CM2005 Object-Oriented Programming

1. Introduction

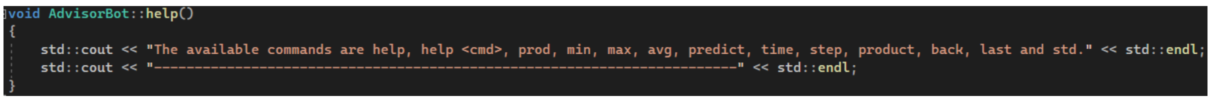
The purpose of AdvisorBot is to help a cryptocurrency investor in analysing crypto transactions. This bot takes in a CSV file filled with crypto transactions and offers several functions that will help the investor in deciding.

2. Commands implemented

Type of command	**Commands**	**Implemented?**
Required	C1: Help	Yes
Required	C2: Help <cmd>	Yes
Required	C3: Prod	Yes
Required	C4: Min	Yes
Required	C5: Max	Yes
Required	C6: Avg	Yes
Required	C7: Predict	Yes
Required	C8: Time	Yes
Required	C9: Step	Yes
Custom	C10: Product	Yes
Custom	C11: Back	Yes
Custom	C12: Further Back	Yes
Custom	C13: Further Forward	Yes
Custom	C14: Last Timestamp	Yes
Custom	C15: Standard Deviation	Yes

2.1. Commands description

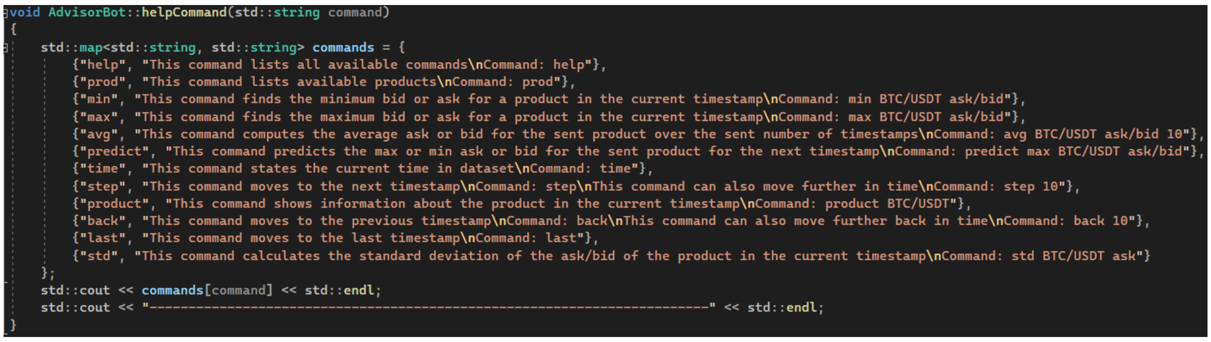
2.1.1. C1: Help



```
void AdvisorBot::help()
{
    std::cout << "The available commands are help, help <cmd>, prod, min, max, avg, predict, time, step, product, back, last and std." << std::endl;
    std::cout << "-------------------------------------------------------------------------" << std::endl;
}
```

The help function helps the user by showing a list of all the available commands for this AdvisorBot.

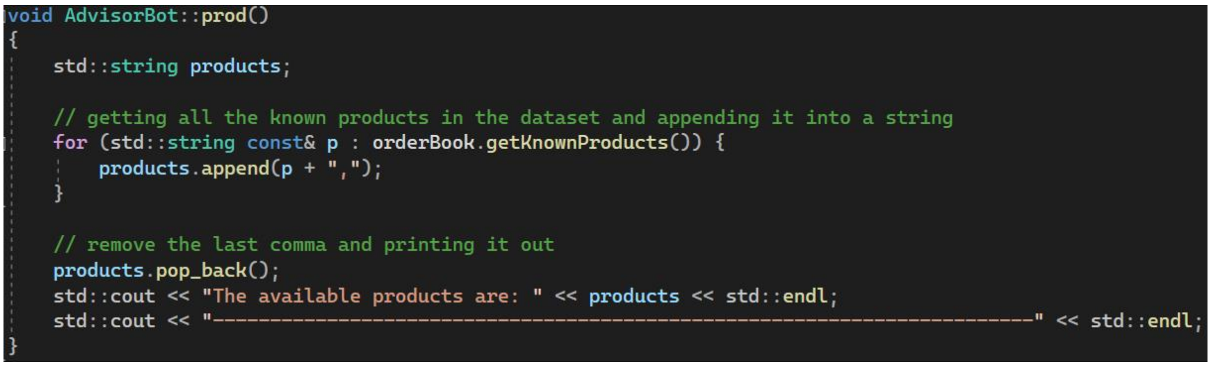
2.1.2. C2: Help <cmd>



```cpp
void AdvisorBot::helpCommand(std::string command)
{
    std::map<std::string, std::string> commands = {
        {"help", "This command lists all available commands\nCommand: help"},
        {"prod", "This command lists available products\nCommand: prod"},
        {"min", "This command finds the minimum bid or ask for a product in the current timestamp\nCommand: min BTC/USDT ask/bid"},
        {"max", "This command finds the maximum bid or ask for a product in the current timestamp\nCommand: max BTC/USDT ask/bid"},
        {"avg", "This command computes the average ask or bid for the sent product over the sent number of timestamps\nCommand: avg BTC/USDT ask/bid 10"},
        {"predict", "This command predicts the max or min ask or bid for the sent product for the next timestamp\nCommand: predict max BTC/USDT ask/bid"},
        {"time", "This command states the current time in dataset\nCommand: time"},
        {"step", "This command moves to the next timestamp\nCommand: step\nThis command can also move further in time\nCommand: step 10"},
        {"product", "This command shows information about the product in the current timestamp\nCommand: product BTC/USDT"},
        {"back", "This command moves to the previous timestamp\nCommand: back\nThis command can also move further back in time\nCommand: back 10"},
        {"last", "This command moves to the last timestamp\nCommand: last"},
        {"std", "This command calculates the standard deviation of the ask/bid of the product in the current timestamp\nCommand: std BTC/USDT ask"}
    };
    std::cout << commands[command] << std::endl;
    std::cout << "--------------------------------------------------------------------------" << std::endl;
}
```

The help <cmd> function outputs the help for a specific command. It takes in a string command as its parameter and uses std::map to locate which string is printed out to the user based on the command chosen by the user.

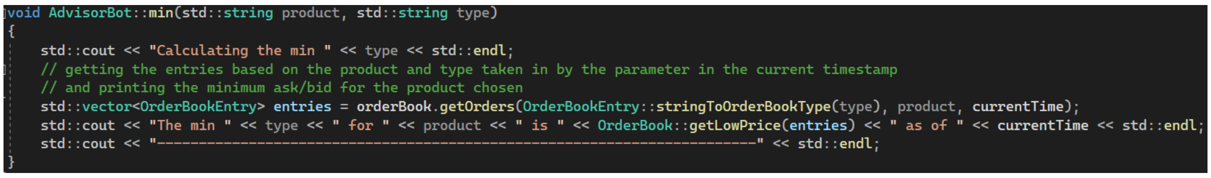
2.1.3. C3: Prod



```cpp
void AdvisorBot::prod()
{
    std::string products;

    // getting all the known products in the dataset and appending it into a string
    for (std::string const& p : orderBook.getKnownProducts()) {
        products.append(p + ",");
    }

    // remove the last comma and printing it out
    products.pop_back();
    std::cout << "The available products are: " << products << std::endl;
    std::cout << "--------------------------------------------------------------------------" << std::endl;
}
```

The prod function lists all the products in the dataset. It uses the function orderBook.getKnownProducts() which gets all the products in the dataset given. The products are then appended to the end of the string with a comma. Once all the products are appended the last comma is removed and the list is printed out. The function getKnownProducts() was taken from the base example code.

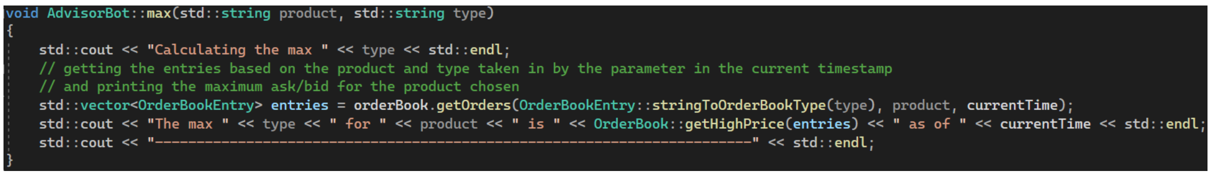
2.1.4. C4: Min



```cpp
void AdvisorBot::min(std::string product, std::string type)
{
    std::cout << "Calculating the min " << type << std::endl;
    // getting the entries based on the product and type taken in by the parameter in the current timestamp
    // and printing the minimum ask/bid for the product chosen
    std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, currentTime);
    std::cout << "The min " << type << " for " << product << " is " << OrderBook::getLowPrice(entries) << " as of " << currentTime << std::endl;
    std::cout << "--------------------------------------------------------------------------" << std::endl;
}
```

The min function finds the lowest price of the product in the current timestamp and outputs it to the user. It takes in the product and order type as its parameter. These 2 parameters and the current timestamp will then determine which entries are taken from the orderbook. Once, all the entries are taken the one with the lowest price value will be found using the OrderBook::getLowPrice and then it is printed out. The function getLowPrice() was taken from the base example code.

