    if (tokenisedUserInput[0] == "help")
    {
        if (tokenisedUserInput.size() == 1) //If the input size equals to 1, than run help command
        {
            helpCommand();
        }

        else if (tokenisedUserInput.size() == 2) //If the input size equals to 2, than run helpcmd command
        {
            helpcmdCommand(tokenisedUserInput);
        }

        else
        {
            std::cout << "Vader: Your lack of valid input disturbing me!" << std::endl; //If the input is invalid due to size, will ask a new input
        }

    }
```



```cpp
    else if (tokenisedUserInput[0] == "prod" && tokenisedUserInput.size() == 1)
    {
        prodCommand();
    }

    else if (tokenisedUserInput[0] == "min")
    {
        if (tokenisedUserInput.size() == 3) //Checks if the input is valid and will not cause a crash
        {
            minCommand(tokenisedUserInput);
        }
        else
        {
            std::cout << "Vader: Your lack of valid input disturbing me!" << std::endl; //If the input is invalid due to size, will ask a new input
        }
    }

    else if (tokenisedUserInput[0] == "max")
    {
        if (tokenisedUserInput.size() == 3) //Checks if the input is valid and will not cause a crash
        {
            maxCommand(tokenisedUserInput);
        }
        else
        {
            std::cout << "Vader: Your lack of valid input disturbing me!" << std::endl; //If the input is invalid due to size, will ask a new input
        }
    }

    else if (tokenisedUserInput[0] == "avg")
    {
        if (tokenisedUserInput.size() == 4) //Checks if the input is valid and will not cause a crash
        {
            avgCommand(tokenisedUserInput);
        }
        else
        {
            std::cout << "Vader: I am most displeased with your apparent lack of valid input." << std::endl; //If the input is invalid due to size, will ask a new input
        }
    }
```



```cpp
    else if (tokenisedUserInput[0] == "predict")
    {
        if (tokenisedUserInput.size() == 4) //Checks if the input is valid and will not cause a crash
        {
            predictCommand(tokenisedUserInput);
        }
        else
        {
            std::cout << "Vader: I see you have constructed a new invalid input." << std::endl; //If the input is invalid due to size, will ask a new input
        }
    }

    else if (tokenisedUserInput[0] == "time" && tokenisedUserInput.size() == 1)
    {
        timeCommand();
    }

    else if (tokenisedUserInput[0] == "step" && tokenisedUserInput.size() == 1)
    {
        stepCommand();
    }

    else if (tokenisedUserInput[0] == "change" && tokenisedUserInput.size() == 4)
    {
        changeCommand(tokenisedUserInput);
    }

    else
    {
        //Invalid Input, returns with an error message.
        std::cout << "Vader: There is no '" << input << "'. You have failed me again with another invalid input" << std::endl;
    }

}
```

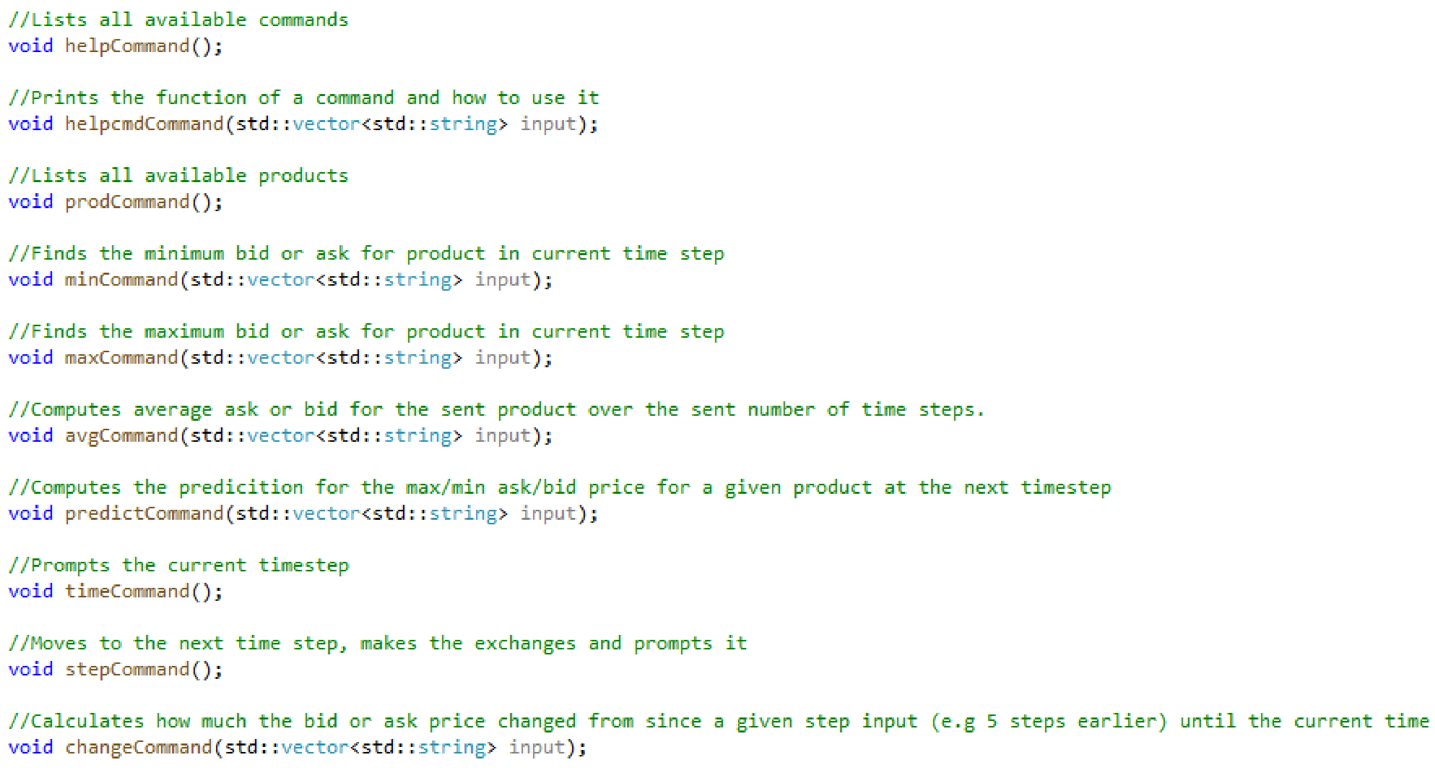
COMMAND FUNCTIONS

GENERAL INFORMATION ON COMMAND FUNCTIONS

The command functions are the main functions which handle the necessary operations for commands to work. In order to do that, they mainly call the more specialized functions from the OrderBook class. They also check (if necessary) for the validity of the inputs, and return with an error message if the input is invalid.

One thing also should be noted that, as all the command functions are functions of MerkelMain class, they have access to the current order book which is called in MerkelMain.h. This results that all of the functions have access to the order book directly.

The command functions and their usages are shown below:



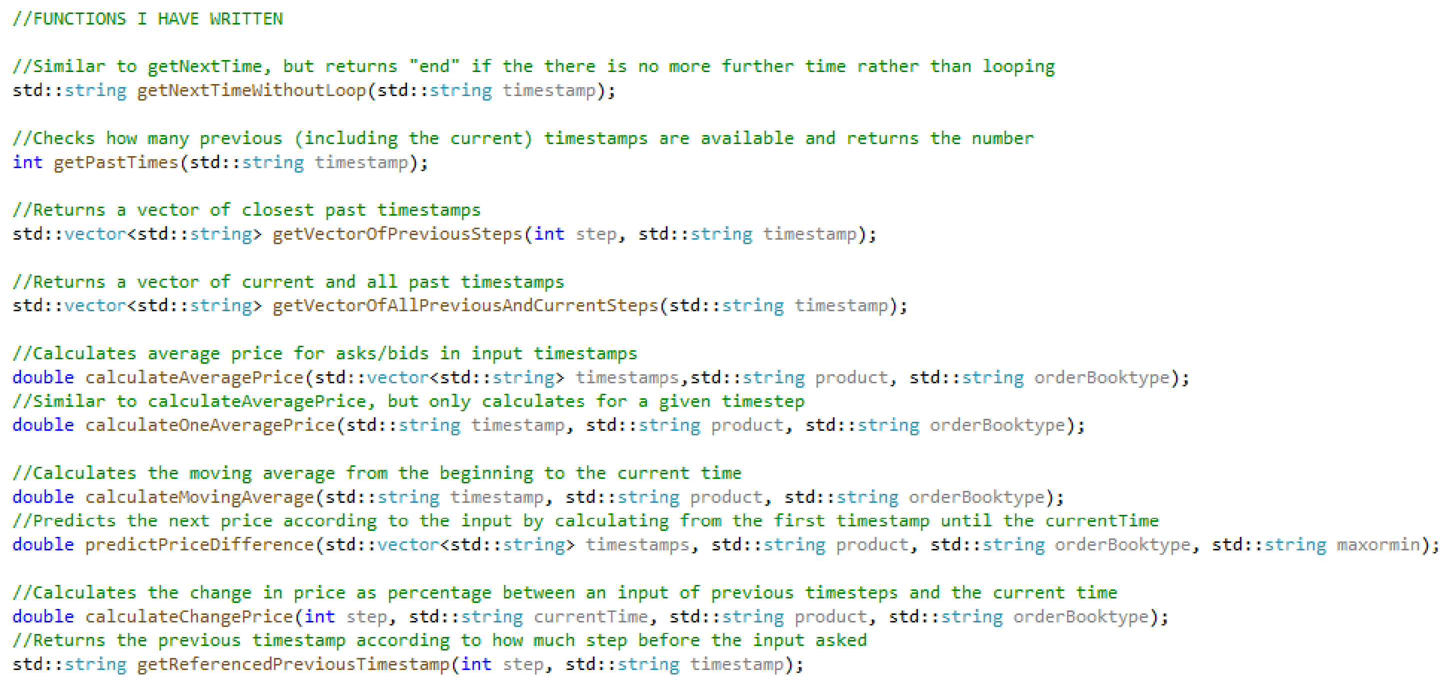
```
//Lists all available commands
void helpCommand();

