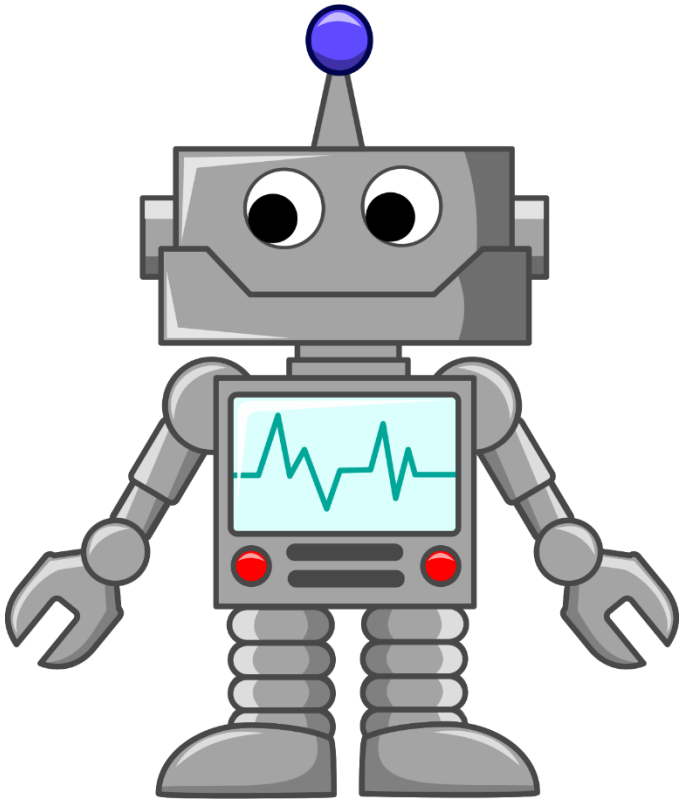

ADVISOR BOT

Your trusty stock market assistant to help you make the right trading decisions

210170537

Object Oriented Programming
26/06/2022

Introduction

The AdvisorBot is a stock trading assistant that gives the user data about the stock market over the current timeframe or a set of timeframes. It allows the user to quickly see important trading information about stock data. It is an adaptation of the MerkelMain project and utilizes some of the same functions while also implementing other interesting informative functions with the sole purpose of providing a user with clarity on a trading environment so that informed decisions can be made when bidding and buying. Some of the new functionalities include seeing all the products being sold and bought in a time step, finding the min, max and average bids or buys over a set of time steps, predicting the min/max ask/bid for a specified product for the next time step, showing the current time, stepping over one or more time steps and calculating the standard deviation for the current time step. This program requires C++17, because of specific commands that are used that are only available in this version of C++.

Table of Commands Implemented

Below is a table with the names of the functions I was able to implement into the AdvisorBot:

	Command Name	**Full Name**	**Implemented (True or False)**
C1	HELP	Help	True
C2	HELP CMD	Help + Command	True
C3	PROD	Products	True
C4	MIN	Minimum	True
C5	MAX	Maximum	True
C6	AVG	Average	True
C7	PREDICT	Prediction	True
C8	TIME	Current Time	True
C9	STEP	Time Step	True
C10	STD	Standard Deviation	True

Command Parsing Code

It is important to validate user input before using it to directly parse a database. This helps prevent the program from short dumping and producing incorrect outputs. This application does not use any user input filters, thus all the validation of user entered data must be validated inside the code to ensure that the expected results are produced.

Tokenizing commands and converting to uppercase

This function is used to take the input data, tokenise it and then convert it to uppercase so that the correct if statements can be triggered.

```
void MerkelMain::processUserOption()
{  // changed this to work with Advisorbot strings
    std::string userInput;
    std::getline( &: std::cin,  &: userInput);
    std::vector<std::string> tokenInput = CSVReader::tokenise( csvLine: userInput,  separator: ' ');

    // convert user input to uppercase
    // The std::size function is only available for C++ 17 and higher
    for (int i = 0; i < std::size( cont: tokenInput); i++)
    {
        std::for_each( first: tokenInput[i].begin(),  last: tokenInput[i].end(),  f: [](char &c) -> void
                    { c = ::toupper( C: c); });
    }
```

This function uses tokenise to separate the user input variables into separate tokens within a vector. This separate inputs can be used to find entries in the database at later stages in the program. There is a loop here that coverts all tokens to uppercase so that the input is not case sensitive.

```
if ((tokenInput[0] == "HELP") && tokenInput.size() == 1)
{
    printHelp();
    return;
}
if ((tokenInput[0] == "HELP") && tokenInput.size() == 2)
{
    printHelpCmd( &: tokenInput[1]);
    return;
}
if ((tokenInput[0] == "PROD") && tokenInput.size() == 1)
{
    printMarketStats();
    return;
}
if ((tokenInput[0] == "MIN") && tokenInput.size() == 3)
{
    // convert all "ask" and "bid" to lower case so they match the database
    std::for_each( first: tokenInput[2].begin(),  last: tokenInput[2].end(),  f: [](char &c) -> void
                { c = ::tolower( C: c); });
    std::cout << tokenInput[2] << std::endl;
    printMin(tokenInput);
    return;
}
if ((tokenInput[0] == "MAX") && tokenInput.size() == 3)
{
    // convert all "ask" and "bid" to lower case so they match the database
    std::for_each( first: tokenInput[2].begin(),  last: tokenInput[2].end(),  f: [](char &c) -> void
                { c = ::tolower( C: c); });
    printMax( tokens: tokenInput);
    return;
}
```

The conversion to uppercase allows the “if” statements to be triggered regardless of the case used by users. There is also a section of as highlighted by the red box that converts all bid and ask strings to lowercase when they are expected, the database contains only lowercase bid and ask strings so this allows the database to be parsed easier.