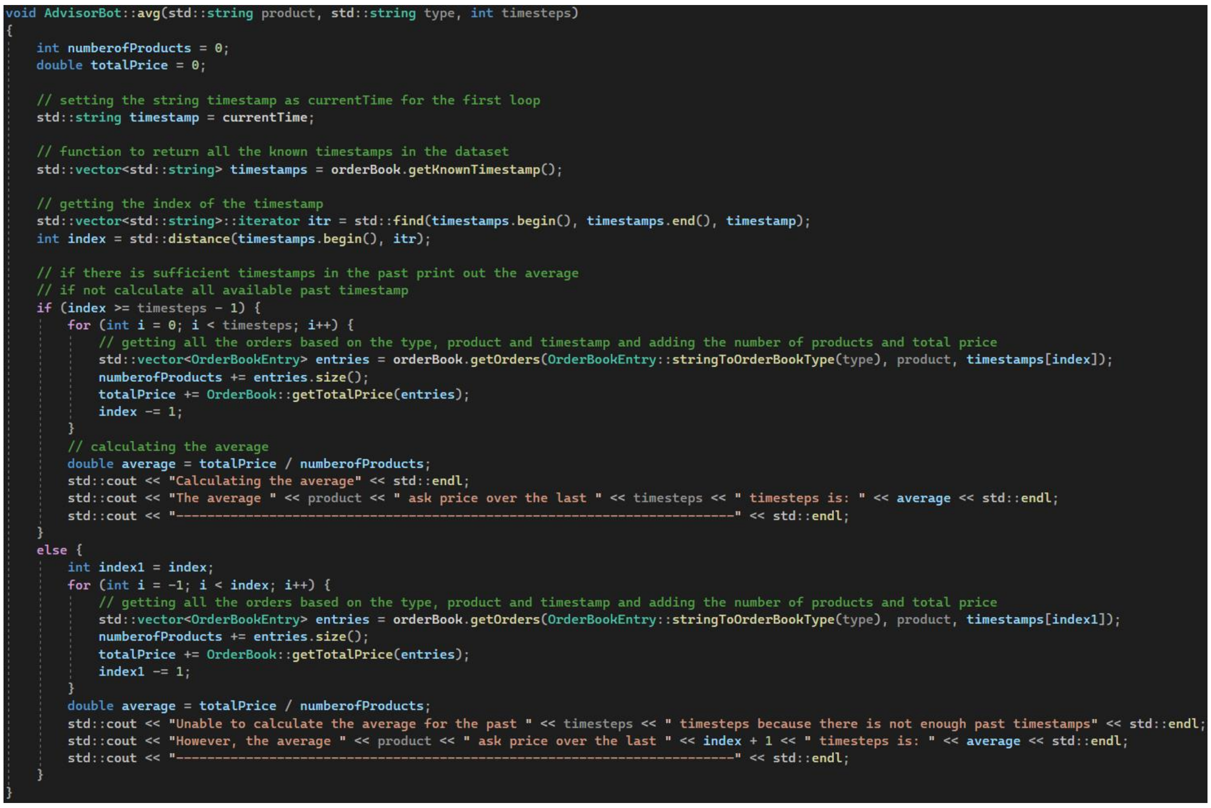
2.1.5. C5: Max



```cpp
void AdvisorBot::max(std::string product, std::string type)
{
    std::cout << "Calculating the max " << type << std::endl;
    // getting the entries based on the product and type taken in by the parameter in the current timestamp
    // and printing the maximum ask/bid for the product chosen
    std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, currentTime);
    std::cout << "The max " << type << " for " << product << " is " << OrderBook::getHighPrice(entries) << " as of " << currentTime << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The max function finds the highest price of the product in the current timestamp and outputs it to the user. It takes in the product and order type as its parameter. These 2 parameters and the current timestamp will then determine which entries are taken from the orderbook. Once, all the entries are taken the one with the highest price value will be found using the OrderBook::getHighPrice and then it is printed out. The function getHighPrice() was taken from the base example code.

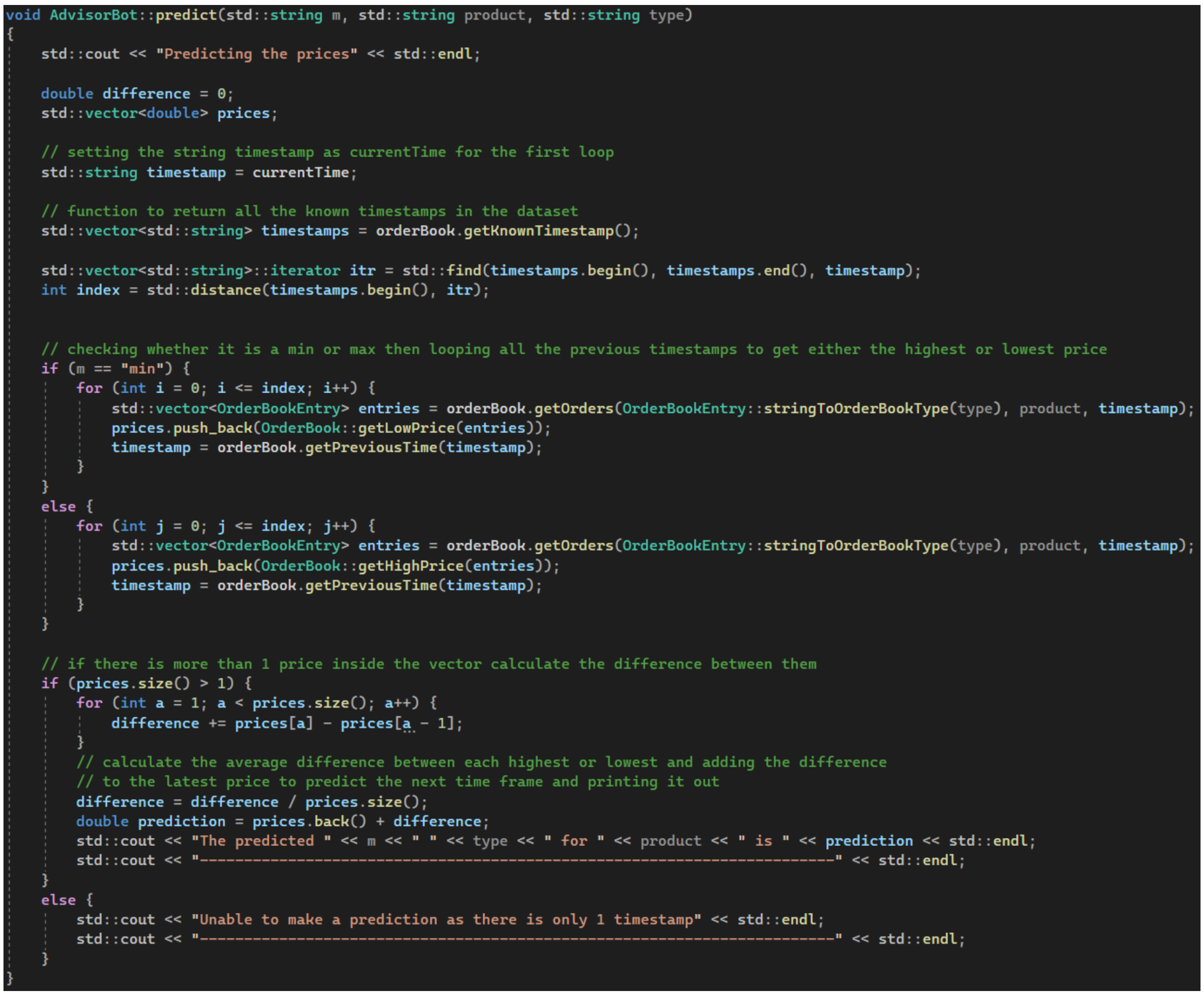
2.1.6. C6: Avg



```cpp
void AdvisorBot::avg(std::string product, std::string type, int timesteps)
{
    int numberofProducts = 0;
    double totalPrice = 0;

    // setting the string timestamp as currentTime for the first loop
    std::string timestamp = currentTime;

    // function to return all the known timestamps in the dataset
    std::vector<std::string> timestamps = orderBook.getKnownTimestamp();

    // getting the index of the timestamp
    std::vector<std::string>::iterator itr = std::find(timestamps.begin(), timestamps.end(), timestamp);
    int index = std::distance(timestamps.begin(), itr);

    // if there is sufficient timestamps in the past print out the average
    // if not calculate all available past timestamp
    if (index >= timesteps - 1) {
        for (int i = 0; i < timesteps; i++) {
            // getting all the orders based on the type, product and timestamp and adding the number of products and total price
            std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, timestamps[index]);
            numberofProducts += entries.size();
            totalPrice += OrderBook::getTotalPrice(entries);
            index -= 1;
        }
        // calculating the average
        double average = totalPrice / numberofProducts;
        std::cout << "Calculating the average" << std::endl;
        std::cout << "The average " << product << " ask price over the last " << timesteps << " timesteps is: " << average << std::endl;
        std::cout << "------------------------------------------------------------------------" << std::endl;
    }
    else {
        int index1 = index;
        for (int i = -1; i < index; i++) {
            // getting all the orders based on the type, product and timestamp and adding the number of products and total price
            std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, timestamps[index1]);
            numberofProducts += entries.size();
            totalPrice += OrderBook::getTotalPrice(entries);
            index1 -= 1;
        }
        double average = totalPrice / numberofProducts;
        std::cout << "Unable to calculate the average for the past " << timesteps << " timesteps because there is not enough past timestamps" << std::endl;
        std::cout << "However, the average " << product << " ask price over the last " << index + 1 << " timesteps is: " << average << std::endl;
        std::cout << "------------------------------------------------------------------------" << std::endl;
    }
}
```

The average function calculates the average ask/bid over the sent number of timesteps. It takes in 3 parameters which are the product, order type, and timesteps. Firstly, the string timestamp will be set as the current time, and also push all the known timestamps into a vector of timestamps. Secondly, the timestamp will then be used with the help of iterator and std::distance to find the index of the current time compared to all the timestamps in the dataset. Lastly, a check will be done to see if the timesteps provided by the user are more or equal to the timestamps that have passed. If true, calculate the average by looping the timesteps and adding the total prices together and dividing by the total number of entries, and printing out the average. However, if it is false, calculate the average based on how many timestamps have passed and print it out. The function getOrders() was taken from the base example code.

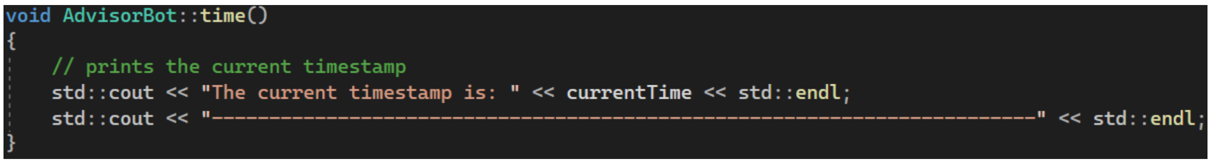
2.1.7. C7: Predict



