QF633 Assignment 3

June 17, 2023

In this assignment, we will continue exploring the L3 Book buidling. The following files can be found in the attached zip file:

```
|-- data
|   |-- l3_market_data.csv          # market data in csv format
|   |-- l3_market_data.drop.csv     # market data in csv format, but with messages drops
|-- src
    |-- book
    |   |-- CMakeLists.txt
    |   |-- l3_book.cpp             # L3Book cpp file
    |   |-- l3_book.h               # L3Book header file
    |-- CMakeLists.txt
    |-- csv_l3_md
    |   |-- CMakeLists.txt
    |   |-- main.cpp                # exeutable to read csv market data, feed into L3Book and print the book
    |-- util
        |-- CMakeLists.txt
        |-- incremental_csv_reader.cpp  # util class IncrementalCSVReader
        |-- incremental_csv_reader.h
        |-- price.h                     # util class Price
```

The project produces a exeutable `csv_l3_md`, which takes two arguments:

- `./csv_l3_md path/to/your/market_data.csv l2`: print the L2 view of the book
- `./csv_l3_md path/to/your/market_data.csv l3`: print the L3 view of the book

The `path/to/your/market_data.csv` should be path to either

- `l3_market_data.csv`: no message drops to validate the book iteration logic and the order execution update handlings.
- `l3_market_data.drop.csv`: some message drops to validate the book uncrossing logic.

You are expected to give implementations in `src/book/l3_book.h` and `src/book/l3_book.cpp`, and create a zip file `Assignment3_YourName.zip`, which contains the completed source codes.

- `L3Book::ForEachLevel` and `L3Book::ForEachOrder` to iterate the book either by level or by order.
- `L3Book::ProcessExec` to handle another type of book update message
- `L3Book::UncrossBookSide` to uncross the book in case of message drops

We have been using the following convention to print the book in the lecture:

```
# the number not inside the parentheses nor the square brackets is the price for the price level
# the number inside the parentheses is the aggregated qty for that price level
# the numbers inside the square brackets are the qties for each order in that price level, following the
      FIFO ordering
                    Bid     |       Ask
    [2, 8] (10)   100.00    |     101.00    (11) [3, 4, 4]
 [1, 1, 7]  (9)    99.00    |     102.00    (11) [7, 4]
        []  (0)     0.00    |     103.00    (12) [1, 1, 10]

# 1. levels are printed following inner levels to outer levels
# 2. for the top buy level:
#     a. the price is 100
#     b. the aggregated qty is 10
#     c. there are two orders in that price level, and the qties for the order are 2 and 8, while 2 is
      queued before 8.
# 3. there're two buy levels and three sell levels, and thus we zero filled the aggregated qty and price for
       the empty 3rd buy level
```

In this assignment, we will use two slightly different representations of the book as illustrated below

L2 View of the Book

```
# In the L2 view of the book, only the aggregated information for each price level is presented:
# a) price, b) aggregated qty, c) order counts
bid_count    bid_qty  bid_price  ask_price    ask_qty  ask_count
    2          10      100.00      101.00          11          3
    3           9       99.00      102.00          11          2
    0           0        0.00      103.00          12          3

# for the top buy level:
#     a. the price is 100
#     b. the aggregated qty is 10
#     c. the number of orders is 3
```

L3 view of the Book

```
# There are four columns:
# a. side: S for Sell side, B for Buy side
# b. price: price for the level
# c. qty: aggregated qty for the level
# d. orders: list of order_qty(order_id), "1(20)" means a order with qty=1 and order_id=20

side      price        qty     orders
S        103.00        12      1(20), 1(17), 10(3)
S        102.00        11      7(1), 4(7)
S        101.00        11      3(7), 4(3), 4(5)
B        100.00        10      2(12), 8(5)
B         99.00         9      1(41), 1(40), 7(28)

# 1. levels are printed from the highest price to lowest price, i.e. outer levels to inner levels for sell
      side and inner levels to outer levels for buy side
# 2. for the top buy level
#     a) the price is 100.00
#     b) the aggregated qty is 10
#     c) there are two orders in that level, order with qty=2, order_id=12 and order with qty=8, order_id=5
```

1 L3 Book Iteration

There are two functions inside the `src/csv_l3_md/main.cpp`:

