**1. Order Structure (Order)**

struct Order {

int orderId;

double price;

int quantity;

Order(int id, double p, int q) : orderId(id), price(p), quantity(q) {}

};

* **Purpose**: Represents an individual order with fields:
  + orderId: Unique identifier for the order.
  + price: Price of the order.
  + quantity: Quantity of the order.
* **Why?**: Encapsulates order details in a reusable and organized way.

**2. OrderBook Class**

class OrderBook {

private:

std::map<int, Order> buyOrders; // Stores buy orders

std::map<int, Order> sellOrders; // Stores sell orders

std::mutex mtx; // Ensures thread-safe access

public:

void addBuyOrder(const Order& order);

void addSellOrder(const Order& order);

void processBuyOrders();

void processSellOrders();

};

* **Purpose**: Manages buy and sell orders in separate maps (buyOrders, sellOrders) and ensures thread-safe operations using std::mutex.
* **Key Functions**:
  1. **addBuyOrder** and **addSellOrder**:
     + Add a buy/sell order to the respective map.
     + Use std::lock\_guard<std::mutex> to prevent simultaneous modifications.
  2. **processBuyOrders** and **processSellOrders**:
     + Process all buy/sell orders in the map.
     + Use std::lock\_guard<std::mutex> to prevent conflicts with other threads modifying the maps.
* **Why?**:
  1. Centralizes the logic for managing and processing orders.
  2. Uses mutex to avoid **race conditions** when multiple threads access buyOrders and sellOrders.

**3. SocketConnection Class**

class SocketConnection {

private:

SOCKET udpSocket, tcpSocket;

sockaddr\_in udpAddr, tcpAddr;

bool tcpConnected;

public:

SocketConnection(const std::string& udpIp, const std::string& tcpIp);

bool connectTcp();

void disconnectTcp();

int receiveUdp(char\* buffer, int bufferSize);

void sendTcp(const std::string& data);

~SocketConnection();

};

* **Purpose**: Handles network communication for the trading system.
  + **UDP Socket**: Receives market data from the exchange.
  + **TCP Socket**: Sends processed order data back to the exchange.
* **Key Functions**:
  + **connectTcp**:
    - Establishes a TCP connection with the exchange.
  + **receiveUdp**:
    - Receives UDP packets containing market data.
  + **sendTcp**:
    - Sends serialized order data to the exchange via TCP.
  + **Destructor**:
    - Cleans up sockets and calls WSACleanup() to release Winsock resources.
* **Why?**:
  + Manages communication efficiently with separate protocols (UDP for fast, connectionless market data; TCP for reliable order submissions).

**4. MarketDataHandler Class**

class MarketDataHandler {

private:

SocketConnection& socketConnection;

OrderBook& orderBook;

public:

MarketDataHandler(SocketConnection& socket, OrderBook& book);

void processMarketData(const std::string& data);

void handleDataStream();

};

* **Purpose**: Decouples the logic for handling market data from the rest of the system.
* **Key Functions**:
  1. **processMarketData**:
     + Parses market data using JSON and updates the OrderBook (buy or sell orders).
     + Determines whether the data corresponds to a buy or sell action and calls the appropriate OrderBook function.
  2. **handleDataStream**:
     + Continuously listens for UDP packets using SocketConnection::receiveUdp.
     + Passes received data to processMarketData.
* **Why?**:
  1. Separates market data processing from socket operations for modularity.
  2. Allows multithreading: handleDataStream can run independently in a thread.

**5. Main Function**

int main() {

const std::string udpIp = "127.0.0.1";

const std::string tcpIp = "127.0.0.1";

SocketConnection socket(udpIp, tcpIp);

OrderBook orderBook;

MarketDataHandler handler(socket, orderBook);

if (socket.connectTcp()) {

// Multithreading

std::thread buyThread(&OrderBook::processBuyOrders, &orderBook);

std::thread sellThread(&OrderBook::processSellOrders, &orderBook);

std::thread dataStreamThread(&MarketDataHandler::handleDataStream, &handler);

buyThread.join();

sellThread.join();

dataStreamThread.join();

}

return 0;

}

* **Purpose**:
  + Sets up the system and manages its threads.
* **Steps**:
  + **Initialize SocketConnection**:
    - Creates UDP and TCP sockets.
    - Connects to the exchange via TCP.
  + **Start Threads**:
    - buyThread: Processes buy orders.
    - sellThread: Processes sell orders.
    - dataStreamThread: Handles incoming market data.
  + **Thread Joining**:
    - Ensures all threads complete before the program exits.
* **Why?**:
  + Achieves concurrency: Buy, sell, and data handling run independently.
  + Ensures proper resource cleanup with join().

**Features**

1. **Multithreading**:
   * Separate threads handle buy orders, sell orders, and data streams.
   * Boosts performance by running tasks in parallel.
2. **Thread Safety with Mutex**:
   * Prevents race conditions when accessing buyOrders and sellOrders.
3. **UDP and TCP Communication**:
   * UDP: For receiving high-speed, connectionless market data.
   * TCP: For reliable submission of processed order data.
4. **JSON Parsing**:
   * Simplifies handling structured market data using the json/json.h library.
5. **Scalable Design**:
   * Modular classes (SocketConnection, OrderBook, MarketDataHandler) make it easy to extend functionality.

**Why We Use These Components**

| **Component** | **Purpose** | **Benefit** |
| --- | --- | --- |
| OrderBook | Centralized management of buy and sell orders. | Thread-safe operations on orders with mutex. |
| SocketConnection | Handles communication via UDP (market data) and TCP (order submission). | Efficient, protocol-specific communication. |
| MarketDataHandler | Decouples market data processing from socket operations. | Modular, extensible design. |
| Multithreading | Concurrent execution of buy/sell processing and market data handling. | Improved performance and responsiveness. |
| Mutex (std::mutex) | Prevents race conditions when accessing shared resources (buyOrders and sellOrders). | Ensures data consistency in a multithreaded environment. |
| JSON Library | Parses structured market data into C++ objects. | Simplifies data handling. |