```cpp
void AdvisorBot::predict(std::string m, std::string product, std::string type)
{
    std::cout << "Predicting the prices" << std::endl;

    double difference = 0;
    std::vector<double> prices;

    // setting the string timestamp as currentTime for the first loop
    std::string timestamp = currentTime;

    // function to return all the known timestamps in the dataset
    std::vector<std::string> timestamps = orderBook.getKnownTimestamp();

    std::vector<std::string>::iterator itr = std::find(timestamps.begin(), timestamps.end(), timestamp);
    int index = std::distance(timestamps.begin(), itr);


    // checking whether it is a min or max then looping all the previous timestamps to get either the highest or lowest price
    if (m == "min") {
        for (int i = 0; i <= index; i++) {
            std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, timestamp);
            prices.push_back(OrderBook::getLowPrice(entries));
            timestamp = orderBook.getPreviousTime(timestamp);
        }
    }
    else {
        for (int j = 0; j <= index; j++) {
            std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, timestamp);
            prices.push_back(OrderBook::getHighPrice(entries));
            timestamp = orderBook.getPreviousTime(timestamp);
        }
    }

    // if there is more than 1 price inside the vector calculate the difference between them
    if (prices.size() > 1) {
        for (int a = 1; a < prices.size(); a++) {
            difference += prices[a] - prices[a - 1];
        }
        // calculate the average difference between each highest or lowest and adding the difference
        // to the latest price to predict the next time frame and printing it out
        difference = difference / prices.size();
        double prediction = prices.back() + difference;
        std::cout << "The predicted " << m << " " << type << " for " << product << " is " << prediction << std::endl;
        std::cout << "------------------------------------------------------------------------" << std::endl;
    }
    else {
        std::cout << "Unable to make a prediction as there is only 1 timestamp" << std::endl;
        std::cout << "------------------------------------------------------------------------" << std::endl;
    }
}
```

The prediction function predicts the max/min product ask/bid in the next timestamp. It takes in 3 parameters, min/max, product, and order type. I decided to calculate the difference between each highest or lowest price and add it to the current highest/lowest value as compared to using the moving average as I feel that the moving average does not take into consideration the previous highest/lowest value. By finding the average difference it would be more accurate. Firstly, the string timestamp will be set as the current time, and also push all the known timestamps into a vector of timestamps. Secondly, the timestamp will then be used with the help of iterator and std::distance to find the index of the current time compared to all the timestamps in the dataset. Third, the entries are then gathered based on the product, order type, and timestamp. The index will then be used to loop this function to gather all the entries in the previous timestamps and push the highest or lowest price into a vector of strings depending on the user’s choice. Lastly, check if there is more than 1 timestamp to predict the next timestamp by finding the average difference between the prices and adding that to the latest value, and printing it out. However, if there aren’t enough timestamps to make a prediction print out an error message. The functions getOrders(), getLowPrice(), and getHighPrice() were taken from the base example code.

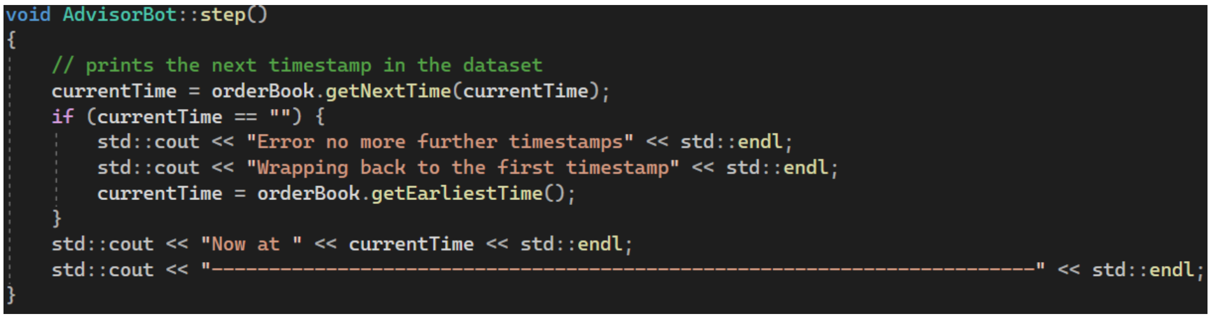
2.1.8. C8: Time



```cpp
void AdvisorBot::time()
{
    // prints the current timestamp
    std::cout << "The current timestamp is: " << currentTime << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The time function prints the current timestamp in the command line.

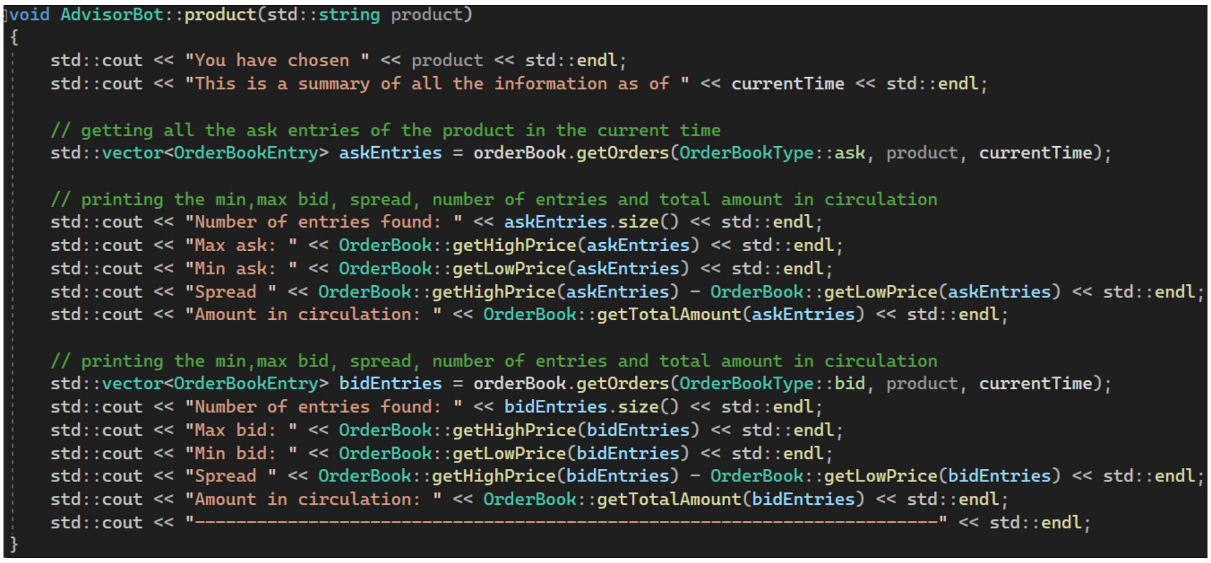
2.1.9. C9: Step



```cpp
void AdvisorBot::step()
{
    // prints the next timestamp in the dataset
    currentTime = orderBook.getNextTime(currentTime);
    if (currentTime == "") {
        std::cout << "Error no more further timestamps" << std::endl;
        std::cout << "Wrapping back to the first timestamp" << std::endl;
        currentTime = orderBook.getEarliestTime();
    }
    std::cout << "Now at " << currentTime << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The step function moves the command line to the next timestamp by using the orderBook.getNextTime() function. However, if there are no further timestamps in the dataset, the time will be set as the earliest time in the dataset. The functions getNextTime() and getEarliestTime() were taken from the base example code.

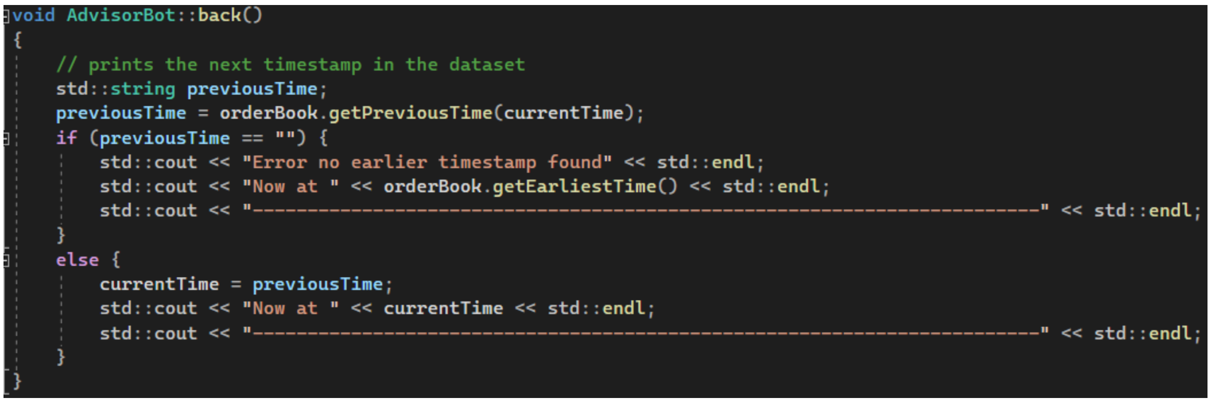
2.1.10. C10: Product