//Prints the function of a command and how to use it
void helpcmdCommand(std::vector<std::string> input);

//Lists all available products
void prodCommand();

//Finds the minimum bid or ask for product in current time step
void minCommand(std::vector<std::string> input);

//Finds the maximum bid or ask for product in current time step
void maxCommand(std::vector<std::string> input);

//Computes average ask or bid for the sent product over the sent number of time steps.
void avgCommand(std::vector<std::string> input);

//Computes the predicition for the max/min ask/bid price for a given product at the next timestep
void predictCommand(std::vector<std::string> input);

//Prompts the current timestep
void timeCommand();

//Moves to the next time step, makes the exchanges and prompts it
void stepCommand();

//Calculates how much the bid or ask price changed from since a given step input (e.g 5 steps earlier) until the current time
void changeCommand(std::vector<std::string> input);
```

9 of the command functions are the ones from the midterm task. The last one, changeCommand function is the custom function I have implemented as Task 2 from the midterm demands. Their purpose and how to call them are shown at the next page, directly from the pictures of “help avg” command.

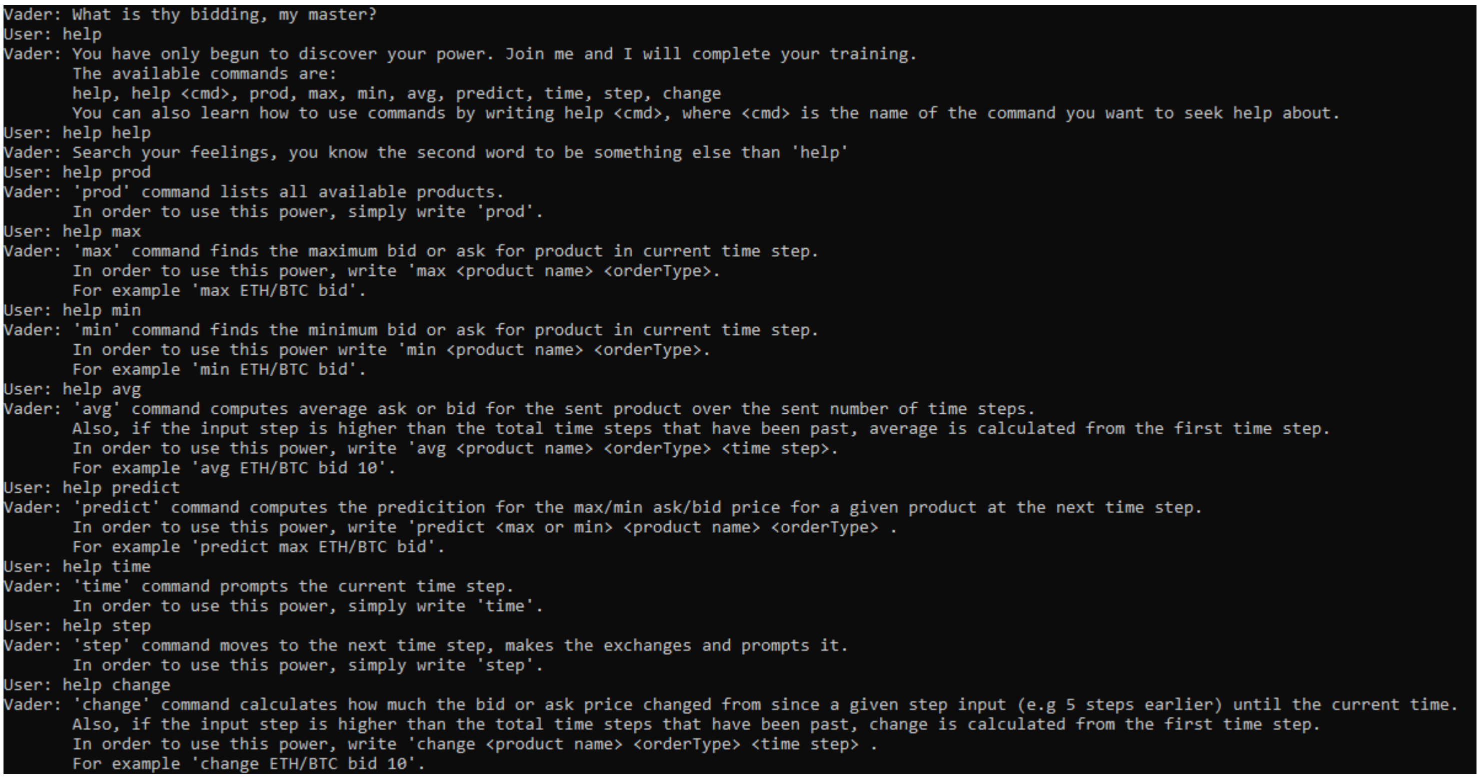
Change command functions heavily use other functions from different libraries and classes. OrderBook class functions are especially used. Some of these functions are the ones which are already written in the starter code. Many others are the ones I have written myself.

The list of the OrderBook functions I have written is shown at the next page.

An example of input validity checkers is also demonstrated under the “changeCommand function” section, where I explained how it works in detail. All the other command functions check the validity of the user input in a similar way but customized to the function.



