University of London

BSc Computer Science

Object Oriented Programming

Midterm Coursework: Advisorbot

Report

Introduction

The purpose of Advisorbot is to analyze the cryptocurrency exchange market. Advisorbot can list all products, report the minim and maximum asks or bids on a product, it can also calculate the average price of a product for a specific set of timeframes. It can predict the movement of a product's minimum or maximum price based on a moving average, tell which timeframe the user is on, move to the next timeframe and provide information on the best and worst growing bids and asks (custom).

Core Functions

Command	Type	Purpose (brief)	Successfully Implemented
Help	Base	Lists all available commands	Yes
Help <cmd>	Base	Provides information on a particular command	Yes
Prod	Base	Lists all products	Yes
min <prod> <ask/bid>	Base	Prints the minimum price of a product's ask or bid in this timeframe	Yes
max <prod> <ask/bid>	Base	Prints the maximum price of a product's ask or bid in this timeframe	Yes
avg <prod> <ask/bid> <steps>	Base	Prints the average price of a product's ask or bid for a specific set of past timeframes	Yes
predict <max/min> <product> <ask/bid>	Base	Predict max or min ask or bid for the sent product for the next time step	Yes
time	Base	Prints the current time in the dataset	Yes
step	Base	Moves to the next time step.	Yes
track	Custom	Shows the best and worst growing products for asks and bids	Yes

Input Parsing

Each command inputted by the user is taken in as a string and tokenized into a string vector called ***tokens***. There are several stages of evaluation for each command:

1. Evaluate whether ***tokens[0]*** corresponds to any existing command's first argument(word).
2. See if the size of ***tokens*** matches the size required for this command. A special exception is made for the "help" and "help <cmd>" commands as the latter is made with function overloading, therefore the same result for ***tokens[0]*** corresponds to 2 different functions, so the program evaluates 2 possible sizes.
3. Next, for each function's argument k, where k>=1, ***tokens[k]*** is evaluated under one of three specific conditions:
    a. If argument is a product, we see whether ***tokens[k]*** corresponds to a valid product. This is achieved using the ***GetKnownProducts()*** static function of the ***OrderBook*** class.
    b. If argument is a bid or ask, a check is made using the ***CSVReader***'s static function ***stringToOBT()*** to validate the order book type (a modification was made to reduce OBTs to ask, bid and unknown) . If ***tokens[k]*** does not correspond to either ask or bid, ***stringToOBT()*** validates it as *unknown* and no function accepts *unknown* as a valid input.
    c. If ***tokens[k]*** must be parsed into an integer, a try/catch block is implemented using the corresponding standard library function – ***stoi()***. The catch block prevents a crash or further execution of the code.

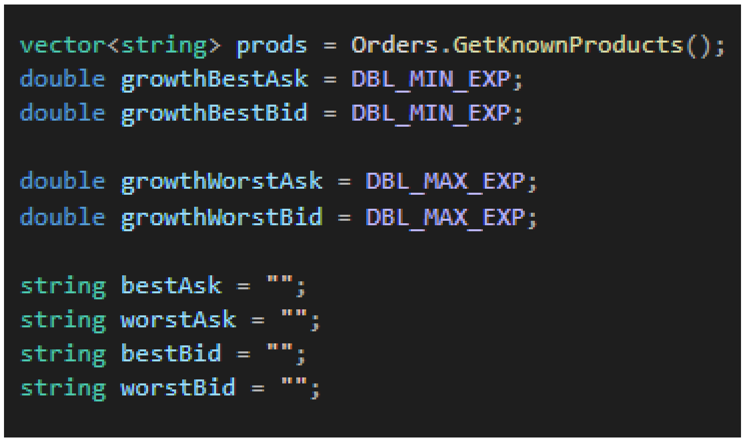
Thanks to the parsing evaluation measures, no faulty inputs pass. Furthermore, there are specific messages to inform the user on their mistake – either an invalid input size, command, OBT, product or integer. All if/else and try/catch blocks used for evaluation simply return void and take the user back to be prompted for their input, avoiding any crashes.

Custom Command

The custom command I implemented for my Advisorbot is "track". Track can show the best and worst products in terms of growth. It displays both the highest and lowest growing bids and asks through their growth percentage.