```cpp
void AdvisorBot::product(std::string product)
{
    std::cout << "You have chosen " << product << std::endl;
    std::cout << "This is a summary of all the information as of " << currentTime << std::endl;

    // getting all the ask entries of the product in the current time
    std::vector<OrderBookEntry> askEntries = orderBook.getOrders(OrderBookType::ask, product, currentTime);

    // printing the min,max bid, spread, number of entries and total amount in circulation
    std::cout << "Number of entries found: " << askEntries.size() << std::endl;
    std::cout << "Max ask: " << OrderBook::getHighPrice(askEntries) << std::endl;
    std::cout << "Min ask: " << OrderBook::getLowPrice(askEntries) << std::endl;
    std::cout << "Spread " << OrderBook::getHighPrice(askEntries) - OrderBook::getLowPrice(askEntries) << std::endl;
    std::cout << "Amount in circulation: " << OrderBook::getTotalAmount(askEntries) << std::endl;

    // printing the min,max bid, spread, number of entries and total amount in circulation
    std::vector<OrderBookEntry> bidEntries = orderBook.getOrders(OrderBookType::bid, product, currentTime);
    std::cout << "Number of entries found: " << bidEntries.size() << std::endl;
    std::cout << "Max bid: " << OrderBook::getHighPrice(bidEntries) << std::endl;
    std::cout << "Min bid: " << OrderBook::getLowPrice(bidEntries) << std::endl;
    std::cout << "Spread " << OrderBook::getHighPrice(bidEntries) - OrderBook::getLowPrice(bidEntries) << std::endl;
    std::cout << "Amount in circulation: " << OrderBook::getTotalAmount(bidEntries) << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The product function outputs information about the product chosen in the current timestamp. It takes in the product as its parameter and uses orderBook.getOrders() to gather all the entries based on the product chosen and prints out the number of entries, max and min ask and bid, spread between the max and min price, and the amount in circulation. The function getOrders() was taken from the base example code.

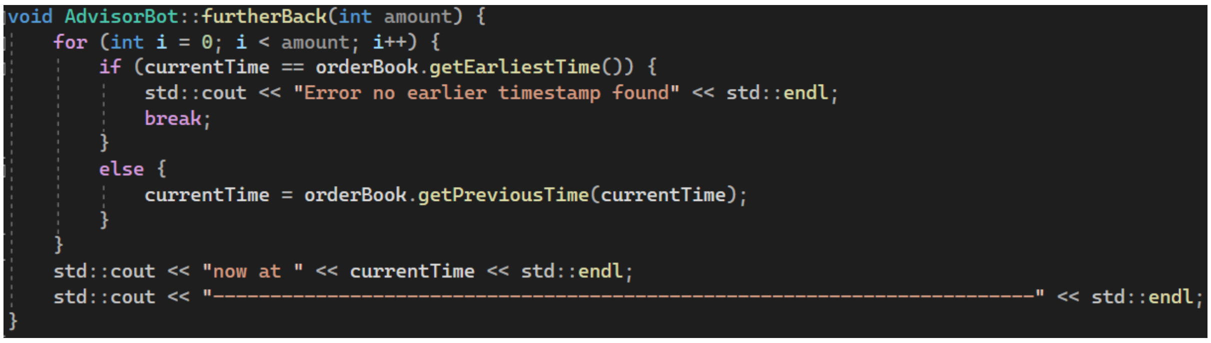
2.1.11. C12: Back



```cpp
void AdvisorBot::back()
{
    // prints the next timestamp in the dataset
    std::string previousTime;
    previousTime = orderBook.getPreviousTime(currentTime);
    if (previousTime == "") {
        std::cout << "Error no earlier timestamp found" << std::endl;
        std::cout << "Now at " << orderBook.getEarliestTime() << std::endl;
        std::cout << "------------------------------------------------------------------------------" << std::endl;
    }
    else {
        currentTime = previousTime;
        std::cout << "Now at " << currentTime << std::endl;
        std::cout << "------------------------------------------------------------------------------" << std::endl;
    }
}
```

The back function moves the user 1 timestamp back. This function uses the orderBook.getPreviousTime() function to return the previous time to the user. However, if there is no more previous timestamp, the timestamp will be set to the earliest time in the orderbook. The function getEarliestTime() was taken from the base example code.

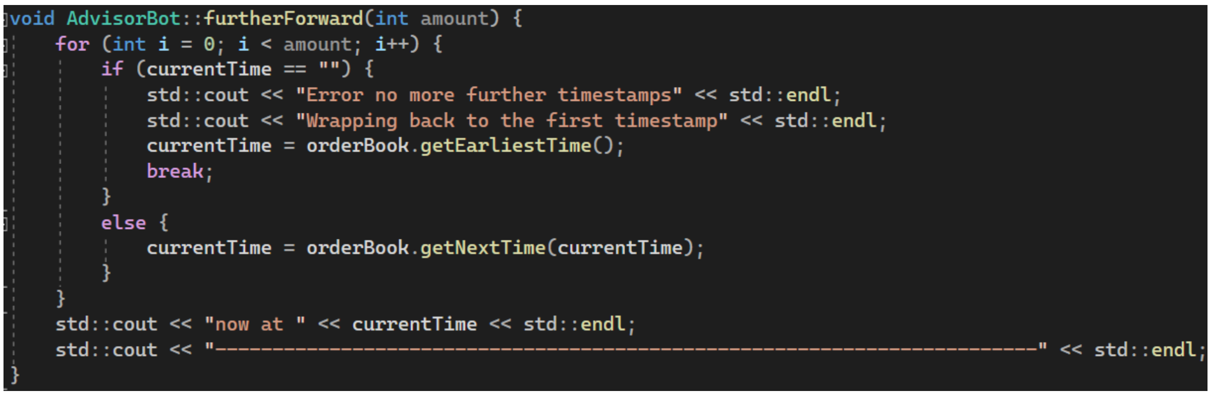
2.1.12. C12: Further Back



```cpp
void AdvisorBot::furtherBack(int amount) {
    for (int i = 0; i < amount; i++) {
        if (currentTime == orderBook.getEarliestTime()) {
            std::cout << "Error no earlier timestamp found" << std::endl;
            break;
        }
        else {
            currentTime = orderBook.getPreviousTime(currentTime);
        }
    }
    std::cout << "now at " << currentTime << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The further back function moves the current time several times back based on the user's choice. It takes in an amount parameter that loops the getPreviousTime() function to move the timestamp back. If there is no earlier timestamp an error message will be printed, and the user will be brought to the first timestamp. The function getEarliestTime() was taken from the based example code.

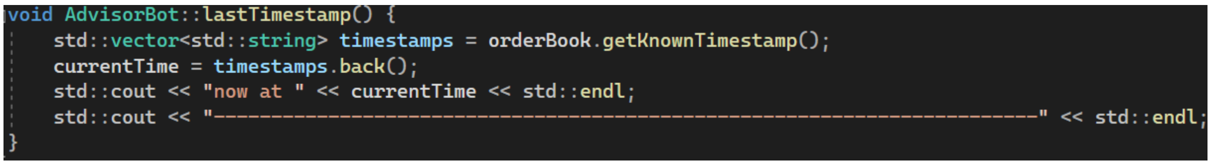
2.1.13. C13: Further Forward



```cpp
void AdvisorBot::furtherForward(int amount) {
    for (int i = 0; i < amount; i++) {
        if (currentTime == "") {
            std::cout << "Error no more further timestamps" << std::endl;
            std::cout << "Wrapping back to the first timestamp" << std::endl;
            currentTime = orderBook.getEarliestTime();
            break;
        }
        else {
            currentTime = orderBook.getNextTime(currentTime);
        }
    }
    std::cout << "now at " << currentTime << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The further forward function moves the current time several times forward based on the user's choice. It takes in an amount parameter that loops the getNextTime() function to move the timestamp forward. If there is no later timestamp an error message will be printed, and the user will be brought to the first timestamp. The function getEarliestTime() was taken from the based example code.

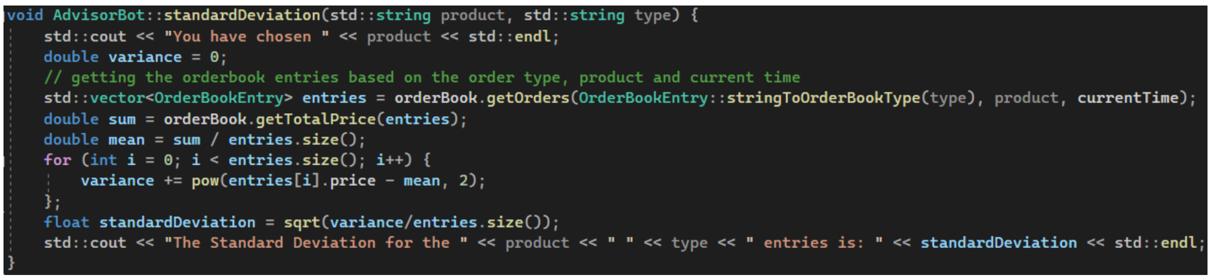
2.1.14. C14: Last Timestamp



```cpp
void AdvisorBot::lastTimestamp() {
    std::vector<std::string> timestamps = orderBook.getKnownTimestamp();
    currentTime = timestamps.back();
    std::cout << "now at " << currentTime << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The function Last Timestamp moves the user to the last timestamp in the dataset. Firstly, all the timestamps in the dataset are gathered by using the getKnownTimestamp() function and the timestamps are pushed to a vector of string timestamps. Lastly, the current time will be set as the last timestamp found.

2.1.15. C15: Standard Deviation

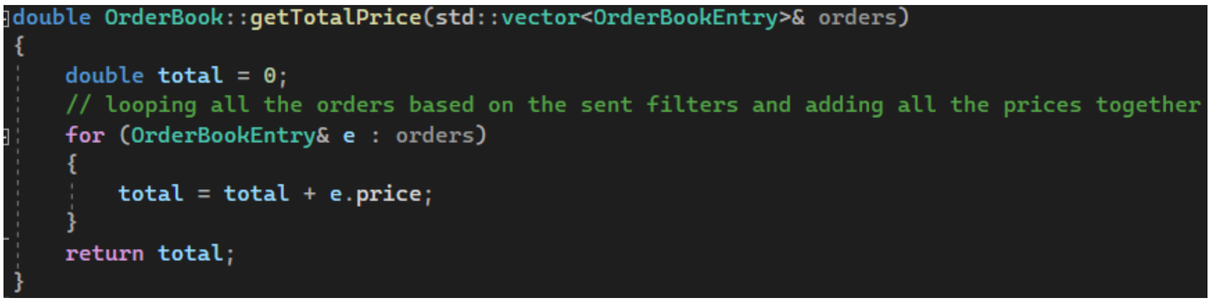


