Coding workshop: allowing the user to place bids and asks

In this worksheet, we are going to implement the place ask and place bid functionality which allows a user to place bids and asks in the order book.

Read input from the user into a string

First we are going to use std::getline to read a string from the user. I should note at this point that in the original videos, I called this function enterOffer. However, that is a bit confusing since we are actually entering an 'ask'. So in these worksheets, I am renaming it to enterAsk.

In MerkelMain::enterAsk:

```
std::cout << "Make an ask - enter the amount: product,price, amount, eg  ETH/BTC,200,0.5" <<
std::string input;
std::getline(std::cin, input);
```

Tokenise

Now tokenise:

```
std::vector<std::string> tokens = CSVReader::tokenise(input, ',');
```

If you are working off the example code, you might find that CSVReader::tokenise is private so we cannot access it from the outside. If that is the case, move CSVReader::tokenise to the public section of the class.

On this point, it is best to start with things set to private. This limits what your classes expose of themselves to the outside world, which limits the potential for bugs. Since there is a clear reason to expose tokenise, this is a good time to do so.

Convert into an OrderBookEntry

Moving to getline

Before we get into user input processing, as noted in the videos, it is not a good idea to combine code like this:

```
int input;
std::cin >> input;
```

with code like this, in the same program:

```
std::string sInput;
std::getline(std::cin, sInput);
```

So let's convert the user input code in the menu to use getline. In MerkelMain.cpp, replace the getUserOption function with this:

```
int MerkelMain::getUserOption()
{
    int userOption = 0;
    std::string line;
    std::cout << "Type in 1-6" << std::endl;
    std::getline(std::cin, line);
    try{
        userOption = std::stoi(line);
    }catch(const std::exception& e)
    {
    }
    std::cout << "You chose: " << userOption << std::endl;
    return userOption;
}
```

Notes:

- We now use getline instead of the stream technique
- We call stoi which can throw an exception if they enter an invalid number
- We catch the exception and return 0 if there is an exception

A better CSVReader::stringsToOBE

The CSVReader has a private function that converts from a vector of strings into an OrderBookEntry. We could make that public and use it here. Here is its signature:

```
static OrderBookEntry stringsToOBE(std::vector<std::string> strings);
```

The problem is that this function is a bit obscure - it is not clear what order the tokens should be in. It would be better to make a more explicit function that specifies which order the tokens go in. Let's do that. Add this function signature to the public section of CSVReader.h:

```
static OrderBookEntry stringsToOBE(std::string price,
                                   std::string amount,
                                   std::string timestamp,
                                   std::string product,
                                   OrderBookType OrderBookType);
```

Then add this implementation to CSVReader.cpp, which is similar to the vectors version we already had:

```
OrderBookEntry CSVReader::stringsToOBE(std::string priceString,
                                       std::string amountString,
                                       std::string timestamp,
                                       std::string product,
                                       OrderBookType orderType)
{
    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;  // throw up to the calling function
    }
    OrderBookEntry obe{price,
                    amount,
                    timestamp,
                    product,
                    orderType};

    return obe;
}
```

Convert the user's tokens into an OrderBookEntry

Now to take the tokens and use them to create an OrderBookEntry:

```
OrderBookEntry obe = CSVReader::stringsToOBE(
            tokens[1],
            tokens[2],
            currentTime,
            tokens[0],
            OrderBookType::ask
        );
```

Challenge: make it robust

Hold on a minute! That was all too easy. Now it is your turn to make the user input processing code robust. Using a combination of input checking and exception handling, update the code in enterAsk, so it is as 'uncrashable' as possible. Refer to CSVReader stringsToOBE for ideas.

Store the new OrderBookEntry object into the order book

Now we need a way to add the new OrderBookEntry object into the order book vector. Let's think about the scope of the various pieces of data we have. We created the OrderBookEntry object obe as a local variable in the MerkelMain::enterAsk function. We need this object to end up stored in the orders data member in the OrderBook class. The orders variable is private to OrderBook and has type std::vector) in the OrderBook class. We could do one of these:

- make the orders variable in OrderBook public so we can push_back our OrderBookEntry to it directly
- add a function member to OrderBook called insertOrder which allows us to pass the OrderBookEntry to the OrderBook

I prefer the latter option, creating a function, as then the OrderBook retains full control over its own data. For example, if OrderBook wanted to check the incoming OrderBookEntry for validity before inserting it, it can do this. If the orders variable were public, it could not check things that are pushed directly onto it.