How data types are converted from tokens

Throughout the code the tokens that are sent to function are converted to the correct data types so that they can be used to find the correct input in the database.

When an order book type (ask or bid) is used in a command. This code is then used to determine whether the order type exists in the OrderBookEntry enum. If it is not an error message is sent and the code exits the function.

```
// now that we can see the product exists, we can assign the order type to a variable
orderType = OrderBookEntry::stringToOrderBookType( s: tokenInput[2]);
// check if the order type is one of the known types
if (orderType == OrderBookType::unknown)
{
    std::cout << "MerkelMain::printMin unknowns order orderType" << std::endl;
    return;
}
```

In functions where product is used in the command, this is assigned to a string variable as such:

```
void MerkelMain::printMin(const std::vector<std::string> &tokenInput)
{
    //Print the minimum amount for either ask or bid for the current timestep
    std::string product = tokenInput[1];
```

When searching the database in these types of function, if no entries match the product that was mentioned an informative error will show as such:

```
// saving all match with parameters orders
entries = orderBook.getOrders( type: orderType, product, timestamp: currentTime);
if (entries.empty())
{
    std::cout << "MerkelMain::printMin No matching orders found" << std::endl;
    return;
}
```

If a number of timesteps are specified in the command then a try catch block is triggered where the string is converted to an integer, if the string is not able to be converted an exception is thrown as such:

```
// assigning value to "numberOfTimeStamps" variable
try
{
    numberOfTimeStamps = std::stoi( str: tokenInput[3]);
}
catch (const std::exception &e)
{
    std::cout << "MerkelMain::printAvg Bad integer " << tokenInput[3] << std::endl;
    return;
}
```

Custom Commands

The custom command that was implemented is the STD command. This aims to calculate the standard deviation of either a bid or an ask for a specific product for the current time step. I also implemented more functionality to the STEP command, where the user can now step over multiple time steps instead of just one.

STD Command

```
456  void MerkelMain::printSTD(const std::vector<std::string> &tokenInput) {
457
458      OrderBookType orderType;
459      std::string product = tokenInput[1];
460
461      orderType = OrderBookEntry::stringToOrderBookType( s: tokenInput[2]);
462      if (orderType == OrderBookType::unknown) {
463          std::cout << "MerkelMain::printSTD unknowns order orderType" << std::endl;
464          return;
465      }
466
467          std::string timeStep = currentTime;
468          std::vector<OrderBookEntry> entries = orderBook.getOrders( type: orderType, product, timestamp: timeStep);
469
470          // print if orders are not found on current timestamp
471          if (entries.empty()) {
472              std::cout << "Orders aren't found at " << timeStep << " timestamp" << std::endl;
473              return;
474          }
475          //Get the average amount for the current timestep
476          double totalAvg = OrderBook::getAvgPrice( &: entries);
477          float mean = totalAvg;
478          float standardDeviation = 0.0;
479
480      for(int i = 0; i < entries.size(); i++)
481      {
482          standardDeviation += pow( x: entries[i].amount - mean, y: 2);
483      }
484
485      std::cout << "The standard deviation is " << sqrt( x: standardDeviation / entries.size()) << " for the current timestep" << std::endl;
486      }
```

This command accepts an input in the format STD product bid/ask. For example STD ETH/BTC ask is an accepted command. This assigns the correct tokens that are parsed from the command in ProcessUserInput( ) into the correct type variables. This also implements a check that the ordertype exists. The current time step is then assigned to the variable timestep. The orders are then retrieved using the tokens as search criteria. The entries are then used to get the average price using the getAvgPrice( ) function within the OrderBook class. A loop is then used to iterate over the values in the entries vector and the cmath library is used to calculate standard deviation. This value is then square rooted to get the standard deviation value which is outputted to the terminal.

STEP enhancement

STEP has been modified to accept an extra 2 parameters. This means that STEP can move forwards and backwards and also can jump over several time steps. The accepted input for the STEP command is either STEP FORWARD/BACK, or STEP FORWARD/BACK 10. For example STEP FORWARD, will move to the next time step, STEP FORWARD 10 will move 10 time steps forward.