```cpp
void AdvisorBot::standardDeviation(std::string product, std::string type) {
    std::cout << "You have chosen " << product << std::endl;
    double variance = 0;
    // getting the orderbook entries based on the order type, product and current time
    std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(type), product, currentTime);
    double sum = orderBook.getTotalPrice(entries);
    double mean = sum / entries.size();
    for (int i = 0; i < entries.size(); i++) {
        variance += pow(entries[i].price - mean, 2);
    };
    float standardDeviation = sqrt(variance/entries.size());
    std::cout << "The Standard Deviation for the " << product << " " << type << " entries is: " << standardDeviation << std::endl;
}
```

The function Standard Deviation calculates the standard deviation of the product and order type chosen. Firstly, the entries are gathered and pushed to a vector. Secondly, the sum is found by using the function getTotalPrice and the mean is found by dividing the total price and the entries.size(). Third, the variance is found by looping the entries' size and finding the difference between each of the prices and its mean and powering it to the power of 2. Lastly, the standard deviation is found by square rooting the variance/entries.size() and printed out to the user.

3. Functions implemented

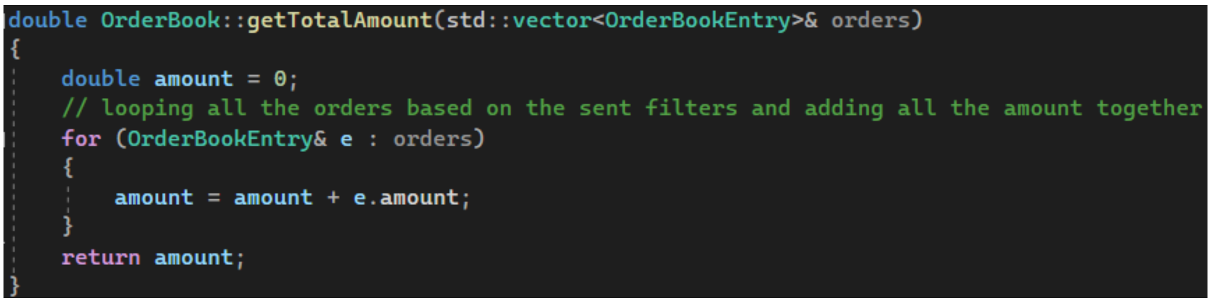
3.1. OrderBook::getTotalPrice()



```cpp
double OrderBook::getTotalPrice(std::vector<OrderBookEntry>& orders)
{
    double total = 0;
    // looping all the orders based on the sent filters and adding all the prices together
    for (OrderBookEntry& e : orders)
    {
        total = total + e.price;
    }
    return total;
}
```

This function takes in a vector of OrderBookEntry as its parameter and tallies all the prices within the vector and returns it to the user. This function was created for the average command in AdvisorBot.

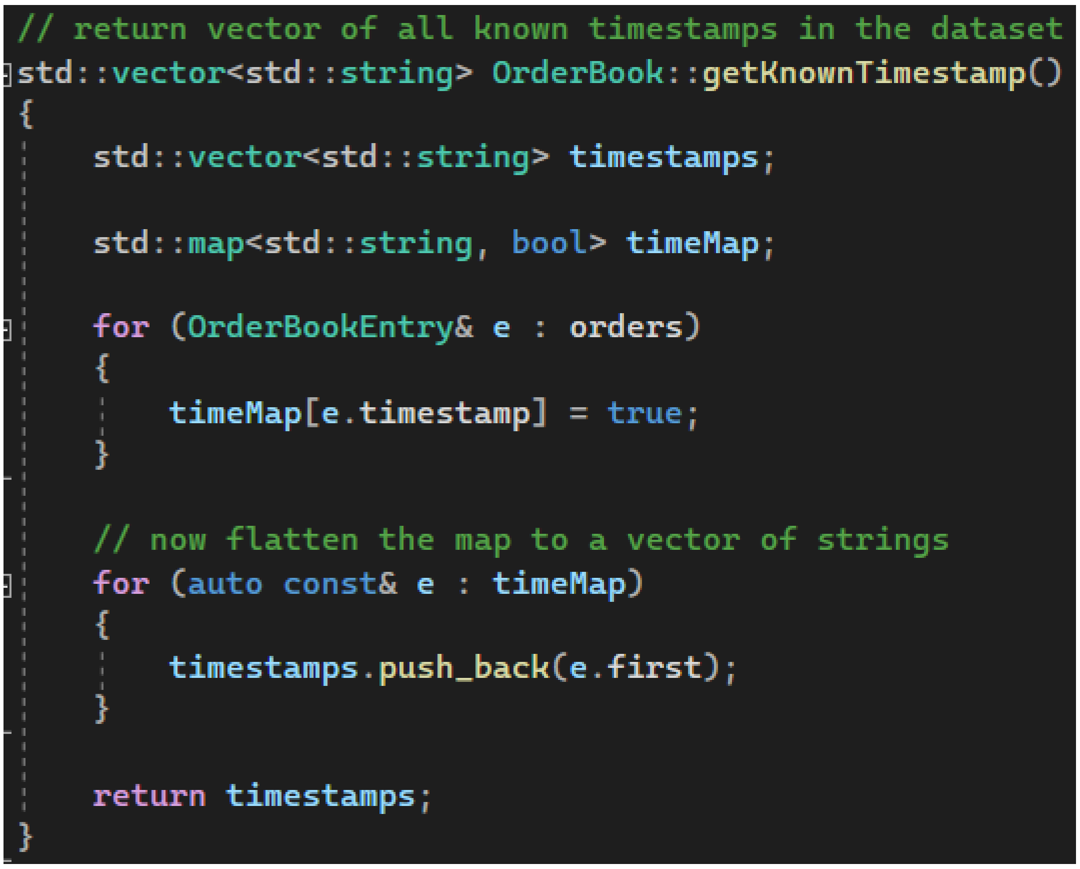
3.2. OrderBook::getTotalAmount()



```cpp
double OrderBook::getTotalAmount(std::vector<OrderBookEntry>& orders)
{
    double amount = 0;
    // looping all the orders based on the sent filters and adding all the amount together
    for (OrderBookEntry& e : orders)
    {
        amount = amount + e.amount;
    }
    return amount;
}
```

This function takes in a vector of OrderBookEntry as its parameter and tallies all the amount within the vector and returns it to the user. This function was created for the product command in AdvisorBot.

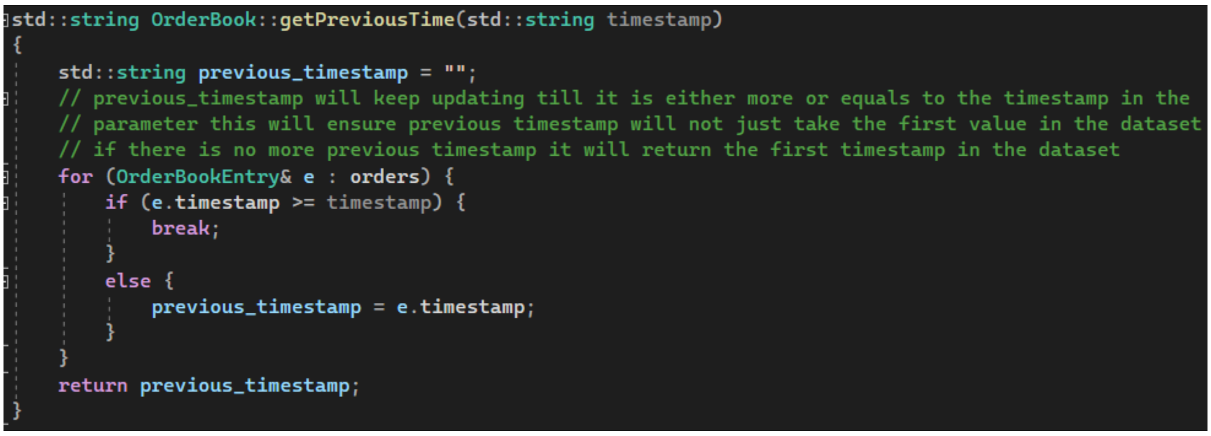
3.3. OrderBook::getKnownTimestamp()



```
// return vector of all known timestamps in the dataset
std::vector<std::string> OrderBook::getKnownTimestamp()
{
    std::vector<std::string> timestamps;

