Midterm Report

Introduction:

My program, AdvisorBot, is a Cryptocurrency Exchange Platform Simulator. It allows for users, especially those new to cryptocurrency, to explore the realm of asset trading in a safe, risk free environment.

Commands Implemented:

Command	Implementation Success
help	Yes
help cmd	Yes
prod	Yes
min	Yes
max	Yes
avg	Yes
predict	Yes
time	Yes
step	Yes
reset (own)	Yes
wallet	Yes
bid	Yes
ask	Yes

What code is mine:

What I used directly from MerkelRex:

- enterAsk()
- enterBid()
- printWallet()
- getUserOption()

- The CSVReader Object Class
- The OrderBook Object Class
- The OrderBookEntry Object Class
- The Wallet Object Class

As Prof. Sean would say, “There’s no need to reinvent the wheel.”

What Is unique to me:

- printMenu()
- printHelp()
- goToNextTimeframe()
- processUserOption()
- getMinPrice() - reinvented in AdvisorBot
- getMaxPrice() - reinvented in AdvisorBot
- resetOrderBook()

Furthermore, regarding the code that I used directly from the MerkelRex app, I needed to implement changes into the code to adapt it to the vision of AdvisorBot, so it may not be identical to that of MerkelRex’s.

Command Parsing Code

My command parsing code works in a few different steps:

1. Takes the user input, and tokenises it.
2. Once tokenised, it counts how many tokens are present (indicating what course of action to take).
3. If there is one token, either the user typed 1 of 8 commands, help, prod, time, step, reset, wallet, bid or ask.
4. If there are two tokens, the user entered a help command, either help min, help max, help predict, help avg, help bid or help ask. For each of these, the function would print a console log with an example of how the specified function would be put to use.

5. If there are three tokens, the user entered either a min or max command.
6. If there are four tokens, the user either entered an avg command or a predict command.

The code would use if-statements to form a tree-like structure of pathways that I would check the tokens to determine whether they were recognisable commands and whether they were used in the right context.

Regarding the conversion from input to appropriate data types, my function would:

- Use stringToOrderBookType to convert a string to an order type.
- Use functions like std::stoi(....) to convert strings to integers.
- Use stringsToOBE(...) to convert string values into an OrderBookEntry Object.

Custom Command

The custom command that I implemented was simply a timeframe reset command. It would call onto the function - getEarliestTime(), to fetch the earliest existing time in the orderbook. It would then assign that time to the currentTime variable, which in turn would reset the timeframe.

Code Optimisation

To optimise my code, I was sure to implement a few techniques:

1. Using references when parsing over values to prevent the code from creating copy variables that take up space of their own.
2. Refrained from creating unnecessary variables.
3. I used built in function where possible, instead of creating my own. For example, std::stod(...) and std::stoi(...).

```
double price, amount;
try {
    price = std::stod(priceString);
    amount = std::stod(amountString);
}catch(const std::exception& e){
    std::cout << "CSVReader::stringsToOBE Bad float! " << priceString<< std::endl;
    std::cout << "CSVReader::stringsToOBE Bad float! " << amountString<< std::endl;
    throw;
}
```

4. When using If Then statements that contain multiple conditions, say p OR q, I listed the most likely condition first, which prevented the program from wasting unnecessary time running checking unlikely conditions. For example, in the user input parsing code, it is more likely that the user will make an error when entering a currency pair (ETH/BTC) as opposed to entering an order type (bid/ask), so I put the currency pair validating condition first.

```
// Checking that all inputs are valid
if (std::find(prodVec.begin(), prodVec.end(), tokens[1]) != prodVec.end() && OrderBookEntry::stringToOrderBookType(tokens[2]) != OrderBookType::unknown)
{
    currPair = tokens[1];
    // Creating orderbook type
    orderType = OrderBookEntry::stringToOrderBookType(tokens[2]);
    // Getting amount of timesteps to take
    timesteps = std::stoi(tokens[3]);
    // Calculating average over given timesteps
    avg = AdvisorBot::getAvgPrice(currPair, orderType, currentTime, timesteps);
    std::cout << "The average price " << currPair << " for the last " << timesteps << " timesteps is " << avg << std::endl;

}
else {
    std::cout << "Invalid Currency Pair or Order Type. Example: avg ETH/BTC ask 10" << std::endl;
}
```