The BACK functionality is implemented using the goToPrevTimeframe( ) function in the MerkelMain class

```
void MerkelMain::gotoPrevTimeframe(int timeStep)
{
    //This allows users to jump to the previous time frame and skip timeframes if needed
    std::cout << "Going to the previous time frame." << std::endl;
    for(int i = 0; i < timeStep; i++ )
    {
        currentTime = orderBook.getPrevTime( timestamp: currentTime);
    }
    if (currentTime.empty())
    {
        std::cout << "MerkelMain::gotoNextTimeframe reached the first timestamp" << std::endl;
        std::cout << "Current time will be set to the last timestamp" << std::endl;
        currentTime = orderBook.setCurrentTime( a: orderBook.getEntriesSize());
    }
    std::cout << "new timestamp: " << currentTime << std::endl;
}

```

It iterates over a loop to change the timeframe according to the number of time steps specified. If no time steps are specified, this is defaulted to one. It also takes into account if the timeframes have run out and gives an informative message. The new function implemented in orderBook looks as follows:

```
std::string OrderBook::getPrevTime(std::string timestamp)
{
    std::string next_timestamp = "";
    for (OrderBookEntry &e : orders)
    {
        if (e.timestamp >= timestamp)
        {
            break;
        }
        else
        {
            next_timestamp = e.timestamp;
        }
    }
    return next_timestamp;
}
```

The FORWARD functionality works exactly the same as the BACK functionality, except this moves the time step forward and if the last time step is reached then this loops back to the first timestamp, this also prints out the number of sales that occurred over the time steps. This is done in the matchAsksToBids( ) method in the orderBook class (this was enhanced for the project):

```cpp
void MerkelMain::gotoNextTimeframe(int timeStep)
{
    /*
     * Since I have changed OrderBook::matchAsksToBids function so it returns sales of all products and also changed
     * how MerkelMain::gotoNextTimeframe prints them
     */
    /*
     * And to make it more clear for the user, I changed OrderBook::getNextTime so MerkelMain::gotoNextTimeframe loop
     * to the first timestamp if it reaches the final timestamp
     */
    std::cout << "Going to the next time frame. " << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids( timestamp: currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;

    std::string currentProduct = "";
    for (OrderBookEntry &e : sales)
    {
        if (currentProduct != e.product)
        {
            currentProduct = e.product;
            std::cout << currentProduct << std::endl;
        }

        std::cout << "Sale price: " << e.price << " amount: " << e.amount << std::endl;
    }

    for(int i = 0; i < timeStep; i++ )
    {
        currentTime = orderBook.getNextTime( timestamp: currentTime);
    }

    if (currentTime.empty())
    {
        std::cout << "MerkelMain::gotoNextTimeframe reached the last timestamp" << std::endl;
        std::cout << "Current time will be set to the first timestamp" << std::endl;
        currentTime = orderBook.setCurrentTime( a: 1);
    }
    std::cout << "new timestamp: " << currentTime << std::endl;
}
```

Code Optimization

The code was optimized for the AdvisorBot by changing the matchAsksToBids( ) method in the orderBook function to find a match for every product instead of just one for each time step. It also calculates whether any sales were made for this time step by comparing bid and ask prices. If the bid price is greater than or equal to the ask price then a sale has been made and this variable is adjusted to account for the fact that the ask may be lower than the bid, the sales price is then adjusted accordingly.

```cpp
std::vector<OrderBookEntry> OrderBook::matchAsksToBids(std::string timestamp)
{
    // This has been changed to only take in the timestamp as an input
    /* This has been optimized to find matches for every product in the time step instead of only one *.
     */
    // it also finds all the sales for the current time step
    std::vector<OrderBookEntry> sales;

    // loop over all products
    std::vector<std::string> products = getKnownProducts( currentTime: timestamp);
    for (std::string &product : products)
    {
        // create a vector for all the asks in current time step
        std::vector<OrderBookEntry> asks = getOrders( type: OrderBookType::ask, product, timestamp);
        // create a vector for al bids in current time step
        std::vector<OrderBookEntry> bids = getOrders( type: OrderBookType::bid, product, timestamp);

        // sorts the asks and bids so that iteration is effective
        std::sort( first: asks.begin(),  last: asks.end(),  comp: OrderBookEntry::compareByPriceAsc);
        std::sort( first: bids.begin(),  last: bids.end(),  comp: OrderBookEntry::compareByPriceDesc);

        // loop over all asks and bids to find matches
        for (OrderBookEntry &ask : asks)
        {
            if (ask.amount == 0)
            {
                continue;
            }
            for (OrderBookEntry &bid : bids)
            {
                if (bid.price >= ask.price)
                {
                    if (bid.amount == 0)
                    {
                        continue;
                    }
                    //creating the sale of a product because the sale criteria is met
                    OrderBookEntry sale{ price: ask.price,  amount: 0, timestamp, product,  orderType: OrderBookType::asksale};
```

```cpp
                    OrderBookEntry sale{ price: ask.price,  amount: 0, timestamp, product,  orderType: OrderBookType::asksale};

                    //adjusting the sale amount according to the actual amount of the product
                    sale.amount = std::min(bid.amount, ask.amount);
                    ask.amount -= sale.amount;
                    bid.amount -= sale.amount;

                    //Adding this to the database
                    sales.push_back(sale);
                }
            }
        }
    }

    return sales;
}
```