```cpp
//FUNCTIONS I HAVE WRITTEN

//Similar to getNextTime, but returns "end" if the there is no more further time rather than looping
std::string getNextTimeWithoutLoop(std::string timestamp);

//Checks how many previous (including the current) timestamps are available and returns the number
int getPastTimes(std::string timestamp);

//Returns a vector of closest past timestamps
std::vector<std::string> getVectorOfPreviousSteps(int step, std::string timestamp);

//Returns a vector of current and all past timestamps
std::vector<std::string> getVectorOfAllPreviousAndCurrentSteps(std::string timestamp);

//Calculates average price for asks/bids in input timestamps
double calculateAveragePrice(std::vector<std::string> timestamps,std::string product, std::string orderBooktype);
//Similar to calculateAveragePrice, but only calculates for a given timestep
double calculateOneAveragePrice(std::string timestamp, std::string product, std::string orderBooktype);

//Calculates the moving average from the beginning to the current time
double calculateMovingAverage(std::string timestamp, std::string product, std::string orderBooktype);
//Predicts the next price according to the input by calculating from the first timestamp until the currentTime
double predictPriceDifference(std::vector<std::string> timestamps, std::string product, std::string orderBooktype, std::string maxormin);

//Calculates the change in price as percentage between an input of previous timesteps and the current time
double calculateChangePrice(int step, std::string currentTime, std::string product, std::string orderBooktype);
//Returns the previous timestamp according to how much step before the input asked
std::string getReferencedPreviousTimestamp(int step, std::string timestamp);
```



```
Vader: What is thy bidding, my master?
User: help
Vader: You have only begun to discover your power. Join me and I will complete your training.
       The available commands are:
       help, help <cmd>, prod, max, min, avg, predict, time, step, change
       You can also learn how to use commands by writing help <cmd>, where <cmd> is the name of the command you want to seek help about.
User: help help
Vader: Search your feelings, you know the second word to be something else than 'help'
User: help prod
Vader: 'prod' command lists all available products.
       In order to use this power, simply write 'prod'.
User: help max
Vader: 'max' command finds the maximum bid or ask for product in current time step.
       In order to use this power, write 'max <product name> <orderType>.
       For example 'max ETH/BTC bid'.
User: help min
Vader: 'min' command finds the minimum bid or ask for product in current time step.
       In order to use this power write 'min <product name> <orderType>.
       For example 'min ETH/BTC bid'.
User: help avg
Vader: 'avg' command computes average ask or bid for the sent product over the sent number of time steps.
       Also, if the input step is higher than the total time steps that have been past, average is calculated from the first time step.
       In order to use this power, write 'avg <product name> <orderType> <time step>.
       For example 'avg ETH/BTC bid 10'.
User: help predict
Vader: 'predict' command computes the predicition for the max/min ask/bid price for a given product at the next time step.
       In order to use this power, write 'predict <max or min> <product name> <orderType> .
       For example 'predict max ETH/BTC bid'.
User: help time
Vader: 'time' command prompts the current time step.
       In order to use this power, simply write 'time'.
User: help step
Vader: 'step' command moves to the next time step, makes the exchanges and prompts it.
       In order to use this power, simply write 'step'.
User: help change
Vader: 'change' command calculates how much the bid or ask price changed from since a given step input (e.g 5 steps earlier) until the current time.
       Also, if the input step is higher than the total time steps that have been past, change is calculated from the first time step.
       In order to use this power, write 'change <product name> <orderType> <time step> .
       For example 'change ETH/BTC bid 10'.
```

'change' COMMAND

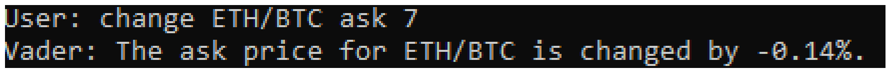
'change' command is my own command as Task 2 demands. The purpose of this command is to calculate how much a bid or ask price has changed since a given time step, in percentage.

In order to use this command, the user needs to enter an input of 4 arguments as change <product name> <orderType> <time step>.

change : name of the command.
<product name> : name of the product that the user wants to learn about.
<orderType> : ask or bid, depending on which one the user wants to learn about.
<time step> : How much earlier the user wants to compare with the current time, in steps.

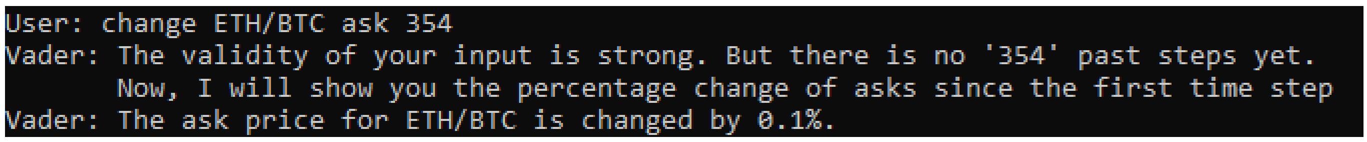
It should be noted that the comparison is made between the average prices.

Below, there is an example of how the command is called and the resulting output. The user queries for what is the change in the ask price of ETH/BTC between 7 steps earlier and the current time. As a result, the advisor bot returns with that average ETH/BTC ask price decreased by 0.14 percent in 7 steps.



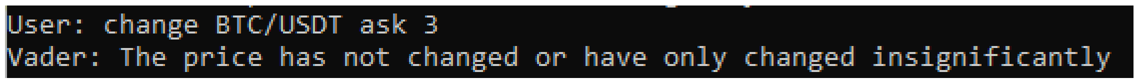
```
User: change ETH/BTC ask 7
Vader: The ask price for ETH/BTC is changed by -0.14%.
```

One scenario with this command is that the user might enter a step higher than the total time steps that have passed. In this scenario, the bot returns with a message that there are not that many previous steps, so the calculation is made between the first time step and the current time step. The example is as shown below:



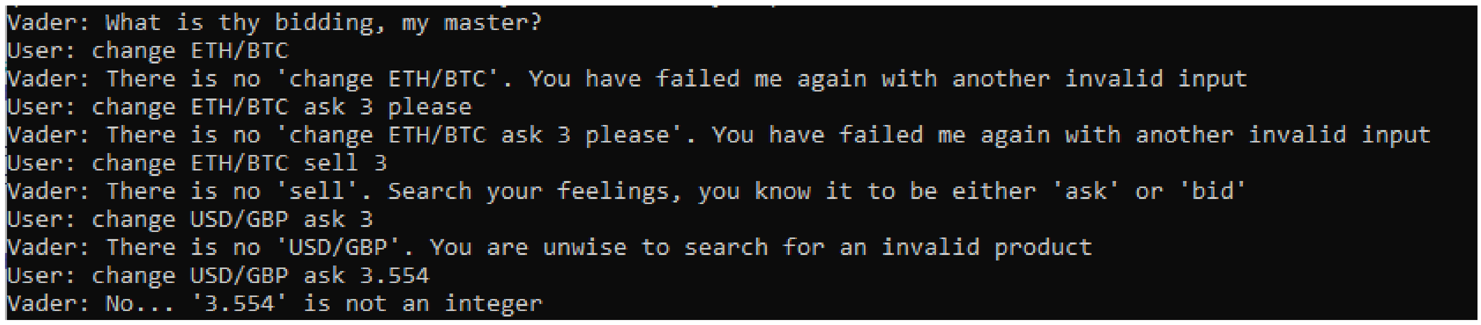
```
User: change ETH/BTC ask 354
Vader: The validity of your input is strong. But there is no '354' past steps yet.
       Now, I will show you the percentage change of asks since the first time step
Vader: The ask price for ETH/BTC is changed by 0.1%.
```

If there is no change in price, or if the change is very small, the bot will indicate that “The price has not changed or have only changed insignificantly”. The bot rounds the changes until 2 decimals, which results that any change smaller than 0.005% will be rounded to 0.00% and will be behaved as insignificant.



