Objective

Create a matching engine that displays results based off of incoming orders.

Limit Order Stream (Input):

O/X/P | ID | SYM | B/S | Q | P

Result Stream (Output):

F/X/P/E | ID | SYM | B/S | Q | P

When does one result?

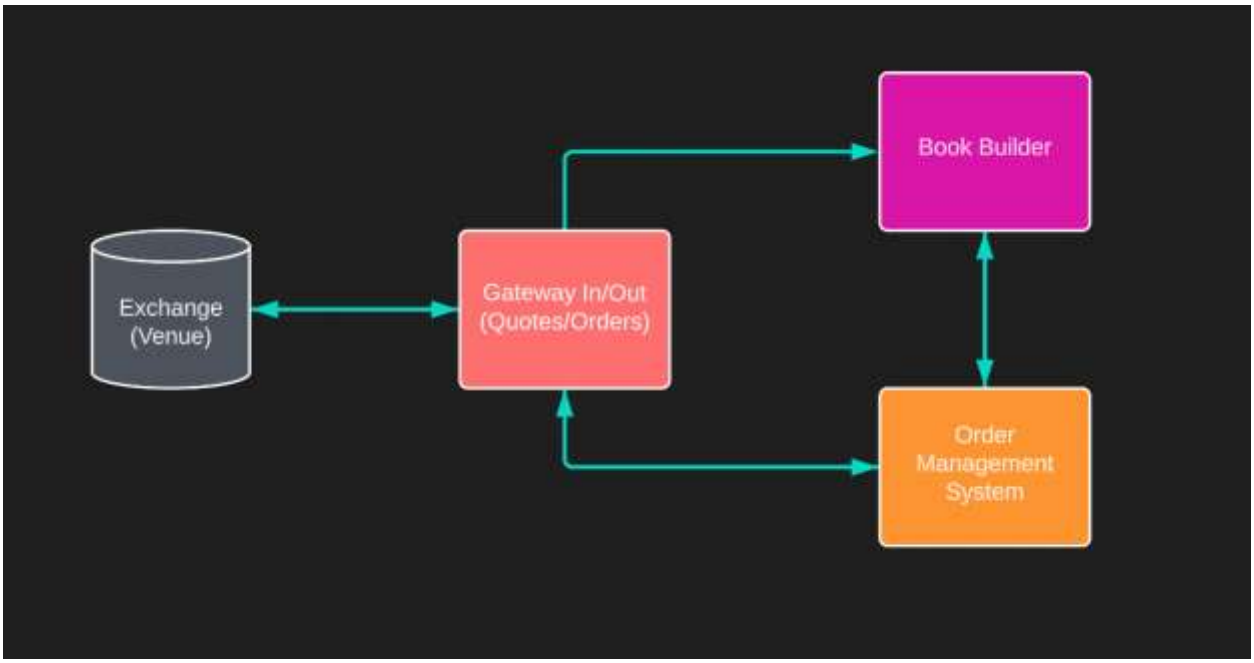
1. F – When a B & S order match by price, match them by the minimum quantity
    a. If an order where B_Price > S_Price exists, then match them by using price-time (FIFO) priority; this means tracking both price and time of each order is critical
2. X – When an order is canceled
3. P – When printing all open orders, by decreasing price
4. E – When ANY error occurs, be descriptive

What data structures will be needed?

1. Need a data structure to store orders
    a. Use a struct that stores all important order information
        i. Type, oid, side, qty, px, and symbol
    b. This struct has a name of **Order**
2. Need a data structure that will be able to hold the open orders, one for buy and one for sell
    a. This data structure needs to be able sort books by price, and then by time
        i. This is imperative for finding the best price-time (FIFO) priority in crossing
    b. This data structure needs to also hold multiple symbols
    c. Use an unordered_map<string, multimap> where the string allows for multiple symbols, and the multimap allows for buy and sell books for each symbol. The maps have names of **buyBooks** and **sellBooks**
        i. The buy and sell unordered_map will have a key of symbol, and value of multimap
            1. The multimap has a key of <double, time> for price and time
                a. This allows for FIFO priority
            2. The multimap has a value of Order
                a. This holds the current order for a given price and time
            3. Multimap is chosen instead of an unordered_multimap as we need to make sure we deal with buys and sells in order of price, unordered multimap would be less efficient in this case as its not sorted
            4. Multimap has a name of **order_book_t**
3. Need a data structure that will be able to effectively print the open orders
    a. Since the orders need to be printed by price and prices can differ between symbols, a new data structure of map<price, vector<order.ids>> is utilized

    b. This data structure has a name of **orderMap_sortedbyPrice**
    c. This stores all open orders by price, and if multiple orders for a price exist, they are stored sequentially in the vector
4. Need a data structure to reference an order directly by order.id, use an unordered_map<order.id, Order>
    a. This is useful for certain operations like removing orders, as the order can be removed from all data structures knowing just the order.id, and order.id lookup is O(1)
    b. This data structure has a name of **orderMap**
5. For canceled orders, simply print the canceled order ID
    a. Remove orders directly using the given order.id
6. For printed orders, use the orderMap_sortedbyPrice
    a. Iterate in descending order of price and print the order information
7. Errors will occur if:
    a. Input Syntax is incorrect (Unique error message for each)
        i. Incorrect Order Action (Not O/X/P)
        ii. Duplicate Order ID, Non Positive 32-Bit Int
        iii. Incorrect Symbol -> Either Empty string or greater than 8 size string
        iv. Incorrect B/S Syntax (Anything other than B or S is input)
        v. Non Positive 16-Bit Int Quantity
        vi. Non Positive Double for Price, or not in 7.5 format
        vii. No order ID matches canceled order

