Coding workshop: completing the trade

In this worksheet, we are going to write the code to complete the whole trading process. This code will check the Wallet to see if the user has sufficient funds to place a bid or ask, then if their order is matched, it will update the contents of the Wallet accordingly.

Checking the status of the program

At this point, you should have the Wallet class fully implemented, as shown in the previous worksheet. You should have the OrderBook class with a working matching engine and the capability to add orders. The MerkMain class should be able to receive user orders and bids and insert them into the order book. If you are not sure you have all of this, you can refer to the code provided at the start of topics 1-5 and at the end of this topic.

Check asks and bids against the Wallet

We will start by adding a function to the Wallet, which allows it to check if it can fulfill a given OrderBookEntry object.

First of all, include the OrderBookEntry.h file in Wallet.h:

```
#include "OrderBookEntry.h"
```

Next, add the following function signature to the public section of the Wallet class in Wallet.h:

```
bool canFulfillOrder(OrderBookEntry order);
```

This function should return true if the Wallet contains sufficient funds to fulfill this order. The function needs to calculate the total cost of a bid or a sale, then use the Wallet::containsCurrency function to check if the Wallet has that much currency available. Here are the formulae to calculate the cost of asks and bids:

```
// for an ask:
amount = order.amount;
// for a bid:
amount = order.amount * order.price
```

Write the body of the canFulfillOrder in Wallet.cpp. It should do the following:

- check the OrderBookType of the OrderBookEntry
- calculate the amount according to the type
- use containsCurrency to decide if it can fulfill that order
- return true if it can, false if it cannot fulfill the order

Call the canFulfillOrder function before placing orders

Now we can call canFulfillOrder before we add orders to the order book. First of all, add a Wallet as a data member to MerkelMain.h private section:

```
private:
    ...
    Wallet wallet;
    ...
```

Put some cash in the Wallet in the MerkelMain::init function in MerkelMain.cpp:

```
void MerkelMain::init()
{
    ...
    wallet.insertCurrency("BTC", 10);
    ...
```

Now call the canFulfillOrder MerkelMain.cpp MerkelMain::enterAsk:

```
OrderBookEntry obe = CSVReader::stringsToOBE(
    tokens[1],
    tokens[2],
    currentTime,
    tokens[0],
    OrderBookType::ask
);
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
}
else {
    std::cout << "Wallet has insufficient funds . " << std::endl;
}
```

Do the same for enterBid. Test the code by putting something in the Wallet then attempting to place asks and bids that you can and cannot fulfill. Does the Wallet correctly detect if it has enough cash?

Label the asks and bids created by the simulation user

The next step is to add labels to the OrderBookEntry objects that are created in MerkelMain::enterAsk and MerkelMain::enterBid. We will use the labels to identify OrderBookEntry objects that were generated by the simulation user, as opposed to being generated from the CSV dataset. We need to identify the user's orders so we can update their Wallet accordingly.

Add this data member to the OrderBookEntry header file public section:

```
std::string username;
```

Change the OrderBookEntry constructor to this:

```
OrderBookEntry( double _price,
                double _amount,
                std::string _timestamp,
                std::string _product,
                OrderBookType _orderType,
                std::string username = "dataset");
```

Note that we have set the default value of username to "dataset". You can read more about setting default parameter values here https://www.cplusplus.com/doc/tutorial/functions/ in the section 'Default values in parameters'. Or for degree students, textbook Chapter 8, p283.

Then, in the MerkelMain::enterAsk and MerkelMain::enterBid functions, set the username of the OrderBookEntry, e.g.:

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

identify trades associated with the simulation user and update the Wallet accordingly

Now we have labelled the OrderBookEntry objects created by the simulation user, we need to make sure the matching engine keeps track of these labels when it creates the sales. Remember that the matching engine is in this function in OrderBook.cpp:

```
std::vector<OrderBookEntry> OrderBook::matchAsksToBids(std::string product, std::string time
```

The function returns a vector of OrderBookEntry items which have their orderType set to OrderBookType::sale. The problem is, from the sale label, we do not know who it belongs to, or indeed, if it came from an ask or a bid.

We could add a username to the sales, then we know which sales resulted from the simulation user. But then we do not know how to process the sale against their Wallet - do we process it as an ask or a bid? This means we need a way to know that a sale is from the simulation user *and* if it was originally an ask or a bid.

We are going to add some more options to the OrderBookType class enum to achieve this double label. In the OrderBookEntry header:

```
enum class OrderBookType{bid, ask, unknown, asksale, bidsale};
```

Now in OrderBook.cpp, OrderBook::matchAsksToBids, after we create the OrderBookEntry sale variable:

```
OrderBookEntry sale{ask.price, 0, timestamp,
                    product,
                    OrderBookType::asksale};
                    // note default username

if (bid.username == "simuser")
{
    sale.username = "simuser";
    sale.orderType = OrderBookType::bidsale;
}
if (ask.username == "simuser")
{
    sale.username = "simuser";
    sale.orderType =  OrderBookType::asksale;
}
```

Add a function to Wallet to process orders

Add this function to the Wallet header file public section:

```
void processSale(OrderBookEntry& sale);
```

The implementation is very similar to canFulfillOrder, except that it has to calculate both outgoing and incoming currency. Here is the code for the function, as seen in the video:

```
void Wallet::processSale(OrderBookEntry& sale)
{
    std::vector<std::string> currs = CSVReader::tokenise(sale.product, '/');
    // ask
    if (sale.orderType == OrderBookType::asksale)
    {
        double outgoingAmount = sale.amount;
        std::string outgoingCurrency = currs[0];
        double incomingAmount = sale.amount * sale.price;
        std::string incomingCurrency = currs[1];

        currencies[incomingCurrency] += incomingAmount;
        currencies[outgoingCurrency] -= outgoingAmount;

    }
```

```
    // bid
    if (sale.orderType == OrderBookType::bidsale)
    {
        double incomingAmount = sale.amount;
        std::string incomingCurrency = currs[0];
        double outgoingAmount = sale.amount * sale.price;
        std::string outgoingCurrency = currs[1];

        currencies[incomingCurrency] += incomingAmount;
        currencies[outgoingCurrency] -= outgoingAmount;
    }
}
```

Tell MerkelMain to process sales against the Wallet

Now we have the username set to simuser and the type set to asksale or bidsale, so we know who made it and how to process it. It probably looks 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);
}
```

It is the inner loop that we are interested in:

```
for (OrderBookEntry& sale : sales)
{
    std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
}
```

Over to you - add some code to this inner loop which does the following:

- checks the username on the sale
- if the usernane is simuser call Wallet::processSale

You might want to think about refactoring the gotoNextTimeframe function into two functions as it now looks a little too nested and seems to do too many

different things. Perhaps you could have a processSale function on MerkelMain?

Now test it

Time to play around with the simulation, which is now ostensibly complete. Fill up your Wallet and go trading!

Conclusion

In this worksheet, we have worked on several classes to complete the functionality we need to process trades against the user's Wallet.