# Appendix

## C++ Code

### EnumTraderType

#### EnumTraderType.h

1. #pragma once
3. **enum** TraderType
4. {
5. NULL\_TRADER=-1, RANDOM\_TRADER=0, LARGE\_RANDOM\_TRADER=1, POSITION\_TRADER=2, MOMENTUM\_TRADER=3
6. };
8. **static** std::string TraderTypeToString(TraderType type)
9. {
10. **switch** (type)
11. {
12. **case** RANDOM\_TRADER:
13. **return** "RANDOM\_TRADER";
14. **case** LARGE\_RANDOM\_TRADER:
15. **return** "LARGE\_RANDOM\_TRADER";
16. **case** POSITION\_TRADER:
17. **return** "POSITION\_TRADER";
18. **case** MOMENTUM\_TRADER:
19. **return** "MOMENTUM\_TRADER";
20. **default**:
21. **return** "Unknown";
22. }
23. }
25. **static** TraderType StringToTraderType(std::string text)
26. {
27. **if** (text == "RANDOM\_TRADER")
28. **return** RANDOM\_TRADER;
29. **if** (text == "LARGE\_RANDOM\_TRADER")
30. **return** LARGE\_RANDOM\_TRADER;
31. **if** (text == "POSITION\_TRADER")
32. **return** POSITION\_TRADER;
33. **if** (text == "MOMENTUM\_TRADER")
34. **return** MOMENTUM\_TRADER;
35. **else**
36. **return** NULL\_TRADER;
37. }

### Trader

#### Trader.h

1. #pragma once
3. #include "Logger.h"
4. #include "Trade.h"
5. #include "Order.h"
6. #include "EnumTraderType.h"
7. #include "OpenClStructs.h"
8. #include "OrderBook.h"
10. **class** OrderBook;
12. **class** Trader
13. {
14. **protected**:
15. **int** \_id;
16. **static** **int** \_lastId;
17. TraderType \_type;
19. **int** \_currentT;
20. **int** \_processT;
21. **int** \_lastCompleteT;
23. **double** \_startCash;
24. **int** \_startVolume;
25. **double** \_cashPosition;
26. **double** \_cashPosWOrders;
28. std::list<Order> \_pendingOrders;
29. std::list<Order> \_completedOrdersV;
30. std::vector<Trade> \_tradesV;
32. std::map<std::string,**int**> \_stockPositions;
33. std::map<std::string,**int**> \_stockPosWOrders;
35. **static** **int** getNextId();
37. **void** updateCashPos(**double** amount);
38. **void** updateCashPosWOrder(**double** amount);
40. **void** addPendingOrder(Order order);
41. **void** removePendingOrder(Order order);
42. **void** completedPendingOrder(Order order);
44. **void** addStockPosition(Stock\* stock, **int** volume);
45. **void** removeStockPosition(Stock\* stock);
47. **void** updateStockPosition(Stock\* stock, **int** amount);
48. **void** updateStockPosWOrder(Stock\* stock, **int** amount);
50. std::string logName;
51. std::ofstream \_completedOrders;
52. std::ofstream \_trades;
53. std::string \_directory;
54. **static** **const** std::string \_tradesFileName;
55. **static** **const** std::string \_completedOrdersFileName;
57. **protected**:
58. **void** WriteTradeToFile(std::string text);
59. **void** WriteCompletedOrderToFile(std::string text);
61. **public**:
62. Trader();
63. Trader(Trader& trader);
64. Trader(Stock\* stock, **int** volume, **double** cash, **int** processTime, TraderType type);
65. ~Trader();
67. **void** update(**int** time);
68. **void** notify(Trade\* trade);
70. std::string toString();
71. std::string toString(std::string symbol);
72. **int** GetId();
73. **int** GetCurrentT();
74. **int** GetProcessT();
75. **int** GetLastCompleteT();
76. **double** GetCashPos();
77. **double** GetCashPosWO();
78. **double** GetStartCash();
79. **int** GetStartVol();
80. TraderType GetType();
81. std::list<Order> GetPendingOrders();
82. std::list<Order> GetCompletedOrders();
83. std::vector<Trade> GetTrades();
84. std::map<std::string, **int**> GetStockPositions();
85. std::map<std::string, **int**> GetStockPosWO();
87. **bool** ready();
89. TraderCL getTraderCL(std::string symbol);
90. **void** processTraderCL(TraderCL tcl, OrderBook\* book);
92. **size\_t** SizeOf();
94. **double** GetProfit();
95. };

#### Trader.cpp

1. #include "stdafx.h"
2. #include "Trader.h"
4. **int** Trader::\_lastId = -1;
5. **const** std::string Trader::\_tradesFileName = "Trades.csv";
6. **const** std::string Trader::\_completedOrdersFileName = "Orders.csv";
8. Trader::Trader()
9. {
10. \_id = getNextId();
11. \_currentT = 0;
12. \_processT = INT\_MAX;
13. \_lastCompleteT = 0;
14. \_cashPosition = 0.0;
15. \_cashPosWOrders = 0.0;
16. \_startCash = 0.0;
17. \_startVolume = 0;
18. logName = "Trader" + Utils::ItoS(\_id);
20. std::stringstream dir1;
21. dir1 << Logger::GetInstance()->GetDirectory() << "Traders\\";
22. std::wstring ws1 = Utils::s2ws(dir1.str());
23. CreateDirectory(ws1.c\_str(), NULL);
24. dir1 << "Trader" << \_id << "-" << \_processT << "\\";
25. ws1 = Utils::s2ws(dir1.str());
26. CreateDirectory(ws1.c\_str(), NULL);
27. \_directory = dir1.str();
29. \_trades.open(\_directory + \_tradesFileName);
30. **this**->WriteTradeToFile(Trade::toStringHeaderCSV());
31. \_completedOrders.open(\_directory + \_completedOrdersFileName);
32. **this**->WriteCompletedOrderToFile(Order::toStringHeaderCSV());
33. }
35. Trader::Trader(Trader& trader)
36. {
37. \_id = trader.GetId();
38. \_currentT = trader.GetCurrentT();
39. \_processT = trader.GetProcessT();
40. \_lastCompleteT = trader.GetLastCompleteT();
41. \_cashPosition = trader.GetCashPos();
42. \_cashPosWOrders = trader.GetCashPosWO();
43. \_startCash = trader.GetStartCash();
44. \_startVolume = trader.GetStartVol();
45. \_type = trader.GetType();
46. \_pendingOrders = trader.GetPendingOrders();
47. \_completedOrdersV = trader.GetCompletedOrders();
48. \_tradesV = trader.GetTrades();
49. \_stockPositions = trader.GetStockPositions();
50. \_stockPosWOrders = trader.GetStockPosWO();
51. logName = "Trader" + Utils::ItoS(\_id);
53. std::stringstream dir1;
54. dir1 << Logger::GetInstance()->GetDirectory() << "Traders\\";
55. std::wstring ws1 = Utils::s2ws(dir1.str());
56. CreateDirectory(ws1.c\_str(), NULL);
57. dir1 << "Trader" << \_id << "-" << \_processT << "\\";
58. ws1 = Utils::s2ws(dir1.str());
59. CreateDirectory(ws1.c\_str(), NULL);
60. \_directory = dir1.str();
62. \_trades.open(\_directory + \_tradesFileName);
63. **this**->WriteTradeToFile(Trade::toStringHeaderCSV());
64. \_completedOrders.open(\_directory + \_completedOrdersFileName);
65. **this**->WriteCompletedOrderToFile(Order::toStringHeaderCSV());
67. }
69. Trader::Trader(Stock\* stock, **int** volume, **double** cash, **int** processTime, TraderType type)
70. {
71. \_id = getNextId();
72. \_currentT = 0;
73. \_processT = processTime;
74. \_lastCompleteT = 0;
75. \_cashPosition = cash;
76. \_cashPosWOrders = cash;
77. \_startCash = cash;
78. \_startVolume = volume;
80. \_stockPositions.insert(std::make\_pair(stock->getSymbol(), volume));
81. \_stockPosWOrders.insert(std::make\_pair(stock->getSymbol(), volume));
83. \_type = type;
85. logName = "Trader" + Utils::ItoS(\_id);
86. //\_logger = Logger::GetInstance(LOGLEVEL);
88. std::stringstream dir1;
89. dir1 << Logger::GetInstance()->GetDirectory() << "Traders\\";
90. std::wstring ws1 = Utils::s2ws(dir1.str());
91. CreateDirectory(ws1.c\_str(), NULL);
92. dir1 << "Trader" << \_id << "-" << \_processT << "\\";
93. ws1 = Utils::s2ws(dir1.str());
94. CreateDirectory(ws1.c\_str(), NULL);
95. \_directory = dir1.str();
97. \_trades.open(\_directory + \_tradesFileName);
98. **this**->WriteTradeToFile(Trade::toStringHeaderCSV());
99. \_completedOrders.open(\_directory + \_completedOrdersFileName);
100. **this**->WriteCompletedOrderToFile(Order::toStringHeaderCSV());
101. }
103. Trader::~Trader()
104. {
105. \_trades.close();
106. \_completedOrders.close();
107. }
109. **void** Trader::update(**int** time)
110. {
111. \_currentT = time;
112. }
114. **void** Trader::notify(Trade\* trade)
115. {
116. **if** (trade->getBuyOrder().getParticipant() == \_id)
117. {
118. updateCashPos(- (trade->getPrice() \* trade->getSize()));
119. updateStockPosition(&trade->getBuyOrder().getStock(), trade->getSize());
120. completedPendingOrder(trade->getBuyOrder());
121. \_tradesV.push\_back(\*trade);
122. **this**->WriteTradeToFile(trade->toStringCSV());
123. }
124. **else** **if** (trade->getSellOrder().getParticipant() == \_id)
125. {
126. updateCashPos( (trade->getPrice() \* trade->getSize()));
127. updateStockPosition(&trade->getSellOrder().getStock(), -trade->getSize());
128. completedPendingOrder(trade->getSellOrder());
129. \_tradesV.push\_back(\*trade);
130. **this**->WriteTradeToFile(trade->toStringCSV());
131. }
132. }
134. std::string Trader::toString()
135. {
136. std::stringstream tempSS;
137. **char** str[256];
138. sprintf\_s(str, "Trader[%d]: Cash[$%.2f], PositionCash[$%.2f], ", \_id, \_cashPosition, \_cashPosWOrders);
139. tempSS << str;
141. auto it1 = \_stockPositions.begin();
142. auto it2 = \_stockPosWOrders.begin();
144. **while** (it1 != \_stockPositions.end() && it2 != \_stockPosWOrders.end())
145. {
146. std::stringstream temp;
147. temp << "Holding:" << it1->first << "[" << it1->second << "], PositionHolding:" << it2->first << "[" << it2->second << "], ";
148. tempSS << temp.str();
149. it1++;
150. it2++;
151. }
153. **return** tempSS.str();
154. }
156. std::string Trader::toString(std::string symbol)
157. {
158. **char** str[128];
159. sprintf\_s(str, "Trader[%d]: Cash[$%.2f], PositionCash[$%.2f], Holding:%s[%d], PositionHolding:%s[%d]", \_id, \_cashPosition, \_cashPosWOrders, symbol, \_stockPositions[symbol], symbol, \_stockPosWOrders[symbol]);
160. std::string temp(str);
161. **return** temp;
162. }
164. **int** Trader::GetId()
165. {
166. **return** \_id;
167. }
169. **int** Trader::GetCurrentT()
170. {
171. **return** \_currentT;
172. }
174. **int** Trader::GetProcessT()
175. {
176. **return** \_processT;
177. }
179. **int** Trader::GetLastCompleteT()
180. {
181. **return** \_lastCompleteT;
182. }
184. **double** Trader::GetCashPos()
185. {
186. **return** \_cashPosition;
187. }
189. **double** Trader::GetCashPosWO()
190. {
191. **return** \_cashPosWOrders;
192. }
194. **double** Trader::GetStartCash()
195. {
196. **return** \_startCash;
197. }
199. **int** Trader::GetStartVol()
200. {
201. **return** \_startVolume;
202. }
204. TraderType Trader::GetType()
205. {
206. **return** \_type;
207. }
209. std::list<Order> Trader::GetPendingOrders()
210. {
211. **return** \_pendingOrders;
212. }
214. std::list<Order> Trader::GetCompletedOrders()
215. {
216. **return** \_completedOrdersV;
217. }
219. std::vector<Trade> Trader::GetTrades()
220. {
221. **return** \_tradesV;
222. }
224. std::map<std::string, **int**> Trader::GetStockPositions()
225. {
226. **return** \_stockPositions;
227. }
229. std::map<std::string, **int**> Trader::GetStockPosWO()
230. {
231. **return** \_stockPosWOrders;
232. }
234. //Protected:
236. **int** Trader::getNextId()
237. {
238. \_lastId++;
239. **return** \_lastId;
240. }
242. **void** Trader::updateCashPos(**double** amount)
243. {
244. \_cashPosition += amount;
245. }
247. **void** Trader::updateCashPosWOrder(**double** amount)
248. {
249. \_cashPosition += amount;
250. }
252. **void** Trader::addPendingOrder(Order order)
253. {
254. \_pendingOrders.push\_back(order);
255. }
257. **void** Trader::removePendingOrder(Order order)
258. {
259. \_pendingOrders.remove(order);
260. }
262. **void** Trader::completedPendingOrder(Order order)
263. {
264. \_pendingOrders.remove(order);
265. **this**->WriteCompletedOrderToFile(order.toStringCSV());
266. \_completedOrdersV.push\_back(order);
267. }
269. **void** Trader::addStockPosition(Stock\* stock, **int** volume)
270. {
271. \_stockPositions.insert(std::make\_pair(stock->getSymbol(), volume));
272. \_stockPosWOrders.insert(std::make\_pair(stock->getSymbol(), volume));
273. }
275. **void** Trader::removeStockPosition(Stock\* stock)
276. {
277. \_stockPositions.erase(stock->getSymbol());
278. \_stockPosWOrders.erase(stock->getSymbol());
279. }
281. **void** Trader::updateStockPosition(Stock\* stock, **int** amount)
282. {
283. \_stockPositions[stock->getSymbol()] += amount;
284. }
286. **void** Trader::updateStockPosWOrder(Stock\* stock, **int** amount)
287. {
288. \_stockPosWOrders[stock->getSymbol()] += amount;
289. }
291. **bool** Trader::ready()
292. {
293. **return** \_currentT >= \_lastCompleteT + \_processT;
294. }
296. TraderCL Trader::getTraderCL(std::string symbol)
297. {
298. TraderCL tcl(\_cashPosition, \_cashPosWOrders, \_stockPositions[symbol], \_stockPosWOrders[symbol], \_startCash, \_startVolume, \_id, \_type);
299. **return** tcl;
300. }
302. **void** Trader::processTraderCL(TraderCL tcl, OrderBook\* book)
303. {
304. //verify that the correct traderCL has been passed
305. **if** (tcl.id != \_id)
306. **throw** **new** std::exception("Incorrect TraderCL passed to Trader");
308. **if** (tcl.type != \_type)
309. **throw** **new** std::exception("Trader type mismatch");
311. //Only if its time to update
312. **if** (\_currentT >= \_lastCompleteT + \_processT)
313. {
314. //Submit order
315. **if** (tcl.volume < 0)  //Sell Order
316. {
317. book->submitOrder(Order(SELL, -tcl.volume, tcl.price, tcl.id, tcl.isMarket, book->getStock(), \_currentT));
318. }
319. **else** **if** (tcl.volume > 0) //Buy Order
320. {
321. book->submitOrder(Order(BUY, tcl.volume, tcl.price, tcl.id, tcl.isMarket, book->getStock(), \_currentT));
322. }
324. //Update the positions
325. //\_cashPosition = tcl.cashPos;
326. \_cashPosWOrders = tcl.cashPosWO;
327. //\_stockPositions[book->getStock()->getSymbol()] = tcl.volPos;
328. \_stockPosWOrders[book->getStock()->getSymbol()] = tcl.volPosWO;
330. \_lastCompleteT = \_currentT;
332. Logger::GetInstance()->Info(logName, Utils::Merge("Updated Trader:", **this**->toString()));
333. }
334. }
336. **size\_t** Trader::SizeOf()
337. {
338. **size\_t** result = 0;
339. result += **sizeof**(Trader);
340. //result += sizeof(Order)\*(\_pendingOrders.size() + \_completedOrders.size());
341. //result += sizeof(Trade\*)\*\_trades.capacity();
342. result += **sizeof**(std::map<std::string,**int**>)\*(\_stockPositions.size() + \_stockPosWOrders.size());
343. **return** result;
344. }
346. **void** Trader::WriteCompletedOrderToFile(std::string text)
347. {
348. \_completedOrders << text << std::endl;
349. }
351. **void** Trader::WriteTradeToFile(std::string text)
352. {
353. \_trades << text << std::endl;
354. }
356. **double** Trader::GetProfit()
357. {
358. **return** \_cashPosition - \_startCash;
359. }

### IRule

#### IRule.h

1. #pragma once
3. #include "OrderBook.h"
4. #include "Trade.h"
5. #include "Logger.h"
7. **class** OrderBook;
9. **class** IRule
10. {
11. **public**:
12. **virtual** **bool** fitsCriteria(OrderBook\* orderBook, Order\* order) = 0;
13. **virtual** Trade\* processRule(OrderBook\*& orderBook, Order\*& order) = 0;
14. **virtual** std::string ToString() = 0;
15. };

### LimitLimitRule

#### LimitLimitRule.h

1. #pragma once
3. #include "IRule.h"
5. **class** LimitLimitRule : **public** IRule
6. {
7. **public**:
8. LimitLimitRule(**void**);
9. ~LimitLimitRule(**void**);
11. **bool** fitsCriteria(OrderBook\* orderBook, Order\* order);
12. Trade\* processRule(OrderBook\*& orderBook, Order\*& order);
13. std::string ToString()  {**return** "LimitLimit Rule";}
14. };

#### LimitLimitRule.cpp

1. #include "StdAfx.h"
2. #include "LimitLimitRule.h"

5. LimitLimitRule::LimitLimitRule(**void**)
6. {}
8. LimitLimitRule::~LimitLimitRule(**void**)
9. {}
11. **bool** LimitLimitRule::fitsCriteria(OrderBook\* orderBook, Order\* order)
12. {
13. **bool** result = **false**;
15. **if** (order->isLimit())
16. {
17. std::list<Order> orders;
19. **if** (order->isBuy())
20. orders = orderBook->getSellOrders();
21. **else** **if** (order->isSell())
22. orders = orderBook->getBuyOrders();
23. **else**
24. **throw** **new** std::exception("Trade of unknown direction. Unable to check criteria match");
26. std::list<Order>::iterator it;
27. **for** (it = orders.begin(); it != orders.end(); it++)
28. {
29. **if** (it->isLimit() && it->getParticipant() != order->getParticipant())
30. {
31. result = **true**;
32. **break**;
33. }
34. }
35. }
37. **return** result;
38. }
40. Trade\* LimitLimitRule::processRule(OrderBook\*& orderBook, Order\*& order)
41. {
42. Order\* matchedOrder = NULL;
43. Trade\* trade = NULL;
44. **int** size;
45. **double** price;
47. std::list<Order> orders;
49. **if** (order->isBuy())
50. orders = orderBook->getSellOrders();
51. **else**
52. orders = orderBook->getBuyOrders();
54. matchedOrder = &orders.front();
56. auto it = orders.begin();
57. **while** (matchedOrder->getParticipant() == order->getParticipant())
58. {
59. **if** (it->getParticipant() != order->getParticipant() && it->isLimit() && order->isLimit())
60. {
61. matchedOrder = &\*it;
62. **break**;
63. }
65. it++;
66. **if** (it == orders.end())
67. **return** NULL;
68. }
70. **if** ((order->isSell() && (matchedOrder->getPrice() >= order->getPrice()))
71. || (order->isBuy() && (matchedOrder->getPrice() <= order->getPrice())))
72. {
73. size = matchedOrder->getSize();
74. price = matchedOrder->getPrice();
76. **if** (matchedOrder->getSize() > order->getSize())
77. size = order->getSize();
79. trade = **new** Trade(\*matchedOrder, \*order, price, size, orderBook->getTime());
81. orderBook->updateOrderSize(order, (order->getSize() - size));
82. orderBook->updateOrderSize(matchedOrder, (matchedOrder->getSize() - size));
84. orderBook->setLastPrice(price);
85. }
87. matchedOrder = NULL;
88. **delete** matchedOrder;
90. **return** trade;
91. }

### LimitMarketRule

#### LimitMarketRule.h

1. #pragma once
3. #include "IRule.h"
5. **class** LimitMarketRule : **public** IRule
6. {
7. **public**:
8. LimitMarketRule(**void**);
9. ~LimitMarketRule(**void**);
11. **bool** fitsCriteria(OrderBook\* orderBook, Order\* order);
12. Trade\* processRule(OrderBook\*& orderBook, Order\*& order);
13. std::string ToString(){ **return** "LimitMarket Rule";}
14. };

#### LimitMarketRule.cpp

1. #include "StdAfx.h"
2. #include "LimitMarketRule.h"

5. LimitMarketRule::LimitMarketRule(**void**)
6. {}
8. LimitMarketRule::~LimitMarketRule(**void**)
9. {}
11. **bool** LimitMarketRule::fitsCriteria(OrderBook\* orderBook, Order\* order)
12. {
13. **bool** result = **false**;
15. **if** (order->isLimit())
16. {
17. std::list<Order> orders;
19. **if** (order->isBuy())
20. orders = orderBook->getSellOrders();
21. **else** **if** (order->isSell())
22. orders = orderBook->getBuyOrders();
23. **else**
24. **throw** **new** std::exception("Trade of unknown direction. Unable to check critera");
26. std::list<Order>::iterator it;
27. **for** (it = orders.begin(); it != orders.end(); it++)
28. {
29. **if** (it->isMarket() && it->getParticipant() != order->getParticipant())
30. {
31. result = **true**;
32. **break**;
33. }
34. }
35. }
37. **return** result;
38. }
40. Trade\* LimitMarketRule::processRule(OrderBook\*& orderBook, Order\*& order)
41. {
42. Order\* matchedOrder = NULL;
43. Trade\* trade = NULL;
45. std::list<Order> orders;
47. **if** (order->isBuy())
48. orders = orderBook->getSellOrders();
49. **else**
50. orders = orderBook->getBuyOrders();
52. matchedOrder = &orders.front();
54. auto it = orders.begin();
55. **while** (matchedOrder->getParticipant() == order->getParticipant())
56. {
57. **if** (it->getParticipant() != order->getParticipant() && it->isMarket() && order->isLimit())
58. {
59. matchedOrder = &\*it;
60. **break**;
61. }
63. it++;
64. **if** (it == orders.end())
65. **return** NULL;
66. }
68. **double** price = order->getPrice();
69. **int** size = matchedOrder->getSize();
71. **if** (matchedOrder->getSize() > order->getSize())
72. size = order->getSize();
74. trade = **new** Trade(\*order, \*matchedOrder, price, size, orderBook->getTime());
76. orderBook->updateOrderSize(order, (order->getSize() - size));
77. orderBook->updateOrderSize(matchedOrder, (matchedOrder->getSize() - size));
79. orderBook->setLastPrice(price);
81. matchedOrder = NULL;
82. **delete** matchedOrder;
84. **return** trade;
85. }

### MarketLimitRule

#### MarketLimitRule.h

1. #pragma once
3. #include "IRule.h"
5. **class** MarketLimitRule : **public** IRule
6. {
7. **public**:
8. MarketLimitRule(**void**);
9. ~MarketLimitRule(**void**);
11. **bool** fitsCriteria(OrderBook\* orderBook, Order\* order);
12. Trade\* processRule(OrderBook\*& orderBook, Order\*& order);
13. std::string ToString(){ **return** "MarketLimit Rule";}
14. };

#### MarketLimitRule.cpp

1. #include "StdAfx.h"
2. #include "MarketLimitRule.h"