```
User: change BTC/USDT ask 3
Vader: The price has not changed or have only changed insignificantly
```

The command also checks for invalid inputs. If an invalid input is entered, an error message is returned to the user and a new input will be asked. Below, there are examples of invalid inputs and the resulting outputs.



```
Vader: What is thy bidding, my master?
User: change ETH/BTC
Vader: There is no 'change ETH/BTC'. You have failed me again with another invalid input
User: change ETH/BTC ask 3 please
Vader: There is no 'change ETH/BTC ask 3 please'. You have failed me again with another invalid input
User: change ETH/BTC sell 3
Vader: There is no 'sell'. Search your feelings, you know it to be either 'ask' or 'bid'
User: change USD/GBP ask 3
Vader: There is no 'USD/GBP'. You are unwise to search for an invalid product
User: change USD/GBP ask 3.554
Vader: No... '3.554' is not an integer
```

The command also handles the scenario of no asks or bids having been made for a particular timestamp. In this case, it checks the prices of earlier steps. These scenarios and the logic is explained in the detailed analysis of the algorithm.

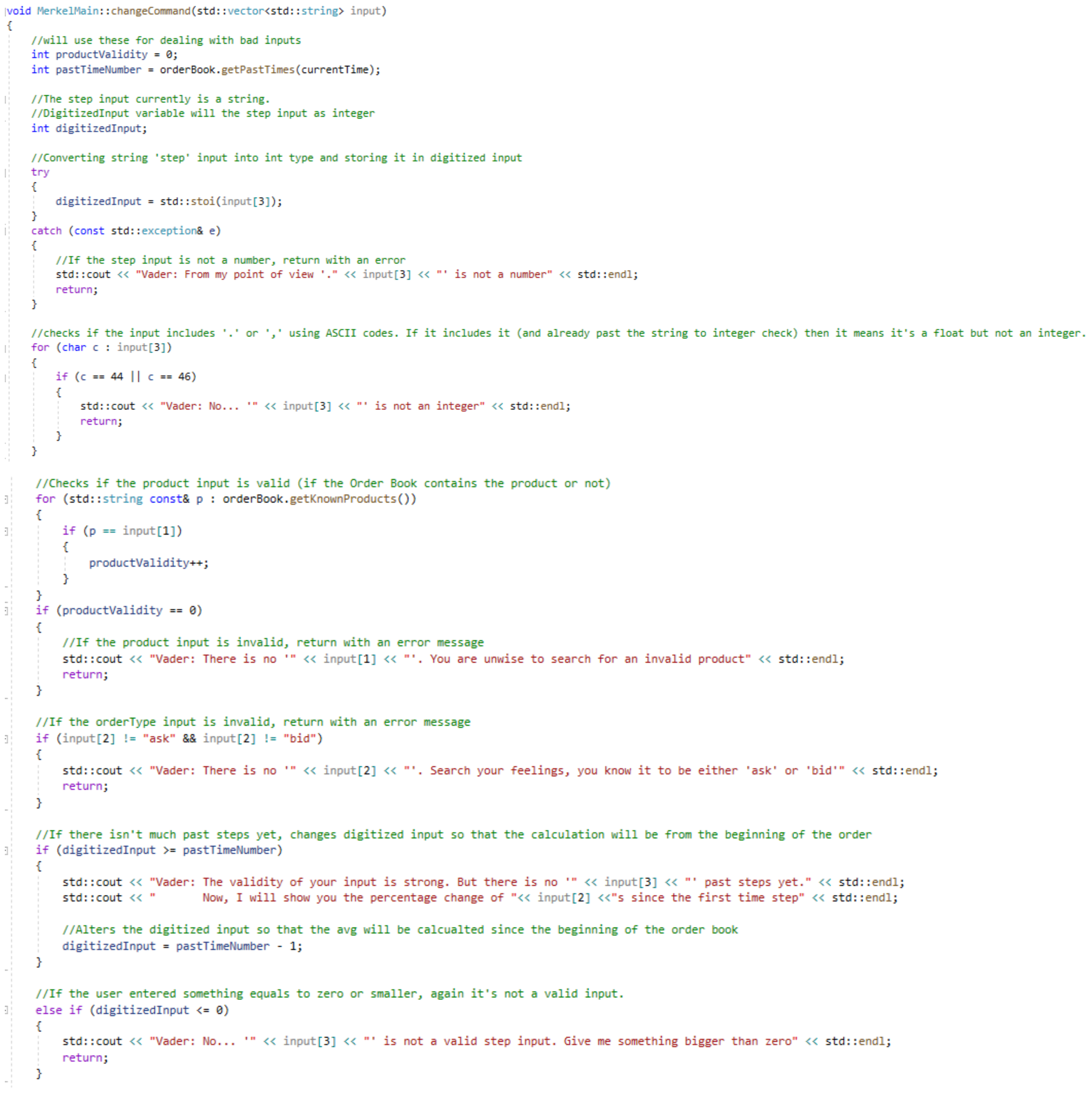
IMPLEMENTATION OF ‘change’ COMMAND

The main handling function for ‘change’ command is MerkelMain::changeCommand function. It is a void function under MerkelMain and takes only one argument, std::vector<std::string> input, which is no other than the tokenized user input. It is called in the same way as all the other command functions, which are explained under “General Information on Command Functions” section.

The algorithm starts with storing ‘step’ input as an int. Initially, even though the user enters an integer, the type of the input is a vector of strings. Therefore, the algorithm creates an int type variable called ‘digitizedInput’ and stores the step input as an int.

At the same time, several checks about input validity are made. The checkers and what do they check for are shown below. For every invalid input, an error message is sent to the user.

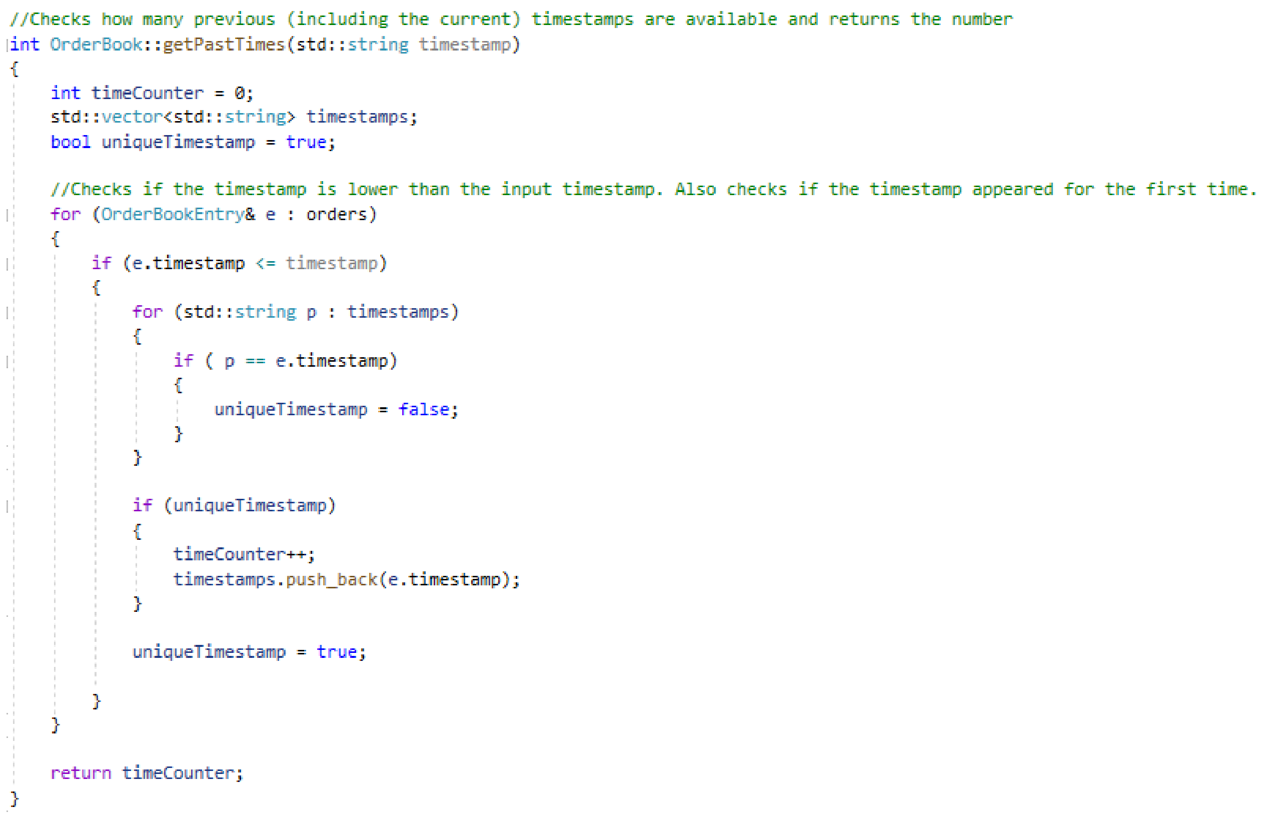
Also note that, input[1] is the ‘product’ input, input[2] is the ‘order type’ input and input[3] is the ‘step’ input.



