QF633 Assignment 2

May 20, 2023

In this assignment, we will explore building a simple but very efficient order manager using the `<algorithm>` library.

An order is minimally represented by the following data structure:

```
struct Order {
    int price;              // the order's price, and int type used instead of double type for simplicity
    int open_qty;           // the open qty of the order
    std::string order_id;   // unique order identifier to identify the order
    bool is_sell;           // order side, true - buy order, false - sell order
};
```

An order manager is a class which manages all the open orders, i.e. order with `open_qty > 0`. The order manager should keep the records in a `std::vector` in a sorted manner:

- orders with lower prices should be placed before orders with higher prices
- orders with the same price should be sorted according to the order of insertion, i.e. FIFO order

```
class OrderManager {
public:
    // add a new record to the order manager
    void Add(int price, int qty, const std::string& order_id, bool is_sell);

    // update the open_qty for an existing order record and delete the record if updated_open_qty== 0
    void Update(const std::string& order_id, int updated_open_qty);

    // print all orders following the order specified, i.e.
    // - from lower prices to high prices
    // - following FIFO if same price
    void PrintAllOrders() const;

private:
    std::vector<Order> orders_;
};
```

With following code snippets, the expected output is inline commented.

```
    OrderManager order_manager;

    order_manager.Add(100, 2, "o1", true);
    order_manager.Add(99, 1, "o2", true);
    order_manager.Add(100, 3, "o3", true);
    order_manager.Add(101, 4, "o4", true);
    order_manager.PrintAllOrders();
    // price=99 open_qty=1 order_id=o2 is_sell=1
    // price=100 open_qty=2 order_id=o1 is_sell=1
    // price=100 open_qty=3 order_id=o3 is_sell=1
    // price=101 open_qty=4 order_id=o4 is_sell=1

    order_manager.Update("o3", 1);
    order_manager.Update("o2", 0);
    order_manager.Add(98, 5, "o5", false);
    order_manager.Add(101, 1, "o6", true);
```

```
    order_manager.Add(100, 2, "o7", true);
    order_manager.PrintAllOrders();
    // price=98 open_qty=5 order_id=o5 is_sell=0
    // price=100 open_qty=2 order_id=o1 is_sell=1
    // price=100 open_qty=1 order_id=o3 is_sell=1
    // price=100 open_qty=2 order_id=o7 is_sell=1
    // price=101 open_qty=4 order_id=o4 is_sell=1
    // price=101 open_qty=1 order_id=o6 is_sell=1
```

Approach 1: Add and then sort

One obvious approach is to simply append the new record to the `std::vector`, and then *sort* the `std::vector` to the desired order. Can you please given an implementation following this approach?

Hint: please take note the following description from std::sort documentation: *Sorts the elements in the range [first, last) in non-descending order. The order of equal elements is not guaranteed to be preserved.*.

Approach 2: Add while preserving the ordering

The other alternative approach is to insert the new report to the correct position of the `std::vector`, i.e. firstly find the position to insert the new record while preserving the ordering and secondly to insert the new record to the found location. With this approach, it's not necessary to call *sort* separately.

We could linearly scan the `std::vector` to find the correct position to insert, but a better way is to use binary search. During the lecture, we have introduced std::binary_search which can tell if an element exists in an given range. There are another two important functions in the *binary search operations* category, namely std::lower_bound and std::upper_bound. Can you use either function to find the correct position to insert the new record for this approach?

The following files have been provided in the zip file:

```
- order.h: declaration for struct Order
- order.cpp: definition for struct Order
- order_manager.h: declaration for class Order Manager
- order_manager_v1.cpp: implementation to be provided using approach 1
- order_manager_v2.cpp: implementation to be provided using approach 2
- main.cpp: main function which gives two simple test cases
- CMakeLists.txt: cmake project file to be provided
```

You are expected to:

- provide an implementation using approach 1 inside `order_manager_v1.cpp`.
- provide an implementation using approach 2 inside `order_manager_v2.cpp`.
- complete the `CMakeLists.txt` to compile an executable `order_manager_v1` for approach 1 and an executable `order_manager_v2` for approach 2.
- provide a PDF document `assignment2.pdf`, which describes
  - the time complexity for `OrderManager::Add`, and `OrderManager::Update` for both approaches.
  - rationale of how to make sure the desired ordering is maintained for both approaches

- create a zip file as `Assignment2_YourName.zip`, which contains 1) the pdf report, 2) the completed source codes as well as the `CMakeLists.txt`, and submit to eLearn.

[Bonus point]: You are encouraged to use functions from `<algorithm>` rather than writing plain for loop.