“An introduction describing the purpose of your AdvisorBot”

Advisorbot is a command-line program that helps a cryptocurrency investor analyze data available on an exchange. The user can enter commands to see a list of available products, print the minimum or maximum value of a product, print the average value of a product over a given number of timesteps, predict the next maximum or minimum value of a product, print the current time, move to the next timestep, or print a list of current orders. The user can also enter the "help" command to see a list of available commands or get help information for a specific command. The user can enter the "exit" command to terminate the program gracefully. Advisorbot provides the user with a convenient way to access and analyze product data in a command-line interface.

“A table reporting which commands you were able to implement and which you were not able to implement”

Command	Implementation Status
help	Success
help cmd	Success
prod	Success
min	Success
max	Success
avg	Success
predict	Success
time	Success
step	Success
list	Success
exit	Success

“A description of your command parsing code. In particular, you should explain how you validated user input and how you converted user inputs to appropriate data types to execute the commands”

The `handleUserCommand(std::string &userCommand)` function first tokenizes the user's command using the space character as a delimiter and stores the resulting tokens in a `std::vector<std::string>`. It then checks the first token to determine which command the user has entered. If the first token is `"help"`, the function checks whether an additional argument was provided. If no additional argument was provided, it calls the

`printHelp()` function to print a list of all available commands. If an additional argument was provided, it checks whether the argument corresponds to a valid command by looking it up in a `std::map` called `helpMap`. If the argument is present in `helpMap`, the function calls the `printHelpForCmd(cmd[1])` function to print help information for the specific command. If the argument is not present in `helpMap`, the function prints an error message.

For the other commands, the function checks the number of arguments provided and the format of the arguments to ensure that they are valid. For example, for the `"min"` and `"max"` commands, the function expects a single argument specifying the product for which to print the minimum or maximum value. For the `"avg"` command, the function expects two arguments: the product and the number of timesteps over which to compute the average. If the number of arguments or the format of the arguments is not as expected, the function prints an error message and throws an exception.

Once the function has verified that the command and its arguments are valid, it calls the appropriate function to execute the command. For example, if the command is `"prod"`, it calls the `printAvailableProducts()` function to print a list of available products. If the command is `"min"` or `"max"`, it calls the `printProductMinMaxOfType(cmd)` function to print the minimum or maximum value of the specified product. If the command is `"list"`, it calls the `printAllCurrentOrdersOfType(cmd[1])` function to print a list of current orders of the specified type (bid or ask). If the command is `"exit"`, it calls the `terminateGracefully()` function to terminate the program gracefully.

“A description of your custom command and how you implemented it.”

```
void AdvisorMain::printAllCurrentOrdersOfType(const std::string &orderType) {
    if (orderType.empty() || !orderBook.isValidOrderType(orderType)) {
        std::cout << "Invalid argument for list <bid/ask>: " << orderType << std::endl;
        throw std::invalid_argument("Invalid argument for list <bid/ask>");
    }

    std::vector<OrderBookEntry> orders;
    orders = orderBook.getOrders(OrderBookEntry::stringToOrderBookType(orderType), "", currentTime.first);

    if (orders.empty()) {
        std::cout << BOTPROMPT << "No " << orderType << "s found for current time step: ("
                  << currentTime.first << ")." << std::endl;
    } else {
        std::cout << BOTPROMPT << orderType << "s for current time step (" << currentTime.first << "):" << std::endl;
        for (const OrderBookEntry &e: orders) {
            std::cout << e.toString() << std::endl;
        }
    }
}
```

The `"list"` command allows the user to view a list of current orders of a specified type (either "bid" or "ask") on the exchange. The above code implements the `printAllCurrentOrdersOfType (const std::string &orderType)` function, which is responsible for printing the list of current orders of the specified type.

The function takes a single parameter, a `std::string` called `orderType`, which specifies the type of orders to print. It first checks whether `orderType` is empty or not a valid order

type by calling the `isValidOrderType()` member function of the orderBook object. If `orderType` is invalid, the function prints an error message and throws an exception.

If `orderType` is valid, the function calls the `getOrders()` member function of the `orderBook` object to get a list of orders of the specified type for the current timestep. It then checks whether the list of orders is empty. If the list is empty, the function prints a message indicating that no orders were found for the current timestep. If the list is not empty, the function prints a message indicating the current timestep and then iterates through the list of orders, calling the `toString()` member function of each `OrderBookEntry` object to print the details of the order.

The output of the `"list"` command is a list of orders of the specified type, including the order ID, product, quantity, and price for each order. The list is printed to the console for the user to view.

“A description of how you optimized the exchange code”

The updated CSVReader::readCSV function has several improvements over the original version that make it more robust and user-friendly:

- The original version used an `std::ifstream` object to read the input file line by line. This can be inefficient for large files because it reads the file one character at a time. The updated version uses the `fopen`, `fgets`, and `fclose` functions to read the file in blocks, which may be more efficient for large files.

  The `fopen` function opens a file and returns a pointer to a FILE object that can be used to access the file. The `fgets` function reads a line of data from the file pointed to by the FILE object, and stores it in a buffer. The `fclose` function closes the file and releases any associated resources. By using these functions, the updated `CSVReader::readCSV` function reads the input file in blocks instead of reading it line by line, which may be more efficient for large files.

- The original version did not include any error handling to handle cases where the input file could not be opened. If the file could not be opened, the function would have returned an empty vector without any indication that there was an error. The updated version includes error handling to handle this case, and throws a std::runtime_error exception if the file cannot be opened. This allows the user to handle the error appropriately and take corrective action if necessary.

  To handle errors in the original version, the function would have needed to check the status of the `std::ifstream` object after it was opened. If the object was not in a good state, it would have indicated that there was an error opening the file. However, this approach is less robust than using an exception because it requires the user to explicitly check the status of the object, and does not provide any information about the specific error that occurred.

By using an exception, the updated `CSVReader::readCSV` function can signal that an error has occurred in a more straightforward and user-friendly way. The user can catch the exception and handle the error by displaying an error message or taking other corrective action.

- The updated version includes a message that prints the number of entries read from the input file. This may be useful for debugging or for understanding the performance of the function. In the original version, this information was not provided, which made it more difficult to understand the behavior of the function.

  By printing the number of entries read from the input file, the updated `CSVReader::readCSV` function provides additional information that may be useful for understanding the performance of the function. The user can use this information to track the progress of the function, and to identify any potential issues that may be affecting the performance of the function.

  For example, if the number of entries read is significantly lower than the expected number of entries in the input file, it may indicate that there is an issue with the data being processed. The user can use this information to debug the issue and take corrective action. Similarly, if the number of entries read is significantly higher than the expected number of entries in the input file, it may indicate that there are duplicate entries in the input file, or that there is an issue with the function itself. The user can use this information to troubleshoot the issue and take corrective action.

- The updated version includes error handling to catch and handle errors in the data being processed. If an error is encountered, the function prints an error message instead of throwing an exception. This allows the function to continue processing the remaining lines in the input file instead of terminating prematurely.

  In the original version, if an error was encountered in the data, the function would have thrown an exception and terminated prematurely. This could have made it difficult for the user to understand the cause of the error, and to take corrective action.

  By including error handling to catch and handle errors gracefully, the updated `CSVReader::readCSV` function provides a more user-friendly experience. The user can identify the cause of the error by reading the error message, and can take corrective action to fix the error and continue processing the input file.