- `void PrintLevel2Book(...)`: The function prints the "L2 View of the Book". It relies on the `L3Book::ForEachLevel` to iterate the levels on both sides simultaneously, from inner levels to outer levels.

```
class L3Book {
public:
    template <typename F>
    void ForEachLevel(F&& f) const {
    // f could be either a lambda function or a functor with the following function signature
    //   bool process_level(double bid_price, int64_t bid_qty, int64_t bid_count,
    //                      double ask_price, int64_t ask_qty, int64_t ask_count);

    // 1. For the above book snapshot, below is the sequence of events that should be delivered to f
        f(100.0, 10, 2, 101.0, 11, 3);
        f(99.0,  9,  3, 102.0, 11, 2);
        f( 0.0,  0,  0, 103.0, 12, 3);
    }
    // 2. NOTE: the return type of "f" is bool, and this function should consider f's return value:
    // -> true: continue the iteration if there's any more level
    // -> false: break the loop and stop the iteration
    // with the above event sequence , if f(99.0,  9,  3, 102.0, 11, 2) returns false, there should
    // be no call with f( 0.0,  0,  0, 103.0, 12, 3)
};
```

- `void PrintLevel3Book(...)`: The function prints the "L3 View of the Book". It relies on the `L3Book::ForEachOrder` to firstly iterate the sell side from outer levels to inner levels and then buy side from inner levels to outer levels. For each price level, it iterates the price level following the priority of the order, i.e. the FIFO ordering.

```
class L3Book {
public:
    template <typename F>
    void ForEachOrder(bool is_buy, bool inner_to_outer, F&& f) const {
    // - is_buy: the side of the book to iterate.
    //     true - iterate buy side book;
    //     false - iterate sell side book
    // - inner_to_outer: whether to iterate from inner levels to outer levels or from outer levels
    // to inner levels. In "L3 view of the Book" printing, we first iterate the sell side with
    // inner_to_outer=false (i.e. higher price to lower price), and then buy side with
    // inner_to_outer=true (i.e. higher price to lower price)
    // - f: could be either a lambda function or a functor with the following function signature
    //     bool process_order(bool is_buy, double level_price, int64_t level_qty,
    //                        int64_t qty, int64_t order_id);

    // For the above book snapshot, below is the sequence of events that should be delivered to f
        f(false, 103.00, 12, 1, 20);
        f(false, 103.00, 12, 1, 17);
        f(false, 103.00, 12, 10, 3);
        // calls to orders with price = 102 skipped here
        f(false, 101, 11, 3, 7);
        f(false, 101, 11, 4, 3);
        f(false, 101, 11, 4, 5);
        f(true,  100, 10, 2, 12);
        f(true,  100, 10, 8, 5);
        // calls to orders with price = 99.0 skipped here
    }
    // 2. NOTE: the return type of "f" is bool to control the iteration
    //       true: continue the iteration if there's any more order
    //       false: break the loop and stop the iteration

    // 3. HINT: there's a reversed iterator, i.e. rbegin(), rend(), with which you can iterate
    //    the container in reversed order.
    // a) In the L3Book, the Order pointers are saved in the Level in reversed order, i.e. Order
    //    added later is infront of orders added earlier. You could use the reversed iterator to
    //    iterate the container.
    // b) In the L3Book, both Buy side and Sell side books are stored from inner levels to outer
    //    levels. You could also  use reversed iterator to iterate the book from outer levels to
    //     inner levels.
};
```

2 Execution Messages Handlings

During the lecture, we have discussed the implementation of Add, Delete and Replace operations. In addition to the above three, there's another common type of order update message, i.e. the Order Execution Update. It is used to inform a order with given order_id is executed by exec_qty:

- partial execution: the exec_qty is less than the open_qty. It's similar to the inplace Replace Operation.
- full execution: the exec_qty is equal to the open_qty. It's similar to the Delete operation.

```
# L3 book view before the update
side      price        qty     orders
   S      99.30         18     4(15), 5(17), 7(25), 2(11),
   S      99.20         16     3(1), 6(8), 7(5),
   B      99.10          8     1(3), 2(6), 5(7),
   B      99.00          5     3(2), 2(9),
   B      98.90          5     4(10), 1(12),

# L3 book view after processing the execution msg: side=Sell, order_id=1, price=99.2, exec_qty=1.
# i.e. the order with order_id=1 is partially filled by a qty=1, so the Order's open_qty and the Level's
      aggregated qty are both reduced by 1
side      price        qty     orders
   S      99.30         18     4(15), 5(17), 7(25), 2(11),
   S      99.20         15     2(1), 6(8), 7(5),   # <- the 1st order is partially executed
   B      99.10          8     1(3), 2(6), 5(7),
   B      99.00          5     3(2), 2(9),
   B      98.90          5     4(10), 1(12),

# L3 book view after processing the execution msg: side=Sell, order_id=1, price=99.2, exec_qty=2
# i.e. the order with order_id=1 is now fully filled, so the order should be deleted and Levels's aggregated
       qty is reduced by 2
side      price        qty     orders
   S      99.30         18     4(15), 5(17), 7(25), 2(11),
   S      99.20         13     6(8), 7(5),   # <- the 1st order is fully executed
   B      99.10          8     1(3), 2(6), 5(7),
   B      99.00          5     3(2), 2(9),
   B      98.90          5     4(10), 1(12),
```

3 Book Uncrossing

In real life, there could be packet drops during the delivering of the order book update messages. And one possible consequence is crossed book, as explained in the following example.