Implementation:

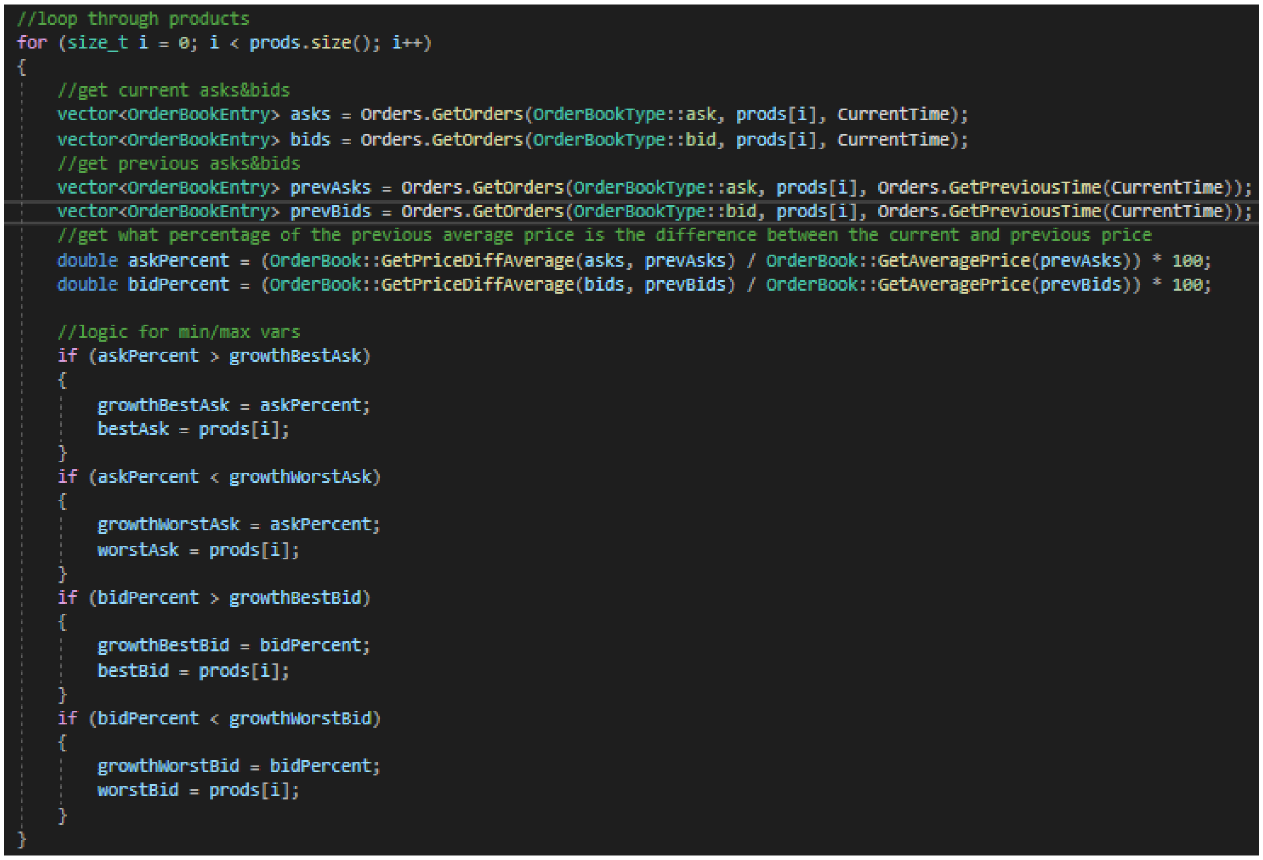
1. Firstly, some starting variables are initialized:



```
vector<string> prods = Orders.GetKnownProducts();
double growthBestAsk = DBL_MIN_EXP;
double growthBestBid = DBL_MIN_EXP;

double growthWorstAsk = DBL_MAX_EXP;
double growthWorstBid = DBL_MAX_EXP;

string bestAsk = "";
string worstAsk = "";
string bestBid = "";
string worstBid = "";
```

   a. prods – a string vector consisting of all known products
   b. growthBest<Ask/Bid> - a double set to the lowest possible value for a double.
   c. growthWorst<Ask/Bid> - a double set to the highest possible value for a double.
   d. <best/worst><Ask/Bid> - empty strings to contain the product names for the best and worst asks and bids.

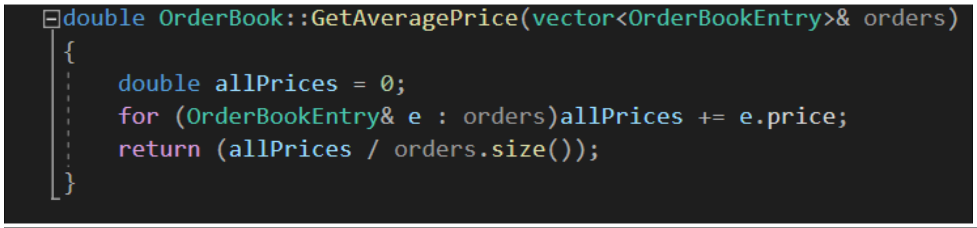
2. Next, a for loop will loop through each product in the prods vector.