```cpp
void MerkelMain::changeCommand(std::vector<std::string> input)
{
    //will use these for dealing with bad inputs
    int productValidity = 0;
    int pastTimeNumber = orderBook.getPastTimes(currentTime);

    //The step input currently is a string.
    //DigitizedInput variable will the step input as integer
    int digitizedInput;

    //Converting string 'step' input into int type and storing it in digitized input
    try
    {
        digitizedInput = std::stoi(input[3]);
    }
    catch (const std::exception& e)
    {
        //If the step input is not a number, return with an error
        std::cout << "Vader: From my point of view '." << input[3] << "' is not a number" << std::endl;
        return;
    }

    //checks if the input includes '.' or ',' using ASCII codes. If it includes it (and already past the string to integer check) then it means it's a float but not an integer.
    for (char c : input[3])
    {
        if (c == 44 || c == 46)
        {
            std::cout << "Vader: No... '" << input[3] << "' is not an integer" << std::endl;
            return;
        }
    }

    //Checks if the product input is valid (if the Order Book contains the product or not)
    for (std::string const& p : orderBook.getKnownProducts())
    {
        if (p == input[1])
        {
            productValidity++;
        }
    }
    if (productValidity == 0)
    {
        //If the product input is invalid, return with an error message
        std::cout << "Vader: There is no '" << input[1] << "'. You are unwise to search for an invalid product" << std::endl;
        return;
    }

    //If the orderType input is invalid, return with an error message
    if (input[2] != "ask" && input[2] != "bid")
    {
        std::cout << "Vader: There is no '" << input[2] << "'. Search your feelings, you know it to be either 'ask' or 'bid'" << std::endl;
        return;
    }

    //If there isn't much past steps yet, changes digitized input so that the calculation will be from the beginning of the order
    if (digitizedInput >= pastTimeNumber)
    {
        std::cout << "Vader: The validity of your input is strong. But there is no '" << input[3] << "' past steps yet." << std::endl;
        std::cout << "       Now, I will show you the percentage change of "<< input[2] <<"s since the first time step" << std::endl;

        //Alters the digitized input so that the avg will be calcualted since the beginning of the order book
        digitizedInput = pastTimeNumber - 1;
    }

    //If the user entered something equals to zero or smaller, again it's not a valid input.
    else if (digitizedInput <= 0)
    {
        std::cout << "Vader: No... '" << input[3] << "' is not a valid step input. Give me something bigger than zero" << std::endl;
        return;
    }
```

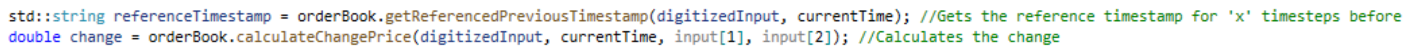
One thing to note here is that the function also calls an OrderBook class function getPastTimes. This function returns the number of how many time steps have passed including the time step in the argument (actually the argument is ‘timestamp’) which is the current time for changeCommand. The result is stored in pastTimeNumber variable. This is used in order to see if the user asks for more than available previous time steps. If that happens, no error is returned but the calculation will be made from the first time step, along with a message of the situation to the user.

The algorithm of getPastTimes is shown below:



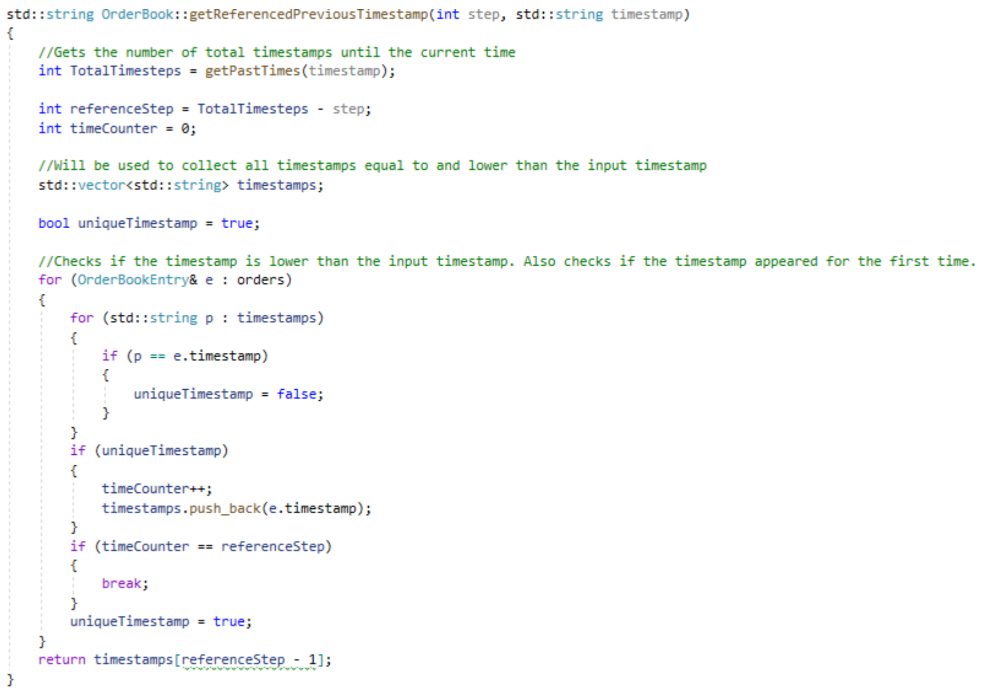
```
//Checks how many previous (including the current) timestamps are available and returns the number
int OrderBook::getPastTimes(std::string timestamp)
{
    int timeCounter = 0;
    std::vector<std::string> timestamps;
    bool uniqueTimestamp = true;

    //Checks if the timestamp is lower than the input timestamp. Also checks if the timestamp appeared for the first time.
    for (OrderBookEntry& e : orders)
    {
        if (e.timestamp <= timestamp)
        {
            for (std::string p : timestamps)
            {
                if ( p == e.timestamp)
                {
                    uniqueTimestamp = false;
                }
            }

            if (uniqueTimestamp)
            {
                timeCounter++;
                timestamps.push_back(e.timestamp);
            }

            uniqueTimestamp = true;