So add this to the OrderBook class public section in OrderBook.h:

```
void insertOrder(OrderBookEntry& order);
```

Then write the implementation in OrderBook.cpp, simply this:

```
void OrderBook::insertOrder(OrderBookEntry& order)
{
    orders.push_back(order);
}
```

Sorting the OrderBook

Now we really have a problem. The title of this section in the worksheet should give you a clue. Various parts of the code in OrderBook assume that the orders are sorted by timestamp. We just pushed back a new OrderBookEntry to the end of the orders vector. What is wrong with that?

the problem

Imagine a scenario where we have the following timestamped orders in the order book:

```
2020/03/17 17:01:24.884492
2020/03/17 17:01:24.884492
2020/03/17 17:01:24.884492
```

```
2020/03/17 17:01:30.099017
2020/03/17 17:01:30.099017
2020/03/17 17:01:30.099017
```

The simulation current time is 2020/03/17 17:01:24.884492. We create a new order with the timestamp set to the current time and stick it on the end of the vector:

```
2020/03/17 17:01:24.884492
2020/03/17 17:01:24.884492
2020/03/17 17:01:24.884492
2020/03/17 17:01:30.099017
2020/03/17 17:01:30.099017
2020/03/17 17:01:30.099017
2020/03/17 17:01:24.884492 -- our new order
```

Now the orders are not sorted by timestamp.

So we need to sort them.

The solution - sorting the orders vector

There is a sort function in the standard library. It works like this:

```
std::sort(orders.begin(), orders.end(), customFunction);
```

You need to add this line to the file that uses std::sort:

```
#include <algorithm>
```

It is our job to write customFunction. customFunction receives two objects of the type found in the vector. It must return true if the first object goes before the second object and false if the second object goes before the first object.

We are going to define customFunction in OrderBookEntry.h's public section. This makes sense as it is the OrderBookEntry class that should know how to sort OrderBookEntry objects.

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

This function will return true if e1 goes before e2, and false if not. Note that it is static and the types are const references. This gives us a minimal performance overhead (reference, do not copy) and minimal access (read-only).

Why is the function implementation in the header file?

We put the function implementation in the header file, instead of putting it out in the CPP file. In general, I prefer to keep header files as lean and readable as possible. Putting function implementations in header files is, therefore, a bad thing from this perspective as all that implementation makes for a messy header file. There is also a technical reason for putting implementations in CPP files. When you compile, the includes are recompiled every time any file that includes those includes changes, even if the include did not change. However, the compiled implementations need not be recompiled if its CPP files are not changed. On larger builds, this is a serious consideration - you don't want to recompile the entire program every time you make a small edit to a CPP file.

So why did I put the implementation in the header files? Only it is such a simple function, it did not make the header file messy, and it saved me putting that implementation somewhere else. This could be an interesting topic for discussion - go ahead!

Use the custom sorting function

Now we can tell std::sort to use this function:

```
std::sort(orders.begin(), orders.end(), OrderBookEntry::compareByTimestamp);
```

So the complete version of insertOrder is:

```
void OrderBook::insertOrder(OrderBookEntry& order)
{
    orders.push_back(order);
    std::sort(orders.begin(), orders.end(), OrderBookEntry::compareByTimestamp);
}
```

Now the order book is re-sorted every time we add a new order. You can check this by stepping through time and inserting asks into the order book in different time windows and verifying the count of orders goes up in that time frame. Go ahead and check it.

Implement enterBid

Now write the code for enterBid, which is very similar to enterAsk, except the OrderBookType is OrderBookType::bid. Go ahead and implement enterBid.

Conclusion

In this worksheet, we implemented the enter ask and enter bid functionality. This allows users to place orders in the order book.