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

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

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

    return timestamps;
}
```

This function returns all the known timestamps in the dataset provided. This function is used in multiple commands within the AdvisorBot such as average, predict, etc. This function is used with the iterator and std::distance to find the index of the current time. This will help me commands in finding out whether has there been enough timestamps to calculate the average or predict the next time stamp.

3.4. OrderBook::getPreviousTime()

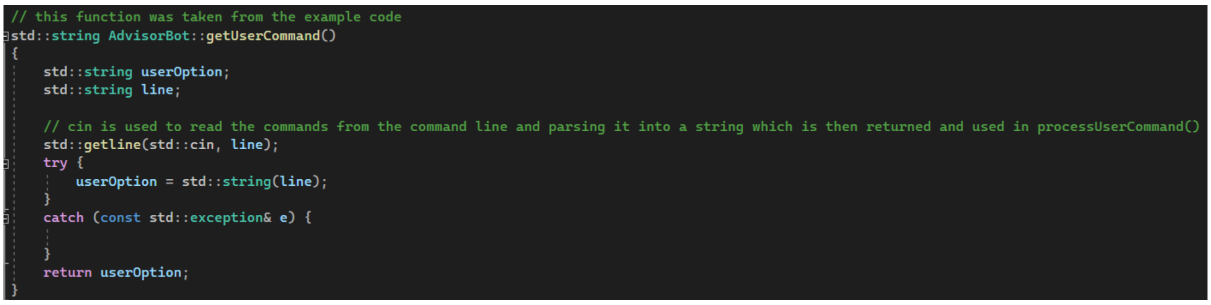


```
std::string OrderBook::getPreviousTime(std::string timestamp)
{
    std::string previous_timestamp = "";
    // previous_timestamp will keep updating till it is either more or equals to the timestamp in the
    // parameter this will ensure previous timestamp will not just take the first value in the dataset
    // if there is no more previous timestamp it will return the first timestamp in the dataset
    for (OrderBookEntry& e : orders) {
        if (e.timestamp >= timestamp) {
            break;
        }
        else {
            previous_timestamp = e.timestamp;
        }
    }
    return previous_timestamp;
}
```

This function returns the previous timestamp to the user. This function takes in a timestamp as its parameter, and it will be used to compare to all the other timestamps in the dataset. With every loop, the string previous_timestamp will be updated till the if statement is met. Whereby the previous_timestamp has to be either more or equal to the timestamp provided. This is to ensure that the function will return the next smallest timestamp instead of just returning the first timestamp.

4. Command parsing codes

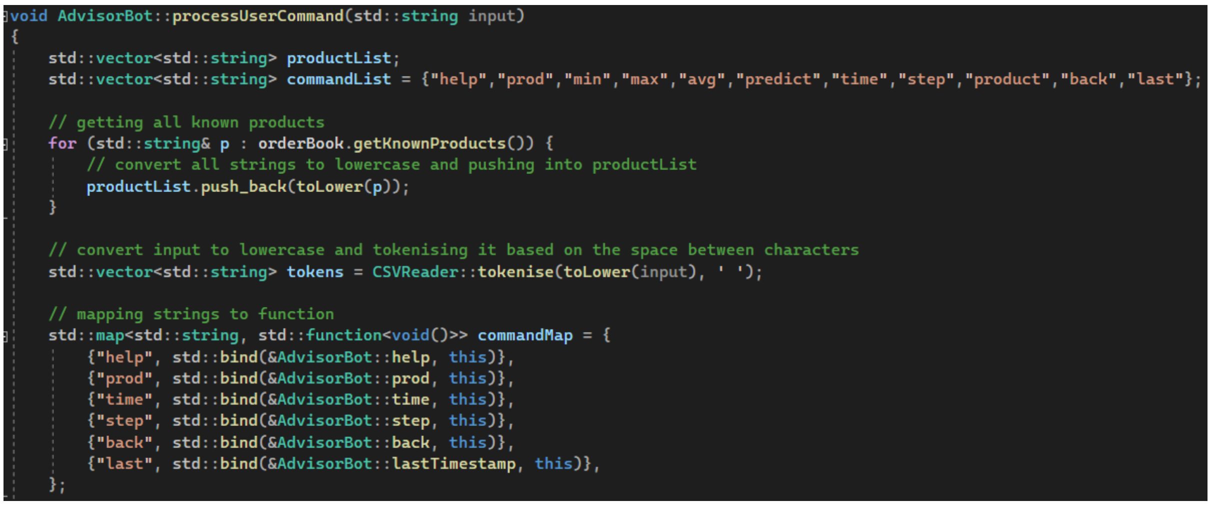
4.1. AdvisorBot::getUserInput



```
// this function was taken from the example code
std::string AdvisorBot::getUserCommand()
{
    std::string userOption;
    std::string line;

    // cin is used to read the commands from the command line and parsing it into a string which is then returned and used in processUserCommand()
    std::getline(std::cin, line);
    try {
        userOption = std::string(line);
    }
    catch (const std::exception& e) {

    }
    return userOption;
}
```

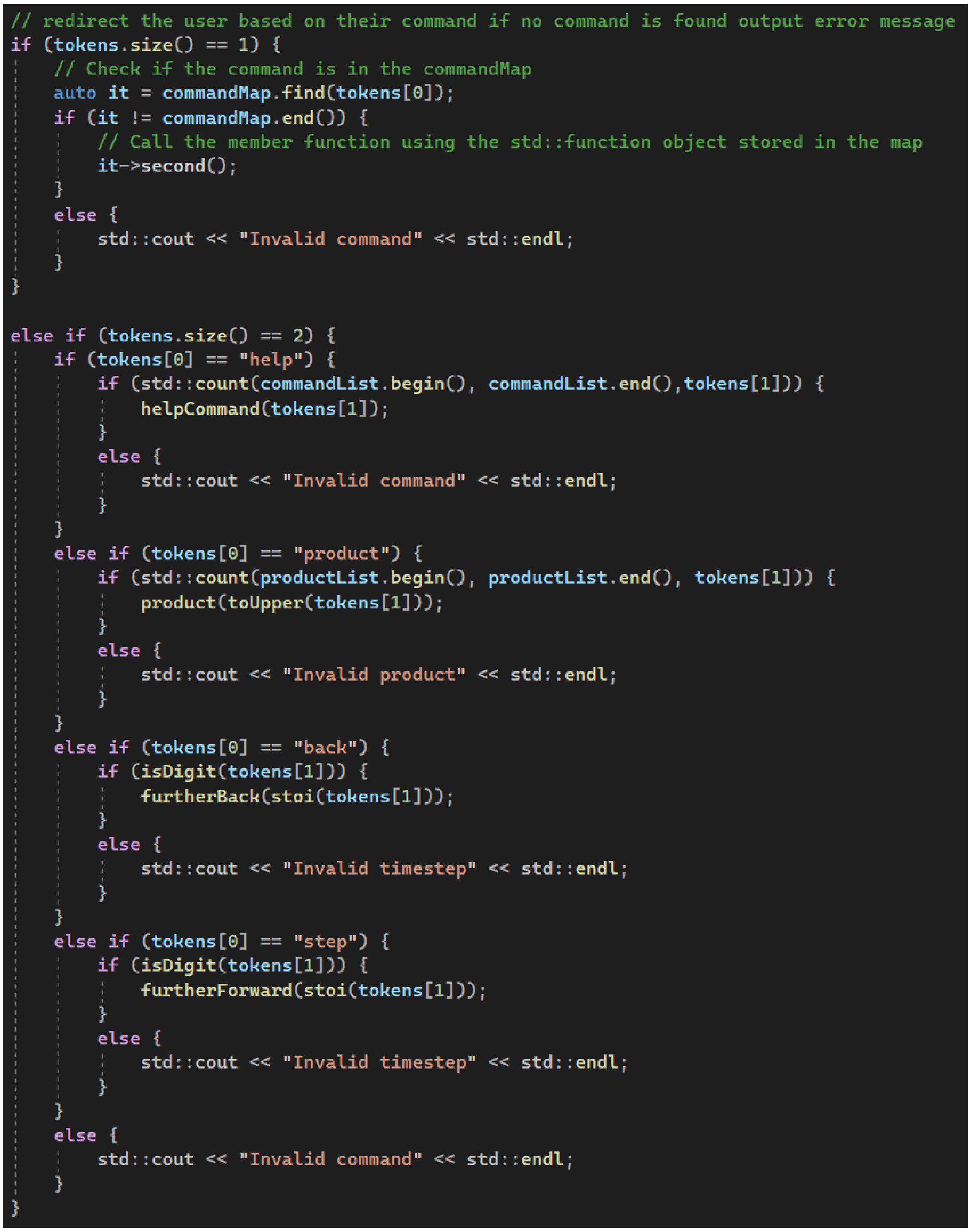
This function was taken from the base example code. Firstly, it initialises the strings userOption and line. Secondly, it uses std::getline() to get the input from the command line. Lastly, the input would be passed to userOption and returned to the user.

4.2. AdvisorBot::processUserCommand

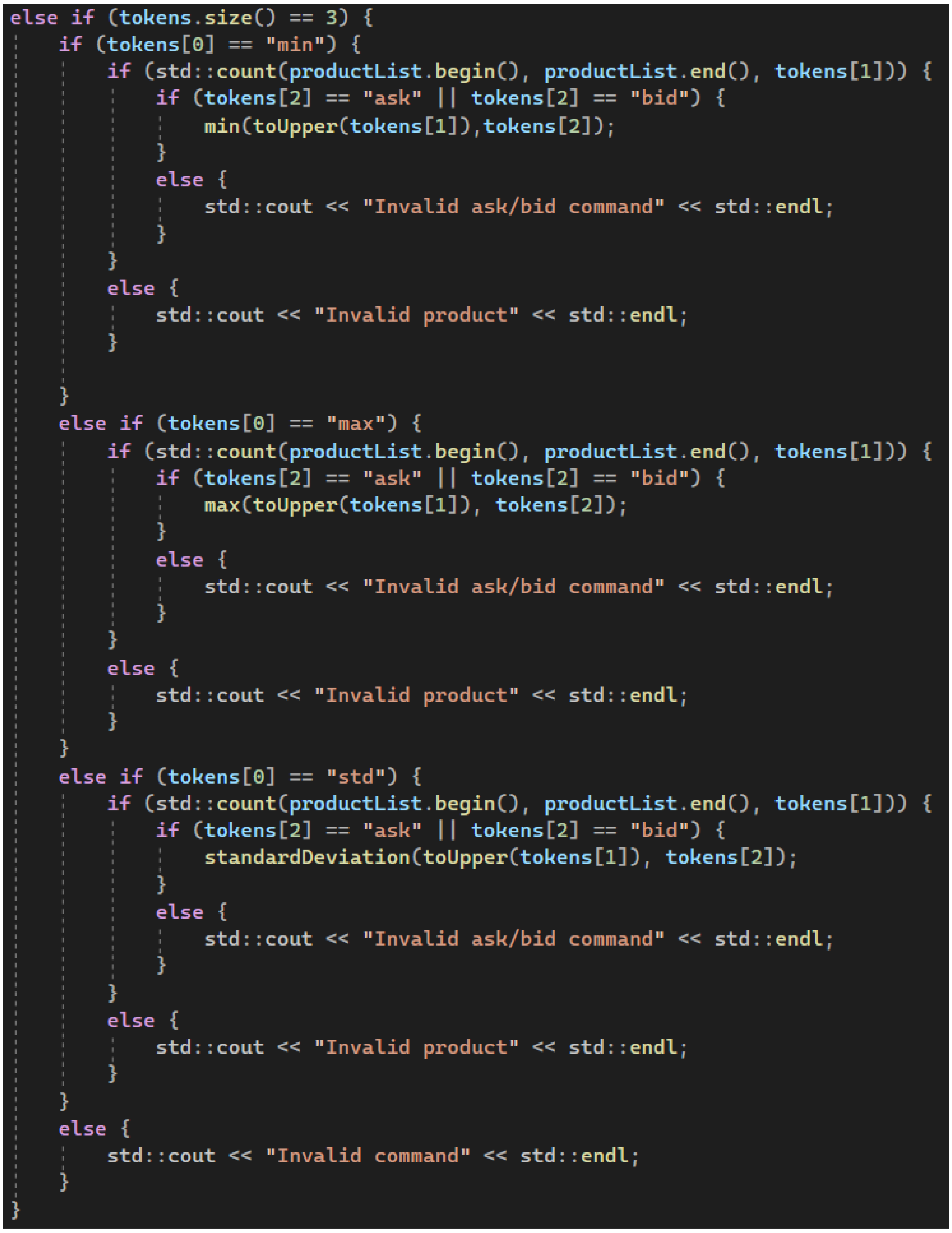