        }
    }

    return timeCounter;
}
```

If everything is ok with the input, then the algorithm executes the functions in order to calculate the change in price. The calculation is made by the following piece of code, which includes another two OrderBook class functions, getReferencedPreviousTimestamp function and calculateChangePrice function.



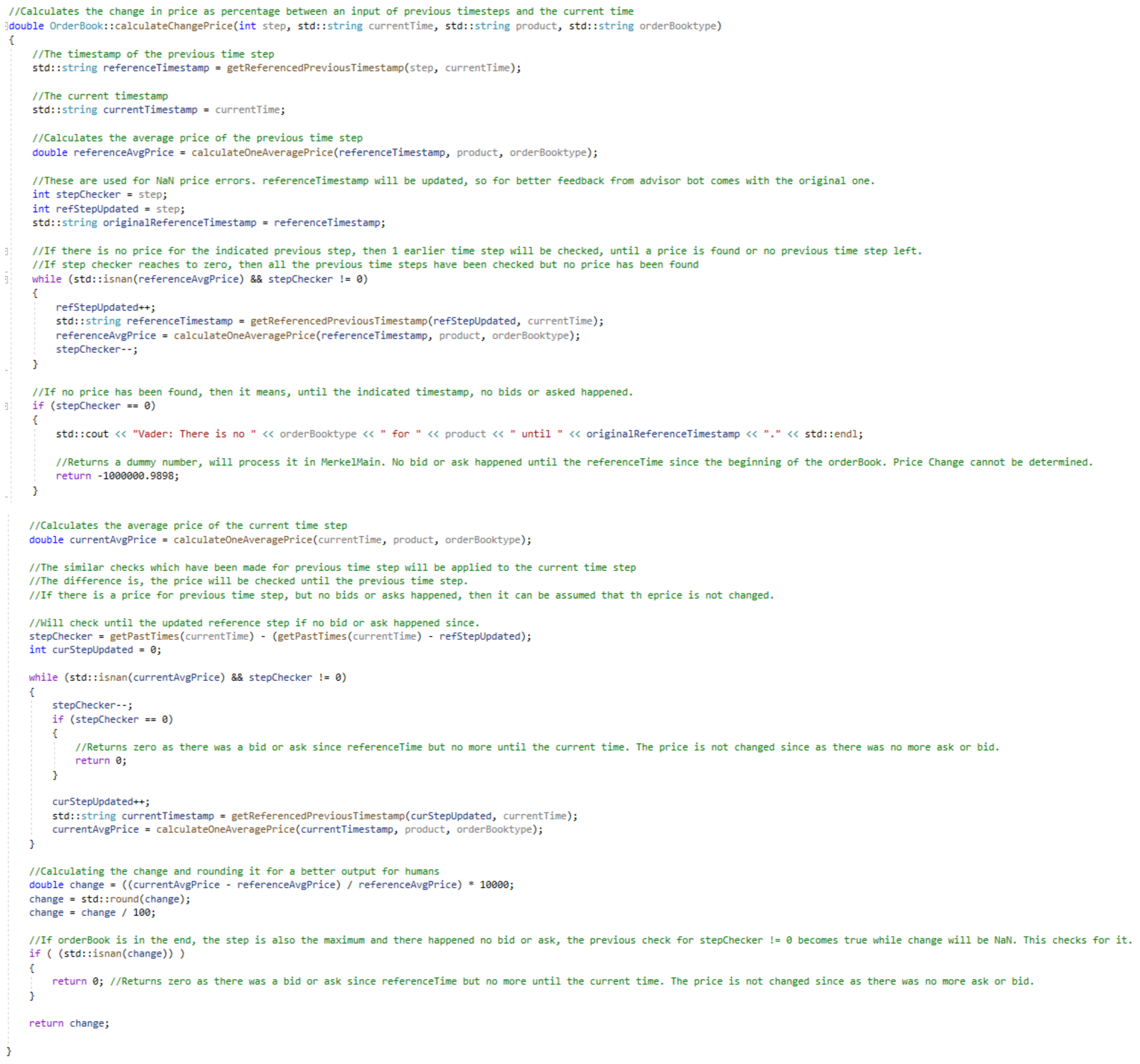
```
std::string referenceTimestamp = orderBook.getReferencedPreviousTimestamp(digitizedInput, currentTime); //Gets the reference timestamp for 'x' timesteps before
double change = orderBook.calculateChangePrice(digitizedInput, currentTime, input[1], input[2]); //Calculates the change
```

getReferencedPreviousTimestamp returns the timestamp of the queried previous time step. The algorithm of the function is as shown at the following page.



```cpp
std::string OrderBook::getReferencedPreviousTimestamp(int step, std::string timestamp)
{
    //Gets the number of total timestamps until the current time
    int TotalTimesteps = getPastTimes(timestamp);

    int referenceStep = TotalTimesteps - step;
    int timeCounter = 0;

    //Will be used to collect all timestamps equal to and lower than the input timestamp
    std::vector<std::string> timestamps;

    bool uniqueTimestamp = true;

    //Checks if the timestamp is lower than the input timestamp. Also checks if the timestamp appeared for the first time.
    for (OrderBookEntry& e : orders)
    {
        for (std::string p : timestamps)
        {
            if (p == e.timestamp)
            {
                uniqueTimestamp = false;
            }
        }
        if (uniqueTimestamp)
        {
            timeCounter++;
            timestamps.push_back(e.timestamp);
        }
        if (timeCounter == referenceStep)
        {
            break;
        }
        uniqueTimestamp = true;
    }
    return timestamps[referenceStep - 1];
}
```

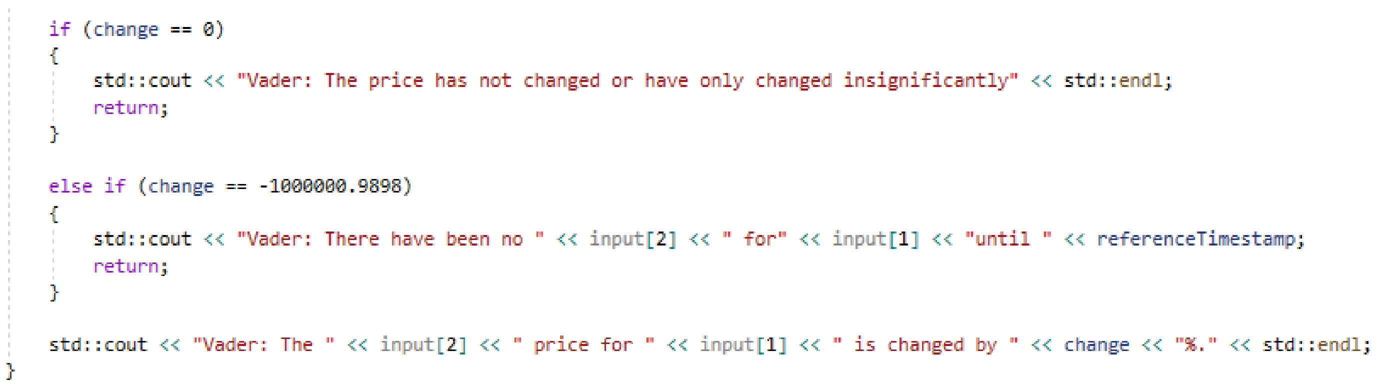
calculateChangePrice is the actual function which calculates the changes in prices. The algorithm is:



```cpp
//Calculates the change in price as percentage between an input of previous timesteps and the current time
double OrderBook::calculateChangePrice(int step, std::string currentTime, std::string product, std::string orderBooktype)
{
    //The timestamp of the previous time step
    std::string referenceTimestamp = getReferencedPreviousTimestamp(step, currentTime);

    //The current timestamp
    std::string currentTimestamp = currentTime;

    //Calculates the average price of the previous time step
    double referenceAvgPrice = calculateOneAveragePrice(referenceTimestamp, product, orderBooktype);