5. MarketLimitRule::MarketLimitRule(**void**)
6. {}
8. MarketLimitRule::~MarketLimitRule(**void**)
9. {}
11. **bool** MarketLimitRule::fitsCriteria(OrderBook\* orderBook, Order\* order)
12. {
13. **bool** result = **false**;
15. **if** (order->isMarket())
16. {
17. std::list<Order> orders;
19. **if** (order->isBuy())
20. orders = orderBook->getSellOrders();
21. **else** **if** (order->isSell())
22. orders = orderBook->getBuyOrders();
23. **else**
24. **throw** **new** std::exception("Trade of unknown direction. Unable to check criteria");
26. std::list<Order>::iterator it;
27. **for** (it = orders.begin(); it != orders.end(); it++)
28. {
29. **if** (it->isLimit() && it->getParticipant() != order->getParticipant())
30. {
31. result = **true**;
32. **break**;
33. }
34. }
35. }
37. **return** result;
38. }
40. Trade\* MarketLimitRule::processRule(OrderBook\*& orderBook, Order\*& order)
41. {
42. Order\* matchedOrder = NULL;
43. Trade\* trade = NULL;
45. std::list<Order> orders;
47. **if** (order->isBuy())
48. orders = orderBook->getSellOrders();
49. **else**
50. orders = orderBook->getBuyOrders();
52. matchedOrder = &orders.front();
54. auto it = orders.begin();
55. **while** (matchedOrder->getParticipant() == order->getParticipant())
56. {
57. **if** (it->getParticipant() != order->getParticipant() && it->isLimit() && order->isMarket())
58. {
59. matchedOrder = &\*it;
60. **break**;
61. }
63. it++;
64. **if** (it == orders.end())
65. **return** NULL;
66. }
68. **int** size = matchedOrder->getSize();
69. **double** price = matchedOrder->getPrice();
71. **if** (matchedOrder->getSize() > order->getSize())
72. size = order->getSize();
74. trade = **new** Trade(\*matchedOrder, \*order, price, size, orderBook->getTime());
76. orderBook->updateOrderSize(order, (order->getSize() - size));
77. orderBook->updateOrderSize(matchedOrder, (matchedOrder->getSize() - size));
79. orderBook->setLastPrice(price);
81. matchedOrder = NULL;
82. **delete** matchedOrder;
84. **return** trade;
85. }

### MarketMarketRule

#### MarketMarketRule.h

1. #pragma once
3. #include "IRule.h"
5. **class** MarketMarketRule : **public** IRule
6. {
7. **public**:
8. MarketMarketRule(**void**);
9. ~MarketMarketRule(**void**);
11. **bool** fitsCriteria(OrderBook\* orderBook, Order\* order);
12. Trade\* processRule(OrderBook\*& orderBook, Order\*& order);
13. std::string ToString(){**return** "MarketMarket Rule";}
14. };

#### MarketMarketRule.cpp

1. #include "StdAfx.h"
2. #include "MarketMarketRule.h"

5. MarketMarketRule::MarketMarketRule()
6. {}
8. MarketMarketRule::~MarketMarketRule()
9. {}
11. **bool** MarketMarketRule::fitsCriteria(OrderBook\* orderBook, Order\* order)
12. {
13. **bool** result = **false**;
15. **if** (order->isMarket())
16. {
17. std::list<Order> orders;
18. **if** (order->isBuy())
19. orders = orderBook->getSellOrders();
20. **else** **if** (order->isSell())
21. orders = orderBook->getBuyOrders();
22. **else**
23. **throw** **new** std::exception("Trade of unknown direction. Unable to check criteria match");
25. std::list<Order>::iterator it;
26. **for** (it = orders.begin(); it != orders.end(); it++)
27. {
28. **if** (it->isMarket() && it->getParticipant() != order->getParticipant())
29. {
30. result = **true**;
31. **break**;
32. }
33. }
34. }
36. **return** result;
37. }
39. Trade\* MarketMarketRule::processRule(OrderBook\*& orderBook, Order\*& order)
40. {
41. Order\* matchedOrder = NULL;
42. Trade\* trade = NULL;
44. std::list<Order> orders;
46. **if** (order->isBuy())
47. orders = orderBook->getSellOrders();
48. **else**
49. orders = orderBook->getBuyOrders();
51. matchedOrder = &orders.front();
53. auto it = orders.begin();
54. **while** (matchedOrder->getParticipant() == order->getParticipant())
55. {
56. **if** (it->getParticipant() != order->getParticipant() && it->isMarket() && order->isMarket())
57. {
58. matchedOrder = &\*it;
59. **break**;
60. }
62. it++;
63. **if** (it == orders.end())
64. **return** NULL;
65. }
67. **int** size = matchedOrder->getSize();
68. **double** price = orderBook->getLastPrice().price;
70. **if** (matchedOrder->getSize() > order->getSize())
71. size = order->getSize();
73. trade = **new** Trade(\*order, \*matchedOrder, price, size, orderBook->getTime());
75. orderBook->updateOrderSize(order, (order->getSize() - size));
76. orderBook->updateOrderSize(matchedOrder, (matchedOrder->getSize() - size));
78. orderBook->setLastPrice(price);
80. matchedOrder = NULL;
81. **delete** matchedOrder;
83. **return** trade;
84. }

### RuleManager

#### RuleManager.h

1. #pragma once

4. #include "IRule.h"
5. #include "Logger.h"
7. **class** OrderBook;
8. **class** IRule;
10. **class** RuleManager
11. {
12. **private**:
13. std::list<IRule\*> \_rules;
15. **static** **const** std::string logName;
17. **public**:
18. RuleManager(**void**);
19. ~RuleManager(**void**);
21. **void** addRule(IRule\* rule);
22. **void** removeRule(IRule\* rule);
23. **void** applyRules(OrderBook\* orderBook, Order\* order);
24. };

#### RuleManager.cpp

1. #include "StdAfx.h"
2. #include "RuleManager.h"
4. **const** std::string RuleManager::logName = "RuleManager";
6. RuleManager::RuleManager(**void**)
7. {}
9. RuleManager::~RuleManager(**void**)
10. {
11. **for** (auto it=\_rules.begin(); it != \_rules.end(); it++)
12. {
13. **delete** \*it;
14. }
15. }
17. **void** RuleManager::addRule(IRule\* rule)
18. {
19. \_rules.push\_back(rule);
20. Logger::GetInstance()->Debug(logName, Utils::Merge("Added rule:", rule->ToString()));
21. }
23. **void** RuleManager::removeRule(IRule\* rule)
24. {
25. \_rules.remove(rule);
26. Logger::GetInstance()->Debug(logName, Utils::Merge("Removed rule:", rule->ToString()));
27. }
29. **void** RuleManager::applyRules(OrderBook\* orderBook, Order\* order)
30. {
31. Logger::GetInstance()->Debug(logName, "Applying Rules");
33. std::list<IRule\*>::iterator& ruleIt = \_rules.begin();
35. **for** (ruleIt = \_rules.begin(); ruleIt != \_rules.end(); ruleIt++)
36. {
37. IRule\* rule = (\*ruleIt);
39. Logger::GetInstance()->Debug(logName, Utils::Merge("Applying Rule:", rule->ToString()));
40. **while** (rule->fitsCriteria(orderBook, order))
41. {
42. Trade\* trade = rule->processRule(orderBook, order);
44. **if** (trade == NULL)
45. **break**;
46. **else**
47. {
48. orderBook->publishTrade((\*trade));
49. **delete** trade;
50. }
52. **if** (order == NULL || order->getSize() == 0)
53. **break**;
54. }
56. **if** (order == NULL || order->getSize() == 0)
57. **break**;
58. }
60. Logger::GetInstance()->Debug(logName, "DONE");
61. }

### Rules

#### Rules.h

1. #pragma once
3. #include "MarketMarketRule.h"
4. #include "LimitLimitRule.h"
5. #include "LimitMarketRule.h"
6. #include "MarketLimitRule.h"

### Logger

#### Logger.h

1. #pragma once
3. #include "WallTimer.h"
4. #include "Utils.h"
5. #include "Trade.h"
6. #include "Seed.h"
8. #define MAXLINECOUNT 50000
10. **class** Logger
11. {
12. **private**:
13. **static** **const** **int** info = 1;
14. **static** **const** **int** debug = 0;
15. **static** **const** **int** warn = 2;
16. **static** **const** **int** error = 3;
18. //Set these values in Logger.cpp
19. **static** **const** std::string infoCol;
20. **static** **const** std::string debugCol;
21. **static** **const** std::string warnCol;
22. **static** **const** std::string errorCol;
24. **int** \_level;
25. **int** \_time;
26. //long \_t;
27. **int** \_lineCount;
28. **int** \_fileCount;
29. **int** \_run;
30. std::string \_name;
32. std::ofstream \_log;
33. std::ofstream \_fullLog;
34. std::ofstream \_data;
35. std::ofstream \_orders;
36. std::ofstream \_trades;
37. std::ofstream \_prices;
38. std::ofstream \_buyOrders;
39. std::ofstream \_sellOrders;
40. std::string \_directory;
41. std::string \_overallDirectory;
42. std::string \_fileName;
44. **static** **bool** \_instanceFlag;
45. **static** Logger\* \_instance;
46. WallTimer \_timer;
47. Logger(**int** level);
49. **public**:
50. **static** Logger\* GetInstance(**int** level=1);
51. **void** SetLevel(**int** level);
52. **int** GetLevel();
54. **void** NextRun();
56. **void** SetTime(**int** time, **bool** refresh=**false**);
57. **int** GetTime();
59. ~Logger();
61. **void** Info(std::string source, std::string text);
62. **void** Debug(std::string source, std::string text);
63. **void** Warn(std::string source, std::string text);
64. **void** Error(std::string source, std::string text);
65. **void** Data(std::string text);
66. **void** Order(std::string text);
67. **void** Trade(std::string text);
69. **void** Prices(**double** text);
70. **void** BuyOrders(**int** text);
71. **void** SellOrders(**int** text);
73. std::string GetDirectory();
74. };

#### Logger.cpp

1. #include "stdafx.h"
2. #include "Logger.h"
4. **bool** Logger::\_instanceFlag = **false**;
6. Logger\* Logger::\_instance = NULL;
8. **const** std::string Logger::infoCol = "\"green\"";
9. **const** std::string Logger::debugCol = "\"white\"";
10. **const** std::string Logger::warnCol = "\"yellow\"";
11. **const** std::string Logger::errorCol = "\"red\"";
13. Logger::Logger(**int** level)
14. {
15. \_lineCount = 0;
16. \_fileCount = 1;
17. \_level = level;
18. \_time = 0;
19. //\_t = time(0);
20. \_run = 1;
22. **char** directory[128];
23. **char** name[128];
24. \_name = Seed::GetInstance()->GetName();
25. sprintf\_s(directory, "C:\\Outputs\\Log-%s\\", \_name.c\_str());
26. \_directory = std::string(directory);
27. std::wstring ws = Utils::s2ws(\_directory);
29. //Check if the directory already exists
30. **DWORD** ftyp = GetFileAttributesA(\_directory.c\_str());
31. **if** (ftyp == INVALID\_FILE\_ATTRIBUTES)
32. {
33. CreateDirectory(ws.c\_str(), NULL);
34. }
35. //This case the directory exists. append a number
36. **else** **if** (ftyp == FILE\_ATTRIBUTE\_DIRECTORY)
37. {
38. **int** i = 0;
39. **while** (ftyp == FILE\_ATTRIBUTE\_DIRECTORY)
40. {
41. i++;
42. **char** temp[128];
43. sprintf\_s(temp, "C:\\Outputs\\Log-%s-%d\\", \_name.c\_str(), i);
44. \_directory = std::string(temp);
45. ws = Utils::s2ws(\_directory);
46. ftyp = GetFileAttributesA(\_directory.c\_str());
47. }
48. CreateDirectory(ws.c\_str(), NULL);
49. }
51. \_overallDirectory = \_directory;
53. \_prices.open(\_directory + "Prices.csv", std::ios\_base::app);
54. \_buyOrders.open(\_directory + "BuyOrders.csv", std::ios\_base::app);
55. \_sellOrders.open(\_directory + "SellOrders.csv", std::ios\_base::app);
57. **for** (**int** i=1; i <= 60000; i++)
58. {
59. \_prices << i << ",";
60. \_sellOrders << i << ",";
61. \_buyOrders << i << ",";
62. }
63. \_prices << std::endl;
64. \_buyOrders << std::endl;
65. \_sellOrders << std::endl;
67. **char** directory2[128];
68. sprintf\_s(directory, "%sLog-%d\\", \_directory.c\_str(), \_run);
69. \_directory = std::string(directory);
70. std::wstring ws1 = Utils::s2ws(\_directory);
71. CreateDirectory(ws1.c\_str(), NULL);
73. sprintf\_s(name, "Log-%d.html", \_fileCount);
74. \_fileName = std::string(name);
75. \_log.open(\_directory + \_fileName, std::ios\_base::app);
76. \_fullLog.open(\_directory + "Log.html", std::ios\_base::app);
77. \_log << "<html><body bgcolor=\"#000000\">\n";
78. //\_fullLog << "<html><body bgcolor=\"#000000\">\n";
79. \_lineCount++;
80. \_timer.Start();
82. \_data.open(\_directory + "Data.csv", std::ios\_base::app);
84. \_orders.open(\_directory + "Orders.csv", std::ios\_base::app);
85. \_orders << "Price,Volume,Type,Market(1)/Limit(0),ParticipantId,Stock,Time,OrderId" << std::endl;
87. \_trades.open(\_directory + "Trades.csv", std::ios\_base::app);
88. \_trades << Trade::toStringHeaderCSV() << std::endl;
89. }
91. Logger::~Logger()
92. {
93. **if** (\_instanceFlag)
94. {
95. \_log << "<\\body><\\html>";
96. \_fullLog << "<\\body><\\html>";
97. \_log.close();
98. \_fullLog.close();
99. \_data.close();
100. \_orders.close();
101. \_trades.close();
102. \_prices.close();
103. \_buyOrders.close();
104. \_sellOrders.close();
105. \_instanceFlag = **false**;
106. **delete** \_instance;
107. \_instance = NULL;
108. }
109. }
111. Logger\* Logger::GetInstance(**int** level)
112. {
113. **if** (!\_instanceFlag)
114. {
115. \_instance = **new** Logger(level);
116. \_instanceFlag = **true**;
117. **return** \_instance;
118. }
119. **else**
120. {
121. **return** \_instance;
122. }
123. }
125. **void** Logger::SetLevel(**int** level)
126. {
127. \_level = level;
128. }
130. **int** Logger::GetLevel()
131. {
132. **return** \_level;
133. }
135. **void** Logger::NextRun()
136. {
137. \_run++;
138. \_log << "<\\body><\\html>";
139. \_fullLog << "<\\body><\\html>";
140. \_log.close();
141. \_fullLog.close();
142. \_data.close();
143. \_orders.close();
144. \_trades.close();
145. \_prices << std::endl;
146. \_buyOrders << std::endl;
147. \_sellOrders << std::endl;
149. **char** directory[128];
150. **char** name[128];
151. sprintf\_s(directory, "%sLog-%d\\", \_overallDirectory.c\_str(), \_run);
152. \_directory = std::string(directory);
153. std::wstring ws = Utils::s2ws(\_directory);
154. CreateDirectory(ws.c\_str(), NULL);
156. sprintf\_s(name, "Log%d.html", \_fileCount);
157. \_fileName = std::string(name);
158. \_log.open(\_directory + \_fileName, std::ios\_base::app);
159. \_fullLog.open(\_directory + "Log.html", std::ios\_base::app);
160. \_log << "<html><body bgcolor=\"#000000\">\n";
161. //\_fullLog << "<html><body bgcolor=\"#000000\">\n";
162. \_lineCount++;
163. \_timer.Start();
165. \_data.open(\_directory + "Data.csv", std::ios\_base::app);
167. \_orders.open(\_directory + "Orders.csv", std::ios\_base::app);
168. \_orders << "Price,Volume,Type,Market(1)/Limit(0),ParticipantId,Stock,Time,OrderId" << std::endl;
170. \_trades.open(\_directory + "Trades.csv", std::ios\_base::app);
171. \_trades << Trade::toStringHeaderCSV() << std::endl;
172. }
174. **void** Logger::SetTime(**int** time, **bool** refresh)
175. {
176. **if** (\_lineCount > MAXLINECOUNT)
177. {
178. \_log << "<\\body><\\html>";
179. \_log.close();
180. \_lineCount = 0;
181. \_fileCount++;
183. **char** name[128];
184. sprintf\_s(name, "Log-%d.html", \_fileCount);
185. \_fileName = std::string(name);
186. \_log.open(\_directory + \_fileName, std::ios\_base::app);
187. \_log << "<html><body bgcolor=\"#000000\">\n";
188. \_lineCount++;
189. }
191. **if** (refresh)
192. {
193. \_timer.Start();
194. Debug("Logger", Utils::DtoS((**double**)0));
195. }
196. \_time = time;
197. }
199. **int** Logger::GetTime()
200. {
201. **return** \_time;
202. }
204. **void** Logger::Info(std::string source, std::string text)
205. {
206. **if** (\_level <= info)
207. {
208. std::stringstream buffer;
209. buffer << "<font color=" << infoCol << "><i>" << \_time << "</i> : " << \_timer.GetCounter() << "ms - <b><i>INFO</i></b> - <b>" << source << "</b> : "
210. << text << "</font><br>\n";
211. \_log << buffer.str();
212. //\_fullLog << buffer.str();
213. \_lineCount++;
214. }
215. }
217. **void** Logger::Debug(std::string source, std::string text)
218. {
219. **if** (\_level <= debug)
220. {
221. std::stringstream buffer;
222. buffer << "<font color=" << debugCol << "><i>" << \_time << "</i> : " << \_timer.GetCounter() << "ms - <b><i>DEBUG</i></b> - <b>" << source << "</b> : "
223. << text << "</font><br>\n";
224. \_log << buffer.str();
225. //\_fullLog << buffer.str();
226. \_lineCount++;
227. }
228. }
230. **void** Logger::Warn(std::string source, std::string text)
231. {
232. **if** (\_level <= warn)
233. {
234. std::stringstream buffer;
235. buffer << "<font color=" << warnCol << "><i>" << \_time << "</i> : " << \_timer.GetCounter() << "ms - <b><i>WARN</i></b> - <b>" << source << "</b> : "
236. << text << "</font><br>\n";
237. \_log << buffer.str();
238. //\_fullLog << buffer.str();
239. \_lineCount++;
240. }
241. }
243. **void** Logger::Error(std::string source, std::string text)
244. {
245. **if** (\_level <= error)
246. {
247. std::stringstream buffer;
248. buffer << "<font color=" << errorCol << "><i>" << \_time << "</i> : " << \_timer.GetCounter() << "ms - <b><i>ERROR</i></b> - <b>" << source << "</b> : "
249. << text << "</font><br>\n";
250. \_log << buffer.str();
251. //\_fullLog << buffer.str();
252. \_lineCount++;
253. }
254. }
256. **void** Logger::Data(std::string text)
257. {
258. \_data << text << std::endl;
259. }
261. **void** Logger::Order(std::string text)
262. {
263. \_orders << text << std::endl;
264. }
266. **void** Logger::Trade(std::string text)
267. {
268. \_trades << text << std::endl;
269. }
271. std::string Logger::GetDirectory()
272. {
273. **return** \_directory;
274. }
276. **void** Logger::Prices(**double** text)
277. {
278. \_prices << text << ",";
279. }
281. **void** Logger::BuyOrders(**int** text)
282. {
283. \_buyOrders << text << ",";
284. }
286. **void** Logger::SellOrders(**int** text)
287. {
288. \_sellOrders << text << ",";
289. }

### OpenClController

#### OpenClController.h

1. #pragma once
3. #include "OpenClDevice.h"
4. #include "OpenClStructs.h"
6. #define \_\_NO\_STD\_VECTOR
8. **class** OpenClController
9. {
10. **private**:
11. cl::vector<cl::Platform> \_platformList;
12. cl\_context\_properties \_contextProperties[3];
13. cl::Context \_context;
14. cl::vector<cl::Device> \_devices;
15. cl::CommandQueue \_queue;
16. OpenClDevice\* \_device;
17. TraderCLArray \_tradersBuffer;
18. MarketDataCL \_data;
19. **static** **const** std::string logName;
20. std::string \_kernelText;
22. **bool** \_profiling;
24. **static** **bool** \_instanceFlag;
25. **static** OpenClController\* \_instance;
26. OpenClController();
27. ~OpenClController();
28. **public**:
29. **static** OpenClController\* GetInstance();

32. **void** RefreshBuffers(TraderCLArray tb, MarketDataCL data);
33. **void** SetText(std::string kernelText);
34. **void** SetupFirstTime(TraderCLArray tradersBuffer, std::string kernelName, **int** rtCount, **int** lrtCount, **int** ptCount, **int** mtCount, **bool** profiling = **false**);
35. **void** UpdateBuffersAndArgs();
36. **double** Run(cl::NDRange globalRange, cl::NDRange localRange);
38. **void** DisplayPlatformInfo(cl::vector<cl::Platform> platformList, **int** deviceType);
39. };

#### OpenClController.cpp

1. #include "StdAfx.h"
2. #include "OpenClController.h"
4. **const** std::string OpenClController::logName = "OpenClController";
6. **bool** OpenClController::\_instanceFlag = **false**;
8. OpenClController\* OpenClController::\_instance = NULL;
10. OpenClController::OpenClController()
11. {
12. cl::Platform::get(&\_platformList);
14. //DisplayPlatformInfo(\_platformList, CL\_DEVICE\_TYPE\_GPU);
16. \_contextProperties[0] = CL\_CONTEXT\_PLATFORM;
17. \_contextProperties[1] = (cl\_context\_properties)(\_platformList[0])();
18. \_contextProperties[2] = 0;
20. \_context = cl::Context(CL\_DEVICE\_TYPE\_GPU, \_contextProperties);
22. \_devices = \_context.getInfo<CL\_CONTEXT\_DEVICES>();
24. //\_logger = Logger::GetInstance(LOGLEVEL);
25. /\*std::ifstream file("Traders.cl");
26. if (file.is\_open())
27. {
28. std::string prog(std::istreambuf\_iterator<char>(file), (std::istreambuf\_iterator<char>()));
29. \_kernelText = prog;
30. }
31. file.close();\*/
32. }