```
void AdvisorBot::processUserCommand(std::string input)
{
    std::vector<std::string> productList;
    std::vector<std::string> commandList = {"help","prod","min","max","avg","predict","time","step","product","back","last"};

    // getting all known products
    for (std::string& p : orderBook.getKnownProducts()) {
        // convert all strings to lowercase and pushing into productList
        productList.push_back(toLower(p));
    }

    // convert input to lowercase and tokenising it based on the space between characters
    std::vector<std::string> tokens = CSVReader::tokenise(toLower(input), ' ');

    // mapping strings to function
    std::map<std::string, std::function<void()>> commandMap = {
        {"help", std::bind(&AdvisorBot::help, this)},
        {"prod", std::bind(&AdvisorBot::prod, this)},
        {"time", std::bind(&AdvisorBot::time, this)},
        {"step", std::bind(&AdvisorBot::step, this)},
        {"back", std::bind(&AdvisorBot::back, this)},
        {"last", std::bind(&AdvisorBot::lastTimestamp, this)},
    };
```

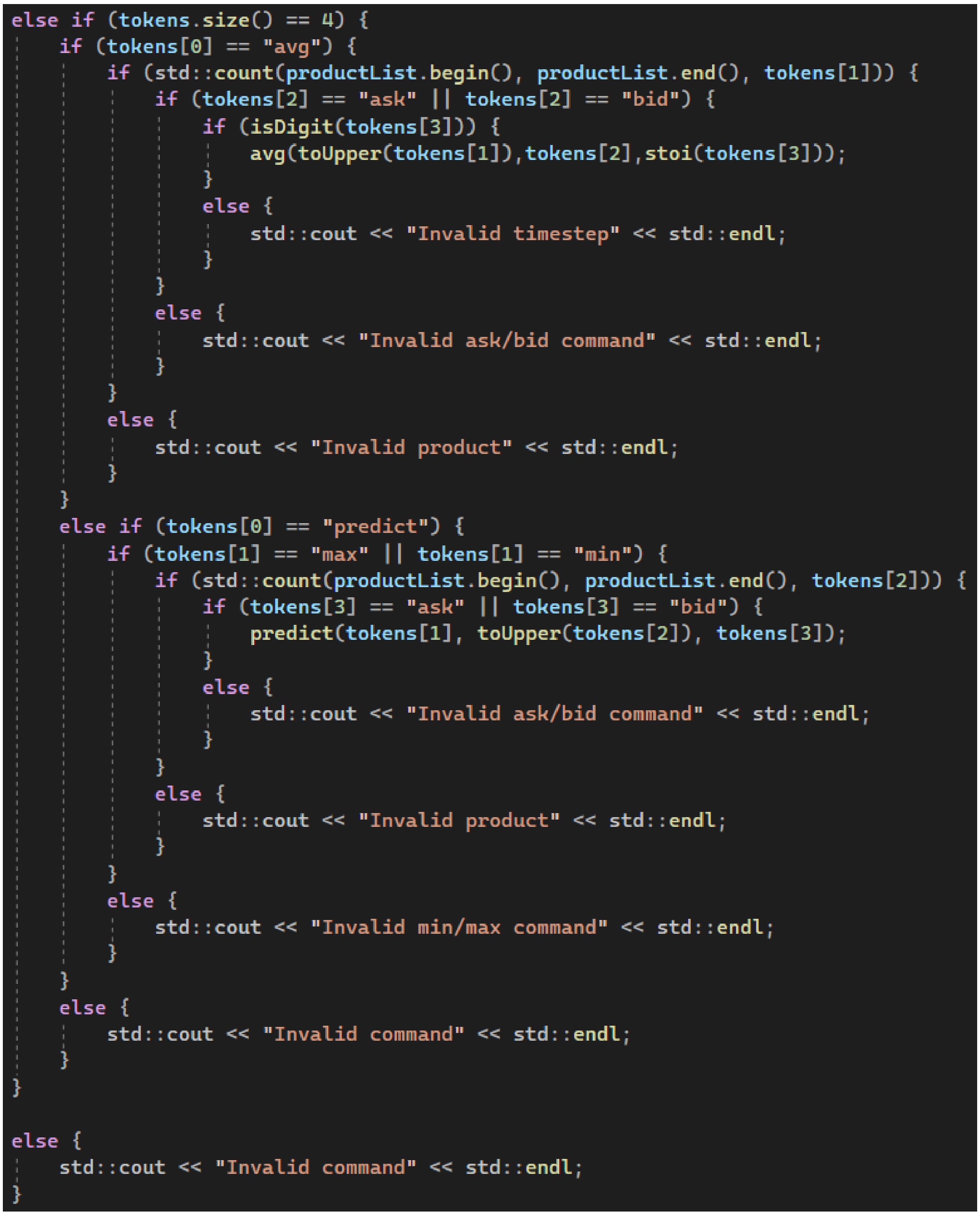
This vector of strings productList and commandList are initialised. Using the getKnownProducts() function all the products are converted into lowercase and pushed into the productList vector. Then the input from the user is also converted to lowercase and tokenised. I used the std::map to map commands with only 1 token which are help, prod, time, step, back, last, etc. I decided to convert all the strings to lowercase to case-insensitive input provided by the user. This would also help in redirecting the user as I used if/else to redirect the user and based on the case of the input it could return an error message instead of redirecting the user to their command.



```cpp
// redirect the user based on their command if no command is found output error message
if (tokens.size() == 1) {
    // Check if the command is in the commandMap
    auto it = commandMap.find(tokens[0]);
    if (it != commandMap.end()) {
        // Call the member function using the std::function object stored in the map
        it->second();
    }
    else {
        std::cout << "Invalid command" << std::endl;
    }
}

else if (tokens.size() == 2) {
    if (tokens[0] == "help") {
        if (std::count(commandList.begin(), commandList.end(),tokens[1])) {
            helpCommand(tokens[1]);
        }
        else {
            std::cout << "Invalid command" << std::endl;
        }
    }
    else if (tokens[0] == "product") {
        if (std::count(productList.begin(), productList.end(), tokens[1])) {
            product(toUpper(tokens[1]));
        }
        else {
            std::cout << "Invalid product" << std::endl;
        }
    }
    else if (tokens[0] == "back") {
        if (isDigit(tokens[1])) {
            furtherBack(stoi(tokens[1]));
        }
        else {
            std::cout << "Invalid timestep" << std::endl;
        }
    }
    else if (tokens[0] == "step") {
        if (isDigit(tokens[1])) {
            furtherForward(stoi(tokens[1]));
        }
        else {
            std::cout << "Invalid timestep" << std::endl;
        }
    }
    else {
        std::cout << "Invalid command" << std::endl;
    }
}
```



```cpp
else if (tokens.size() == 3) {
    if (tokens[0] == "min") {
        if (std::count(productList.begin(), productList.end(), tokens[1])) {
            if (tokens[2] == "ask" || tokens[2] == "bid") {
                min(toUpper(tokens[1]),tokens[2]);
            }
            else {
                std::cout << "Invalid ask/bid command" << std::endl;
            }
        }
        else {
            std::cout << "Invalid product" << std::endl;
        }

