Coding workshop: separating the orders into bids and asks. Printing stats per product.

In this worksheet, we will work on an OrderBook class which models the order book data set. It behaves a little like a database, allowing us to retrieve orders from the order book using filters.

Add a new class to your project

You might want to start with the starter code for this topic if you had some problems with the code in the previous topic, or you might want to keep working inside your own project.

Add a new class to your project called OrderBook. We are going to start with the class definition, which has all the data and function members we plan to add in this worksheet.

The header file OrderBook.h should contain this:

```
#pragma once
#include "OrderBookEntry.h"
#include "CSVReader.h"
#include <string>
#include <vector>

class OrderBook
{
    public:
    /** construct, reading a csv data file */
        OrderBook(std::string filename);
    /** return vector of all know products in the dataset*/
        std::vector<std::string> getKnownProducts();
    /** return vector of Orders according to the sent filters*/
        std::vector<OrderBookEntry> getOrders(OrderBookType type,
                                              std::string product,
                                              std::string timestamp);
    /** return the price of the highest bid in the sent set */
        static double getHighPrice(std::vector<OrderBookEntry>& orders);
    /** return the price of the lowest bid in the sent set */
        static double getLowPrice(std::vector<OrderBookEntry>& orders);

    private:
        std::vector<OrderBookEntry> orders;

};
```

Study this code and identify the functions you need to implement. Then go into your OrderBook.cpp file and write empty wrappers for these functions which just return the minimum code to allow it to be compiled. For example, here is getLowPrice:

```
double OrderBook::getLowPrice(std::vector<OrderBookEntry>& orders)
{
    double min{0};
    return min;
}
```

We will put in the implementations later! Go ahead and put the empty functions into OrderBook.cpp and compile the project.

Integrate the OrderBook with the MerlelMain class

We are going to update the MerkelMain class, so it uses the OrderBook class as a wrapper around the dataset instead of directly storing the OrderBookEntry objects in a vector. In MerkelMain.h, add a private field called orderBook:

```
#include "OrderBook.h"

class MerkelMain
{
    ....
    private:
        // this is the new way
        // we will represent the orders
        OrderBook orderBook{"20200317.csv"};

        // we don't want this any more as
        // OrderBook will store this data itself
        //std::vector<OrderBookEntry> orders;

    ...
```

Note that the orders variable declaration in the above code is commented out.

To get this to compile, you will need to go into MerkelMain.cpp and comment out any lines that refer to the orders data member that we just removed, e.g.:

```
void MerkelMain::loadOrderBook()
{
    // comment this out as the orders field has gone!
    //orders = CSVReader::readCSV("20200317.csv");
}
```

Once you've commented out any lines in MerkelMain that make calls to the orders vector, you should be able to compile your program. Note that your main function should look like this:

```
#include "MerkelMain.h"
#include <iostream>

int main()
{
    MerkelMain app{};
    app.init();
}
```

Since OrderBook is a data member of MerkelMain, the line:

```
MerkelMain app{};
```

will trigger a call to the constructor of MerkelMain, then the constructor of OrderBook.

The constructor

Now we have the framework of the newly organised program in place, we can start filling in the code.

The first thing we will work on in the OrderBook class is the constructor. It needs to do what the MerkelMain loadOrderBook function used to do, i.e.

```
orders = CSVReader::readCSV(filename);
```

Put that code into the OrderBook constructor. 'filename' is a variable that is passed into the constructor. This works because the constructor signature is like this:

```
OrderBook(std::string filename);
```

Test it - run the program and check that the OrderBook object stored in the MerkelMain object has loaded data from the data file.

The getKnownProducts

Next, we will work on the getKnownProducts function in the OrderBook class. This runs through the vector of OrderBookEntry objects and generates a unique list of products. A product is a currency pair such as ETH/BTC, stored as a std::string.