35. OpenClController::~OpenClController()
36. {
37. **if** (\_instanceFlag)
38. {
39. \_instanceFlag = **false**;
40. **delete** \_device;
41. **delete** \_instance;
42. \_instance = NULL;
43. }
44. }
46. OpenClController\* OpenClController::GetInstance()
47. {
48. **if** (!\_instanceFlag)
49. {
50. \_instanceFlag = **true**;
51. \_instance = **new** OpenClController();
52. **return** \_instance;
53. }
54. **else**
55. {
56. **return** \_instance;
57. }
58. }
60. **void** OpenClController::RefreshBuffers(TraderCLArray tb, MarketDataCL data)
61. {
62. Logger::GetInstance()->Debug(logName, "Refreshing Traders and Market Data Buffers");
63. \_tradersBuffer = tb;
64. \_data = data;
65. Logger::GetInstance()->Debug(logName, "DONE");
66. }
68. **void** OpenClController::SetText(std::string kernelText)
69. {
70. \_kernelText = kernelText;
71. }
73. **void** OpenClController::SetupFirstTime(TraderCLArray tradersBuffer, std::string kernelName, **int** rtCount, **int** lrtCount, **int** ptCount, **int** mtCount, **bool** profiling)
74. {
75. \_profiling = profiling;
76. \_tradersBuffer = tradersBuffer;
78. **if** (\_profiling)
79. \_queue = cl::CommandQueue(\_context, \_devices[0], CL\_QUEUE\_PROFILING\_ENABLE);
80. **else**
81. \_queue = cl::CommandQueue(\_context, \_devices[0], 0);
83. \_device = **new** OpenClDevice(\_context, \_devices, "Traders.cl", \_profiling);
85. Logger::GetInstance()->Debug(logName, "Setting up Build Options");
86. \_device->SetupBuildOptions(rtCount, lrtCount, ptCount, mtCount);
87. Logger::GetInstance()->Debug(logName, "DONE");
89. Logger::GetInstance()->Debug(logName, "Building Kernel");
90. **try**
91. {
92. \_device->BuildKernel(kernelName, \_kernelText);
93. }
94. **catch** (...)
95. {
96. std::stringstream temp1; temp1 << "Failed in SetupFirstTime:BuildKernel - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
97. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
98. Logger::GetInstance()->Error(logName, temp);
99. **throw** **new** std::exception(temp.c\_str());
100. }
101. Logger::GetInstance()->Debug(logName, "DONE");
102. }
104. **void** OpenClController::UpdateBuffersAndArgs()
105. {
106. Logger::GetInstance()->Debug(logName, "Setting up Buffers");
107. **try**
108. {
109. \_device->SetupBuffers(\_tradersBuffer, \_data);
110. }
111. **catch** (...)
112. {
113. std::stringstream temp1; temp1 << "Failed in UpdateBuffersAndArgs:SetupBuffer - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
114. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
115. Logger::GetInstance()->Error(logName, temp);
116. **throw** **new** std::exception(temp.c\_str());
117. }
118. Logger::GetInstance()->Debug(logName, "DONE");
120. Logger::GetInstance()->Debug(logName, "Setting up Args");
121. **try**
122. {
123. \_device->SetupKernelArgs();
124. }
125. **catch** (...)
126. {
127. std::stringstream temp1; temp1 << "Failed in UpdateBuffersAndArgs:SetupKernelArgs - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
128. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
129. Logger::GetInstance()->Error(logName, temp);
130. **throw** **new** std::exception(temp.c\_str());
131. }
132. Logger::GetInstance()->Debug(logName, "DONE");
133. }
135. **double** OpenClController::Run(cl::NDRange globalRange, cl::NDRange localRange)
136. {
137. cl::Event finishEvent;
138. Logger::GetInstance()->Debug(logName, "Enqueueing Buffers");
139. **try**
140. {
141. \_device->EnqueueWriteBuffers(\_queue);
142. }
143. **catch** (...)
144. {
145. std::stringstream temp1; temp1 << "Failed in Run:EnqueueWriteBuffers - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
146. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
147. Logger::GetInstance()->Error(logName, temp);
148. **throw** **new** std::exception(temp.c\_str());
149. }
150. Logger::GetInstance()->Debug(logName, "DONE");
152. Logger::GetInstance()->Debug(logName, "Enqueueing Kernel");
153. **try**
154. {
155. \_queue.enqueueNDRangeKernel(\_device->GetKernel(), cl::NullRange, globalRange, localRange, NULL, &finishEvent);
156. }
157. **catch** (...)
158. {
159. std::stringstream temp1; temp1 << "Failed in Run:EnqueueNDRangeKernel - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
160. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
161. Logger::GetInstance()->Error(logName, temp);
162. **throw** **new** std::exception(temp.c\_str());
163. }
164. Logger::GetInstance()->Debug(logName, "DONE");
166. Logger::GetInstance()->Debug(logName, "Enqueueing Read of Buffers");
167. **double** time;
168. **try**
169. {
170. time = \_device->EnqueueRead(\_queue, finishEvent);
171. }
172. **catch** (...)
173. {
174. std::stringstream temp1; temp1 << "Failed in Run:EnqueueRead - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
175. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
176. Logger::GetInstance()->Error(logName, temp);
177. **throw** **new** std::exception(temp.c\_str());
178. }
179. Logger::GetInstance()->Debug(logName, "DONE");
181. **try**
182. {
183. \_queue.flush();
184. \_queue.finish();
185. }
186. **catch** (...)
187. {
188. std::stringstream temp1; temp1 << "Failed in Run:Flush/Finish - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
189. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
190. Logger::GetInstance()->Error(logName, temp);
191. **throw** **new** std::exception(temp.c\_str());
192. }
194. std::stringstream temp;
195. temp << "Kernel Executed in time: " << time << "ms";
196. Logger::GetInstance()->Info(logName, temp.str());
197. **return** time;
198. }
200. #pragma region Helper Methods
202. **void** OpenClController::DisplayPlatformInfo(cl::vector<cl::Platform> platformList, **int** deviceType)
203. {
204. std::string platformVendor;
205. platformList[0].getInfo((cl\_platform\_info)CL\_PLATFORM\_VENDOR,
206. &platformVendor);
208. std::cout << "device Type "
209. << ((deviceType==CL\_DEVICE\_TYPE\_GPU)?"GPU":"CPU") << std::endl;
210. std::cout << "Platform is by: " << platformVendor << "\n";
211. }
213. #pragma endregion

### OpenClDevice

#### OpenClDevice.h

1. #pragma once
3. #define \_\_CL\_ENABLE\_EXCEPTIONS
4. #define \_\_NO\_STD\_VECTOR
5. #include <CL\cl.hpp>
7. #include "OpenClStructs.h"
8. #include "Logger.h"
9. #include "Utils.h"
10. #include "Seed.h"

13. #define \_\_NO\_STD\_VECTOR
15. **class** OpenClDevice
16. {
17. **private**:
18. //OCL variables
19. cl::Context \_context;
20. cl::vector<cl::Device> \_devices;
21. cl::Kernel \_kernel;
22. **const** **char**\* \_kernelFile;
23. std::string \_buildOptions;
24. std::string \_kernelName;
25. //TODO buffers
26. cl::Buffer \_tradersBuffer;
27. cl::Buffer \_marketDataBuffer;
29. //Traders
30. TraderCLArray \_traders;
31. MarketDataCL \_data;
33. //Other
34. **bool** \_profiling;
35. **static** **const** std::string logName;
37. **protected**:
38. **void** Try(cl\_int err);
40. **public**:
41. OpenClDevice(cl::Context& context, cl::vector<cl::Device>& devices, **const** **char**\* kernelFile, **bool** profiling);
42. ~OpenClDevice(**void**);
43. **void** SetupBuildOptions(**int** rtCount, **int** lrtCount, **int** ptCount, **int** mtCount);
44. **void** BuildKernel(std::string kernelName, std::string kernelText="");
45. **void** SetupBuffers(TraderCLArray tradersBuffer, MarketDataCL data);
46. **void** SetupKernelArgs();
47. **void** EnqueueWriteBuffers(cl::CommandQueue& queue, cl::Event\* writeEvent = (cl::Event\*)0);
48. **double** EnqueueRead(cl::CommandQueue& queue, cl::Event& finishEvent);
50. **static** **const** **char**\* clErr(cl\_int err);
52. cl::Kernel GetKernel();
54. **void** EnableProfiling();
55. **void** DisableProfiling();
56. };

#### OpenClDevice.cpp

1. #include "StdAfx.h"
2. #include "OpenClDevice.h"
4. **const** std::string OpenClDevice::logName = "OpenClDevice";
6. OpenClDevice::OpenClDevice(cl::Context& context, cl::vector<cl::Device>& devices, **const** **char**\* kernelFile, **bool** profiling)
7. {
8. \_context = context;
9. \_devices = devices;
10. \_kernelFile = kernelFile;
11. \_profiling = profiling;
12. srand(Seed::GetInstance()->GetSeed());
13. }

16. OpenClDevice::~OpenClDevice(**void**)
17. {}
19. **void** OpenClDevice::SetupBuildOptions(**int** rtCount, **int** lrtCount, **int** ptCount, **int** mtCount)
20. {
21. **if** (rtCount > -1)
22. {
23. **char** buf[1024];
24. //TODO fix
25. /\*sprintf\_s(buf, "-D %s=%d -D %s=%d -D %s=%d -D %s=%.2f -D %s=%d -D %s=%d", \_list.RT\_BUYSELL\_THRESH.Name(), \_list.RT\_BUYSELL\_THRESH.Value(),
26. \_list.RT\_COUNT.Name(), \_list.RT\_COUNT.Value(),
27. \_list.RT\_MARKET\_THRESH.Name(), \_list.RT\_MARKET\_THRESH.Value(),
28. \_list.RT\_PRICE\_CHANGE.Name(), \_list.RT\_PRICE\_CHANGE.Value(),
29. \_list.RT\_PRICE\_SIZE.Name(), \_list.RT\_PRICE\_SIZE.Value(),
30. \_list.RT\_SIZE.Name(), \_list.RT\_SIZE.Value());\*/
31. sprintf\_s(buf, "-D RT\_BUYSELL\_THRESH=2 -D RT\_MARKET\_THRESH=2 -D RT\_SIZE=1000 -D RT\_PRICE\_CHANGE=0.01 -D RT\_PRICE\_SIZE=10 -D RT\_COUNT=%d ", rtCount);
32. \_buildOptions += std::string(buf);
33. std::stringstream temp;
34. temp << "Added " << std::string(buf) << " to build options";
35. Logger::GetInstance()->Debug(logName, temp.str());
36. }
37. **if** (lrtCount > -1)
38. {
39. **char** buf[64];
40. sprintf\_s(buf, "-D LRT\_COUNT=%d ", lrtCount);
41. \_buildOptions += std::string(buf);
42. std::stringstream temp;
43. temp << "Added " << std::string(buf) << " to build options";
44. Logger::GetInstance()->Debug(logName, temp.str());
45. }
46. **if** (ptCount > -1)
47. {
48. **char** buf[128];
49. sprintf\_s(buf, "-D PT\_SELL\_THRESH=100 -D PT\_BUY\_THRESH=100 -D PT\_BOUNDS=10 -D PT\_COUNT=%d ", ptCount);
50. \_buildOptions += std::string(buf);
51. std::stringstream temp;
52. temp << "Added " << std::string(buf) << " to build options";
53. Logger::GetInstance()->Debug(logName, temp.str());
54. }
55. **if** (mtCount > -1)
56. {
57. **char** buf[128];
58. sprintf\_s(buf, "-D MT\_SIZE\_THRESH=100 -D MT\_SHORT\_RANGE=10 -D MT\_COUNT=%d ", mtCount);
59. \_buildOptions += std::string(buf);
60. std::stringstream temp;
61. temp << "Added " << std::string(buf) << " to build options";
62. Logger::GetInstance()->Debug(logName, temp.str());
63. }
65. std::stringstream temp;
66. temp << "Build Options are: " << \_buildOptions;
67. Logger::GetInstance()->Info(logName, temp.str());
68. }
70. **void** OpenClDevice::BuildKernel(std::string kernelName, std::string kernelText)
71. {
72. \_kernelName = kernelName;
74. //Setup the program
75. std::stringstream tempSS;
76. tempSS << "Building file: " << \_kernelFile;
77. Logger::GetInstance()->Debug(logName, tempSS.str());
78. tempSS.clear();
80. cl::Program program;
81. **try**
82. {
83. assert(kernelText != "");
84. cl::Program::Sources source(1, std::make\_pair(kernelText.c\_str(), kernelText.length()+1));
85. program = cl::Program(\_context, source);
86. }
87. **catch** (...)
88. {
89. std::stringstream temp1; temp1 << "Failed in BuildKernel:assert(kernelText) - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
90. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
91. Logger::GetInstance()->Error(logName, temp);
92. **throw** **new** std::exception(temp.c\_str());
93. }
95. Logger::GetInstance()->Debug(logName, "DONE");

98. //build the program
99. **try**
100. {
101. Logger::GetInstance()->Debug(logName, "Compiling...");
102. program.build(\_devices, \_buildOptions.c\_str());
103. Logger::GetInstance()->Debug(logName, "DONE");
104. }
105. **catch** (...)
106. {
107. std::stringstream temp1; temp1 << "Failed in BuildKernel:program.build - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
108. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
109. Logger::GetInstance()->Error(logName, temp);
110. std::stringstream temp2; temp2 << program.getBuildInfo<CL\_PROGRAM\_BUILD\_LOG>(\_devices[0]) << std::endl;
111. Logger::GetInstance()->Error(logName, temp2.str());
112. std::string temp3 = Utils::Merge(temp, temp2.str());
113. **throw** **new** std::exception(temp3.c\_str());
114. }
116. **try**
117. {
118. \_kernel = cl::Kernel(program, \_kernelName.c\_str());
119. }
120. **catch** (...)
121. {
122. std::stringstream temp1; temp1 << "Failed in BuildKernel:cl::Kernel(program,\_kernelName.c\_str()) - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
123. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
124. Logger::GetInstance()->Error(logName, temp);
125. **throw** **new** std::exception(temp.c\_str());
126. }
127. }
129. **void** OpenClDevice::SetupBuffers(TraderCLArray tradersBuffer, MarketDataCL data)
130. {
131. \_traders = tradersBuffer;
132. \_data = data;
134. cl\_int err;
135. \_tradersBuffer = cl::Buffer(\_context, CL\_MEM\_READ\_WRITE | CL\_MEM\_USE\_HOST\_PTR, \_traders.number\***sizeof**(TraderCL), \_traders.traders, &err);
136. **try**
137. {
138. Try(err);
139. }
140. **catch** (...)
141. {
142. std::stringstream temp1; temp1 << "Failed in SetupBuffers:\_tradersBuffer - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
143. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
144. Logger::GetInstance()->Error(logName, temp);
145. **throw** **new** std::exception(temp.c\_str());
146. }
148. \_marketDataBuffer = cl::Buffer(\_context, CL\_MEM\_READ\_WRITE | CL\_MEM\_USE\_HOST\_PTR, \_data.numPastPrices\***sizeof**(PastPrice), \_data.prices, &err);
149. **try**
150. {
151. Try(err);
152. }
153. **catch** (...)
154. {
155. std::stringstream temp1; temp1 << "Failed in SetupBuffers:\_marketDataBuffer - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
156. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
157. Logger::GetInstance()->Error(logName, temp);
158. **throw** **new** std::exception(temp.c\_str());
159. }
160. }
162. **void** OpenClDevice::SetupKernelArgs()
163. {
164. **try**
165. {
166. Try(\_kernel.setArg(0, (cl\_ulong)Seed::GetInstance()->GetSeed()));
167. }
168. **catch** (...)
169. {
170. std::stringstream temp1; temp1 << "Failed in SetupKernelArgs:Arg(0) - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
171. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
172. Logger::GetInstance()->Error(logName, temp);
173. **throw** **new** std::exception(temp.c\_str());
174. }
176. **try**
177. {
178. Try(\_kernel.setArg(1, \_tradersBuffer));
179. }
180. **catch** (...)
181. {
182. std::stringstream temp1; temp1 << "Failed in SetupKernelArgs:Arg(1) - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
183. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
184. Logger::GetInstance()->Error(logName, temp);
185. **throw** **new** std::exception(temp.c\_str());
186. }
188. **try**
189. {
190. Try(\_kernel.setArg(2, \_marketDataBuffer));
191. }
192. **catch** (...)
193. {
194. std::stringstream temp1; temp1 << "Failed in SetupKernelArgs:Arg(2) - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
195. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
196. Logger::GetInstance()->Error(logName, temp);
197. **throw** **new** std::exception(temp.c\_str());
198. }
200. **try**
201. {
202. Try(\_kernel.setArg(3, MarketDataSmallCL(\_data.buyVolume, \_data.sellVolume, \_data.getLatestPrice().price, \_data.numPastPrices)));
203. }
204. **catch** (...)
205. {
206. std::stringstream temp1; temp1 << "Failed in SetupKernelArgs:Arg(3) - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
207. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
208. Logger::GetInstance()->Error(logName, temp);
209. **throw** **new** std::exception(temp.c\_str());
210. }
211. }
213. **void** OpenClDevice::EnqueueWriteBuffers(cl::CommandQueue& queue, cl::Event\* writeEvent)
214. {
215. **try**
216. {
217. Try(queue.enqueueWriteBuffer(\_tradersBuffer, CL\_TRUE, 0, \_traders.number\***sizeof**(TraderCL), \_traders.traders, NULL, writeEvent));
218. }
219. **catch** (...)
220. {
221. std::stringstream temp1; temp1 << "Failed in EnqueueWriteBuffers:queue.enqueueWriteBuffer - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
222. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
223. Logger::GetInstance()->Error(logName, temp);
224. **throw** **new** std::exception(temp.c\_str());
225. }
226. }
228. **double** OpenClDevice::EnqueueRead(cl::CommandQueue& queue, cl::Event& finishEvent)
229. {
230. finishEvent.wait();
232. **double** time;
234. **if** (\_profiling)
235. {
236. cl\_ulong start = finishEvent.getProfilingInfo<CL\_PROFILING\_COMMAND\_START>();
237. cl\_ulong end = finishEvent.getProfilingInfo<CL\_PROFILING\_COMMAND\_END>();
238. time = 1.e-6 \* (end-start); //ms 1.e-9 for seconds
239. }
241. **try**
242. {
243. Try(queue.enqueueReadBuffer(\_tradersBuffer, CL\_TRUE, NULL, \_traders.number\***sizeof**(TraderCL), \_traders.traders));
244. }
245. **catch** (...)
246. {
247. std::stringstream temp1; temp1 << "Failed in EnqueueRead:queue.enqueueReadBuffer - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
248. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
249. Logger::GetInstance()->Error(logName, temp);
250. **throw** **new** std::exception(temp.c\_str());
251. }
252. **return** time;
253. }
255. #pragma region Helper Methods
257. cl::Kernel OpenClDevice::GetKernel()
258. {
259. **return** \_kernel;
260. }
262. **void** OpenClDevice::EnableProfiling()
263. {
264. \_profiling = **true**;
265. }
267. **void** OpenClDevice::DisableProfiling()
268. {
269. \_profiling = **false**;
270. }
272. **void** OpenClDevice::Try(cl\_int err)
273. {
274. **if** (err != CL\_SUCCESS)
275. {
276. **throw** **new** std::exception(clErr(err));
277. }
278. }
280. **const** **char**\* OpenClDevice::clErr(cl\_int err)
281. {
282. **switch** (err) {
283. **case** CL\_SUCCESS:                            **return** "Success!";
284. **case** CL\_DEVICE\_NOT\_FOUND:                   **return** "Device not found.";
285. **case** CL\_DEVICE\_NOT\_AVAILABLE:               **return** "Device not available";
286. **case** CL\_COMPILER\_NOT\_AVAILABLE:             **return** "Compiler not available";
287. **case** CL\_MEM\_OBJECT\_ALLOCATION\_FAILURE:      **return** "Memory object allocation failure";
288. **case** CL\_OUT\_OF\_RESOURCES:                   **return** "Out of resources";
289. **case** CL\_OUT\_OF\_HOST\_MEMORY:                 **return** "Out of host memory";
290. **case** CL\_PROFILING\_INFO\_NOT\_AVAILABLE:       **return** "Profiling information not available";
291. **case** CL\_MEM\_COPY\_OVERLAP:                   **return** "Memory copy overlap";
292. **case** CL\_IMAGE\_FORMAT\_MISMATCH:              **return** "Image format mismatch";
293. **case** CL\_IMAGE\_FORMAT\_NOT\_SUPPORTED:         **return** "Image format not supported";
294. **case** CL\_BUILD\_PROGRAM\_FAILURE:              **return** "Program build failure";
295. **case** CL\_MAP\_FAILURE:                        **return** "Map failure";
296. **case** CL\_INVALID\_VALUE:                      **return** "Invalid value";
297. **case** CL\_INVALID\_DEVICE\_TYPE:                **return** "Invalid device type";
298. **case** CL\_INVALID\_PLATFORM:                   **return** "Invalid platform";
299. **case** CL\_INVALID\_DEVICE:                     **return** "Invalid device";
300. **case** CL\_INVALID\_CONTEXT:                    **return** "Invalid context";
301. **case** CL\_INVALID\_QUEUE\_PROPERTIES:           **return** "Invalid queue properties";
302. **case** CL\_INVALID\_COMMAND\_QUEUE:              **return** "Invalid command queue";
303. **case** CL\_INVALID\_HOST\_PTR:                   **return** "Invalid host pointer";
304. **case** CL\_INVALID\_MEM\_OBJECT:                 **return** "Invalid memory object";
305. **case** CL\_INVALID\_IMAGE\_FORMAT\_DESCRIPTOR:    **return** "Invalid image format descriptor";
306. **case** CL\_INVALID\_IMAGE\_SIZE:                 **return** "Invalid image size";
307. **case** CL\_INVALID\_SAMPLER:                    **return** "Invalid sampler";
308. **case** CL\_INVALID\_BINARY:                     **return** "Invalid binary";
309. **case** CL\_INVALID\_BUILD\_OPTIONS:              **return** "Invalid build options";
310. **case** CL\_INVALID\_PROGRAM:                    **return** "Invalid program";
311. **case** CL\_INVALID\_PROGRAM\_EXECUTABLE:         **return** "Invalid program executable";
312. **case** CL\_INVALID\_KERNEL\_NAME:                **return** "Invalid kernel name";
313. **case** CL\_INVALID\_KERNEL\_DEFINITION:          **return** "Invalid kernel definition";
314. **case** CL\_INVALID\_KERNEL:                     **return** "Invalid kernel";
315. **case** CL\_INVALID\_ARG\_INDEX:                  **return** "Invalid argument index";
316. **case** CL\_INVALID\_ARG\_VALUE:                  **return** "Invalid argument value";
317. **case** CL\_INVALID\_ARG\_SIZE:                   **return** "Invalid argument size";
318. **case** CL\_INVALID\_KERNEL\_ARGS:                **return** "Invalid kernel arguments";
319. **case** CL\_INVALID\_WORK\_DIMENSION:             **return** "Invalid work dimension";
320. **case** CL\_INVALID\_WORK\_GROUP\_SIZE:            **return** "Invalid work group size";
321. **case** CL\_INVALID\_WORK\_ITEM\_SIZE:             **return** "Invalid work item size";
322. **case** CL\_INVALID\_GLOBAL\_OFFSET:              **return** "Invalid global offset";
323. **case** CL\_INVALID\_EVENT\_WAIT\_LIST:            **return** "Invalid event wait list";
324. **case** CL\_INVALID\_EVENT:                      **return** "Invalid event";
325. **case** CL\_INVALID\_OPERATION:                  **return** "Invalid operation";
326. **case** CL\_INVALID\_GL\_OBJECT:                  **return** "Invalid OpenGL object";
327. **case** CL\_INVALID\_BUFFER\_SIZE:                **return** "Invalid buffer size";
328. **case** CL\_INVALID\_MIP\_LEVEL:                  **return** "Invalid mip-map level";
329. **default**: **return** "<Unknown error code>";
330. }
331. }
333. #pragma endregion

### OpenClStructs

#### OpenClStructs.h

1. #pragma once
3. **struct** TraderCL
4. {
5. **double** cashPos, cashPosWO;
6. **int** volPos, volPosWO;
7. **double** startCash;
8. **int** startVol;
9. **int** id, type;
10. **bool** isMarket;
11. **double** price;
12. **int** volume;
14. TraderCL(**double** cp=0, **double** cpwo=0, **int** vp=0, **int** vpwo=0, **double** sc=0, **int** sv=0, **int** ID=-1, **int** t=-1, **bool** im=**true**, **double** p=0, **int** v=0) :
15. cashPos(cp), cashPosWO(cpwo), volPos(vp), volPosWO(vpwo), startCash(sc), startVol(sv), id(ID), type(t), isMarket(im), price(p), volume(v){};
16. };
18. **struct** TraderCLArray
19. {
20. TraderCL\* traders;
21. **int** number, randomNumber, largeRandomNumber, positionNumber, momentumNumber;
23. TraderCLArray(TraderCL\* t=NULL, **int** n=0, **int** rn=0, **int** lrn=0, **int** pn=0, **int** mn=0) :
24. traders(t), number(n), randomNumber(rn), largeRandomNumber(lrn), positionNumber(pn), momentumNumber(mn){};
25. };
27. **struct** PastPrice
28. {
29. **double** price;
30. **int** time;
32. PastPrice(**double** p=0, **int** t=-1) : price(p), time(t) {};
33. };
35. **struct** MarketDataSmallCL
36. {
37. **int** buyVolume;
38. **int** sellVolume;
39. **double** lastPrice;
40. **int** numPastPrices;
42. MarketDataSmallCL(**int** bv=0, **int** sv=0, **double** lp=0, **int** npm=0) : buyVolume(bv), sellVolume(sv), lastPrice(lp), numPastPrices(npm){};
43. };
45. **struct** MarketDataCL
46. {
47. **int** buyVolume;
48. **int** sellVolume;
49. PastPrice\* prices;
50. **int** numPastPrices;
52. MarketDataCL(**int** bv=0, **int** sv=0, PastPrice\* p=NULL, **int** npp=0) : buyVolume(bv), sellVolume(sv), prices(p), numPastPrices(npp) {};
53. PastPrice getLatestPrice() { **return** prices[numPastPrices-1]; };
54. };