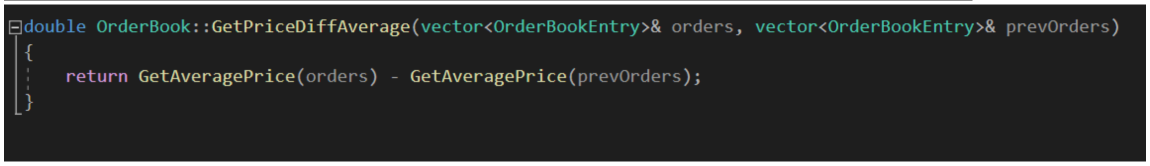
```
//loop through products
for (size_t i = 0; i < prods.size(); i++)
{
    //get current asks&bids
    vector<OrderBookEntry> asks = Orders.GetOrders(OrderBookType::ask, prods[i], CurrentTime);
    vector<OrderBookEntry> bids = Orders.GetOrders(OrderBookType::bid, prods[i], CurrentTime);
    //get previous asks&bids
    vector<OrderBookEntry> prevAsks = Orders.GetOrders(OrderBookType::ask, prods[i], Orders.GetPreviousTime(CurrentTime));
    vector<OrderBookEntry> prevBids = Orders.GetOrders(OrderBookType::bid, prods[i], Orders.GetPreviousTime(CurrentTime));
    //get what percentage of the previous average price is the difference between the current and previous price
    double askPercent = (OrderBook::GetPriceDiffAverage(asks, prevAsks) / OrderBook::GetAveragePrice(prevAsks)) * 100;
    double bidPercent = (OrderBook::GetPriceDiffAverage(bids, prevBids) / OrderBook::GetAveragePrice(prevBids)) * 100;

    //logic for min/max vars
    if (askPercent > growthBestAsk)
    {
        growthBestAsk = askPercent;
        bestAsk = prods[i];
    }
    if (askPercent < growthWorstAsk)
    {
        growthWorstAsk = askPercent;
        worstAsk = prods[i];
    }
    if (bidPercent > growthBestBid)
    {
        growthBestBid = bidPercent;
        bestBid = prods[i];
    }
    if (bidPercent < growthWorstBid)
    {
        growthWorstBid = bidPercent;
        worstBid = prods[i];
    }
}
```

   a. 4 OBE vectors are made – they all take prods[i] as their product, prev/asks take ask and prev/bids take bid as their OBT, asks and bids use current time, while prevAsks and prevBids use the previous timeframe.
   b. askPercent and bidPercent are doubles made with a simple logic. The goal is to show what(x) percentage of previous asks/bids (p) is the difference (d). Or:
      i. (x%)*p = d
      ii. (x / 100) * p = d
      iii. x/100 = d / p
      iv. x = (d / p) * 100
   c. The final part of the for loop is comparison logic to see if our current percentages are better than the best growths or worse than the worst growths and record the doubles and products strings. The initial values of our best/worst growths are set so any <ask/bid>Percent will overwrite them (in the context of maximum/minimum growths I found when testing data).
   d. Notes:
      “OrderBook::GetPriceDiffAverage()” and “OrderBook::GetPriceDiffAverage()” are

functions I made:



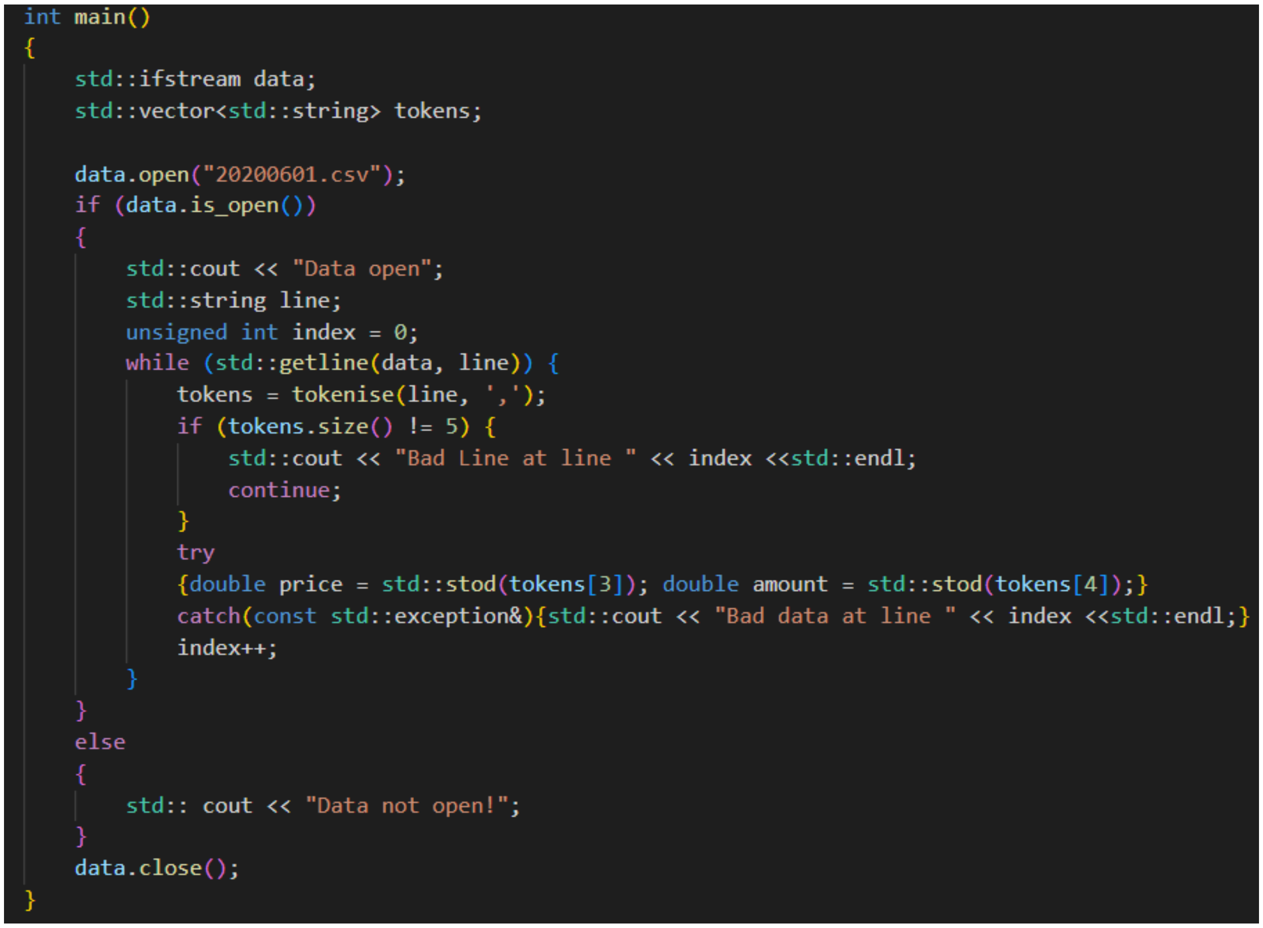
```
double OrderBook::GetAveragePrice(vector<OrderBookEntry>& orders)
{
    double allPrices = 0;
    for (OrderBookEntry& e : orders)allPrices += e.price;
    return (allPrices / orders.size());
}
```



```
double OrderBook::GetPriceDiffAverage(vector<OrderBookEntry>& orders, vector<OrderBookEntry>& prevOrders)
{
    return GetAveragePrice(orders) - GetAveragePrice(prevOrders);
}
```

Optimization

What was a blow to the execution time was the sheer data size. At 60MB+ with over a million entries, the dataset used to take 2+ minutes to load. What I used was to search online for ways to optimize input file stream parsing and a solution I came by on accident was that try/catch blocks are very costly. What I did was use the "TokeniserTest.CPP" (used during the MerkelrEx courses) to set up a test for data validity:



```
int main()
{
    std::ifstream data;
    std::vector<std::string> tokens;

    data.open("20200601.csv");
    if (data.is_open())
    {
        std::cout << "Data open";
        std::string line;
        unsigned int index = 0;
        while (std::getline(data, line)) {
            tokens = tokenise(line, ',');
            if (tokens.size() != 5) {
                std::cout << "Bad Line at line " << index <<std::endl;
                continue;
            }
            try
            {double price = std::stod(tokens[3]); double amount = std::stod(tokens[4]);}
            catch(const std::exception&){std::cout << "Bad data at line " << index <<std::endl;}
            index++;
        }
    }
    else
    {
        std:: cout << "Data not open!";
    }
    data.close();
}
```

When all data was checked no bad data was present, therefore I removed the try/catch blocks. Loading time was reduced to roughly 30 seconds. Furthermore, while code was borrowed from MerkelrEx, it was heavily edited to be lighter and only what was useful remained or was added (excluding main function):

1. OrderBookEntry now only has:
    a. OrderBookType – ask, bid, uknown.
    b. Price, amount, timestamp, product fields.
    c. Constructor function (without username)
2. CSVReader remains roughly the same, however the public stringsToOBE() function was removed.
3. OrderBook contains:
    a. Same constructor function and orders field.
    b. Get<Previous/Nex/Earilest>Time and Get<High/Low>Price functions remain roughly the same.
    c. GetAveragePrice() and GetPriceDiffAverage() are my own functions.
4. LineArt is a class I made for quickly writing out console messages with aesthetics such as dash/equals sign borders and endlines.

Note: any function or code snippets from a function referring to sells or users or updating the dataset was removed from MerkelrEx classes and all of their functions, which is why Wallet was not used as a class.