Coding workshop: implementing the matching engine

In this worksheet, you are going to implement the matching engine. The matching engine matches the bids in the order book to the asks in the order book and identifies which trades should take place. Primarily, if someone offers to buy at or above the price that someone else offers to sell for, a trade happens.

Key characteristics of the algorithm

There are many ways to design a matching algorithm. The algorithm I have designed aims to have the following characteristics:

- The lowest ask is processed first.
- The highest bid that matches an ask is given priority over lower, matching bids.
- The lowest price is paid - so if a bid is offering to pay more than an ask, the bidder will only pay as much as the asker wants.
- Partial sales are allowed - if an ask and a bid match but the amounts do not match, the largest possible amount is sold
- Partially matched bids or asks can be re-processed and matched against further bids or asks

Some extra sorting functions

The algorithm below relies on you being able to sort the bids and asks according to their prices, ascending and descending. At the moment, we have this function in OrderBookEntry.h which allows sorting by timestamp:

```
static bool compareByTimestamp(const OrderBookEntry& e1, const OrderBookEntry& e2)
{
    return e1.timestamp < e2.timestamp;
}
```

Here are the functions you need:

```
static bool compareByPriceAsc(OrderBookEntry& e1, OrderBookEntry& e2)
{
    return e1.price < e2.price;
}
static bool compareByPriceDesc(OrderBookEntry& e1, OrderBookEntry& e2)
{
    return e1.price > e2.price;
}
```

Add those to OrderBookEntry.h.

The pseudocode algorithm for the matching engine

Here is the pseudocode algorithm.

```
// asks = orderbook.asks in this timeframe

// bids = orderbook.bids in this timeframe

// sales = []

// sort asks lowest first
// sort bids highest first

// for ask in asks:

//     for bid in bids:

//         if bid.price >= ask.price # we have a match

//             sale = new orderbookentry()
//             sale.price = ask.price

//             if bid.amount == ask.amount: # bid completely clears ask
//                 sale.amount = ask.amount
//                 sales.append(sale)
//                 bid.amount = 0 # make sure the bid is not processed again
//                 # can do no more with this ask
//                 # go onto the next ask
//                 break

//             if bid.amount > ask.amount:  # ask is completely gone slice the bid
//                 sale.amount = ask.amount
//                 sales.append(sale)
//                 # we adjust the bid in place
//                 # so it can be used to process the next ask
//                 bid.amount = bid.amount - ask.amount
//                 # ask is completely gone, so go to next ask
//                 break

//             if bid.amount < ask.amount # bid is completely gone, slice the ask
//                 sale.amount = bid.amount
//                 sales.append(sale)
//                 # update the ask
//                 # and allow further bids to process the remaining amount
//                 ask.amount = ask.amount - bid.amount
//                 bid.amount = 0 # make sure the bid is not processed again
//                  # some ask remains so go to the next bid
//                  continue
// return sales
```

Implement the algorithm in C++

Your job is to translate this algorithm into C++. You have already seen me doing it in the videos and now it is your turn!

The algorithm should be placed into a function in the public section of the OrderBook class. It should have the following signature:

```
std::vector<OrderBookEntry> matchAsksToBids(std::string product, std::string timestamp);
```

Note that this is not a static function so it can work with the orders data member of the OrderBook class. You will need to add another OrderBookType: OrderBookType::sale.

Call the matchinmatchAsksToBids function from MerkelMain

Once you have the algorithm implemented, you can call it from MerkelMain gotoNextTimeframe. Something like this:

```
void MerkelMain::gotoNextTimeframe()
{
    std::cout << "Going to next time frame. " << std::endl;
    for (std::string& p : orderBook.getKnownProducts())
    {
        std::cout << "matching " << p << std::endl;
        std::vector<OrderBookEntry> sales =  orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;
        for (OrderBookEntry& sale : sales)
        {
            std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::e
        }
    }
    currentTime = orderBook.getNextTime(currentTime);
}
```

Testing the algorithm

Now it is time to make some test data and test that your algorithm is operating correctly. You can create a new CSV file with just a few rows in it and edit the values in those rows to test out particular scenarios. Remember that the CSV file is specified in MerkelMain.h in the orderBook data member:

```
OrderBook orderBook{"20200317.csv"};
```

Create a CSV file called test.csv and put this in it:

```
2020/03/17 17:01:24.884492,ETH/BTC,ask,0.021873,1.
2020/03/17 17:01:24.884492,ETH/BTC,bid,0.021873,1.
```

Update MerkelMain.h:

```
OrderBook orderBook{"test.csv"};
```

Now compile and run the program. Then from the menu, select continue. It will process the ask and the bid, producing a perfect match. The output should be something like this:

```
You chose: 6
Going to next time frame.
matching ETH/BTC
max ask 0.021873
min ask 0.021873
max bid 0.021873
min bid 0.021873
bid price is right
Sales: 1
Sale price: 0.021873 amount 1
```

Create CSV data to test the following scenarios:

- No bids, one ask
- Bid that only wants to buy some of the ask (bid amount is smaller than ask amount)
- One ask that is matched by multiple bids check that the highest bid is chosen first
- Lowest ask is processed first
- Bid is higher than ask, sale price is set to the lower, ask value

Try to think of other scenarios you'd like to test, with reference to the set of characteristics listed at the beginning of the worksheet.

Conclusion

In this worksheet, we have implemented a reasonably sophisticated algorithm from pseudocode into C++ code. We then integrated the algorithm to the main program. In the end, we tested the algorithm by feeding it test data to see if it operates as required.