### Order

#### Order.h

1. #pragma once
3. #include "Stock.h"
5. **enum** OrderType { BUY = 1, SELL = 0 };
7. **class** Order
8. {
9. **private**:
10. **int** \_size;
11. OrderType \_type;
12. **bool** \_isMarket;
13. **int** \_participant;
14. **int** \_orderNumber;
15. **double** \_price;
16. Stock\* \_stock;
17. **int** \_time;
19. **static** **int** \_lastOrderNumber;
21. **static** **const** **int** BUY = 1;
22. **static** **const** **int** SELL = 0;
24. **protected**:
25. **static** **int** getNextOrderNumber();
26. **const** **char**\* OrderTypeToString(OrderType type);
27. OrderType StringToOrderType(std::string text);
29. **public**:
30. Order(){}
31. Order(OrderType type, **int** size, **double** price, **int** participant, **bool** isMarket, Stock\* stock, **int** time);
32. ~Order(**void**);
34. **void** print();
35. std::string toString();
36. **static** std::string toStringHeaderCSV();
37. std::string toStringCSV();
38. **bool** equals(Order\* order);
39. **static** **bool** compareBuys(Order order1, Order order2);
40. **static** **bool** compareSells(Order order1, Order order2);
41. **bool** isBuy();
42. **bool** isSell();
44. **bool** operator==(Order order);
46. //Getters and Setters
47. **int** getSize();
48. OrderType getOrderType();
49. **bool** isMarket();
50. **bool** isLimit();
51. **int** getParticipant();
52. **int** getOrderNumber();
53. **double** getPrice();
54. **int** getStockId();
55. Stock getStock();
56. **int** getTime();
58. **void** setSize(**int** size);
59. **void** setOrderType(OrderType type);
60. **void** setIsMarket(**bool** isMarket);
61. **void** setParticipant(**int** participant);
62. **void** setOrderNumber(**int** orderNumber);
63. **void** setPrice(**double** price);
64. **void** setStock(Stock\* stock);
65. **void** setTime(**int** time);
66. };

#### Order.cpp

1. #include "StdAfx.h"
2. #include "Order.h"
4. **int** Order::\_lastOrderNumber = -1;
6. **int** Order::getNextOrderNumber()
7. {
8. \_lastOrderNumber++;
9. **return** \_lastOrderNumber;
10. }
12. Order::Order(OrderType type, **int** size, **double** price, **int** participant, **bool** isMarket, Stock\* stock, **int** time)
13. {
14. **if** (size <= 0)
15. **throw** **new** std::exception("Orders cannot be sized 0 or less. Size passed was: " + size);
17. \_type = type;
18. \_size = size;
19. \_price = price;
20. \_participant = participant;
21. \_isMarket = isMarket;
22. \_stock = stock;
23. \_time = time;
24. \_orderNumber = getNextOrderNumber();
26. **if** (\_isMarket)
27. {
28. **if** (\_type == BUY)
29. \_price = INT\_MAX;
30. **else**
31. \_price = 0;
32. }
33. }
35. Order::~Order(**void**)
36. {}
38. **void** Order::print()
39. {
40. **char** str[512];
41. sprintf\_s(str, "Price: %.2f, Volume: %d, Type: %s, OrderNumber: %d, Market: %d", \_price, \_size, OrderTypeToString(\_type), \_orderNumber, \_isMarket);
42. std::cout << str << std::endl;
43. }
45. std::string Order::toString()
46. {
47. **char** str[512];
48. sprintf\_s(str, "Price: %.2f, Volume: %d, Type: %s, Market Order: %d, Participant: %d, Stock:[%s], Time: %d, OrderNumber: %d",
49. \_price, \_size, OrderTypeToString(\_type), \_isMarket, \_participant, \_stock->toString().c\_str(), \_time, \_orderNumber);
50. **return** str;
51. }
53. std::string Order::toStringHeaderCSV()
54. {
55. **return** "Price,Volume,Type,Market(1)/Limit(0),ParticipantId,Stock,Time,OrderId";
56. }
58. std::string Order::toStringCSV()
59. {
60. **char** str[128];
61. sprintf\_s(str, "%.2f,%d,%s,%d,%d,%s,%d,%d", \_price, \_size, OrderTypeToString(\_type), \_isMarket, \_participant, \_stock->toString().c\_str(), \_time, \_orderNumber);
62. **return** std::string(str);
63. }
65. **const** **char**\* Order::OrderTypeToString(OrderType type)
66. {
67. **if** (type == BUY)
68. **return** "BUY";
69. **else**
70. **return** "SELL";
71. }
73. OrderType Order::StringToOrderType(std::string text)
74. {
75. **if** (text == "BUY")
76. **return** OrderType::BUY;
77. **else**
78. **return** OrderType::SELL;
79. }
81. **bool** Order::equals(Order\* order)
82. {
83. **return** order->getOrderNumber() == **this**->getOrderNumber() && order->getTime() == **this**->getTime();
84. }
86. **bool** Order::compareBuys(Order order1, Order order2)
87. {
88. //Check if they are both market order or both have the same price
89. **if** (order2.isMarket() && order1.isMarket() || order2.getPrice() == order1.getPrice())
90. {
91. //Check order numbers
92. **if** (order1.getOrderNumber() < order2.getOrderNumber())
93. **return** **true**;
94. **else** **if** (order1.getOrderNumber() == order2.getOrderNumber())
95. **return** **true**;
96. **else**
97. **return** **false**;
98. }
99. **else**
100. {
101. **if** (order1.isMarket())
102. **return** **true**;
103. **if** (order2.isMarket())
104. **return** **false**;
105. **if** (order1.getPrice() > order2.getPrice())
106. **return** **true**;
107. **else**
108. **return** **false**;
109. }
110. }
112. **bool** Order::compareSells(Order order1, Order order2)
113. {
114. //Check if they are both market or have the same price
115. **if** (order2.isMarket() && order1.isMarket() || order2.getPrice() == order1.getPrice())
116. {
117. **if** (order1.getOrderNumber() < order2.getOrderNumber())
118. **return** **true**;
119. **else** **if** (order1.getOrderNumber() == order2.getOrderNumber())
120. **return** **true**;
121. **else**
122. **return** **false**;
123. }
124. **else**
125. {
126. **if** (order1.isMarket())
127. **return** **true**;
128. **if** (order2.isMarket())
129. **return** **false**;
130. **if** (order1.getPrice() > order2.getPrice())
131. **return** **false**;
132. **else**
133. **return** **true**;
134. }
135. }
137. **bool** Order::isBuy()
138. {
139. **if** (**this**->getOrderType() == BUY)
140. **return** **true**;
141. **else**
142. **return** **false**;
143. }
145. **bool** Order::isSell()
146. {
147. **if** (**this**->getOrderType() == SELL)
148. **return** **true**;
149. **else**
150. **return** **false**;
151. }
153. **bool** Order::operator==(Order order)
154. {
155. **return** order.equals(**this**);
156. }
158. //Getters and Setters
159. #pragma region Getters
161. **int** Order::getSize()
162. {
163. **return** \_size;
164. }
166. OrderType Order::getOrderType()
167. {
168. **return** \_type;
169. }
171. **bool** Order::isMarket()
172. {
173. **return** \_isMarket;
174. }
176. **bool** Order::isLimit()
177. {
178. **return** \_isMarket == **false**;
179. }
181. **int** Order::getParticipant()
182. {
183. **return** \_participant;
184. }
186. **int** Order::getOrderNumber()
187. {
188. **return** \_orderNumber;
189. }
191. **double** Order::getPrice()
192. {
193. **return** \_price;
194. }
196. **int** Order::getStockId()
197. {
198. **return** \_stock->getStockId();
199. }
201. Stock Order::getStock()
202. {
203. **return** (\*\_stock);
204. }
206. **int** Order::getTime()
207. {
208. **return** \_time;
209. }
211. #pragma endregion
213. #pragma region Setters
215. **void** Order::setSize(**int** size)
216. {
217. \_size = size;
218. }
220. **void** Order::setOrderType(OrderType type)
221. {
222. \_type = type;
223. }
225. **void** Order::setIsMarket(**bool** isMarket)
226. {
227. \_isMarket = isMarket;
228. }
230. **void** Order::setParticipant(**int** participant)
231. {
232. \_participant = participant;
233. }
235. **void** Order::setOrderNumber(**int** orderNumber)
236. {
237. \_orderNumber = orderNumber;
238. }
240. **void** Order::setPrice(**double** price)
241. {
242. \_price = price;
243. }
245. **void** Order::setStock(Stock\* stock)
246. {
247. \_stock = stock;
248. }
250. **void** Order::setTime(**int** time)
251. {
252. \_time = time;
253. }
255. #pragma endregion

### OrderBook

#### OrderBook.h

1. #pragma once
3. #include "Trade.h"
4. #include "Stock.h"
5. #include "Order.h"
6. #include "RuleManager.h"
7. #include "OrderRequest.h"
8. #include "OrderQueue.h"
9. #include "TraderManager.h"
10. #include "WallTimer.h"
11. #include "OpenClStructs.h"
12. #include "Logger.h"
13. #include "Utils.h"
14. #include "Trader.h"
15. #include "TBBlog.h"
17. **class** TraderManager;
18. **class** RuleManager;
19. **class** IRule;
20. **class** Trader;
21. **class** WallTimer;
23. **class** OrderBook
24. {
25. **private**:
26. Stock\* \_stock;
27. std::list<Order> \_buyOrders;
28. std::list<Order> \_sellOrders;
29. std::vector<Trade> \_trades;
30. OrderQueue queue;
31. **double** \_lastPrice;
32. std::vector<PastPrice> \_prices;
33. std::vector<**double**> \_allPrices;
34. **double** \_openPrice;
35. **double** \_threshold;
36. RuleManager\* \_ruleManager;
37. TraderManager\* \_traderManager;
38. **int** \_time;
40. **bool** \_performanceAnalytics;
41. WallTimer\* \_timer;
42. **double** \_matchTime;
43. **double** \_traderProcTime;
44. std::vector<**double**> \_matchTimes;
45. std::vector<**double**> \_traderProcTimes;
46. std::vector<**double**> \_oclProcTimes;
47. **size\_t** \_tradesSize;
48. **size\_t** \_ordersSize;
49. std::vector<**double**> \_pastReturns;
50. std::vector<**double**> \_spreads;
52. //Helpers for Traders
53. **int** \_buyMarketOrders, \_buyLimitOrders;
54. **int** \_sellMarketOrders, \_sellLimitOrders;
55. **int** \_buyVolume, \_sellVolume;
57. **static** **const** std::string logName;
59. Order\* getOrderPtr(Order order);
60. Order\* getOrderPtr(**int** id);
61. Order GetOrder(**int** id);
62. Trade\* getTradePtr(Trade trade);
64. **void** notifyTraders(Trade\* trade);
66. **public**:
67. OrderBook(Stock\* stock, TraderManager\* traderManager, **bool** perfAnalytics=**false**);
68. OrderBook(Stock\* stock, TraderManager\* traderManager, **double** openPrice, **bool** perfAnalytics=**false**);
69. ~OrderBook(**void**);
71. **void** print();
72. **void** printPrice();
73. **void** printTrades();
74. **void** printBrief();
76. **void** registerTrader(Trader\* trader);
77. **void** unRegisterTrader(Trader\* trader);
79. **void** processTraders();
81. **void** addRule(IRule\* rule);
82. **void** removeRule(IRule\* rule);
84. **void** submitOrder(Order order);
85. **void** deSubmitOrder(**int** orderId);
87. **void** addOrder(Order order);
88. **void** removeOrder(Order order);
89. **void** updateOrderSize(Order\*& order, **int** size);
90. **void** matchOrders();
91. **void** publishTrade(Trade trade);
93. RuleManager\* getRuleManager();
94. **void** setRuleManager(RuleManager\* ruleManager);
96. TraderManager\* getTraderManager();
97. **void** setTraderManager(TraderManager\* traderManager);
99. std::list<Order> getBuyOrders();
100. std::list<Order> getSellOrders();
101. **int** getNumBuyOrders();
102. **int** getNumSellOrders();
104. **void** setLastPrice(**double** price);
105. PastPrice getLastPrice();
106. std::vector<PastPrice> getLastPrices();
107. std::vector<PastPrice> getLastPricesN(**int** N);
108. PastPrice\* getLastPricesPArray();
109. PastPrice\* getLastPricesPArrayN(**int** N);
111. **int** GetMarketBuyCount();
112. **int** GetMarketSellCount();
113. **int** GetLimitBuyCount();
114. **int** GetLimitSellCount();
116. **int** GetBuyVolume();
117. **int** GetSellVolume();
119. **void** setTime(**int** time);
120. **int** getTime();
121. **void** update();
123. Stock\* getStock();
125. **void** enablePerfAnalytics();
126. **void** disablePerfAnalytics();
128. //Helpers for traders
129. **bool** BuyContainsMarket();
130. **bool** BuyContainsLimit();
131. **bool** SellContainsMarket();
132. **bool** SellContainsLimit();
133. **int** GetBuyMarketCount();
134. **int** GetBuyLimitCount();
135. **int** GetSellMarketCount();
136. **int** GetSellLimitCount();
137. **int** getBuyVolume();
138. **int** getSellVolume();
139. TraderManager\* GetTraderManager();
141. **double** GetAveMatchTime();
142. **double** GetMaxMatchTime();
143. **double** GetAveTraderProcTime();
144. **double** GetMaxTraderProcTime();
145. **double** GetAveOclProcTime();
146. **double** GetMaxOclProcTime();
147. **size\_t** GetBookSize();
148. **double** GetBid();
149. **double** GetCall();
150. **double** GetSpread();
151. **double** GetMinPrice();
152. **double** GetAvePrice();
153. **double** GetMaxPrice();
154. **double** GetTradesPerSecond();
155. **double** GetMinReturns1ms();
156. **double** GetAveReturns1ms();
157. **double** GetMaxReturns1ms();
158. **double** GetMinReturns1s();
159. **double** GetAveReturns1s();
160. **double** GetMaxReturns1s();
161. **double** GetReturn1m();
162. **double** GetMinRTProfit();
163. **double** GetAveRTProfit();
164. **double** GetMaxRTProfit();
165. **double** GetMinLRTProfit();
166. **double** GetAveLRTProfit();
167. **double** GetMaxLRTProfit();
168. **double** GetMinPTProfit();
169. **double** GetAvePTProfit();
170. **double** GetMaxPTProfit();
171. **double** GetMinMTProfit();
172. **double** GetAveMTProfit();
173. **double** GetMaxMTProfit();
174. **double** GetAveProfit();
175. **double** GetVolatilityPerMin();
176. **double** GetAveSpread();
177. **int** GetMinTraderProcessT();
178. **int** GetAveTraderProcessT();
179. **int** GetMaxTraderProcessT();
180. };