    }
    else if (tokens[0] == "max") {
        if (std::count(productList.begin(), productList.end(), tokens[1])) {
            if (tokens[2] == "ask" || tokens[2] == "bid") {
                max(toUpper(tokens[1]), tokens[2]);
            }
            else {
                std::cout << "Invalid ask/bid command" << std::endl;
            }
        }
        else {
            std::cout << "Invalid product" << std::endl;
        }
    }
    else if (tokens[0] == "std") {
        if (std::count(productList.begin(), productList.end(), tokens[1])) {
            if (tokens[2] == "ask" || tokens[2] == "bid") {
                standardDeviation(toUpper(tokens[1]), tokens[2]);
            }
            else {
                std::cout << "Invalid ask/bid command" << std::endl;
            }
        }
        else {
            std::cout << "Invalid product" << std::endl;
        }
    }
    else {
        std::cout << "Invalid command" << std::endl;
    }
}
```

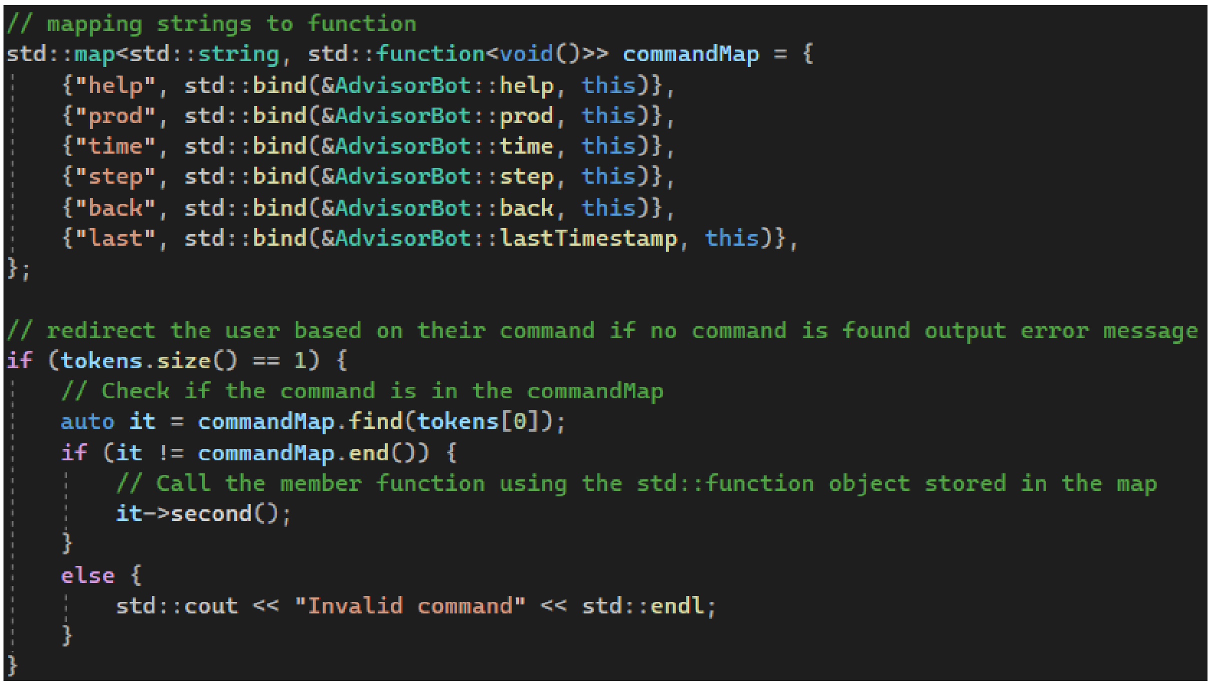


```
else if (tokens.size() == 4) {
    if (tokens[0] == "avg") {
        if (std::count(productList.begin(), productList.end(), tokens[1])) {
            if (tokens[2] == "ask" || tokens[2] == "bid") {
                if (isDigit(tokens[3])) {
                    avg(toUpper(tokens[1]),tokens[2],stoi(tokens[3]));
                }
                else {
                    std::cout << "Invalid timestep" << std::endl;
                }
            }
            else {
                std::cout << "Invalid ask/bid command" << std::endl;
            }
        }
        else {
            std::cout << "Invalid product" << std::endl;
        }
    }
    else if (tokens[0] == "predict") {
        if (tokens[1] == "max" || tokens[1] == "min") {
            if (std::count(productList.begin(), productList.end(), tokens[2])) {
                if (tokens[3] == "ask" || tokens[3] == "bid") {
                    predict(tokens[1], toUpper(tokens[2]), tokens[3]);
                }
                else {
                    std::cout << "Invalid ask/bid command" << std::endl;
                }
            }
            else {
                std::cout << "Invalid product" << std::endl;
            }
        }
        else {
            std::cout << "Invalid min/max command" << std::endl;
        }
    }
    else {
        std::cout << "Invalid command" << std::endl;
    }
}

else {
    std::cout << "Invalid command" << std::endl;
}
```

These are all processUserCommand codes that will redirect the user based on the commands inputted into the command line. I decided to use if/else statements as it was the easiest to implement and edit. I started the if/else statement based on their token size. However, if there is only 1 token, I used the commandmap.find() to make use of my commandMap. If there are more than 1 tokens I used the standard if/else to continue to code. By understanding what input each command needs I can successfully redirect the user based on the tokens provided by the user. I did all the checks before passing the input into the command to prevent any unnecessary redirection to the command. For commands which require checking on more than 2 strings such as command and product. I used the std::count to check whether the input provided by the user can be found inside the vector of strings I created. Before redirecting the user to the command, I would convert the tokens to their appropriate data types. For the product, I would convert it back into uppercase and for the commands which require an integer, I would stoi the string to convert it into an integer and pass those converted values into the function parameter.

5. Optimising the exchange code

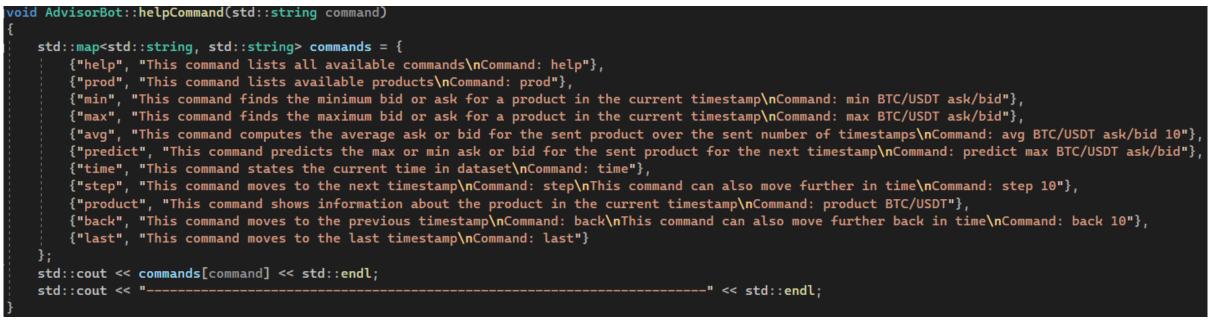
5.1. AdvisorBot::processUserCommand



```
// mapping strings to function
std::map<std::string, std::function<void()>> commandMap = {
    {"help", std::bind(&AdvisorBot::help, this)},
    {"prod", std::bind(&AdvisorBot::prod, this)},
    {"time", std::bind(&AdvisorBot::time, this)},
    {"step", std::bind(&AdvisorBot::step, this)},
    {"back", std::bind(&AdvisorBot::back, this)},
    {"last", std::bind(&AdvisorBot::lastTimestamp, this)},
};

// redirect the user based on their command if no command is found output error message
if (tokens.size() == 1) {
    // Check if the command is in the commandMap
    auto it = commandMap.find(tokens[0]);
    if (it != commandMap.end()) {
        // Call the member function using the std::function object stored in the map
        it->second();
    }
    else {
        std::cout << "Invalid command" << std::endl;
    }
}
```

Instead of using the standard if else to redirect the user to the command, I decided to use std::map to store a string and a function. However, I only used std::map for commands which have only 1 token. However, as compared to before, there were many if/else removed from the code which make it more readable and optimized. This can clearly be seen in the screenshot above where the if/else statement has been reduced to 6 lines.

5.2. AdvisorBot::helpCommand



```cpp
void AdvisorBot::helpCommand(std::string command)
{
    std::map<std::string, std::string> commands = {
        {"help", "This command lists all available commands\nCommand: help"},
        {"prod", "This command lists available products\nCommand: prod"},
        {"min", "This command finds the minimum bid or ask for a product in the current timestamp\nCommand: min BTC/USDT ask/bid"},
        {"max", "This command finds the maximum bid or ask for a product in the current timestamp\nCommand: max BTC/USDT ask/bid"},
        {"avg", "This command computes the average ask or bid for the sent product over the sent number of timestamps\nCommand: avg BTC/USDT ask/bid 10"},
        {"predict", "This command predicts the max or min ask or bid for the sent product for the next timestamp\nCommand: predict max BTC/USDT ask/bid"},
        {"time", "This command states the current time in dataset\nCommand: time"},
        {"step", "This command moves to the next timestamp\nCommand: step\nThis command can also move further in time\nCommand: step 10"},
        {"product", "This command shows information about the product in the current timestamp\nCommand: product BTC/USDT"},
        {"back", "This command moves to the previous timestamp\nCommand: back\nThis command can also move further back in time\nCommand: back 10"},
        {"last", "This command moves to the last timestamp\nCommand: last"}
    };
    std::cout << commands[command] << std::endl;
    std::cout << "------------------------------------------------------------------------" << std::endl;
}
```

The std::map is also used in the helpCommand function. This also greatly reduces the number of codes needed for this command in AdvisorBot. I used std::map to map 2 strings whereby the first string would be the function and the second would be the explanation of it. Once I was done mapping the function and its explanation, I just had to compare the command taken in by the parameter and those commands to print out the explanation.