    //These are used for NaN price errors. referenceTimestamp will be updated, so for better feedback from advisor bot comes with the original one.
    int stepChecker = step;
    int refStepUpdated = step;
    std::string originalReferenceTimestamp = referenceTimestamp;

    //If there is no price for the indicated previous step, then 1 earlier time step will be checked, until a price is found or no previous time step left.
    //If step checker reaches to zero, then all the previous time steps have been checked but no price has been found
    while (std::isnan(referenceAvgPrice) && stepChecker != 0)
    {
        refStepUpdated++;
        std::string referenceTimestamp = getReferencedPreviousTimestamp(refStepUpdated, currentTime);
        referenceAvgPrice = calculateOneAveragePrice(referenceTimestamp, product, orderBooktype);
        stepChecker--;
    }

    //If no price has been found, then it means, until the indicated timestamp, no bids or asked happened.
    if (stepChecker == 0)
    {
        std::cout << "Vader: There is no " << orderBooktype << " for " << product << " until " << originalReferenceTimestamp << "." << std::endl;

        //Returns a dummy number, will process it in MerkelMain. No bid or ask happened until the referenceTime since the beginning of the orderBook. Price Change cannot be determined.
        return -1000000.9898;
    }

    //Calculates the average price of the current time step
    double currentAvgPrice = calculateOneAveragePrice(currentTime, product, orderBooktype);

    //The similar checks which have been made for previous time step will be applied to the current time step
    //The difference is, the price will be checked until the previous time step.
    //If there is a price for previous time step, but no bids or asks happened, then it can be assumed that th eprice is not changed.

    //Will check until the updated reference step if no bid or ask happened since.
    stepChecker = getPastTimes(currentTime) - (getPastTimes(currentTime) - refStepUpdated);
    int curStepUpdated = 0;

    while (std::isnan(currentAvgPrice) && stepChecker != 0)
    {
        stepChecker--;
        if (stepChecker == 0)
        {
            //Returns zero as there was a bid or ask since referenceTime but no more until the current time. The price is not changed since as there was no more ask or bid.
            return 0;
        }

        curStepUpdated++;
        std::string currentTimestamp = getReferencedPreviousTimestamp(curStepUpdated, currentTime);
        currentAvgPrice = calculateOneAveragePrice(currentTimestamp, product, orderBooktype);
    }

    //Calculating the change and rounding it for a better output for humans
    double change = ((currentAvgPrice - referenceAvgPrice) / referenceAvgPrice) * 10000;
    change = std::round(change);
    change = change / 100;

    //If orderBook is in the end, the step is also the maximum and there happened no bid or ask, the previous check for stepChecker != 0 becomes true while change will be NaN. This checks for it.
    if ( (std::isnan(change)) )
    {
        return 0; //Returns zero as there was a bid or ask since referenceTime but no more until the current time. The price is not changed since as there was no more ask or bid.
    }

    return change;

}
```

In the end, calculateChangePrice function returns the percentage of the difference. Depending on the result, changeCommand prompts the result to the user.

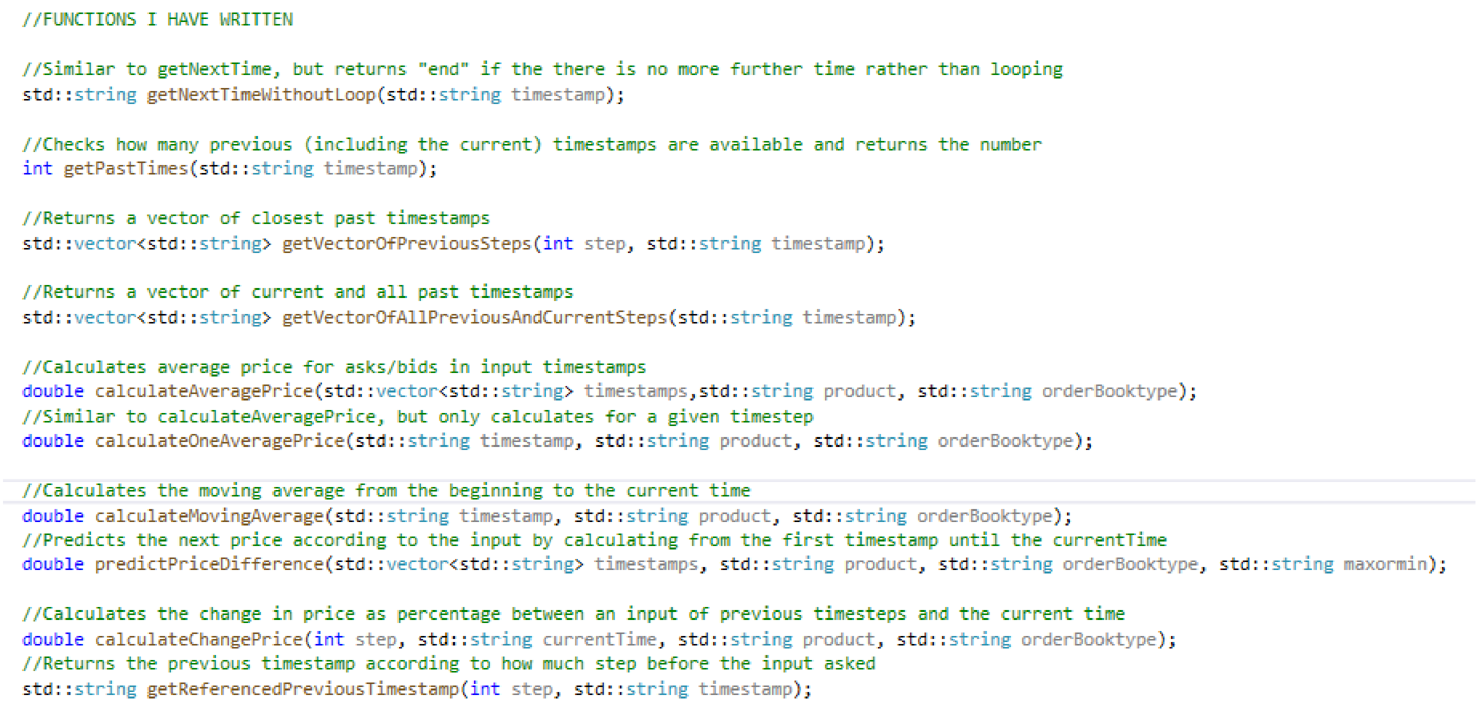
There is also a slight detail that, under certain conditions, calculateChangePrice may return a dummy number -1000000.9898. It happens when there has never been an ask or bid for the indicated product until the indicated previous time step. These result can never been returned if the actual change is this number, as calculateChangePrice function returns a number with two decimals, so it is safe.



```
    if (change == 0)
    {
        std::cout << "Vader: The price has not changed or have only changed insignificantly" << std::endl;
        return;
    }

    else if (change == -1000000.9898)
    {
        std::cout << "Vader: There have been no " << input[2] << " for" << input[1] << "until " << referenceTimestamp;
        return;
    }

    std::cout << "Vader: The " << input[2] << " price for " << input[1] << " is changed by " << change << "%." << std::endl;
}
```

OTHER OrderBook FUNCTIONS

While implementing command functions, I have also written many new OrderBook class commands. Below, there is a list of these functions along with what they are doing:



```
//FUNCTIONS I HAVE WRITTEN

//Similar to getNextTime, but returns "end" if the there is no more further time rather than looping
std::string getNextTimeWithoutLoop(std::string timestamp);

//Checks how many previous (including the current) timestamps are available and returns the number
int getPastTimes(std::string timestamp);

//Returns a vector of closest past timestamps
std::vector<std::string> getVectorOfPreviousSteps(int step, std::string timestamp);

//Returns a vector of current and all past timestamps
std::vector<std::string> getVectorOfAllPreviousAndCurrentSteps(std::string timestamp);

