Tibra Coding Assignment

Our thanks for accepting the challenge of the Tibra coding assignment. This document describes the problem to be solved and the constraints around the solution.

Overview

The problem to be solved is to provide an implementation of a simplified stock exchange. More specifically, there is a C++ interface class below, which your implementation should implement. In order to prove the correctness of your implementation, we also ask you to provide a suite of unit tests.

The exchange interface described below is asynchronous by design. Client classes are expected to call the functions on the interface, and replies and other notifications are to be given via events.

The exchange interface events are modeled below as function pointers, which would be set by the client to an appropriate function. If it helps, you might want to change these to use `boost::function` or `std::tr1::function` objects. In either case you can assume that the client would provide a handler for these events and it is your implementations responsibility to fire these events (i.e. call these functions) at the appropriate times.

For the purposes of this exercise you can assume that the exchange is single-threaded and there is a single client. No network or IPC code is required, the implementation and its tests can be all run in a single process.

Exchange Interface

The purpose of the exchange is to maintain collections of orders to buy or sell stock. In this simplistic exchange there is no order matching (i.e. trading) but you might like to consider how this would be added in future. This means that the orders can "cross", such that there may exist a buy order which is at a higher price than a sell order for a given stock. This is similar to a real exchange which is in "pre-open".

The methods on the exchange are as follows:

- `AddOrder` which is used to add orders. Some information about the order is provided by the client, discussed below. Once the `AddOrder` operation is complete, an `OnOrderAdded` event is fired, indicating success or failure. The client matches this event with the original `AddOrder` operation through the `clientOrderId` value which is preserved by the exchange.
- `RemoveOrder` which is used to remove orders. The only information required is the `exchangeOrderId` which is provided by the exchange in the `OnOrderAdded` event when an order is added successfully. Once the `RemoveOrder` operation completes, the `OnOrderRemoved` event is fired to indicate success or failure.

The events on the exchange are as follows:

- `OnOrderAdded` is fired when an `AddOrder` operation completes.
- `OnOrderRemoved` is fired when an `RemoveOrder` operation completes.
- `OnBestPriceFeed` is fired when the best price changes.

The `AddOrder` operation requires the following information:

- `stockCode`: a string identify the stock. The exchange should support the following stock codes: BHP, RIO, ANZ.
- `bidSide`: a boolean to indicate whether the order is to buy (true) or sell (false) stock.
- `price`: the price at which the client wants to buy or sell. Every non-zero positive integer can be assumed to be a valid price.
- `volume`: the number of shares client wishes to buy or sell. Every non-zero positive integer can be assumed to be a valid volume.
- `clientOrderId`: an integer supplied by the client, which it uses to identify the corresponding order when the `OnOrderAdded` event is fired. There are no requirements for the implementation to validate these numbers; the client can be assumed to be well-behaved in this respect.

The error codes specified in the interface declaration should be used in all the callback events where appropriate

The `exchangeOrderId` is to be generated by the implementation and must be unique for every order.

The current exchange interface is designed to be a "broadcast exchange" (i.e., feeds are published for all supported stocks).

Best Price

You should only publish best price feeds (`OnBestPriceFeed` callback) when the best price (ie the highest bid or lowest ask price) changes, OR when the total volume of orders at the current best price changes. For example, assume the following order book state:

BID	ASK
price: 100.00, volume: 200	[empty]
price: 90.00, volume: 100	
price: 40.00, volume: 5	

If a new order is added on the bid side at any price < $100.00 then there is no price feed generated because it is not at the best level. However, if an order is added on the bid side with a price >= $100.00 or any order is added on the ask side then a best price feed needs to be published. Similarly if there is a new buy order of $100 with volume 50, there will need to be a new best price with a bid of $100 and volume 250.

In cases when there is no bid/ask price you should publish a price/volume of 0.

You can assume upon starting the application there are no orders/prices in the market.

Terminology

Order

An order is how you express an interest to buy or sell a specified volume of something in a securities market, at a specified price. If you are buying, you are placing a bid; conversely if you are selling you are placing an offer (ask).

Order Book (Price Information)

An Order Book keeps track of all individual orders placed into a market at all times. It organises the orders based on the security the order is for (e.g., BHP), order type or side (bid or ask), order price, and the time sequence in which orders were entered at each price level. The volume of each order must also be stored.

By way of example:

Order Book (keeps track of individual volumes at each price level):

BID	ASK
price: 100.00, volume: 200	price: 150.00, volume: 200, volume: 5
price: 90.00, volume: 100	price: 200.00, volume: 1
price: 40.00, volume: 5, volume: 1	price: 250.00, volume: 10
	price: 300.00, volumes: 10

Price Feed

A price feed is a message/call-back (`OnBestPriceFeed`) that is generated by the exchange indicating the current “best” bid or ask price, and the TOTAL volume available in the market at that price. Eg, from the Order Book described above, the current “best prices” would be:

Best Price:

BID	ASK
price: 100.00, volume: 200	price: 150.00, volume: 205

A Price Feed is generated whenever the “best price” for the bid or ask side of the Order Book changes.

Solution Constraints

You should use your best C++ development practices to demonstrate how you would code in a production environment (i.e., highest levels of quality).

Implementation of your solution should include a simple standalone/console application (i.e., no graphical user interface or network interface required). This application should demonstrate the correctness of your solution in a suite of automated unit tests. You are free to use a unit testing framework such as Boost.Test if desired. The quality of your unit tests are considered equally important as the solution itself. Please be mindful of this and ensure your unit tests cover all important requirements for the solution. A good quality unit test suite would test the implementation only through the public interface, `IStockExchange`.

You should code to be as performance oriented as possible and be mindful of time and space complexity of algorithms/data structures.

It is expected all errors are handled/reported unless stated otherwise.

It will be highly regarded if you can detail, in bullet points, how the design of the interface could be improved.

Please email your solution to the email address supplied to you during your interview. If done using Visual Studio/Windows please send the entire solution (source code + VS .vcproj/.sln files), if done on Linux etc, please send source files and a Makefile.

IStockExchange

```
namespace Tibra { namespace Exchange {

// IStockExchange interface
class IStockExchange
{
public:
    // Public Methods

    virtual ~IStockExchange() {}

    virtual void AddOrder(
        const std::string& stockCode,
        bool bidSide,
        int price,
        int volume,
        int clientOrderId) = 0;

    virtual void RemoveOrder(
        int exchangeOrderId) = 0;

    // Public Events

    void (*OnOrderAdded)(
        int exchangeOrderId,
        int clientOrderId,
        int errorCode);

    void (*OnOrderRemoved)(
        int exchangeOrderId,
        int errorCode);

    void (*OnBestPriceFeed)(
        const std::string& stockCode,
        int bidPrice,
        int bidVolume,
        int askPrice,
        int askVolume);

    // Define error codes that can be returned by implemented call-backs
    //
    enum tCallBackErrorCodes
    {
        eNO_ERROR = 0,
        eINTERNAL_SYSTEM_FAILURE = 1,
        eNONEXISTENT_ORDER = 2,
        eUNKNOWN_STOCK_CODE = 3,
        eINVALID_ORDER_PRICE = 4,
        eINVALID_ORDER_VOLUME = 5,
    };
};

}};
```