#### OrderBook.cpp

1. #include "StdAfx.h"
2. #include "OrderBook.h"
4. **const** std::string OrderBook::logName = "OrderBook";
6. OrderBook::OrderBook(Stock\* stock, TraderManager\* traderManager, **bool** perfAnalytics)
7. {
8. \_threshold = 0.01;
9. \_stock = stock;
10. \_ruleManager = **new** RuleManager();
11. \_traderManager = traderManager;
12. \_openPrice = stock->getLastPrice();
13. \_lastPrice = \_openPrice;
14. \_time = 0;
15. \_prices.push\_back(PastPrice(\_lastPrice, \_time));
16. \_performanceAnalytics = perfAnalytics;
17. **if** (\_performanceAnalytics)
18. {
19. \_timer = **new** WallTimer();
20. \_matchTime = 0;
21. \_traderProcTime = 0;
22. std::string temp = "Time,Price,Bid,Call,Spread,BuyOrderCount,SellOrderCount,TradesCount,MatchTime,OclProcTime,TraderProcTime";
23. //std::cout << temp << std::endl;
24. Logger::GetInstance()->Data(temp);
25. }
26. **else**
27. {
28. \_timer = NULL;
29. \_matchTime = NULL;
30. \_traderProcTime = NULL;
31. std::string temp = "Time,Price,Bid,Call,Spread,NumberBuyOrder,NumberSellOrder,NumberTrades,MinPrice,AvePrice,MaxPrice";
32. //std::cout << temp << std::endl;
33. Logger::GetInstance()->Data(temp);
34. }
35. \_buyMarketOrders = 0;
36. \_buyLimitOrders = 0;
37. \_sellMarketOrders = 0;
38. \_sellLimitOrders = 0;
39. \_buyVolume = 0;
40. \_sellVolume = 0;
42. **try**
43. {
44. \_traderManager->Init();
45. }
46. **catch** (...)
47. {
48. std::stringstream temp1; temp1 << "Failed in OrderBook():\_traderManager->Init() - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
49. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
50. Logger::GetInstance()->Error(logName, temp);
51. std::cerr << temp << std::endl;
52. std::getchar();
53. exit(-1);
54. }
55. }
57. OrderBook::OrderBook(Stock\* stock, TraderManager\* traderManager, **double** openPrice, **bool** perfAnalytics)
58. {
59. \_threshold = 0.01;
60. \_stock = stock;
61. \_ruleManager = **new** RuleManager();
62. \_traderManager = traderManager;
63. \_openPrice = openPrice;
64. \_lastPrice = stock->getLastPrice();
65. \_time = 0;
66. \_prices.push\_back(PastPrice(\_lastPrice, \_time));
67. \_performanceAnalytics = perfAnalytics;
68. **if** (\_performanceAnalytics)
69. {
70. \_timer = **new** WallTimer();
71. \_matchTime = 0;
72. \_traderProcTime = 0;
73. std::string temp = "Time,Price,Bid,Call,Spread,BuyOrderCount,SellOrderCount,TradesCount,MatchTime,OclProcTime,TraderProcTime";
74. //std::cout << temp << std::endl;
75. Logger::GetInstance()->Data(temp);
76. }
77. **else**
78. {
79. \_timer = NULL;
80. \_matchTime = NULL;
81. \_traderProcTime = NULL;
82. std::string temp = "Time,Price,Bid,Call,Spread,NumberBuyOrder,NumberSellOrder,NumberTrades";
83. //std::cout << temp << std::endl;
84. Logger::GetInstance()->Data(temp);
85. }
86. \_buyMarketOrders = 0;
87. \_buyLimitOrders = 0;
88. \_sellMarketOrders = 0;
89. \_sellLimitOrders = 0;
90. \_buyVolume = 0;
91. \_sellVolume = 0;
93. **try**
94. {
95. \_traderManager->Init();
96. }
97. **catch** (...)
98. {
99. std::stringstream temp1; temp1 << "Failed in OrderBook():\_traderManager->Init() - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
100. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
101. Logger::GetInstance()->Error(logName, temp);
102. **throw** **new** std::exception(temp.c\_str());
103. }
104. }
106. OrderBook::~OrderBook(**void**)
107. {
108. **if** (\_performanceAnalytics)
109. {
110. **delete** \_timer;
111. }
112. **delete** \_ruleManager;
113. }
115. Order\* OrderBook::getOrderPtr(Order order)
116. {
117. Logger::GetInstance()->Debug(logName, Utils::Merge("Getting Pointer for order:", order.toString()));
118. std::list<Order>::iterator it;
120. **if** (order.isBuy())
121. {
122. **for** (it = \_buyOrders.begin(); it != \_buyOrders.end(); it++)
123. {
124. **if** (it->equals(&order))
125. {
126. Logger::GetInstance()->Debug(logName, "DONE");
127. **return** &\*it;
128. }
129. }
130. }
131. **else**
132. {
133. **for** (it = \_sellOrders.begin(); it != \_sellOrders.end(); it++)
134. {
135. **if** (it->equals(&order))
136. {
137. Logger::GetInstance()->Debug(logName, "DONE");
138. **return** &\*it;
139. }
140. }
141. }
143. Logger::GetInstance()->Debug(logName, "DONE");
144. **return** NULL;
145. }
147. Order\* OrderBook::getOrderPtr(**int** id)
148. {
149. Logger::GetInstance()->Debug(logName, Utils::Merge("Getting Pointer for order id:", Utils::ItoS(id)));
150. std::list<Order>::iterator it;
152. //First search buy orders
153. **for** (it = \_buyOrders.begin(); it != \_buyOrders.end(); it++)
154. {
155. **if** (it->getOrderNumber() == id)
156. {
157. Logger::GetInstance()->Debug(logName, "DONE");
158. **return** &\*it;
159. }
160. }
162. //If not found search sell orders
163. **for** (it = \_sellOrders.begin(); it != \_sellOrders.end(); it++)
164. {
165. **if** (it->getOrderNumber() == id)
166. {
167. Logger::GetInstance()->Debug(logName, "DONE");
168. **return** &\*it;
169. }
170. }
172. //if nothing is found
173. Logger::GetInstance()->Debug(logName, "DONE");
174. **return** NULL;
175. }
177. Order OrderBook::GetOrder(**int** id)
178. {
179. Logger::GetInstance()->Debug(logName, Utils::Merge("Getting Pointer for order id:", Utils::ItoS(id)));
180. std::list<Order>::iterator it;
182. //First search buy orders
183. **for** (it = \_buyOrders.begin(); it != \_buyOrders.end(); it++)
184. {
185. **if** (it->getOrderNumber() == id)
186. {
187. Logger::GetInstance()->Debug(logName, "DONE");
188. **return** \*it;
189. }
190. }
192. //If not found search sell orders
193. **for** (it = \_sellOrders.begin(); it != \_sellOrders.end(); it++)
194. {
195. **if** (it->getOrderNumber() == id)
196. {
197. Logger::GetInstance()->Debug(logName, "DONE");
198. **return** \*it;
199. }
200. }
202. //if nothing is found
203. Logger::GetInstance()->Debug(logName, "DONE");
204. **throw** **new** std::exception(Utils::Merge("No order found with id:", Utils::ItoS(id)).c\_str());
205. }
207. Trade\* OrderBook::getTradePtr(Trade trade)
208. {
209. Logger::GetInstance()->Debug(logName, Utils::Merge("Getting Pointer for trade:", trade.toString()));
210. std::vector<Trade>::iterator it;
212. **for** (it = \_trades.begin(); it != \_trades.end(); it++)
213. {
214. **if** (it->equals(&trade))
215. {
216. Logger::GetInstance()->Debug(logName, "DONE");
217. **return** &\*it;
218. }
219. }
221. Logger::GetInstance()->Debug(logName, "DONE");
222. **return** NULL;
223. }
225. **void** OrderBook::notifyTraders(Trade\* trade)
226. {
227. Logger::GetInstance()->Debug(logName, "Notifying Traders of trade");
228. \_traderManager->notify(trade);
229. Logger::GetInstance()->Debug(logName, "DONE");
230. }
232. **void** OrderBook::print()
233. {
234. std::stringstream stream;
236. stream << "==============================\n";
238. stream << "---------------" << **this**->getTime() << "---------------\n";
240. std::list<Order>::iterator it;
242. **for** (it = \_buyOrders.begin(); it != \_buyOrders.end(); it++)
243. {
244. stream << it->toString() << "\n";
245. }
247. stream << "\n";
249. **for** (it = \_sellOrders.begin(); it != \_sellOrders.end(); it++)
250. {
251. stream << it->toString() << "\n";
252. }
254. stream << "------------Trades------------\n";
256. **for** (**int** i = 0; i < \_trades.size(); i++)
257. {
258. stream << \_trades.at(i).toString() << "\n";
259. }
261. std::cout << stream.str() << std::endl;
262. }
264. **void** OrderBook::printPrice()
265. {
266. std::cout << \_lastPrice << std::endl;
267. }
269. **void** OrderBook::printTrades()
270. {
271. std::stringstream stream;
273. stream << "------------Trades------------\n";
275. **for** (**int** i = 0; i < \_trades.size(); i++)
276. {
277. stream << \_trades.at(i).toString() << "\n";
278. }
280. std::cout << stream.str() << std::endl;
281. }
283. **void** OrderBook::printBrief()
284. {
285. std::stringstream stream;
287. **if** (\_performanceAnalytics)
288. {
289. /\*char T[512];
290. sprintf\_s(T, "%d,%.2f,%d,%d,%d,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f\t\t",
291. \_time, this->getLastPrice().price, this->getBuyOrders().size(), this->getSellOrders().size(), \_trades.size(),
292. \_matchTime, GetAveMatchTime(), GetMaxMatchTime(),
293. \_traderProcTime, GetAveTraderProcTime(), GetMaxTraderProcTime(),
294. \_traderManager->getProcessTime(), GetAveOclProcTime(), GetMaxOclProcTime());
295. stream << T << "\r";\*/
296. **char** T[512];
297. sprintf\_s(T, "%d,%.2f,%.2f,%.2f,%.2f,%d,%d,%d,%.3f,%.3f,%.3f",
298. \_time, **this**->getLastPrice().price, **this**->GetBid(), **this**->GetCall(), **this**->GetSpread(),
299. **this**->getBuyOrders().size(), **this**->getSellOrders().size(), \_trades.size(),
300. \_matchTime, \_traderProcTime, \_traderManager->getProcessTime());
302. /\*sprintf\_s(T, "%d,%.2f,%d,%d,%d,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f\t\t",
303. \_time, this->getLastPrice().price, this->getBuyOrders().size(), this->getSellOrders().size(), \_trades.size(),
304. \_matchTime, 1.0f, 1.0f,
305. \_traderProcTime, 1.0f, 1.0f,
306. \_traderManager->getProcessTime(), 1.0f, 1.0f);\*/
307. stream << T << "\r";
308. }
309. **else**
310. {
311. **char** T[128];
312. sprintf\_s(T, "%d,%.2f,%d,%d,%d\t\t", \_time, **this**->getLastPrice().price, **this**->getBuyOrders().size(), **this**->getSellOrders().size(), \_trades.size());
313. stream << T << "\r";
314. }
316. std::cout << stream.str();
318. std::ofstream output;
319. }
321. **void** OrderBook::registerTrader(Trader\* trader)
322. {
323. Logger::GetInstance()->Debug(logName, "Registering Trader");
324. \_traderManager->addTrader(trader);
325. Logger::GetInstance()->Debug(logName, "DONE");
326. }
328. **void** OrderBook::unRegisterTrader(Trader\* trader)
329. {
330. Logger::GetInstance()->Debug(logName, "Unregistering Trader");
331. \_traderManager->removeTrader(trader);
332. Logger::GetInstance()->Debug(logName, "DONE");
333. }
335. **void** OrderBook::processTraders()
336. {
337. Logger::GetInstance()->Debug(logName, "Processing Traders");
339. **if** (\_performanceAnalytics)
340. \_timer->Start();
341. **try**
342. {
343. \_traderManager->process(**this**);
344. }
345. **catch** (...)
346. {
347. std::stringstream temp1; temp1 << "Failed in processTraders:\_traderManager->process() - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
348. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
349. Logger::GetInstance()->Error(logName, temp);
350. std::cerr << temp << std::endl;
351. std::getchar();
352. exit(-1);
353. }
354. **if** (\_performanceAnalytics)
355. \_traderProcTime = \_timer->GetCounter();
357. Logger::GetInstance()->Debug(logName, "DONE");
358. }
360. **void** OrderBook::addRule(IRule\* rule)
361. {
362. Logger::GetInstance()->Info(logName, Utils::Merge("Adding rule:", rule->ToString()));
363. \_ruleManager->addRule(rule);
364. }
366. **void** OrderBook::removeRule(IRule\* rule)
367. {
368. Logger::GetInstance()->Info(logName, Utils::Merge("Removing rule:", rule->ToString()));
369. \_ruleManager->removeRule(rule);
370. }
372. **void** OrderBook::submitOrder(Order order)
373. {
374. Logger::GetInstance()->Debug(logName, Utils::Merge("Submitting order to queue. Order:", order.toString()));
375. queue.enqueue(OrderRequest(order, **true**));
376. }
378. **void** OrderBook::deSubmitOrder(**int** orderId)
379. {
380. Logger::GetInstance()->Debug(logName, Utils::Merge("Submitting cancel request to queue for order id:", Utils::ItoS(orderId)));
381. queue.enqueue(OrderRequest(GetOrder(orderId), **false**));
382. }
384. **void** OrderBook::addOrder(Order order)
385. {
386. Logger::GetInstance()->Order(order.toStringCSV());
387. Logger::GetInstance()->Debug(logName, "Adding order to appropriate order list");
388. **if** (order.isBuy())
389. {
390. \_buyOrders.push\_back(order);
391. Logger::GetInstance()->Info(logName, Utils::Merge("Added buy order:", order.toString()));
393. Logger::GetInstance()->Debug(logName, "Sorting Buy orders");
394. \_buyOrders.sort(Order::compareBuys);
395. Logger::GetInstance()->Debug(logName, "DONE");
397. **if** (order.isMarket())
398. {
399. \_buyMarketOrders++;
400. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Buy Market Orders:", Utils::ItoS(\_buyMarketOrders)));
401. }
402. **else** **if** (order.isLimit())
403. {
404. \_buyLimitOrders++;
405. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Buy Limit Orders:", Utils::ItoS(\_buyLimitOrders)));
406. }
408. \_buyVolume += order.getSize();
409. Logger::GetInstance()->Info(logName, Utils::Merge("Total Buy Volume on the book:", Utils::ItoS(\_buyVolume)));
410. }
411. **else**
412. {
413. \_sellOrders.push\_back(order);
414. Logger::GetInstance()->Info(logName, Utils::Merge("Added sell order:", order.toString()));
416. Logger::GetInstance()->Debug(logName, "Sorting Sell orders");
417. \_sellOrders.sort(Order::compareSells);
418. Logger::GetInstance()->Debug(logName, "DONE");
420. **if** (order.isMarket())
421. {
422. \_sellMarketOrders++;
423. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Sell Market Orders:", Utils::ItoS(\_sellMarketOrders)));
424. }
425. **else** **if**(order.isLimit())
426. {
427. \_sellLimitOrders++;
428. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Sell Limit Orders:", Utils::ItoS(\_sellLimitOrders)));
429. }
431. \_sellVolume += order.getSize();
432. Logger::GetInstance()->Info(logName, Utils::Merge("Total Sell Volume on the book:", Utils::ItoS(\_sellVolume)));
433. }
434. Logger::GetInstance()->Debug(logName, "DONE");
435. }
437. **void** OrderBook::removeOrder(Order order)
438. {
439. Logger::GetInstance()->Debug(logName, "Removing order to appropriate order list");
440. **if** (order.isBuy())
441. {
442. \_buyOrders.remove(order);
443. Logger::GetInstance()->Info(logName, Utils::Merge("Removed buy order:", order.toString()));
445. Logger::GetInstance()->Debug(logName, "Sorting Buy orders");
446. \_buyOrders.sort(Order::compareBuys);
447. Logger::GetInstance()->Debug(logName, "DONE");
449. **if** (order.isMarket())
450. {
451. \_buyMarketOrders--;
452. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Buy Market Orders:", Utils::ItoS(\_buyMarketOrders)));
453. }
454. **else** **if** (order.isLimit())
455. {
456. \_buyLimitOrders--;
457. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Buy Limit Orders:", Utils::ItoS(\_buyLimitOrders)));
458. }
460. \_buyVolume -= order.getSize();
461. Logger::GetInstance()->Info(logName, Utils::Merge("Total Buy Volume on the book:", Utils::ItoS(\_buyVolume)));
462. }
463. **else**
464. {
465. \_sellOrders.remove(order);
466. Logger::GetInstance()->Info(logName, Utils::Merge("Removed sell order:", order.toString()));
468. Logger::GetInstance()->Debug(logName, "Sorting Sell orders");
469. \_sellOrders.sort(Order::compareSells);
470. Logger::GetInstance()->Debug(logName, "DONE");
472. **if** (order.isMarket())
473. {
474. \_sellMarketOrders--;
475. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Sell Market Orders:", Utils::ItoS(\_sellMarketOrders)));
476. }
477. **else** **if** (order.isLimit())
478. {
479. \_sellLimitOrders--;
480. Logger::GetInstance()->Info(logName, Utils::Merge("Number of Sell Limit Orders:", Utils::ItoS(\_sellLimitOrders)));
481. }
483. \_sellVolume -= order.getSize();
484. Logger::GetInstance()->Info(logName, Utils::Merge("Total Sell Volume on the book:", Utils::ItoS(\_sellVolume)));
485. }
486. Logger::GetInstance()->Debug(logName, "DONE");
487. }
489. **void** OrderBook::updateOrderSize(Order\*& order, **int** size)
490. {
491. Logger::GetInstance()->Debug(logName, Utils::Merge(Utils::Merge("Updating order size to:", Utils::ItoS(size)), order->toString()));
492. **if** (size <= 0)
493. {
494. **this**->removeOrder(\*order);
495. //if (order->isBuy())
496. //  \_buyOrders.remove(\*order);
497. //else
498. //  \_sellOrders.remove(\*order);
499. order = NULL;
500. }
501. **else**
502. {
503. **if** (order->isBuy())
504. \_buyVolume -= (order->getSize() - size);
505. **else** **if** (order->isSell())
506. \_sellVolume -= (order->getSize() - size);
507. order->setSize(size);
508. }
509. Logger::GetInstance()->Debug(logName, "DONE");
510. }
512. **void** OrderBook::matchOrders()
513. {
514. Logger::GetInstance()->Debug(logName, "Matching Orders");
516. **if** (\_performanceAnalytics)
517. \_timer->Start();
518. **if** (!queue.isEmpty())
519. {
520. OrderRequest request = queue.dequeue();
521. Logger::GetInstance()->Info(logName, Utils::Merge("Dequeued request:", request.ToString()));
523. Logger::GetInstance()->Debug(logName, "Processing Request");
524. **if** (request.isInsert())
525. {
526. **try**
527. {
528. **this**->addOrder(request.getOrder());
530. \_ruleManager->applyRules(**this**, getOrderPtr(request.getOrder()));
531. }
532. **catch** (...)
533. {
534. std::stringstream temp1; temp1 << "Failed in matchOrders:addOrder/applyRules - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
535. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
536. Logger::GetInstance()->Error(logName, temp);
537. std::cerr << temp << std::endl;
538. std::getchar();
539. exit(-1);
540. }
541. }
542. **else**
543. {
544. Order tempOrder = request.getOrder();
545. **this**->removeOrder(tempOrder);
547. **try**
548. {
549. \_ruleManager->applyRules(**this**, &tempOrder);
550. }
551. **catch** (...)
552. {
553. std::stringstream temp1; temp1 << "Failed in matchOrders:applyRules - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
554. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
555. Logger::GetInstance()->Error(logName, temp);
556. std::cerr << temp << std::endl;
557. std::getchar();
558. exit(-1);
559. }
560. }
562. Logger::GetInstance()->Debug(logName, "DONE");
563. }
564. **if** (\_performanceAnalytics)
565. \_matchTime = \_timer->GetCounter();
566. }
568. **void** OrderBook::publishTrade(Trade trade)
569. {
570. Logger::GetInstance()->Info(logName, Utils::Merge("Publishing trade:", trade.toString()));
571. \_stock->setLastPrice(trade.getPrice());
573. Logger::GetInstance()->Info(logName, Utils::Merge("Last Trade Price:", Utils::DtoS(\_stock->getLastPrice())));
574. Logger::GetInstance()->Trade(trade.toStringCSV());
575. //\_tradesCount++;
576. \_trades.push\_back(trade);
577. Trade\* tradePtr = getTradePtr(trade);
579. Logger::GetInstance()->Debug(logName, "Notifying Traders");
580. **this**->notifyTraders(tradePtr);
581. //this->notifyTraders(&trade);
582. Logger::GetInstance()->Debug(logName, "DONE");
583. }
585. RuleManager\* OrderBook::getRuleManager()
586. {
587. **return** \_ruleManager;
588. }
590. **void** OrderBook::setRuleManager(RuleManager\* ruleManager)
591. {
592. \_ruleManager = ruleManager;
593. }
595. std::list<Order> OrderBook::getBuyOrders()
596. {
597. **return** \_buyOrders;
598. }
600. std::list<Order> OrderBook::getSellOrders()
601. {
602. **return** \_sellOrders;
603. }
605. **int** OrderBook::getNumBuyOrders()
606. {
607. **return** \_buyOrders.size();
608. }
610. **int** OrderBook::getNumSellOrders()
611. {
612. **return** \_sellOrders.size();
613. }
615. **void** OrderBook::setLastPrice(**double** price)
616. {
617. **if** (price > 500 || price < 20)
618. std::cout << "error";
620. \_lastPrice = price;
621. Logger::GetInstance()->Info(logName, Utils::Merge("Set last Price: $", Utils::DtoS(\_lastPrice)));
622. /\*char temp[16];
623. sprintf\_s(temp, "%.2f", price);
624. std::string temp2(temp);\*/
625. }
627. PastPrice OrderBook::getLastPrice()
628. {
629. **return** PastPrice(\_lastPrice, \_time);
630. }
632. std::vector<PastPrice> OrderBook::getLastPrices()
633. {
634. **return** \_prices;
635. }
637. std::vector<PastPrice> OrderBook::getLastPricesN(**int** N)
638. {
639. **if** (N > \_prices.size())
640. **throw** **new** std::exception("Range is larger than length of past prices data");
642. std::vector<PastPrice> result;
644. **for** (**int** i = (\_prices.size() - 1); i < (\_prices.size() - (N+1)); i--)
645. {
646. result.push\_back(PastPrice(\_prices.at(i).price, \_prices.at(i).time));
647. }
649. **return** result;
650. }
652. PastPrice\* OrderBook::getLastPricesPArray()
653. {
654. /\*PastPrice\* pPrices = new PastPrice[\_prices.size()-1];
656. for (int i=0; i < \_prices.size(); i++)
657. {
658. pPrices[i].price = \_prices[i].price;
659. pPrices[i].time = \_prices[i].time;
660. }
661. return pPrices;\*/
662. /\*if (\_prices.size() > 4000)
663. {
664. std::vector<PastPrice> temp;
665. for (int i=\_prices.size()-4001; i < \_prices.size(); i++)
666. {
667. temp.push\_back(\_prices[i]);
668. }
669. return &temp[0];
670. }
671. else
672. {\*/
673. **return** &\_prices[0];
674. }
676. PastPrice\* OrderBook::getLastPricesPArrayN(**int** N)
677. {
678. /\*if (N > \_prices.size())
679. throw new std::exception("Range is larger than length of past prices data");
681. PastPrice\* pPrices = new PastPrice[N];
683. for (int i= (\_prices.size()-1-N); i < \_prices.size(); i++)
684. {
685. pPrices[i].price = \_prices[i].price;
686. pPrices[i].time = \_prices[i].time;
687. }
689. return pPrices; \*/
690. **return** &\_prices[0];
691. }
693. **int** OrderBook::GetMarketBuyCount()
694. {
695. **return** \_buyMarketOrders;
696. }
698. **int** OrderBook::GetMarketSellCount()
699. {
700. **return** \_sellMarketOrders;
701. }
703. **int** OrderBook::GetLimitBuyCount()
704. {
705. **return** \_buyLimitOrders;
706. }
708. **int** OrderBook::GetLimitSellCount()
709. {
710. **return** \_sellLimitOrders;
711. }
713. **void** OrderBook::setTime(**int** time)
714. {
715. \_time = time;
716. }
718. **int** OrderBook::getTime()
719. {
720. **return** \_time;
721. }
723. **void** OrderBook::update()
724. {
725. std::stringstream stream;
726. **if** (\_performanceAnalytics)
727. {
728. \_matchTimes.push\_back(\_matchTime);
729. \_traderProcTimes.push\_back(\_traderProcTime);
730. \_oclProcTimes.push\_back(\_traderManager->getProcessTime());
732. \_tradesSize = **sizeof**(\_trades) + **sizeof**(Trade)\*\_trades.capacity();
733. \_ordersSize = (**sizeof**(\_buyOrders) + **sizeof**(Order)\*\_buyOrders.size()) + (**sizeof**(\_sellOrders) + **sizeof**(Order)\*\_sellOrders.size());
735. **char** T[512];
736. /\*sprintf\_s(T, "%d,%.2f,%d,%d,%d,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f",
737. \_time, this->getLastPrice().price, this->getBuyOrders().size(), this->getSellOrders().size(), \_trades.size(),
738. \_matchTime, GetAveMatchTime(), GetMaxMatchTime(),
739. \_traderProcTime, GetAveTraderProcTime(), GetMaxTraderProcTime(),
740. \_traderManager->getProcessTime(), GetAveOclProcTime(), GetMaxOclProcTime());\*/
741. sprintf\_s(T, "%d,%.2f,%.2f,%.2f,%.2f,%d,%d,%d,%.3f,%.3f,%.3f",
742. \_time, **this**->getLastPrice().price, **this**->GetBid(), **this**->GetCall(), **this**->GetSpread(),
743. **this**->getBuyOrders().size(), **this**->getSellOrders().size(), \_trades.size(),
744. \_matchTime, \_traderProcTime, \_traderManager->getProcessTime());
745. stream << T;
746. Logger::GetInstance()->Data(stream.str());
747. Logger::GetInstance()->Prices(**this**->getLastPrice().price);
748. Logger::GetInstance()->BuyOrders(**this**->getBuyOrders().size());
749. Logger::GetInstance()->SellOrders(**this**->getSellOrders().size());
750. }
751. **else**
752. {
753. **char** T[128];
754. sprintf\_s(T, "%d,%.2f,%d,%d,%d", \_time, **this**->getLastPrice().price, **this**->getBuyOrders().size(), **this**->getSellOrders().size(), \_trades.size());
755. stream << T;
756. Logger::GetInstance()->Data(stream.str());
757. }
759. //if the price has changed store the change and time
760. **if** (**this**->getLastPrice().price != \_prices.back().price)
761. {
762. **if** (\_prices.size() > 4000)
763. {
764. **for** (**int** i=0; i < \_prices.size()-1; i++)
765. {
766. \_prices[i] = \_prices[i+1];
767. }
768. \_prices[4000] = PastPrice(**this**->getLastPrice().price, \_time);
769. }
770. **else**
771. {
772. \_prices.push\_back(PastPrice(**this**->getLastPrice().price, \_time));
773. }
774. }
775. \_allPrices.push\_back(**this**->getLastPrice().price);
776. \_spreads.push\_back(**this**->GetSpread());
777. **if** (\_time > 0)
778. \_pastReturns.push\_back(\_allPrices.back() - \_allPrices[\_allPrices.size()-2]);
779. \_time++;
781. Logger::GetInstance()->SetTime(\_time, **true**);
782. \_traderManager->notify(\_time);
783. }
785. Stock\* OrderBook::getStock()
786. {
787. **return** \_stock;
788. }
790. **void** OrderBook::enablePerfAnalytics()
791. {
792. **if** (!\_performanceAnalytics)
793. {
794. \_performanceAnalytics = **true**;
795. \_timer = **new** WallTimer();
796. }
797. }
799. **void** OrderBook::disablePerfAnalytics()
800. {
801. **if** (\_performanceAnalytics)
802. {
803. \_performanceAnalytics = **false**;
804. **if** (\_timer != NULL)
805. **delete** \_timer;
806. \_timer = NULL;
807. }
808. }
810. **bool** OrderBook::BuyContainsMarket()
811. {
812. **return** **this**->GetBuyMarketCount() > 0;
813. }
815. **bool** OrderBook::BuyContainsLimit()
816. {
817. **return** **this**->GetBuyLimitCount() > 0;
818. }
820. **bool** OrderBook::SellContainsMarket()
821. {
822. **return** **this**->GetSellMarketCount() > 0;
823. }
825. **bool** OrderBook::SellContainsLimit()
826. {
827. **return** **this**->GetSellLimitCount() > 0;
828. }
830. **int** OrderBook::GetBuyMarketCount()
831. {
832. **return** \_buyMarketOrders;
833. }
835. **int** OrderBook::GetBuyLimitCount()
836. {
837. **return** \_buyLimitOrders;
838. }
840. **int** OrderBook::GetSellMarketCount()
841. {
842. **return** \_sellMarketOrders;
843. }
845. **int** OrderBook::GetSellLimitCount()
846. {
847. **return** \_sellLimitOrders;
848. }
850. **int** OrderBook::getBuyVolume()
851. {
852. **int** count = 0;
854. **for** (auto it = \_buyOrders.begin(); it != \_buyOrders.end(); it++)
855. {
856. count += it->getSize();
857. }
859. **return** count;
860. //return \_buyVolume;
861. }
863. **int** OrderBook::getSellVolume()
864. {
865. **int** count = 0;
867. **for** (auto it = \_sellOrders.begin(); it != \_sellOrders.end(); it++)
868. {
869. count += it->getSize();
870. }
872. **return** count;
873. //return \_sellVolume;
874. }
876. TraderManager\* OrderBook::GetTraderManager()
877. {
878. **return** \_traderManager;
879. }
881. **double** OrderBook::GetAveMatchTime()
882. {
883. **return** Utils::Mean(\_matchTimes);
884. }
886. **double** OrderBook::GetMaxMatchTime()
887. {
888. **return** Utils::Max(\_matchTimes);
889. }
891. **double** OrderBook::GetAveTraderProcTime()
892. {
893. **return** Utils::Mean(\_traderProcTimes);
894. }
896. **double** OrderBook::GetMaxTraderProcTime()
897. {
898. **return** Utils::Max(\_traderProcTimes);
899. }
901. **double** OrderBook::GetAveOclProcTime()
902. {
903. **return** Utils::Mean(\_oclProcTimes);
904. }
906. **double** OrderBook::GetMaxOclProcTime()
907. {
908. **return** Utils::Max(\_oclProcTimes);
909. }
911. **size\_t** OrderBook::GetBookSize()
912. {
913. **size\_t** result = 0;
914. result += **sizeof**(OrderBook);
915. result += \_traderManager->SizeOf();
916. result += **sizeof**(Trade)\*\_trades.capacity();
917. result += **sizeof**(Order)\*(\_buyOrders.size() + \_sellOrders.size());
918. result += \_prices.size()\***sizeof**(PastPrice) + **sizeof**(std::vector<**double**>);
919. result += \_allPrices.size()\***sizeof**(**double**) + **sizeof**(std::vector<**double**>);
920. result += \_pastReturns.size()\***sizeof**(**double**) + **sizeof**(std::vector<**double**>);
921. **return** result;
922. }
924. **double** OrderBook::GetBid()
925. {
926. **double** bid = -DBL\_MAX;
927. **bool** set = **false**;
928. **for** (auto it = \_buyOrders.begin(); it != \_buyOrders.end(); it++)
929. {
930. **if** (it->isLimit())
931. {
932. bid = max(bid, it->getPrice());
933. set = **true**;
934. }
935. }
937. **if** (!set)
938. {
939. bid = **this**->getLastPrice().price;
940. }
942. **return** bid;
943. }
945. **double** OrderBook::GetCall()
946. {
947. **double** call = DBL\_MAX;
948. **bool** set = **false**;
949. **for** (auto it = \_sellOrders.begin(); it != \_sellOrders.end(); it++)
950. {
951. **if** (it->isLimit())
952. {
953. call = min(call, it->getPrice());
954. set = **true**;
955. }
956. }
958. **if** (!set)
959. {
960. call = **this**->getLastPrice().price;
961. }
963. **return** call;
964. }
966. **double** OrderBook::GetSpread()
967. {
968. **return** **this**->GetCall() - **this**->GetBid();
969. }
971. **double** OrderBook::GetMinPrice()
972. {
973. **return** Utils::Min(\_allPrices);
974. }
976. **double** OrderBook::GetAvePrice()
977. {
978. **return** Utils::Mean(\_allPrices);
979. }
981. **double** OrderBook::GetMaxPrice()
982. {
983. **return** Utils::Max(\_allPrices);
984. }
986. **double** OrderBook::GetTradesPerSecond()
987. {
988. **return** \_trades.size()/((**double**)\_time/1000);
989. }
991. **double** OrderBook::GetMinReturns1ms()
992. {
993. **return** Utils::Min(\_pastReturns);
994. }
996. **double** OrderBook::GetAveReturns1ms()
997. {
998. **return** Utils::Mean(\_pastReturns);
999. }
1001. **double** OrderBook::GetMaxReturns1ms()
1002. {
1003. **return** Utils::Max(\_pastReturns);
1004. }
1006. **double** OrderBook::GetMinReturns1s()
1007. {
1008. std::vector<**double**> tempReturns;
1009. **for** (**int** i=1000; i < \_allPrices.size(); i++)
1010. tempReturns.push\_back(\_allPrices[i]-\_allPrices[i-1000]);
1011. **return** Utils::Min(tempReturns);
1012. }
1014. **double** OrderBook::GetAveReturns1s()
1015. {
1016. std::vector<**double**> tempReturns;
1017. **for** (**int** i=1000; i < \_allPrices.size(); i++)
1018. tempReturns.push\_back(\_allPrices[i]-\_allPrices[i-1000]);
1019. **return** Utils::Mean(tempReturns);
1020. }
1022. **double** OrderBook::GetMaxReturns1s()
1023. {
1024. std::vector<**double**> tempReturns;
1025. **for** (**int** i=1000; i < \_allPrices.size(); i++)
1026. tempReturns.push\_back(\_allPrices[i]-\_allPrices[i-1000]);
1027. **return** Utils::Max(tempReturns);
1028. }
1030. **double** OrderBook::GetReturn1m()
1031. {
1032. **if** (\_time >= 60000)
1033. **return** \_allPrices[59999] - \_openPrice;
1034. **else**
1035. **return** 0;
1036. }
1038. **double** OrderBook::GetMinRTProfit()
1039. {
1040. **return** \_traderManager->GetMinRTProfit();
1041. }
1043. **double** OrderBook::GetAveRTProfit()
1044. {
1045. **return** \_traderManager->GetAveRTProfit();
1046. }
1048. **double** OrderBook::GetMaxRTProfit()
1049. {
1050. **return** \_traderManager->GetMaxRTProfit();
1051. }
1053. **double** OrderBook::GetMinLRTProfit()
1054. {
1055. **return** \_traderManager->GetMinLRTProfit();
1056. }
1058. **double** OrderBook::GetAveLRTProfit()
1059. {
1060. **return** \_traderManager->GetAveLRTProfit();
1061. }
1063. **double** OrderBook::GetMaxLRTProfit()
1064. {
1065. **return** \_traderManager->GetMaxLRTProfit();
1066. }
1068. **double** OrderBook::GetMinPTProfit()
1069. {
1070. **return** \_traderManager->GetMinPTProfit();
1071. }
1073. **double** OrderBook::GetAvePTProfit()
1074. {
1075. **return** \_traderManager->GetAvePTProfit();
1076. }
1078. **double** OrderBook::GetMaxPTProfit()
1079. {
1080. **return** \_traderManager->GetMaxPTProfit();
1081. }
1083. **double** OrderBook::GetMinMTProfit()
1084. {
1085. **return** \_traderManager->GetMinMTProfit();
1086. }
1088. **double** OrderBook::GetAveMTProfit()
1089. {
1090. **return** \_traderManager->GetAveMTProfit();
1091. }
1093. **double** OrderBook::GetMaxMTProfit()
1094. {
1095. **return** \_traderManager->GetMaxMTProfit();
1096. }
1098. **double** OrderBook::GetAveProfit()
1099. {
1100. **return** \_traderManager->GetAveProfit();
1101. }
1103. **double** OrderBook::GetVolatilityPerMin()
1104. {
1105. std::vector<**double**> temp;
1106. temp.resize(\_allPrices.size());
1107. tbb::parallel\_for(tbb::blocked\_range<**int**>(1, \_allPrices.size()-1), TBBlog(\_allPrices, temp));
1108. **double** vol = Utils::Stdev(temp);
1109. vol \*= std::sqrt(60000.0f);
1110. **return** vol;
1111. }
1113. **double** OrderBook::GetAveSpread()
1114. {
1115. **return** Utils::Mean(\_spreads);
1116. }
1118. **int** OrderBook::GetMinTraderProcessT()
1119. {
1120. **return** \_traderManager->GetMinTraderProcessT();
1121. }
1123. **int** OrderBook::GetAveTraderProcessT()
1124. {
1125. **return** \_traderManager->GetAveTraderProcessT();
1126. }
1128. **int** OrderBook::GetMaxTraderProcessT()
1129. {
1130. **return** \_traderManager->GetMaxTraderProcessT();
1131. }