System Architecture



Here is the basic layout of the matching engine. Each component operates as such:

1. Exchange (Venue)
    a. The exchange supplies data on all NYSE (or other exchange) equities

2. Gateway
    a. In
        i. Uses exchange data to gather quotes about specific equities
    b. Out
        i. Uses OMS data to send out orders to the exchange
    c. The gateway supplies these orders and quotes to both the book builder and the OMS for further processing
3. Book Builder
    a. The book builder receives quote updates from the gateway, and order updates from the OMS
    b. It processes these updates to maintain and update the order book data structures for each traded instrument
    c. The book builder does not send anything back to the gateway; its role is to maintain an accurate representation of the order book
4. Order Management System (OMS)
    a. The OMS receives orders and quotes from the gateway
    b. It performs order validation, order processing, order canceling, order printing, and error handling
    c. The OMS communicates with the book builder to ensure consistency between the OMS and the order books
    d. The OMS sends the processed orders back to the gateway for execution

Simple_Cross Architecture

```cpp
#pragma once

#include <string>
#include <list>
#include <unordered_map>
#include <map>
#include <ctime>
#include <vector>

struct alignas(8) Order {
    char type;
    uint32_t oid;
    char side;
    uint32_t qty;
    double px;
    std::string symbol;
};

typedef std::list<std::string> results_t;
//we need the books to be sorted by first price, then by time, to ensure FIFO priority
typedef std::multimap<std::pair<double, std::time_t>, Order> order_book_t;

class SimpleCross {
    private:
        results_t results;
        Order order;
        std::unordered_map<std::string, order_book_t> buyBooks;
        std::unordered_map<std::string, order_book_t> sellBooks;
        std::unordered_map<uint32_t, Order> orderMap;
        std::map<double, std::vector<uint32_t>> orderMap_sortedbyPrice;

        void addOrder(const Order& order) noexcept;
        void removeOrder(uint32_t oid) noexcept;
    public:
        [[nodiscard]] Order parseLine(const std::string& line);
        void validateOrder(const Order& order); //Handle input errors
        [[nodiscard]] results_t processOrder(Order& order);
        [[nodiscard]] results_t processCancel(Order& order);
        [[nodiscard]] results_t processPrint();
        [[nodiscard]] results_t action(const std::string& line);
};

```

1. Order struct organizes order data coming in
    a. Note: Struct is memory aligned along 8 bytes (default for 64 bit system)
    b. Note: std::string symbol may be optimized by char symbol[8] to reduce size, however std::string at small sizes automatically utilizes SSO to reduce size
2. results_t type for results output
3. order_book_t for buy and sell book data structures

    a. Note: This data structure is a multimap where the key is a price-time pair, and the value is the specific order this price-time pair is associated with, this is critical to obtain FIFO price-time during crossing
4. buyBooks and sellBooks for ordering buy and sell books by symbol
5. orderMap for storing all orders by unique id
6. orderMap_sortedbyPrice for printing all open orders by price
7. addOrder and removeOrder update the book builder
8. OMS Functions:
    a. parseLine
        i. initializes the order struct, if parsing incorrect value, throw an exception
    b. validateOrder
        i. validates the order is in a correct format, otherwise throws an exception
    c. processOrder
        i. processes order
        ii. If order is 'O' then books and map are adjusted, and function also performs FIFO cross checking
        iii. If order is 'X' then order is removed from respective book, orderMap, and orderMap_sortedbyPrice (see processCancel)
        iv. If order is 'P' then order is printed using orderMap_sortedbyPrice (see processPrint)
    d. processCancel
        i. cancels order
        ii. updates respective data structures
    e. processPrint
        i. prints sell and buy books, respectively
    f. action
        i. main entry point for each input line

File Organization

1. Repository consists of
    a. This solution manual
    b. A CMakeLists.txt for building the solution
    c. Src folder
        i. Book_builder.cpp
        ii. Order_management_system.cpp
        iii. Main.cpp
    d. Includes folder
        i. Simple_cross.h
        ii. Book_builder.h
        iii. Order_management_system.h
    e. Testcases folder
        i. Actions_default.txt
        ii. IncorrectInputs.txt
        iii. MultipleSymbols.txt

iv. SuccessfulCrosses.txt

Test Cases

Testing occurs in benchmarking and test-driven data sets.

Benchmarking is done using std::chrono to find out the time of execution from first line to last of each file

Test-driven data sets include 4 sets of test cases to test error handling and accuracy of solution. All test cases pass.

Actions_default.txt is the test case given by 3Red Partners.

IncorrectInputs.txt tests the error handling of incorrect inputs into the system.

MultipleSymbols.txt tests the solutions ability to handle multiple symbols, also utilizes cross checking.

SuccessfullCrosses.txt tests the solutions ability to handle cross checked orders.