Without any packet drops

```
# book snapshot before the update
    side      price         qty     orders
       S      99.30          18     4(15), 5(17), 7(25), 2(11),
       S      99.20           7     7(5),
       B      99.10           8     1(3), 2(6), 5(7),
       B      99.00           5     3(2), 2(9),
       B      98.90           5     4(10), 1(12),

# process msg action exec, side: S, order_id: 5, price: 99.2, qty: 7
    side      price         qty     orders
       S      99.30          18     4(15), 5(17), 7(25), 2(11),
       B      99.10           8     1(3), 2(6), 5(7),
       B      99.00           5     3(2), 2(9),
       B      98.90           5     4(10), 1(12),

# process msg action add, side: B, order_id: 30, price: 99.2, qty: 5
    side      price         qty     orders
       S      99.30          18     4(15), 5(17), 7(25), 2(11),
       B      99.20           5     5(30),
       B      99.10           8     1(3), 2(6), 5(7),
       B      99.00           5     3(2), 2(9),
       B      98.90           5     4(10), 1(12),
```

Without packet drops

```
# the second exec msg is lost, and thus the order with order_id=5 is retained in the book, i.e. stale order
# book snapshot before the update
    side      price         qty     orders
       S      99.30          18     4(15), 5(17), 7(25), 2(11),
       S      99.20           7     7(5),
       B      99.10           8     1(3), 2(6), 5(7),
       B      99.00           5     3(2), 2(9),
       B      98.90           5     4(10), 1(12),

# the exec msg is lost, and the order with order_id=5 stays in the book

# process msg action add, side: B, order_id: 30, price: 99.2, qty: 5
 side      price         qty     orders
    S      99.30          18     4(15), 5(17), 7(25), 2(11),
    S      99.20           7     7(5),                 # <- crossed book, bid side and ask side overlapped
    B      99.20           5     5(30),                # <- crossed book
    B      99.10           8     1(3), 2(6), 5(7),
    B      99.00           5     3(2), 2(9),
    B      98.90           5     4(10), 1(12),
```

The crossed book could be troublesome, and we can use some simple logic to uncross the book. When a order is added to the buy side with price P, any sell orders with prices less or equal to P are stale orders and should be purged. Similarly when a order is added to the sell side with price P, any buy orders with prices greater or equal to P are considered to be stale orders. The rationale behind the assumption is that newer operations (i.e. Add operation) should better represent the state of the book at that moment.

Without packet drops and book uncrossing

```
# the second exec msg is lost, and thus the order with order_id=5 is retained in the book, i.e. stale order
# book snapshot before the update
 side      price         qty     orders
    S      99.30          18     4(15), 5(17), 7(25), 2(11),
    S      99.20           7     7(5),
    B      99.10           8     1(3), 2(6), 5(7),
    B      99.00           5     3(2), 2(9),
    B      98.90           5     4(10), 1(12),

# the exec msg is lost, and the order with order_id=5 stays in the book
```

```
# process msg action add, side: B, order_id: 30, price: 99.2, qty: 5.
# temporary crossed book after the Add operation
 side      price         qty     orders
    S      99.30          18     4(15), 5(17), 7(25), 2(11),
    S      99.20           7     7(5),               # <- crossed book, bid side and ask side overlapped
    B      99.20           5     5(30),              # <- crossed book
    B      99.10           8     1(3), 2(6), 5(7),
    B      99.00           5     3(2), 2(9),
    B      98.90           5     4(10), 1(12),

# uncrossed book: the sell order with order_id=5 is purged as the price is equal to the price of the
# newly inserted order, i.e. price of order with order_id=5
 side      price         qty     orders
    S      99.30          18     4(15), 5(17), 7(25), 2(11),
    B      99.20           5     5(30),
    B      99.10           8     1(3), 2(6), 5(7),
    B      99.00           5     3(2), 2(9),
    B      98.90           5     4(10), 1(12),
```