### OrderBookSim

#### OrderBookSim.cpp

1. // OrderBookSim.cpp : Defines the entry point for the console application.
3. #include "stdafx.h"
5. #include "OrderBook.h"
6. #include "RuleManager.h"
7. #include "TraderManager.h"
8. #include "Rules.h"
9. #include "Order.h"
10. #include "WallTimer.h"
11. #include "Trader.h"
12. #include "Seed.h"
14. //#include <vld.h>
16. #define DURATION 60000
17. #define NUMRT 12
18. #define NUMLRT 12
19. #define NUMPT 12
20. #define NUMMT 12
21. #define RUNS 1000
23. **class** Trader;
25. **double** RandomInRange(**int** low=0, **int** high=RAND\_MAX)
26. {
27. **double** result = rand()\*(high-low)/RAND\_MAX;
28. result += low;
29. **return** result;
30. }
32. **int** main(**int** argc, **char**\*\* argv)
33. {
34. **int** runs = 0;
35. **if** (argc >= 2)
36. runs = std::atoi(argv[1]);
37. **int** duration = 60000;
38. **if** (argc >= 3)
39. duration = std::atoi(argv[2]);
40. std::string name = "noname.csv";
41. **if** (argc >= 4)
42. {
43. name = std::string(argv[3]);
44. Seed::GetInstance()->SetName(name);
45. }
46. **int** baseCount = 1000;
47. **if** (argc >= 5)
48. baseCount = std::atoi(argv[4]);
49. **int** loglevel = 1;
50. **if** (argc >= 6)
51. loglevel = std::atoi(argv[5]);
52. **int** numrt = 12;
53. **if** (argc >= 7)
54. numrt = std::atoi(argv[6]);
55. **int** numlrt = 12;
56. **if** (argc >= 8)
57. numlrt = std::atoi(argv[7]);
58. **int** numpt = 12;
59. **if** (argc >= 9)
60. numpt = std::atoi(argv[8]);
61. **int** nummt = 12;
62. **if** (argc >= 10)
63. nummt = std::atoi(argv[9]);
64. **int** a = 2000;
65. **if** (argc >= 12)
66. a = std::atoi(argv[10]);
67. **int** b = 3000;
68. **if** (argc >= 12)
69. b = std::atoi(argv[11]);
70. **bool** waitAtFinish = **true**;
71. **if** (argc >= 13)
72. {
73. **if** (std::atoi(argv[12]) == 1)
74. waitAtFinish = **true**;
75. **else**
76. waitAtFinish = **false**;
77. }
79. //Get high frequency trader counts
80. **int** hfrt = 0;
81. **if** (argc >= 14)
82. hfrt = std::atoi(argv[13]);
84. **int** hflrt = 0;
85. **if** (argc >= 15)
86. hflrt = std::atoi(argv[14]);
88. **int** hfpt = 0;
89. **if** (argc >= 16)
90. hfpt = std::atoi(argv[15]);
92. **int** hfmt = 0;
93. **if** (argc >= 17)
94. hfmt = std::atoi(argv[16]);
96. //Get high frequency trader time range
97. **int** hftA = 1;
98. **if** (argc >= 18)
99. hftA = std::atoi(argv[17]);
101. **int** hftB = 10;
102. **if** (argc >= 19)
103. hftB = std::atoi(argv[18]);
105. std::ifstream file("Traders.cl");
106. **if** (file.is\_open())
107. {
108. std::string prog(std::istreambuf\_iterator<**char**>(file), (std::istreambuf\_iterator<**char**>()));
109. std::string kernelText = prog;
110. OpenClController::GetInstance()->SetText(kernelText);
111. }
112. file.close();
114. std::cout << "Initialised with: Runs[" << runs << "] Duration [" << duration << "] Filename[" << name << "] OutputInc[" << baseCount
115. << "] LogLevel[" << loglevel << "] NumRT[" << numrt << "] NumLRT[" << numlrt << "] NumPT["
116. << numpt << "] NumMt[" << nummt << "] TimeRange[" << a << "-" << b << "]"
117. << " WaitAtFinish[" << waitAtFinish << "] NumHFRT[" << hfrt << "] NumHFLRT[" << hflrt << "] NumHFPT[" << hfpt << "] NumHFMT[" << hfmt
118. << "] HF TimeRange[" << hftA << "-" << hftB << "]" << std::endl;
120. Logger::GetInstance()->SetLevel(loglevel);
121. std::ofstream output;
122. output.open("C:\\Outputs\\" + name);
123. output << "TimeTaken,SizeInMemory,AveMatchTime,MaxMatchTime,AveOclTime,MaxOclTime,AveTraderTime,MaxTraderTime,MinPrice,AvePrice,MaxPrice,AveSpread,TradesPerSecond,VolatilityPerMin,MinReturn1ms,AveReturn1ms,MaxReturn1ms,MinReturn1s,AveReturn1s,MaxReturn1s,Return1m,MinRTProfit,AveRTProfit,MaxRTProfit,MinLRTProfit,AveLRTProfit,MaxLRTProfit,MinPTProfit,AvePTProfit,MaxPTProfit,MinMTProfit,AveMTProfit,MaxMTProfit,AveProfit,MinTraderT,AveTraderT,MaxTraderT,Seed" << std::endl;
124. **for** (**int** i=0; i < runs; i++)
125. {
126. std::cout << name << std::endl;
127. std::cout << "Run " << (i+1) << " of " << runs << std::endl;
128. Seed::GetInstance()->Update();
129. srand(Seed::GetInstance()->GetSeed());
130. std::cout << "Seed: " << Seed::GetInstance()->GetSeed() << std::endl;
131. **int** time = 0;
132. OrderBook\* pBook = NULL;
133. TraderManager\* pTm = NULL;
134. WallTimer timer, timer1;
136. Stock GOOG(0, "GOOG", 100);
137. pTm = **new** TraderManager(**true**);
138. //Init normal traders
139. **for** (**int** j=0; j < numrt; j++)
140. pTm->addTrader(**new** Trader(&GOOG, 1000000, 1000000, std::floor(RandomInRange(a,b)), RANDOM\_TRADER));
141. **for** (**int** j=0; j < numlrt; j++)
142. pTm->addTrader(**new** Trader(&GOOG, 10000000, 10000000, std::floor(RandomInRange(a,b)), LARGE\_RANDOM\_TRADER));
143. **for** (**int** j=0; j < numpt; j++)
144. pTm->addTrader(**new** Trader(&GOOG, 100000, 100000, std::floor(RandomInRange(a,b)), POSITION\_TRADER));
145. **for** (**int** j=0; j < nummt; j++)
146. pTm->addTrader(**new** Trader(&GOOG, 10000, 10000, std::floor(RandomInRange(a,b)), MOMENTUM\_TRADER));
148. //Init high frequency traders
149. **for** (**int** j=0; j < hfrt; j++)
150. pTm->addTrader(**new** Trader(&GOOG, 1000000, 1000000, std::floor(RandomInRange(hftA,hftB)), RANDOM\_TRADER));
151. **for** (**int** j=0; j < hflrt; j++)
152. pTm->addTrader(**new** Trader(&GOOG, 10000000, 10000000, std::floor(RandomInRange(hftA,hftB)), LARGE\_RANDOM\_TRADER));
153. **for** (**int** j=0; j < hfpt; j++)
154. pTm->addTrader(**new** Trader(&GOOG, 100000, 100000, std::floor(RandomInRange(hftA,hftB)), POSITION\_TRADER));
155. **for** (**int** j=0; j < hfmt; j++)
156. pTm->addTrader(**new** Trader(&GOOG, 10000, 10000, std::floor(RandomInRange(hftA,hftB)), MOMENTUM\_TRADER));
158. pBook = **new** OrderBook(&GOOG, pTm, **true**);
159. pBook->addRule(**new** MarketMarketRule());
160. pBook->addRule(**new** MarketLimitRule());
161. pBook->addRule(**new** LimitMarketRule());
162. pBook->addRule(**new** LimitLimitRule());
164. **int** count = baseCount;
165. timer.Start();
167. **while** (time < duration)
168. {
169. //Match orders
170. pBook->matchOrders();
172. //Process traders
173. pBook->processTraders();
175. //Increment time
176. pBook->update();
177. time = pBook->getTime();
179. //Output progress to console
180. **if** (count == 1)
181. {
182. std::cout << "\r" << time << "/" << duration << " - " << ((((**double**)time)\*100)/duration) << "%\t\t\t";
183. count = baseCount;
184. }
185. **else**
186. {
187. count--;
188. }
189. }
190. std::cout << std::endl;
191. **double** timeTaken = timer.GetCounter();
192. time = 0;
193. std::cout << "Took [" << timeTaken << "ms] to complete a simulation lasting [" << duration << "ms]" << std::endl;
194. std::cout << "Performance vs Realtime: " << duration/timeTaken << "x" << std::endl;
195. timer1.Start();
196. output << timeTaken << "," << pBook->GetBookSize() << "," << pBook->GetAveMatchTime() << "," << pBook->GetMaxMatchTime() << ","
197. << pBook->GetAveOclProcTime() << "," << pBook->GetMaxOclProcTime() << "," << pBook->GetAveTraderProcTime() << ","
198. << pBook->GetMaxTraderProcTime() << "," << pBook->GetMinPrice() << "," << pBook->GetAvePrice() << ","
199. << pBook->GetMaxPrice() << "," << pBook->GetAveSpread() << "," << pBook->GetTradesPerSecond() << ","
200. << pBook->GetVolatilityPerMin() << "," << pBook->GetMinReturns1ms() << "," << pBook->GetAveReturns1ms() << ","
201. << pBook->GetMaxReturns1ms() << "," << pBook->GetMinReturns1s() << "," << pBook->GetAveReturns1s() << ","
202. << pBook->GetMaxReturns1s() << "," << pBook->GetReturn1m() << "," << pBook->GetMinRTProfit() << ","
203. << pBook->GetAveRTProfit() << "," << pBook->GetMaxRTProfit() << "," << pBook->GetMinLRTProfit() << ","
204. << pBook->GetAveLRTProfit() << "," << pBook->GetMaxLRTProfit() << "," << pBook->GetMinPTProfit() << ","
205. << pBook->GetAvePTProfit() << "," << pBook->GetMaxPTProfit() << "," << pBook->GetMinMTProfit() << ","
206. << pBook->GetAveMTProfit() << "," << pBook->GetMaxMTProfit() << "," << pBook->GetAveProfit() << ","
207. << pBook->GetMinTraderProcessT() << "," << pBook->GetAveTraderProcessT() << "," << pBook->GetMaxTraderProcessT()
208. << "," << Seed::GetInstance()->GetSeed() << std::endl;
209. **double** calcTime = timer1.GetCounter();
210. std::cout << "Took [" << calcTime << "ms] to complete post-simulation calculations" << std::endl;
211. std::cout << "Overall Time: [" << (timeTaken+calcTime) << "ms] - " << duration/(timeTaken+calcTime) << "x" << std::endl;
213. Logger::GetInstance()->NextRun();
215. **delete** pBook;
216. **delete** pTm;
217. pBook = NULL;
218. pTm = NULL;
219. }
220. output.close();
221. std::cout << "COMPLETE" << std::endl;
222. **if** (waitAtFinish)
223. std::getchar();
224. **return** 0;
225. }

### OrderQueue

#### OrderQueue.h

1. #pragma once
3. #include "OrderRequest.h"
5. **class** OrderQueue
6. {
7. **private**:
8. std::vector<OrderRequest> elements;
10. **public**:
11. OrderQueue(**void**){}
12. ~OrderQueue(**void**){}
14. **void** enqueue(OrderRequest request)
15. {
16. elements.push\_back(request);
17. }
19. OrderRequest dequeue()
20. {
21. OrderRequest t = elements.front();
22. elements.erase(elements.begin());
23. **return** t;
24. }
26. **bool** isEmpty()
27. {
28. **if** (elements.size() == 0)
29. **return** **true**;
30. **else**
31. **return** **false**;
32. }
33. };

### OrderRequest

#### OrderRequest.h

1. #pragma once
3. #include "Order.h"
4. #include "Utils.h"
6. **class** OrderRequest
7. {
8. **private**:
9. **bool** \_isInsertRequest;
10. Order \_order;
11. **public**:
12. OrderRequest(){};
13. OrderRequest(Order order, **bool** isInsertRequest);
14. ~OrderRequest(**void**);
16. **bool** isInsert();
17. **bool** isRemove();
19. Order getOrder();
21. std::string ToString();
22. };

#### OrderRequest.cpp

1. #include "StdAfx.h"
2. #include "OrderRequest.h"

5. OrderRequest::OrderRequest(Order order, **bool** isInserting)
6. {
7. \_order = order;
8. \_isInsertRequest = isInserting;
9. }
11. OrderRequest::~OrderRequest(**void**)
12. {}
14. **bool** OrderRequest::isInsert()
15. {
16. **return** \_isInsertRequest == **true**;
17. }
19. **bool** OrderRequest::isRemove()
20. {
21. **return** \_isInsertRequest == **false**;
22. }
24. Order OrderRequest::getOrder()
25. {
26. **return** \_order;
27. }
29. std::string OrderRequest::ToString()
30. {
31. std::string temp;
32. **if** (**this**->isInsert())
33. temp = "Insert ";
34. **else** **if** (**this**->isRemove())
35. temp = "Remove ";
37. **return** temp + \_order.toString();
38. }

### Seed

#### Seed.h

1. #pragma once
3. **class** Seed
4. {
5. **private**:
6. **long** seed;
7. std::string name;
8. **bool** nameSet;
10. **static** **bool** \_instanceFlag;
11. **static** Seed\* \_instance;
12. Seed();
14. **public**:
15. ~Seed();
16. **static** Seed\* GetInstance();
17. **long** GetSeed();
18. std::string GetName();
19. **void** SetName(std::string text);
20. **void** Update();
21. };

#### Seed.cpp

1. #include "stdafx.h"
2. #include "Seed.h"
4. Seed::Seed()
5. {
6. seed = 0;
7. name = "";
8. nameSet = **false**;
9. }
11. Seed::~Seed()
12. {
13. **if** (\_instanceFlag)
14. {
15. \_instanceFlag = **false**;
16. **delete** \_instance;
17. }
18. }
20. **bool** Seed::\_instanceFlag = **false**;
21. Seed\* Seed::\_instance = NULL;
23. Seed\* Seed::GetInstance()
24. {
25. **if** (!\_instanceFlag)
26. {
27. \_instanceFlag = **true**;
28. \_instance = **new** Seed();
29. **return** \_instance;
30. }
31. **else**
32. {
33. **return** \_instance;
34. }
35. }
37. **long** Seed::GetSeed()
38. {
39. **return** seed;
40. }
42. std::string Seed::GetName()
43. {
44. **if** (nameSet)
45. **return** name;
46. **else**
47. {
48. std::stringstream temp;
49. temp << seed;
50. **return** temp.str();
51. }
52. }
54. **void** Seed::SetName(std::string text)
55. {
56. std::stringstream line;
57. line << text;
58. std::getline(line, name, '.');
59. nameSet = **true**;
60. }
62. **void** Seed::Update()
63. {
64. seed = time(0);
65. }

### Stock

#### Stock.h

1. #pragma once
3. **class** Stock
4. {
5. **private**:
6. std::string \_symbol;
7. **double** \_lastPrice;
8. **int** \_stockId;
9. **public**:
10. Stock(**int** id=-1);
11. Stock(**int** id, std::string symbol);
12. Stock(**int** id, std::string symbol, **double** lastPrice);
13. ~Stock(**void**);
15. **bool** operator==(Stock stock);
17. //Getters and Setters
18. std::string getSymbol();
19. **void** setSymbol(std::string symbol);
20. std::string toString();
21. **double** getLastPrice();
22. **void** setLastPrice(**double** lastPrice);
23. **int** getStockId();
24. };

#### Stock.cpp

1. #include "StdAfx.h"
2. #include "Stock.h"
4. Stock::Stock(**int** id)
5. {
6. \_stockId = id;
7. \_symbol = "0";
8. }
10. Stock::Stock(**int** id, std::string symbol)
11. {
12. \_stockId = id;
13. \_symbol = symbol;
14. }
16. Stock::Stock(**int** id, std::string symbol, **double** lastPrice)
17. {
18. \_stockId = id;
19. \_symbol = symbol;
20. \_lastPrice = lastPrice;
21. }
23. Stock::~Stock(**void**)
24. {}
26. **bool** Stock::operator==(Stock stock)
27. {
28. **if** (stock.getLastPrice() == \_lastPrice && stock.getStockId() == \_stockId && stock.getSymbol() == \_symbol)
29. **return** **true**;
30. **else**
31. **return** **false**;
32. }
34. std::string Stock::getSymbol()
35. {
36. **return** \_symbol;
37. }
39. **void** Stock::setSymbol(std::string symbol)
40. {
41. \_symbol = symbol;
42. }
44. std::string Stock::toString()
45. {
46. **char** str[64];
48. **if** (\_symbol == "0")
49. {
50. sprintf\_s(str, "ID: %d", \_stockId);
51. std::string temp(str);
52. **return** temp;
53. }
54. **else**
55. {
56. sprintf\_s(str, "%d,%s", \_stockId, \_symbol.c\_str());
57. std::string temp(str);
58. **return** temp;
59. }
60. }
62. **double** Stock::getLastPrice()
63. {
64. **return** \_lastPrice;
65. }
67. **void** Stock::setLastPrice(**double** lastPrice)
68. {
69. \_lastPrice = lastPrice;
70. }
72. **int** Stock::getStockId()
73. {
74. **return** \_stockId;
75. }

### TBBlog

#### TBBLog.h

1. #pragma once
3. #include <tbb\tbb.h>
5. **class** TBBlog
6. {
7. **private**:
8. **const** std::vector<**double**>\* \_allPrices;
9. std::vector<**double**>\* \_result;
11. **public**:
12. TBBlog(**const** std::vector<**double**>& allPrices, std::vector<**double**>& prfv)
13. {
14. \_allPrices = &allPrices;
15. \_result = &prfv;
16. }
18. **void** operator() (**const** tbb::blocked\_range<**int**>& range)
19. **const**
20. {
21. **for** (unsigned i = range.begin(); i < range.end(); i++)
22. {
23. **double** returns = std::log(\_allPrices->at(i)/\_allPrices->at(i-1));
24. (\*\_result)[i-1] = returns;
25. }
26. }
27. };

### Tests

#### Tests.h

1. #pragma once
3. #include "Stock.h"
4. #include "Order.h"

7. **class** Tests
8. {
9. **public**:
10. Tests(**void**);
11. ~Tests(**void**);
13. **static** **bool** TestStock();
14. **static** **bool** TestOrder();
15. };

#### Tests.cpp

1. #include "StdAfx.h"
2. #include "Tests.h"

