User Manual

To compile the client code:
g++ -std=c++11 .\client.cpp -lws2_32 -o client

To compile the server code:
g++ -std=c++11 .\server.cpp -lws2_32 -o server

To run the executables:
server.exe
client.exe [ip-address] (**localhost** in this case) (eg. client.exe localhost)

To compile and run the testing code:
g++ -std=c++11 .\test.cpp -o test
test.exe

Formats for orders and packet

Order:
order(float price,int quantity,bool isBuy,string name)
Example: Buying 10 “AAA” stock for $9.50 -> order(9.50, 10, true, “AAA”)
Float is chosen because precision of double is not needed.
Int should be able to cover any quantity of stock

Packet format:
Price|Quantity|isBuy|Name|
Example: Buying 10 “AAA” stock for $9.50 -> “950|10|1|AAA”

Test Plan

All of my tests are included in test.cpp.

We first test the orders to see if they are able to handle <=0 prices and quantity. The code should throw an exception to both of them.

```
ID: 0 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 1
Price cannot be negative or zero: 0.000000
ID: 2 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 1
Price cannot be negative or zero: -0.500000
ID: 4 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 1
Quantity cannot be negative or zero: 0
ID: 6 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 1
Quantity cannot be negative or zero: -100
```

Next we test the stock orders to see if it is able to handle matching priority and partial fill of orders.

```
----------------------------------------
Testing partial fill of buy orders
Order added: ID: 20 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 0
Order added: ID: 21 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 0
Order added: ID: 22 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 0
Order added: ID: 23 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 0
Order added: ID: 24 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 0
Order added: ID: 25 | Name: aaa | Price: 1.01 | Original quantity: 4 | Remaining Quantity: 4 | isBuy: 1
Order matched: ID: 20 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 0 | isBuy: 0
Order matched: ID: 21 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 0 | isBuy: 0
Order matched: ID: 22 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 0 | isBuy: 0
Order matched: ID: 23 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 0 | isBuy: 0

----------------------------------------
Buy Orders empty
Sell Orders:
ID: 24 | Name: aaa | Price: 1.01 | Original quantity: 1 | Remaining Quantity: 1 | isBuy: 0
```

We then test the parser to see if it is able to handle packets properly.

The final test is to test the matching algorithm. The algorithm works by filling the order which has the highest buy price/ lowest sell price.
For example, if there are outstanding buy orders at $1, $2, $3, $4, and a sell order of $2 comes in, the buy order of $4 will get filled first.
Likewise, if there are outstanding sell orders at $1, $2, $3, $4, and a buy order of $3 comes in, the sell order of $1 will get filled first.

Data Generator

The data generator is done by calling generateValues in parser.cpp, followed by inputting the output of generateValues into generatePacket in parser.cpp. Variables such as number of generated stock names, price and quantity can be changed.