We will use a std::map data structure to implement this. You will need to include map to be able to use it. Add this line to the OrderBook.cpp file:

```
#include <map>
```

Then for the first part of the function, add this to OrderBook.cpp:

```
std::map<std::string,bool> prodMap;

for (const OrderBookEntry& e : orders)
{
    prodMap[e.product] = true;
}
```

This will result in a data structure mapping a unique set of product names to true values, e.g.:

```
key -> value
===============
ETH/BTC -> true
BTC/USDT -> true
```

Now we have identified unique products, we will iterate over the keys in the map and store them out to a vector of strings. That vector will be our list of unique product names. Here is the second part of the function:

```
std::vector<std::string> products;
for (const auto& e : prodMap)
{
    products.push_back(e.first);
}
```

Some things to note:

- We used the auto type here. We could also have used the std::pair type to make it more explicit. Personally, I prefer this as it is more explicit:

  for (const std::pair& e : prodMap) { products.push_back(e.first); }

- The variable name 'e' is not very descriptive. How about this?

  for (const std::pair& productStringBoolPair : prodMap) { products.push_back(productStringBoolPair.first); }

- We call e.first on the pair, which gives us the string part of the pair
- We set the type to const reference as we only need read-only access to the key (e.first)
- Don't forget to return products from the function.

You can find more information about std::map in the textbook, Chapter 19 p730. There is also another example of the syntax that you can use to iterate over items in a map. There are always many ways to do things in C++.

Challenge

Speaking of other ways to do things, there are different ways to build a unique list of products from the orders vector. Can you think of another way to do this? Go ahead and code it up.

The getOrders function

Next up we are going to implement the getOrders function. This function transforms the set of orders as a database which we can query. Here is the signature again:

```
std::vector<OrderBookEntry> getOrders(OrderBookType type,
                                      std::string product,
                                      std::string timestamp);
```

This function does the following:

- Create a local variable of type std::vector which can store the orders matching the filters
- Iterate over OrderBookEntry objects in the orders vector
- If an OrderBookEntry object matches the filters, store it onto the new OrderBookEntry vector
- return the vector at the end.

Go ahead and implement this function yourself.

The getHighPrice function

Now we will implement some statistical functions. The first one identifies the highest price in a given vector of OrderBookEntry objects. The signature looks like this:

```
static double getHighPrice(std::vector<OrderBookEntry>& orders);
```

Some comments on this:

- It is a static function. That means it cannot access any of the data members of the OrderBook class. It just operates on the data it is sent.
- Can we reasonably change the signature to this?

  static double getHighPrice(const std::vector& orders);
- Can you go ahead and implement the function? It should iterate over the sent orders and identify the highest price seen in any order.

The getLowPrice function

This function is very similar to getHighPrice, except it finds the lowest price and returns that. Go ahead and implement this function.

Integrate it back out to MerkelMain

Now we have written these functions, it is time to call them from the MerkelMain class. Update your MerkelMain printMarketStats function to this:

```
void MerkelMain::printMarketStats()
{
    std::string currentTime = "2020/03/17 17:01:24.884492";
    for (const std::string& p : orderBook.getKnownProducts())
    {
        std::cout << "Product: " << p << std::endl;
        std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookType::ask, p, cu
        std::cout << "Asks seen: " << entries.size() << std::endl;
        std::cout << "Max ask: " << OrderBook::getHighPrice(entries) << std::endl;
        std::cout << "Min ask: " << OrderBook::getLowPrice(entries) << std::endl;
    }
}
```

Note that the time is hardcoded for now. Make sure you set it to a correct time from your dataset.

You should see some output like this when you select the print exchange stats option from the menu in the main app:

```
Product: BTC/USDT
Asks seen: 50
Max ask: 5460.12
Min ask: 5352
Product: DOGE/BTC
Asks seen: 50
Max ask: 8e-07
Min ask: 3.1e-07
....
```

The output your program generates will vary depending on the data file you use, but the structure should be like this.

Challenge: do some more advanced stats

Can you compute some more advanced stats? For example, it is common in currency trading to measure the spread between the set of asks and the set of

bids.

Conclusion

In this worksheet, we have implemented an OrderBook class which provides a higher level interface through which we can interact with the order book data. The interface includes statistical functions and filtering functions.