//Calculates average price for asks/bids in input timestamps
double calculateAveragePrice(std::vector<std::string> timestamps,std::string product, std::string orderBooktype);
//Similar to calculateAveragePrice, but only calculates for a given timestep
double calculateOneAveragePrice(std::string timestamp, std::string product, std::string orderBooktype);

//Calculates the moving average from the beginning to the current time
double calculateMovingAverage(std::string timestamp, std::string product, std::string orderBooktype);
//Predicts the next price according to the input by calculating from the first timestamp until the currentTime
double predictPriceDifference(std::vector<std::string> timestamps, std::string product, std::string orderBooktype, std::string maxormin);

//Calculates the change in price as percentage between an input of previous timesteps and the current time
double calculateChangePrice(int step, std::string currentTime, std::string product, std::string orderBooktype);
//Returns the previous timestamp according to how much step before the input asked
std::string getReferencedPreviousTimestamp(int step, std::string timestamp);
```

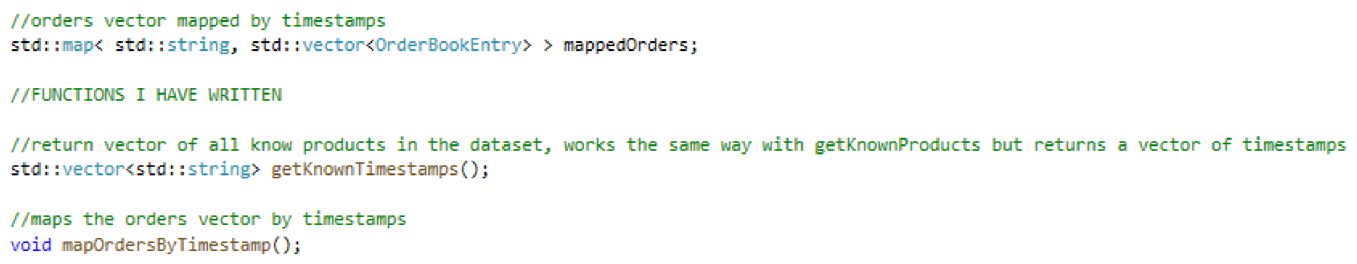
EXCHANGE CODE OPTIMIZATION

The idea of optimizing the exchange code is by using map() function on orders vector.

Originally, getOrders functions go through all of the order vectors in order to find order book entries matching with the ones queried by ask and bid vectors inside matchAsksToBids functions. This is not very efficient, as the for loop in getOrders function goes through all over the orders vector.

However, if we map the order book entries in the order book by their timestamp, the for loop in getOrders goes through a significantly lower number of order book entries.

For that, I have implemented the following map and the functions in OrderBook class

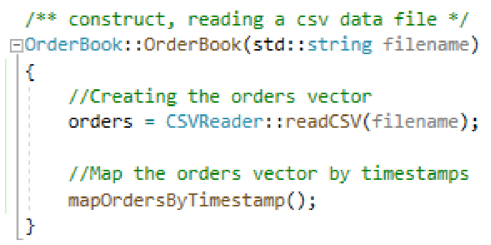


```
//orders vector mapped by timestamps
std::map< std::string, std::vector<OrderBookEntry> > mappedOrders;

//FUNCTIONS I HAVE WRITTEN

//return vector of all know products in the dataset, works the same way with getKnownProducts but returns a vector of timestamps
std::vector<std::string> getKnownTimestamps();

//maps the orders vector by timestamps
void mapOrdersByTimestamp();
```

mappedOrders is the map of orders with the timestamp keys.

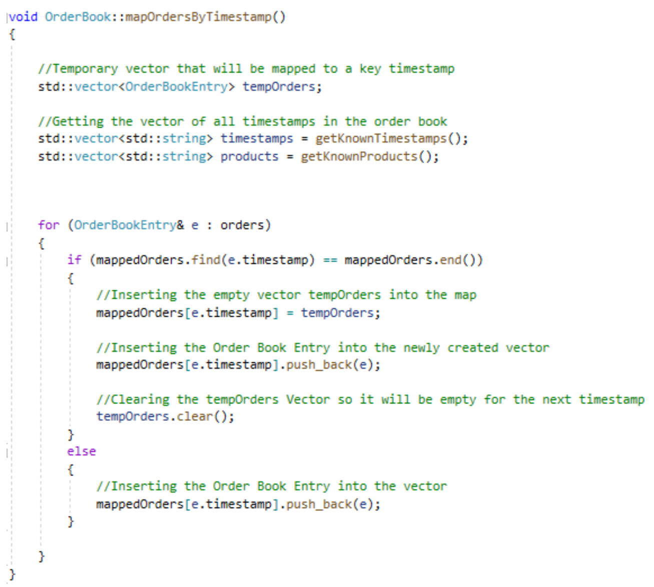
getKnownTimestamps function is very identical to getKnownProducts function from the starter code, with the only exception being that it returns the vector of all the timestamps rather than all the products.

mapOrdersByTimestamp function is the function that actually maps the orders vector by timestamps. It is run as soon as the OrderBook object is created.



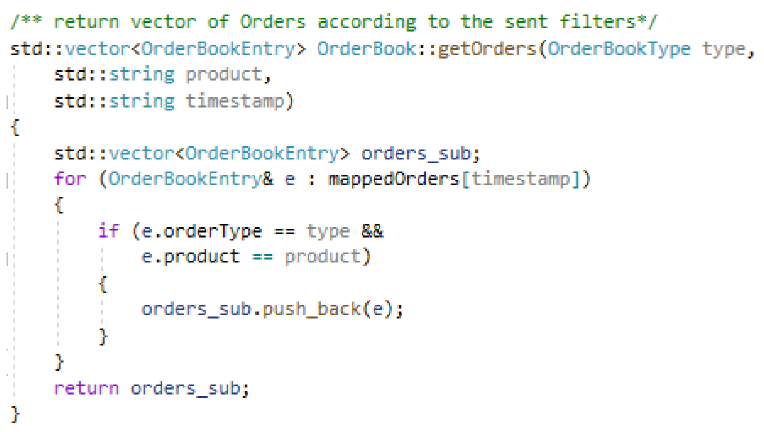
```
/** construct, reading a csv data file */
OrderBook::OrderBook(std::string filename)
{
    //Creating the orders vector
    orders = CSVReader::readCSV(filename);

    //Map the orders vector by timestamps
    mapOrdersByTimestamp();
}
```

The algorithm of the mapOrdersByTimestamp function is as shown below:



```
void OrderBook::mapOrdersByTimestamp()
{

    //Temporary vector that will be mapped to a key timestamp
    std::vector<OrderBookEntry> tempOrders;

    //Getting the vector of all timestamps in the order book
    std::vector<std::string> timestamps = getKnownTimestamps();
    std::vector<std::string> products = getKnownProducts();



    for (OrderBookEntry& e : orders)
    {
        if (mappedOrders.find(e.timestamp) == mappedOrders.end())
        {
            //Inserting the empty vector tempOrders into the map
            mappedOrders[e.timestamp] = tempOrders;

            //Inserting the Order Book Entry into the newly created vector
            mappedOrders[e.timestamp].push_back(e);

            //Clearing the tempOrders Vector so it will be empty for the next timestamp
            tempOrders.clear();
        }
        else
        {
            //Inserting the Order Book Entry into the vector
            mappedOrders[e.timestamp].push_back(e);
        }

    }
}
```

The last step is to modify getOrders function to work with mappedOrders rather than orders function.



```
/** 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 : mappedOrders[timestamp])
    {
        if (e.orderType == type &&
            e.product == product)
        {
            orders_sub.push_back(e);
        }
    }
    return orders_sub;
}
```

And with these little changes in our approach, the efficiency of the exchange code dramatically increases.