Coding workshop: dealing with time

In this worksheet, we are going to implement time in the simulation. We can then step through and look at the state of the order book at specific points in time.

Find a start time

The first step is to figure out what the earliest time is in the order book. We can do that with a function on the OrderBook class. In OrderBook.h, add this to the public section of the class:

```
...
public:
    std::string getEarliestTime();
...
```

The simplest implementation is to assume the dataset is already in chronological order. This should be the case as the data in the data file is sorted chronologically. Put this in OrderBook.cpp:

```
std::string OrderBook::getEarliestTime()
{
    return orders[0].timestamp;
}
```

However, this does not feel super reliable. Why not iterate over the orders and find the smallest timestamp? The timestamps are strings, but we can compare them alphabetically and still get the right result. Consider this program:

```
#include <iostream>
#include <string>

int main()
{
    std::string date1 = "2020/03/17 17:01:24.884492";
    std::string date2 = "2020/03/17 17:01:30.099017";
    if (date1 < date2)
    {
        std::cout << date2 << "comes after " << date1 << std::endl;
    }
    if (date1 > date2)
    {
        std::cout << date1 << "comes after " << date2 << std::endl;
    }
}
```

Run that program - it should reassure you that you can compare timestamps stored as strings and figure out which is the earliest.

Now go ahead and write a more reliable version of OrderBook::getEarliestTime which searches the whole vector of OrderBookEntry objects for the earliest timestamp.

A note about operator overloading

Here is something to ponder. In the example above, we used the $>$ and $<$ operators on strings:

```
if (date1 > date2)
```

But strings are objects, not numbers. How does the language know how to compare string objects? Comparing two strings involves a complicated operation of comparing them character by character.

The answer is *operator overloading*. The idea is that a class can specify some code that is used to execute a particular operator.

You can find out about the relational operators that are available on strings here: https://www.cplusplus.com/reference/string/string/operators/

You can find more information about operator overloading at this link: https://www.cplusplus.com/doc/tutorial/templates/ or (degree students) the textbook covers operator overloading in Chapter 12 p450.

Store the current time

Now that we can find the earliest time in our dataset, we need to have a way of remembering the current time in our simulation so that we can carry out operations in a given time window. This is quite simple. In MerkelMain.h, add a private data member to the class:

```
private:
    ...
    std::string currentTime;
    ...
```

Note that it is a good idea to set up fields as private. In general, only make things public if that is essential for the functioning of the program.

Now in the MerkelMain::init function, find and store the currentTime:

```
void MerkelMain::init()
{
    ...
    currentTime = orderBook.getEarliestTime();
    ...
}
```

Now update the MerkelMain printMarketStats function so it uses current time instead of a hardcoded time:

```
...
// get rid of this line!
//std::string currentTime = "2020/03/17 17:01:24.884492";
for (const std::string& p : orderBook.getKnownProducts())
{
    std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookType::ask, p, current
...
```

Move to the next time frame

Now we have the currentTime variable set up, we need to implement the getNextTime on the OrderBook class so we can move to the next time window:

```
class OrderBook
{
    public:
        ...
        std::string getNextTime(const std::string& timestamp);
        ...
```

Note the function signature: timestamp is const and a reference.

Here is an implementation:

```
std::string OrderBook::getNextTime(const std::string& timestamp)
{
    std::string next_timestamp = "";
    for (OrderBookEntry& e : orders)
    {
        if (e.timestamp > timestamp)
        {
            next_timestamp = e.timestamp;
            break;
        }
    }
    return next_timestamp;
}
```

Can you think of any problems with this function? I can think of one - what happens if it does not find a timestamp that is later than the sent parameter timestamp? Adding this block at the end should fix that:

```
if (next_timestamp == "")
{
    next_timestamp = orders[0].timestamp;
}
```

That will set the timestamp to the first timestamp if it does not find a later one. The simulation wraps around. Perhaps a better way would be to do this:

```
if (next_timestamp == "")
{
    next_timestamp = getEarliestTime();
}
```

As for the naive version of getEarliestTime, the getNextTime function assumes that the orders are sorted by time.

We will investigate sorting in C++ in a later worksheet. However, if you want to sort the orders right now, then I set you the *challenge* of working out how to use std::sort to sort the orders in the orders vector. See this link: https://www.cplusplus.com/reference/algorithm/sort/ or (degree students) textbook Chapter 19 p755 for information about sorting.

Use getNextTime to move in time

Now we are going to use the getNextTime function to implement moving through time in the simulation. We'll call it in MerkelMain::gotoNextTimeframe, which should be called from one of the menu options:

```
void MerkelMain::gotoNextTimeframe()
{
    std::cout << "Going to next time frame. " << std::endl;
    currentTime = orderBook.getNextTime(currentTime);
}
```

Test it out by moving through time using the menus and printing out the market stats in each time window.

Conclusion

In this worksheet, we have implemented the features we need to be able to work with time in our simulation.