5. Tests::Tests(**void**)
6. {
7. }
9. Tests::~Tests(**void**)
10. {
11. }
13. **bool** Tests::TestStock()
14. {
15. **try**
16. {
17. Stock stock0(0);
18. Stock stock1(1, "AAA");
19. }
20. **catch** (std::exception exception)
21. {
22. std::cout << exception.what() << std::endl;
23. std::getchar();
24. **return** **false**;
25. }
27. Stock stock0(0);
28. Stock stock1(1, "AAA");
29. std::cout << "Successfully created stocks 0 and 1" << std::endl;
31. **try**
32. {
33. **const** std::string resultStr0 = "ID: 0";
34. **const** std::string resultStr1 = "ID: 1, Symbol: AAA";
35. **if** (stock0.toString() != resultStr0)
36. **throw** **new** std::exception("Strings do not match 0");
37. **if** (stock1.toString() != resultStr1)
38. **throw** **new** std::exception("Strings do not match 1");
39. }
40. **catch** (std::exception exception)
41. {
42. std::cout << exception.what() << std::endl;
43. std::getchar();
44. **return** **false**;
45. }
47. std::cout << "Successfully stringed stocks 0 and 1" << std::endl;
48. std::cout << stock0.toString() << "\t" << stock1.toString() << std::endl;
50. **return** **true**;;
51. }
53. **bool** Tests::TestOrder()
54. {
55. Stock stock0(0, "AAA");
56. Stock stock1(1, "BBB");
57. Stock stock2(2, "CCC");
59. **try**
60. {
61. Order order1(BUY, 100, 1.05, 99, **true**, &stock0, 0);
62. Order order2(SELL, 100, 1.10, 98, **true**, &stock0, 0);
63. }
64. **catch** (std::exception exception)
65. {
66. std::cout << exception.what() << std::endl;
67. std::getchar();
68. **return** **false**;
69. }
71. //Create orders necessary for testing methods: toString(), equals, compareBuys, and compareSells
72. Order order1(BUY, 100, 1.05, 99, **true**, &stock0, 0);
73. Order order2(BUY, 100, 1.05, 98, **true**, &stock0, 0);
75. //Check toString and if order number increments
76. **try**
77. {
78. **const** std::string resultStr1("Price: 99999.00, Volume: 100, Type: 1, Market Order: 1, Participant: 99, Stock:[ID: 0, Symbol: AAA], Time: 0, OrderNumber: 2");
79. **const** std::string resultStr2("Price: 99999.00, Volume: 100, Type: 1, Market Order: 1, Participant: 98, Stock:[ID: 0, Symbol: AAA], Time: 0, OrderNumber: 3");
80. **if** (order1.toString() != resultStr1)
81. {
82. std::cout << resultStr1 << std::endl;
83. std::cout << order1.toString() << std::endl;
84. **throw** **new** std::exception("Strings did not match");
85. }
86. **if** (order2.toString() != resultStr2)
87. {
88. std::cout << resultStr2 << std::endl;
89. std::cout << order2.toString() << std::endl;
90. **throw** **new** std::exception("Strings did not match");
91. }
92. }
93. **catch** (std::exception exception)
94. {
95. std::cout << exception.what() << std::endl;
96. std::getchar();
97. **return** **false**;
98. }
100. std::cout << "Order.toString() PASSES\nOrder number incrementation PASSES" << std::endl;
102. //Check equals
103. **if** (order1.equals(&order1) != **true**)
104. {
105. std::cout << "equals failed" << std::endl;
106. }
108. **if** (order1.equals(&order2) == **true**)
109. {
110. std::cout << "equals failed" << std::endl;
111. }
113. std::cout << "equals PASSES" << std::endl;
115. //Check compareBuys
116. order1 = Order(BUY, 100, 1.00, 99, **true**, &stock0, 0);
117. order2 = Order(BUY, 50, 1.00, 98, **true**, &stock0, 0);
118. Order order6(BUY, 200, 1.00, 94, **false**, &stock0, 0);
119. Order order3(BUY, 70, 1.00, 97, **false**, &stock0, 0);
120. Order order4(BUY, 20, 1.05, 96, **false**, &stock0, 0);
121. Order order5(BUY, 150, 0.95, 95, **false**, &stock0, 0);
123. std::list<Order> buyOrders;
124. buyOrders.push\_back(order1);
125. buyOrders.push\_back(order2);
126. buyOrders.push\_back(order3);
127. buyOrders.push\_back(order4);
128. buyOrders.push\_back(order5);
129. buyOrders.push\_back(order6);
131. **for**(unsigned i=0; i < buyOrders.size(); i++)
132. {
133. std::list<Order>::iterator it = buyOrders.begin();
134. std::advance(it, i);
135. it->print();
136. }
138. buyOrders.sort(Order::compareBuys);
139. std::cout << std::endl;
141. **for**(unsigned i=0; i < buyOrders.size(); i++)
142. {
143. std::list<Order>::iterator it = buyOrders.begin();
144. std::advance(it, i);
145. it->print();
146. }
148. std::cout << "compareBuys, PASSES (by inspection)" << std::endl << std::endl;
150. //Check compareSells
151. order1 = Order(SELL, 100, 1.00, 99, **true**, &stock1, 0);
152. order2 = Order(SELL, 50, 1.00, 98, **false**, &stock1, 0);
153. order3 = Order(SELL, 75, 1.05, 97, **false**, &stock1, 0);
154. order4 = Order(SELL, 25, 1.05, 96, **true**, &stock1, 0);
155. order5 = Order(SELL, 200, 0.95, 95, **true**, &stock1, 0);
156. order6 = Order(SELL, 10, 0.95, 94, **false**, &stock1, 0);
158. std::list<Order> sellOrders;
160. sellOrders.push\_back(order1);
161. sellOrders.push\_back(order2);
162. sellOrders.push\_back(order3);
163. sellOrders.push\_back(order4);
164. sellOrders.push\_back(order5);
165. sellOrders.push\_back(order6);
167. **for**(unsigned i=0; i < sellOrders.size(); i++)
168. {
169. std::list<Order>::iterator it = sellOrders.begin();
170. std::advance(it, i);
171. it->print();
172. }
174. sellOrders.sort(Order::compareSells);
175. std::cout << std::endl;
177. **for**(unsigned i=0; i < sellOrders.size(); i++)
178. {
179. std::list<Order>::iterator it = sellOrders.begin();
180. std::advance(it, i);
181. it->print();
182. }
184. std::cout << "compareSells PASSES (by inspection)" << std::endl << std::endl;
186. **return** **true**;
187. }

### Trade

#### Trade.h

1. #pragma once
3. #include "Order.h"
5. **class** Trade
6. {
7. **private**:
8. **int** \_size;
9. **double** \_price;
10. Order \_buyOrder;
11. Order \_sellOrder;
12. **int** \_time;
13. **int** \_tradeId;
15. **static** **int** \_lastTradeId;
17. **protected**:
18. **int** getNextTradeId();
20. **public**:
21. Trade(Order order1, Order order2, **double** price, **int** size, **int** time);
22. //Trade(Order\* order1, Order\* order2, double price, int size, int time);
23. ~Trade(**void**);
25. std::string toString();
26. std::string toStringCSV();
27. **static** std::string toStringHeaderCSV();
28. **void** print();
30. **bool** equals(Trade\* trade);
32. //Getters and Setters
33. **int** getSize();
34. **double** getPrice();
35. Order getBuyOrder();
36. **const** Order getBuyOrderC();
37. Order getSellOrder();
38. **const** Order getSellOrderC();
39. **int** getTime();
40. **int** getTradeId();
41. };

#### Trade.cpp

1. #include "StdAfx.h"
2. #include "Trade.h"

5. **int** Trade::\_lastTradeId = -1;
7. **int** Trade::getNextTradeId()
8. {
9. \_lastTradeId++;
10. **return** \_lastTradeId;
11. }
13. Trade::Trade(Order order1, Order order2, **double** price, **int** size, **int** time)
14. {
15. **if** (order1.isBuy())
16. {
17. \_buyOrder = order1;
18. \_sellOrder = order2;
19. }
20. **else**
21. {
22. \_buyOrder = order2;
23. \_sellOrder = order1;
24. }
26. \_price = price;
27. \_size = size;
28. \_time = time;
29. \_tradeId = getNextTradeId();
30. }
32. Trade::~Trade(**void**)
33. {
34. }
36. std::string Trade::toString()
37. {
38. **char** str[256];
39. sprintf\_s(str, "%d---Trade ID: %d, Price: %.2f, Volume: %d between ID:[%d] and ID:[%d]. Order Ids: Buy[%d], Sell[%d]",
40. \_time, \_tradeId, \_price, \_size, \_buyOrder.getParticipant(), \_sellOrder.getParticipant(), \_buyOrder.getOrderNumber(), \_sellOrder.getOrderNumber());
41. std::string temp(str);
42. **return** temp;
43. }
45. std::string Trade::toStringCSV()
46. {
47. **char** str[256];
48. sprintf\_s(str, "%d,%d,%.2f,%d,%d,%d,%d,%d",
49. \_time, \_tradeId, \_price, \_size, \_buyOrder.getParticipant(), \_sellOrder.getParticipant(), \_buyOrder.getOrderNumber(), \_sellOrder.getOrderNumber());
50. **return** std::string(str);
51. }
53. std::string Trade::toStringHeaderCSV()
54. {
55. **return** "Time,TradeId,Price,Volume,BuyId,SellId,BuyOrderId,SellOrderId";
56. }
58. **void** Trade::print()
59. {
60. **char** str[64];
61. sprintf\_s(str, "Trade ID: %d, Price: %.2f, Volume: %d", \_tradeId, \_price, \_size);
62. std::cout << str << std::endl;
63. }
65. **bool** Trade::equals(Trade\* trade)
66. {
67. **return** **this**->getTradeId() == trade->getTradeId();
68. }
70. #pragma region Getters
72. **int** Trade::getSize()
73. {
74. **return** \_size;
75. }
77. **double** Trade::getPrice()
78. {
79. **return** \_price;
80. }
82. Order Trade::getBuyOrder()
83. {
84. **return** \_buyOrder;
85. }
87. **const** Order Trade::getBuyOrderC()
88. {
89. **return** \_buyOrder;
90. }
92. Order Trade::getSellOrder()
93. {
94. **return** \_sellOrder;
95. }
97. **const** Order Trade::getSellOrderC()
98. {
99. **return** \_sellOrder;
100. }
102. **int** Trade::getTime()
103. {
104. **return** \_time;
105. }
107. **int** Trade::getTradeId()
108. {
109. **return** \_tradeId;
110. }
112. #pragma endregion

### TraderManager

#### TraderManager.h

1. #pragma once
3. #include "Trader.h"
4. #include "OpenClController.h"
5. #include "OpenClDevice.h"
6. #include "OpenClStructs.h"

9. **class** Trader;
10. **enum** TraderType;
12. **class** TraderManager
13. {
14. **private**:
15. std::vector<Trader\*> \_randomTraders;
16. std::vector<Trader\*> \_largeRandomTraders;
17. std::vector<Trader\*> \_positionTraders;
18. std::vector<Trader\*> \_momentumTraders;
20. std::vector<Trader\*> \_allTraders;
22. TraderCLArray \_tradersBuffer;
23. OpenClController\* ocl;
24. **int** \_totalTraders;
25. **bool** \_firstTime;
26. **double** \_procTime;
27. **int** \_minTraderProcT;
28. **int** \_currentT, \_lastT;
30. **bool** \_profiling;
32. **void** WriteBuffers(std::string symbol);
33. **void** ReadBuffers(OrderBook\* book);
34. Trader\* matchTCLtoTrader(TraderCL tcl, TraderType type);
36. **static** **const** std::string logName;
38. **public**:
39. TraderManager(**bool** profiling = **false**);
40. ~TraderManager();
42. **void** addTrader(Trader\* trader);
43. **void** removeTrader(Trader\* trade);
44. **void** Init();
46. **void** print(std::string symbol);
48. **void** notify(**int** time);
49. **void** notify(Trade\* trade);
50. **void** process(OrderBook\* book);
51. **double** getProcessTime();
52. **bool** checkReady();
54. **size\_t** SizeOf();
56. std::vector<Trader\*> GetAllTraders();
57. **double** GetMinRTProfit();
58. **double** GetAveRTProfit();
59. **double** GetMaxRTProfit();
60. **double** GetMinLRTProfit();
61. **double** GetAveLRTProfit();
62. **double** GetMaxLRTProfit();
63. **double** GetMinPTProfit();
64. **double** GetAvePTProfit();
65. **double** GetMaxPTProfit();
66. **double** GetMinMTProfit();
67. **double** GetAveMTProfit();
68. **double** GetMaxMTProfit();
69. **double** GetAveProfit();
70. **int** GetMinTraderProcessT();
71. **int** GetAveTraderProcessT();
72. **int** GetMaxTraderProcessT();
73. };

#### TraderManager.cpp

1. #include "stdafx.h"
2. #include "TraderManager.h"
4. **const** std::string TraderManager::logName = "TraderManager";
6. TraderManager::TraderManager(**bool** profiling)
7. {
8. \_totalTraders = 0;
9. \_firstTime = **true**;
10. \_minTraderProcT = INT\_MAX;
11. \_currentT = 0;
12. \_lastT = 0;
13. \_procTime = 0;
14. \_profiling = profiling;
15. }
17. TraderManager::~TraderManager()
18. {
19. **for** (**int** i=0; i < \_allTraders.size(); i++)
20. **delete** \_allTraders[i];
21. \_allTraders.clear();
22. \_randomTraders.clear();
23. \_largeRandomTraders.clear();
24. \_positionTraders.clear();
25. \_momentumTraders.clear();
26. }
28. **void** TraderManager::addTrader(Trader\* trader)
29. {
30. Logger::GetInstance()->Debug(logName, "Adding Trader...");
31. **switch** (trader->GetType())
32. {
33. **case** RANDOM\_TRADER:
34. \_randomTraders.push\_back(trader);
35. \_tradersBuffer.randomNumber++;
36. Logger::GetInstance()->Info(logName, Utils::Merge("Added Random Trader:", trader->toString()));
37. **break**;
38. **case** LARGE\_RANDOM\_TRADER:
39. \_largeRandomTraders.push\_back(trader);
40. \_tradersBuffer.largeRandomNumber++;
41. Logger::GetInstance()->Info(logName, Utils::Merge("Added Large Random Trader:", trader->toString()));
42. **break**;
43. **case** POSITION\_TRADER:
44. \_positionTraders.push\_back(trader);
45. \_tradersBuffer.positionNumber++;
46. Logger::GetInstance()->Info(logName, Utils::Merge("Added Position Trader:", trader->toString()));
47. **break**;
48. **case** MOMENTUM\_TRADER:
49. \_momentumTraders.push\_back(trader);
50. \_tradersBuffer.momentumNumber++;
51. Logger::GetInstance()->Info(logName, Utils::Merge("Added Momentum trader:", trader->toString()));
52. **break**;
53. **default**:
54. Logger::GetInstance()->Warn(logName, Utils::Merge("Unable to add trader of unrecognised type:", trader->toString()));
55. **break**;
56. }
57. \_allTraders.push\_back(trader);
58. Logger::GetInstance()->Debug(logName, "DONE");
60. Logger::GetInstance()->Debug(logName, "Determining minimum trader sim process time");
61. \_minTraderProcT = min(\_minTraderProcT, trader->GetProcessT());
62. Logger::GetInstance()->Info(logName, Utils::ItoS(\_minTraderProcT));
63. Logger::GetInstance()->Debug(logName, "DONE");
64. \_tradersBuffer.number++;
65. \_totalTraders++;
66. Logger::GetInstance()->Info(logName, Utils::Merge("Total Number of traders:",Utils::ItoS(\_totalTraders)));
67. }
69. **template**<**class** T>
70. **void** removeFromVector(std::vector<T>& vector, T item)
71. {
72. **for** (**int** i=0; i < vector.size(); i++)
73. {
74. **if** (vector[i] == item)
75. {
76. vector.erase(vector.begin() + i);
77. **break**;
78. }
79. }
80. }
82. **void** TraderManager::removeTrader(Trader\* trader)
83. {
84. Logger::GetInstance()->Debug(logName, "Removing trader...");
85. **switch** (trader->GetType())
86. {
87. **case** RANDOM\_TRADER:
88. removeFromVector(\_randomTraders, trader);
89. \_tradersBuffer.randomNumber--;
90. Logger::GetInstance()->Info(logName, Utils::Merge("Removed Random Trader:", trader->toString()));
91. **break**;
92. **case** LARGE\_RANDOM\_TRADER:
93. removeFromVector(\_largeRandomTraders, trader);
94. \_tradersBuffer.largeRandomNumber--;
95. Logger::GetInstance()->Info(logName, Utils::Merge("Removed Large Random Trader:", trader->toString()));
96. **break**;
97. **case** POSITION\_TRADER:
98. removeFromVector(\_positionTraders, trader);
99. \_tradersBuffer.positionNumber--;
100. Logger::GetInstance()->Info(logName, Utils::Merge("Removed Position Trader:", trader->toString()));
101. **break**;
102. **case** MOMENTUM\_TRADER:
103. removeFromVector(\_momentumTraders, trader);
104. \_tradersBuffer.momentumNumber--;
105. Logger::GetInstance()->Info(logName, Utils::Merge("Removed Momentum Trader:", trader->toString()));
106. **break**;
107. **default**:
108. Logger::GetInstance()->Warn(logName, Utils::Merge("Unable to remove trader of unrecognised type:", trader->toString()));
109. **break**;
110. }
112. removeFromVector(\_allTraders, trader);
114. \_totalTraders--;
115. Logger::GetInstance()->Info(logName, Utils::Merge("Total Number of traders:",Utils::ItoS(\_totalTraders)));
116. }
118. **void** TraderManager::Init()
119. {
120. **if** (\_firstTime)
121. {
122. \_tradersBuffer.traders = **new** TraderCL[\_totalTraders];
123. **try**
124. {
125. OpenClController::GetInstance()->SetupFirstTime(\_tradersBuffer, "ProcessTraders", \_randomTraders.size(), \_largeRandomTraders.size(), \_positionTraders.size(), \_momentumTraders.size(), \_profiling);
126. }
127. **catch** (...)
128. {
129. std::stringstream temp1; temp1 << "Failed in Init:SetupFirstTime - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
130. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
131. Logger::GetInstance()->Error(logName, temp);
132. **throw** **new** std::exception(temp.c\_str());
133. }
134. \_firstTime = **false**;
135. }
136. }
138. **void** TraderManager::print(std::string symbol)
139. {
140. std::stringstream stream;
142. stream << "------------Traders------------\n";
144. stream << "Random Traders\n";
146. **for** (**int** i=0; i < \_allTraders.size(); i++)
147. {
148. stream << \_allTraders[i]->toString(symbol) << std::endl;
149. }
151. std::cout << stream.str() << std::endl;
152. }
154. **void** TraderManager::notify(**int** time)
155. {
156. Logger::GetInstance()->Debug(logName, Utils::Merge("Notifying traders of time:", Utils::ItoS(time)));
157. \_currentT = time;
159. **for** (**int** i=0; i < \_allTraders.size(); i++)
160. {
161. \_allTraders[i]->update(time);
162. }
163. Logger::GetInstance()->Debug(logName, "DONE");
164. }
166. **void** TraderManager::notify(Trade\* trade)
167. {
168. **if** (trade == NULL)
169. **throw** **new** std::exception("Null Trade");
171. **int** buyPid = trade->getBuyOrder().getParticipant();
172. **int** sellPid = trade->getSellOrder().getParticipant();
174. Logger::GetInstance()->Debug(logName, "Notifying Traders of trade...");
175. Logger::GetInstance()->Info(logName, Utils::Merge("Notifying traders of trade:", trade->toString()));
177. **for** (**int** i=0; i < \_allTraders.size(); i++)
178. {
179. **if** (\_allTraders[i]->GetId() == buyPid || \_allTraders[i]->GetId() == sellPid)
180. {
181. \_allTraders[i]->notify(trade);
182. Logger::GetInstance()->Debug(logName, Utils::Merge(Utils::Merge("Notifying Trader:", \_allTraders[i]->toString()), Utils::Merge("of Trade:", trade->toString())));
183. }
184. }
185. Logger::GetInstance()->Debug(logName, "DONE");
186. }
188. **void** TraderManager::process(OrderBook\* book)
189. {
190. Logger::GetInstance()->Debug(logName, "Processing order book");
191. **if** (checkReady())
192. {
193. Logger::GetInstance()->Debug(logName, "Writing Buffers for traders");
194. WriteBuffers(book->getStock()->getSymbol());
195. Logger::GetInstance()->Debug(logName, "DONE");
197. Logger::GetInstance()->Debug(logName, "Setting up Market Data");
198. MarketDataCL data(book->getBuyVolume(), book->getSellVolume(), book->getLastPricesPArray(), book->getLastPrices().size());
199. Logger::GetInstance()->Debug(logName, "DONE");
201. Logger::GetInstance()->Debug(logName, "Refreshing Buffers");
202. OpenClController::GetInstance()->RefreshBuffers(\_tradersBuffer, data);
203. Logger::GetInstance()->Debug(logName, "DONE");
205. **try**
206. {
207. Logger::GetInstance()->Debug(logName, "Updating Buffers and Args");
208. OpenClController::GetInstance()->UpdateBuffersAndArgs();
209. Logger::GetInstance()->Debug(logName, "DONE");
210. }
211. **catch** (...)
212. {
213. std::stringstream temp1; temp1 << "Failed in process:UpdateBuffersAndArgs - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
214. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
215. Logger::GetInstance()->Error(logName, temp);
216. **throw** **new** std::exception(temp.c\_str());
217. }
219. **try**
220. {
221. Logger::GetInstance()->Debug(logName, "Running");
222. \_procTime = OpenClController::GetInstance()->Run(cl::NDRange(\_totalTraders), cl::NDRange(1));
223. Logger::GetInstance()->Debug(logName, "DONE");
224. }
225. **catch** (...)
226. {
227. std::stringstream temp1; temp1 << "Failed in process:Run - " << \_\_FILE\_\_ << " (" << \_\_LINE\_\_ << ")";
228. std::string temp = Utils::MergeException(temp1.str(), Utils::ResurrectException());
229. Logger::GetInstance()->Error(logName, temp);
230. **throw** **new** std::exception(temp.c\_str());
231. }
233. Logger::GetInstance()->Debug(logName, "Reading Buffers back");
234. ReadBuffers(book);
235. Logger::GetInstance()->Debug(logName, "DONE");
236. \_lastT = \_currentT;
237. Logger::GetInstance()->Info(logName, Utils::Merge("Last process time:", Utils::ItoS(\_lastT)));
238. }
239. }
241. //Private
243. **void** TraderManager::WriteBuffers(std::string symbol)
244. {
245. Logger::GetInstance()->Debug(logName, "Writing Random Traders Buffers");
246. **for** (**int** i=0; i < \_allTraders.size(); i++)
247. {
248. \_tradersBuffer.traders[i] = \_allTraders[i]->getTraderCL(symbol);
249. }
250. Logger::GetInstance()->Debug(logName, "DONE");
251. }
253. **void** TraderManager::ReadBuffers(OrderBook\* book)
254. {
255. Logger::GetInstance()->Debug(logName, "Updating local traders from buffers");
256. **for** (**int** i=0; i < \_tradersBuffer.number; i++)
257. {
258. assert(\_tradersBuffer.traders[i].id == \_allTraders[i]->GetId());
259. \_allTraders[i]->processTraderCL(\_tradersBuffer.traders[i], book);
260. }
261. Logger::GetInstance()->Debug(logName, "DONE");
262. }
264. Trader\* TraderManager::matchTCLtoTrader(TraderCL tcl, TraderType type)
265. {
266. **switch** (type)
267. {
268. **case** RANDOM\_TRADER:
269. **for** (**int** i=0; i < \_randomTraders.size(); i++)
270. {
271. **if** (tcl.id == \_randomTraders[i]->GetId() && tcl.type == \_randomTraders[i]->GetType())
272. {
273. **return** \_randomTraders[i];
274. }
275. }
276. **return** NULL;
277. **break**;
278. **case** LARGE\_RANDOM\_TRADER:
279. **for** (**int** i=0; i < \_largeRandomTraders.size(); i++)
280. {
281. **if** (tcl.id == \_largeRandomTraders[i]->GetId() && tcl.type == \_largeRandomTraders[i]->GetType())
282. {
283. **return** \_largeRandomTraders[i];
284. }
285. }
286. **return** NULL;
287. **break**;
288. **case** POSITION\_TRADER:
289. **for** (**int** i=0; i < \_positionTraders.size(); i++)
290. {
291. **if** (tcl.id == \_positionTraders[i]->GetId() && tcl.type == \_positionTraders[i]->GetType())
292. {
293. **return** \_positionTraders[i];
294. }
295. }
296. **return** NULL;
297. **break**;
298. **case** MOMENTUM\_TRADER:
299. **for** (**int** i=0; i < \_momentumTraders.size(); i++)
300. {
301. **if** (tcl.id == \_momentumTraders[i]->GetId() && tcl.type == \_momentumTraders[i]->GetType())
302. {
303. **return** \_momentumTraders[i];
304. }
305. }
306. **return** NULL;
307. **break**;
308. **default**:
309. **return** NULL;
310. **break**;
311. }
312. }
314. **double** TraderManager::getProcessTime()
315. {
316. **return** \_procTime;
317. }
319. **bool** TraderManager::checkReady()
320. {
321. Logger::GetInstance()->Debug(logName, "Checking if traders are ready");
322. **for** (**int** i=0; i < \_allTraders.size(); i++)
323. {
324. **if** (\_allTraders[i]->ready())
325. {
326. Logger::GetInstance()->Info(logName, Utils::Merge("Trader ready:", \_allTraders[i]->toString()));
327. Logger::GetInstance()->Debug(logName, "DONE");
328. **return** **true**;
329. }
330. }
331. Logger::GetInstance()->Debug(logName, "DONE");
333. **return** **false**;
334. }
336. **size\_t** TraderManager::SizeOf()
337. {
338. **size\_t** result = 0;
339. result += **sizeof**(TraderManager);
340. **for** (**int** i=0; i < \_allTraders.size(); i++)
341. {
342. result += \_allTraders[i]->SizeOf();
343. }
344. **return** result;
345. }
347. std::vector<Trader\*> TraderManager::GetAllTraders()
348. {
349. **return** \_allTraders;
350. }
352. **double** TraderManager::GetMinRTProfit()
353. {
354. **if** (\_randomTraders.size() > 0)
355. {
356. **double** minVal = DBL\_MAX;
357. **for** (**int** i=0; i < \_randomTraders.size(); i++)
358. {
359. minVal = min(minVal, \_randomTraders[i]->GetProfit());
360. }
361. **return** minVal;
362. }
363. **else**
364. **return** 0;
365. }
367. **double** TraderManager::GetAveRTProfit()
368. {
369. **if** (\_randomTraders.size() > 0)
370. {
371. **double** acc = 0;
372. **for** (**int** i=0; i < \_randomTraders.size(); i++)
373. {
374. acc += \_randomTraders[i]->GetProfit();
375. }
376. acc /= \_randomTraders.size();
377. **return** acc;
378. }
379. **else**
380. **return** 0;
381. }
383. **double** TraderManager::GetMaxRTProfit()
384. {
385. **if** (\_randomTraders.size() > 0)
386. {
387. **double** maxVal = -DBL\_MAX;
388. **for** (**int** i=0; i < \_randomTraders.size(); i++)
389. {
390. maxVal = max(maxVal, \_randomTraders[i]->GetProfit());
391. }
392. **return** maxVal;
393. }
394. **else**
395. **return** 0;
396. }
398. **double** TraderManager::GetMinLRTProfit()
399. {
400. **if** (\_largeRandomTraders.size() > 0)
401. {
402. **double** minVal = DBL\_MAX;
403. **for** (**int** i=0; i < \_largeRandomTraders.size(); i++)
404. {
405. minVal = min(minVal, \_largeRandomTraders[i]->GetProfit());
406. }
407. **return** minVal;
408. }
409. **else**
410. **return** 0;
411. }
413. **double** TraderManager::GetAveLRTProfit()
414. {
415. **if** (\_largeRandomTraders.size() > 0)
416. {
417. **double** acc = 0;
418. **for** (**int** i=0; i < \_largeRandomTraders.size(); i++)
419. {
420. acc += \_largeRandomTraders[i]->GetProfit();
421. }
422. acc /= \_largeRandomTraders.size();
423. **return** acc;
424. }
425. **else**
426. **return** 0;
427. }
429. **double** TraderManager::GetMaxLRTProfit()
430. {
431. **if** (\_largeRandomTraders.size() > 0)
432. {
433. **double** maxVal = -DBL\_MAX;
434. **for** (**int** i=0; i < \_largeRandomTraders.size(); i++)
435. {
436. maxVal = max(maxVal, \_largeRandomTraders[i]->GetProfit());
437. }
438. **return** maxVal;
439. }
440. **else**
441. **return** 0;
442. }
444. **double** TraderManager::GetMinPTProfit()
445. {
446. **if** (\_positionTraders.size() > 0)
447. {
448. **double** minVal = DBL\_MAX;
449. **for** (**int** i=0; i < \_positionTraders.size(); i++)
450. {
451. minVal = min(minVal, \_positionTraders[i]->GetProfit());
452. }
453. **return** minVal;
454. }
455. **else**
456. **return** 0;
457. }
459. **double** TraderManager::GetAvePTProfit()
460. {
461. **if** (\_positionTraders.size() > 0)
462. {
463. **double** acc = 0;
464. **for** (**int** i=0; i < \_positionTraders.size(); i++)
465. {
466. acc += \_positionTraders[i]->GetProfit();
467. }
468. acc /= \_positionTraders.size();
469. **return** acc;
470. }
471. **else**
472. **return** 0;
473. }
475. **double** TraderManager::GetMaxPTProfit()
476. {
477. **if** (\_positionTraders.size() > 0)
478. {
479. **double** maxVal = -DBL\_MAX;
480. **for** (**int** i=0; i < \_positionTraders.size(); i++)
481. {
482. maxVal = max(maxVal, \_positionTraders[i]->GetProfit());
483. }
484. **return** maxVal;
485. }
486. **else**
487. **return** 0;
488. }
490. **double** TraderManager::GetMinMTProfit()
491. {
492. **if** (\_momentumTraders.size() > 0)
493. {
494. **double** minVal = DBL\_MAX;
495. **for** (**int** i=0; i < \_momentumTraders.size(); i++)
496. {
497. minVal = min(minVal, \_momentumTraders[i]->GetProfit());
498. }
499. **return** minVal;
500. }
501. **else**
502. **return** 0;
503. }
505. **double** TraderManager::GetAveMTProfit()
506. {
507. **if** (\_momentumTraders.size() > 0)
508. {
509. **double** acc = 0;
510. **for** (**int** i=0; i < \_momentumTraders.size(); i++)
511. {
512. acc += \_momentumTraders[i]->GetProfit();
513. }
514. acc /= \_momentumTraders.size();
515. **return** acc;
516. }
517. **else**
518. **return** 0;
519. }
521. **double** TraderManager::GetMaxMTProfit()
522. {
523. **if** (\_momentumTraders.size() > 0)
524. {
525. **double** maxVal = -DBL\_MAX;
526. **for** (**int** i=0; i < \_momentumTraders.size(); i++)
527. {
528. maxVal = max(maxVal, \_momentumTraders[i]->GetProfit());
529. }
530. **return** maxVal;
531. }
532. **else**
533. **return** 0;
534. }
536. **double** TraderManager::GetAveProfit()
537. {
538. **if** (\_allTraders.size() > 0)
539. {
540. **double** acc = 0;
541. **for** (**int** i=0; i < \_allTraders.size(); i++)
542. {
543. acc += \_allTraders[i]->GetProfit();
544. }
545. acc /= \_allTraders.size();
546. **return** acc;
547. }
548. **else**
549. **return** -1;
550. }
552. **int** TraderManager::GetMinTraderProcessT()
553. {
554. **if** (\_allTraders.size() > 0)
555. {
556. **int** val = 9999999999;
557. **for** (**int** i=0; i < \_allTraders.size(); i++)
558. {
559. **int** temp = \_allTraders[i]->GetProcessT();
560. val = min(val, temp);
561. }
562. **return** val;
563. }
564. **else**
565. **return** -1;
566. }
568. **int** TraderManager::GetAveTraderProcessT()
569. {
570. **if** (\_allTraders.size() > 0)
571. {
572. **int** acc = 0;
573. **for** (**int** i=0; i < \_allTraders.size(); i++)
574. {
575. acc += \_allTraders[i]->GetProcessT();
576. }
577. acc /= \_allTraders.size();
578. **return** acc;
579. }
580. **else**
581. **return** -1;
582. }
584. **int** TraderManager::GetMaxTraderProcessT()
585. {
586. **if** (\_allTraders.size() > 0)
587. {
588. **int** val = -DBL\_MAX;
589. **for** (**int** i=0; i < \_allTraders.size(); i++)
590. {
591. val = max(val, \_allTraders[i]->GetProcessT());
592. }
593. **return** val;
594. }
595. **else**
596. **return** -1;
597. }

