CS3211 Assignment 1
2022/2023 Semester 1

Concurrent Matching Engine in C++

Group **B1**	Name	Student #
Member #1	LOW QI XIANG	A0223859U
Member #2	HARRIS MAUNG	A0199798R

We start with a simple sequential orderbook.
The orderbook should contain:

1. 2 sorted map from price to price levels, representing bids and asks
    a. each price level will contain orders at that price, stored in a FIFO data structure for time priority matching
    b. the sorted map allows fast lookup by price for execution and insertion
2. map from order IDs to orders
    a. this will allow fast lookups for cancellations

Hence, we arrived at:

- `std::map<uint32_t, PriceLevel>` for bids and asks -> a self balancing tree.
- `std::unordered_map<uint32_t, Order*>` for the map of orders
- Each price level contains `std::queue<Order*>`

When a new order is added to a book:

1. The order is inserted into the map of orders
2. Lookup the opposite side levels map (bids/asks)
3. From the best price, if the order can match:
    a. Execute order against each resting order in the queue
    b. If the order is filled, **return**
    c. If the queue is empty, move to the next best price
4. Since there is remaining quantity unfilled, lookup the same side levels map
    a. If the level does not exist, create a new level and insert it into the levels map.
    b. Push the order into the queue and return

When a order is requested to be canceled:

1. Lookup the map of orders
2. If order does not exist, or client does not match the order, reject
3. If the order is filled, reject
4. Else, set the remaining quantity to 0 (lazy cancellation)

Challenges arise when we try to parallelise the process.
For maximal parallelism, we have to mark a minimal amount of code as critical sections.

To achieve **inter-book** parallelism:

- the engine will create a detached thread for each request to an orderbook

To achieve **intra-book** parallelism

- the orderbook will create threads to execute against the resting order queue in each price level
- we store the total quantity at each price level, so that the orderbook thread does not need to iterate through each queue before moving on to the next price
- the orderbook will also create a thread to add the new order to the queue
- the queue of resting orders will have a **front and back mutex**, to allow for independent producing (insertions) and consuming (execution/cancellations)

To synchronize these processes, we require:

- a FIFO critical section at the engine
    - this enforces sequential processing of client requests
- a FIFO critical section at the orderbook
    - this also enforces sequential processing of client requests
- front and back mutexes at each queue, as well as a conditional variable
    - the cv enforces the queue is not empty when a consumer attempts to consume

We required a FIFO mutex, however the C++ thread library mutex does not guarantee it. We were able to implement a FIFO mutex with a primitive mutex and a conditional variable. We utilized a head and tail integer to keep track of the queue numbers.

- When locking:
    - lock internal mutex
    - queue number is assigned.
    - condition variable blocks until current queue == queue number
    - lock is unlocked and thread proceeds
- When unlocking:
    - lock internal mutex
    - increment the current ticket number
    - unlock