### Utils

#### Utils.h

1. #pragma once
3. #include "stdafx.h"
4. #include "OpenClDevice.h"
6. **class** OpenClDevice;
8. **class** Utils
9. {
10. **public**:
11. **static** **double** Min(std::vector<**double**> vals)
12. {
13. **double** x = DBL\_MAX;
14. **for** (**int** i=0; i < vals.size(); i++)
15. x = min(x, vals[i]);
16. **return** x;
17. }
19. **static** **double** Max(std::vector<**double**> vals)
20. {
21. **double** x = -DBL\_MAX;
22. **for** (**int** i=0; i < vals.size(); i++)
23. x = max(x, vals[i]);
24. **return** x;
25. }
27. **static** **double** Mean(std::vector<**double**> vals)
28. {
29. **double** acc = 0;
30. **for** (**int** i=0; i < vals.size(); i++)
31. acc += vals[i];
32. acc /= vals.size();
33. **return** acc;
34. }
36. **static** **double** Stdev(std::vector<**double**> vals)
37. {
38. **double** mean = Utils::Mean(vals);
39. **double** acc = 0;
40. **for** (**int** i=0; i < vals.size(); i++)
41. acc += ((vals[i]-mean)\*(vals[i]-mean));
42. acc /= vals.size();
43. **return** std::sqrt(acc);
44. }
46. **static** std::string Merge(std::string s1, **const** **char**\* cc1)
47. {
48. std::stringstream temp;
49. temp << s1 << " " << cc1;
50. **return** temp.str();
51. }
53. **static** std::string Merge(std::string s1, std::string s2)
54. {
55. **return** s1 + " " + s2;
56. }
58. **static** std::string Merge(**const** **char**\* cc1, **const** **char**\* cc2)
59. {
60. std::stringstream temp;
61. temp << cc1 << " " << cc2;
62. **return** temp.str();
63. }
65. **static** std::string ItoS(**int** i)
66. {
67. std::stringstream temp;
68. temp << i;
69. **return** temp.str();
70. }
72. **static** std::string DtoS(**float** f)
73. {
74. std::stringstream temp;
75. temp << f;
76. **return** temp.str();
77. }
79. **static** std::string DtoS(**double** f)
80. {
81. std::stringstream temp;
82. temp << f;
83. **return** temp.str();
84. }
86. **static** std::wstring s2ws(**const** std::string& s)
87. {
88. **int** len;
89. **int** slength = (**int**)s.length() + 1;
90. len = MultiByteToWideChar(CP\_ACP, 0, s.c\_str(), slength, 0, 0);
91. **wchar\_t**\* buf = **new** **wchar\_t**[len];
92. MultiByteToWideChar(CP\_ACP, 0, s.c\_str(), slength, buf, len);
93. std::wstring r(buf);
94. **delete**[] buf;
95. **return** r;
96. }
98. **static** std::string MergeException(std::string location, std::string exceptionInfo)
99. {
100. std::stringstream temp;
101. temp << location << " - " << exceptionInfo << std::endl << "\t";
102. **return** temp.str();
103. }
105. **static** std::string clErr(cl\_int err)
106. {
107. **switch** (err) {
108. **case** CL\_SUCCESS:                            **return** "Success!";
109. **case** CL\_DEVICE\_NOT\_FOUND:                   **return** "Device not found.";
110. **case** CL\_DEVICE\_NOT\_AVAILABLE:               **return** "Device not available";
111. **case** CL\_COMPILER\_NOT\_AVAILABLE:             **return** "Compiler not available";
112. **case** CL\_MEM\_OBJECT\_ALLOCATION\_FAILURE:      **return** "Memory object allocation failure";
113. **case** CL\_OUT\_OF\_RESOURCES:                   **return** "Out of resources";
114. **case** CL\_OUT\_OF\_HOST\_MEMORY:                 **return** "Out of host memory";
115. **case** CL\_PROFILING\_INFO\_NOT\_AVAILABLE:       **return** "Profiling information not available";
116. **case** CL\_MEM\_COPY\_OVERLAP:                   **return** "Memory copy overlap";
117. **case** CL\_IMAGE\_FORMAT\_MISMATCH:              **return** "Image format mismatch";
118. **case** CL\_IMAGE\_FORMAT\_NOT\_SUPPORTED:         **return** "Image format not supported";
119. **case** CL\_BUILD\_PROGRAM\_FAILURE:              **return** "Program build failure";
120. **case** CL\_MAP\_FAILURE:                        **return** "Map failure";
121. **case** CL\_INVALID\_VALUE:                      **return** "Invalid value";
122. **case** CL\_INVALID\_DEVICE\_TYPE:                **return** "Invalid device type";
123. **case** CL\_INVALID\_PLATFORM:                   **return** "Invalid platform";
124. **case** CL\_INVALID\_DEVICE:                     **return** "Invalid device";
125. **case** CL\_INVALID\_CONTEXT:                    **return** "Invalid context";
126. **case** CL\_INVALID\_QUEUE\_PROPERTIES:           **return** "Invalid queue properties";
127. **case** CL\_INVALID\_COMMAND\_QUEUE:              **return** "Invalid command queue";
128. **case** CL\_INVALID\_HOST\_PTR:                   **return** "Invalid host pointer";
129. **case** CL\_INVALID\_MEM\_OBJECT:                 **return** "Invalid memory object";
130. **case** CL\_INVALID\_IMAGE\_FORMAT\_DESCRIPTOR:    **return** "Invalid image format descriptor";
131. **case** CL\_INVALID\_IMAGE\_SIZE:                 **return** "Invalid image size";
132. **case** CL\_INVALID\_SAMPLER:                    **return** "Invalid sampler";
133. **case** CL\_INVALID\_BINARY:                     **return** "Invalid binary";
134. **case** CL\_INVALID\_BUILD\_OPTIONS:              **return** "Invalid build options";
135. **case** CL\_INVALID\_PROGRAM:                    **return** "Invalid program";
136. **case** CL\_INVALID\_PROGRAM\_EXECUTABLE:         **return** "Invalid program executable";
137. **case** CL\_INVALID\_KERNEL\_NAME:                **return** "Invalid kernel name";
138. **case** CL\_INVALID\_KERNEL\_DEFINITION:          **return** "Invalid kernel definition";
139. **case** CL\_INVALID\_KERNEL:                     **return** "Invalid kernel";
140. **case** CL\_INVALID\_ARG\_INDEX:                  **return** "Invalid argument index";
141. **case** CL\_INVALID\_ARG\_VALUE:                  **return** "Invalid argument value";
142. **case** CL\_INVALID\_ARG\_SIZE:                   **return** "Invalid argument size";
143. **case** CL\_INVALID\_KERNEL\_ARGS:                **return** "Invalid kernel arguments";
144. **case** CL\_INVALID\_WORK\_DIMENSION:             **return** "Invalid work dimension";
145. **case** CL\_INVALID\_WORK\_GROUP\_SIZE:            **return** "Invalid work group size";
146. **case** CL\_INVALID\_WORK\_ITEM\_SIZE:             **return** "Invalid work item size";
147. **case** CL\_INVALID\_GLOBAL\_OFFSET:              **return** "Invalid global offset";
148. **case** CL\_INVALID\_EVENT\_WAIT\_LIST:            **return** "Invalid event wait list";
149. **case** CL\_INVALID\_EVENT:                      **return** "Invalid event";
150. **case** CL\_INVALID\_OPERATION:                  **return** "Invalid operation";
151. **case** CL\_INVALID\_GL\_OBJECT:                  **return** "Invalid OpenGL object";
152. **case** CL\_INVALID\_BUFFER\_SIZE:                **return** "Invalid buffer size";
153. **case** CL\_INVALID\_MIP\_LEVEL:                  **return** "Invalid mip-map level";
154. **default**: **return** "<Unknown error code>";
155. }
156. }
158. **static** std::string ResurrectException()
159. {
160. **try**
161. {
162. **throw**;
163. }
164. **catch** (cl::Error& error)
165. {
166. std::stringstream tempSS;
167. tempSS << error.what() << "(" << Utils::clErr(error.err()) << ")";
168. **return** tempSS.str();
169. }
170. **catch** (std::exception& exception)
171. {
172. std::stringstream tempSS;
173. tempSS << exception.what();
174. **return** tempSS.str();
175. }
176. **catch** (...)
177. {
178. **return** "Unknown Exception";
179. }
180. }
181. };

### WallTimer

#### WallTimer.h

1. #pragma once

4. **class** WallTimer
5. {
6. **private**:
7. **double** \_pcFrequency;
8. **\_\_int64** \_start;
9. **bool** \_running;
10. **public**:
11. WallTimer(**void**);
12. ~WallTimer(**void**);
14. **void** Start();
15. **double** GetCounter();
16. **bool** Running();
17. };

#### WallTimer.cpp

1. #include "StdAfx.h"
2. #include "WallTimer.h"

5. WallTimer::WallTimer(**void**)
6. {
7. \_pcFrequency = 0.0;
8. \_start = 0.0;
9. \_running = **false**;
10. }
12. WallTimer::~WallTimer(**void**)
13. {}
15. **void** WallTimer::Start()
16. {
17. LARGE\_INTEGER li;
18. **if** (!QueryPerformanceFrequency(&li))
19. **throw** **new** std::exception("Query Performance Frequency failed");
21. \_pcFrequency = **double**(li.QuadPart)/1000.0;
23. QueryPerformanceCounter(&li);
24. \_start = li.QuadPart;
25. \_running = **true**;
26. }
28. **double** WallTimer::GetCounter()
29. {
30. LARGE\_INTEGER li;
31. QueryPerformanceCounter(&li);
32. **return** **double**(li.QuadPart - \_start)/\_pcFrequency;
33. }
35. **bool** WallTimer::Running()
36. {
37. **return** \_running;
38. }

### Windows Headers

#### stdafx.h

1. // stdafx.h : include file for standard system include files,
2. // or project specific include files that are used frequently, but
3. // are changed infrequently
4. //
6. #pragma once
8. #include "targetver.h"
10. #include <stdio.h>
11. #include <tchar.h>
13. #include <map>
14. #include <sstream>
15. #include <iostream>
16. #include <fstream>
17. #include <time.h>
18. #include <vector>
19. #include <climits>
20. #include <string>
21. #include <list>
22. #include <random>
23. #include <Windows.h>
24. #include <exception>
25. #include <cassert>
26. #include <stdlib.h>
28. #define \_\_CL\_ENABLE\_EXCEPTIONS
29. #define \_\_NO\_STD\_VECTOR
30. #include <CL\cl.hpp>
32. #define LOGLEVEL 1
34. #include <tbb\tbb.h>


38. // TODO: reference additional headers your program requires here

#### targetver.h

1. #pragma once
3. // Including SDKDDKVer.h defines the highest available Windows platform.
5. // If you wish to build your application for a previous Windows platform, include WinSDKVer.h and
6. // set the \_WIN32\_WINNT macro to the platform you wish to support before including SDKDDKVer.h.
8. #include <SDKDDKVer.h>

## OpenCL Code

### Traders.cl

1. #pragma OPENCL EXTENSION cl\_khr\_byte\_addressable\_store : enable
2. #pragma OPENCL EXTENSION cl\_amd\_fp64 : enable
4. // Pre: a<M, b<M
5. // Post: r=(a+b) mod M
6. ulong MWC\_AddMod64(ulong a, ulong b, ulong M)
7. {
8. ulong v=a+b;
9. **if**( (v>=M) || (v<a) )
10. v=v-M;
11. **return** v;
12. }
14. // Pre: a<M,b<M
15. // Post: r=(a\*b) mod M
16. // This could be done more efficently, but it is portable, and should
17. // be easy to understand. It can be replaced with any of the better
18. // modular multiplication algorithms (for example if you know you have
19. // double precision available or something).
20. ulong MWC\_MulMod64(ulong a, ulong b, ulong M)
21. {
22. ulong r=0;
23. **while**(a!=0){
24. **if**(a&1)
25. r=MWC\_AddMod64(r,b,M);
26. b=MWC\_AddMod64(b,b,M);
27. a=a>>1;
28. }
29. **return** r;
30. }

33. // Pre: a<M, e>=0
34. // Post: r=(a^b) mod M
35. // This takes at most ~64^2 modular additions, so probably about 2^15 or so instructions on
36. // most architectures
37. ulong MWC\_PowMod64(ulong a, ulong e, ulong M)
38. {
39. ulong sqr=a, acc=1;
40. **while**(e!=0){
41. **if**(e&1)
42. acc=MWC\_MulMod64(acc,sqr,M);
43. sqr=MWC\_MulMod64(sqr,sqr,M);
44. e=e>>1;
45. }
46. **return** acc;
47. }
49. uint2 MWC\_SkipImpl\_Mod64(uint2 curr, ulong A, ulong M, ulong distance)
50. {
51. ulong m=MWC\_PowMod64(A, distance, M);
52. ulong x=curr.x\*(ulong)A+curr.y;
53. x=MWC\_MulMod64(x, m, M);
54. **return** (uint2)((uint)(x/A), (uint)(x%A));
55. }
57. uint2 MWC\_SeedImpl\_Mod64(ulong A, ulong M, uint vecSize, uint vecOffset, ulong streamBase, ulong streamGap)
58. {
59. // This is an arbitrary constant for starting LCG jumping from. I didn't
60. // want to start from 1, as then you end up with the two or three first values
61. // being a bit poor in ones - once you've decided that, one constant is as
62. // good as any another. There is no deep mathematical reason for it, I just
63. // generated a random number.
64. **enum**{ MWC\_BASEID = 4077358422479273989UL };
66. ulong dist=streamBase + (get\_global\_id(0)\*vecSize+vecOffset)\*streamGap;
67. ulong m=MWC\_PowMod64(A, dist, M);
69. ulong x=MWC\_MulMod64(MWC\_BASEID, m, M);
70. **return** (uint2)((uint)(x/A), (uint)(x%A));
71. }
73. //! Represents the state of a particular generator
74. **typedef** **struct**{ uint x; uint c; } mwc64x\_state\_t;
76. **enum**{ MWC64X\_A = 4294883355U };
77. **enum**{ MWC64X\_M = 18446383549859758079UL };
79. **void** MWC64X\_Step(mwc64x\_state\_t \*s)
80. {
81. uint X=s->x, C=s->c;
83. uint Xn=MWC64X\_A\*X+C;
84. uint carry=(uint)(Xn<C);             // The (Xn<C) will be zero or one for scalar
85. uint Cn=mad\_hi(MWC64X\_A,X,carry);
87. s->x=Xn;
88. s->c=Cn;
89. }
91. **void** MWC64X\_Skip(mwc64x\_state\_t \*s, ulong distance)
92. {
93. uint2 tmp=MWC\_SkipImpl\_Mod64((uint2)(s->x,s->c), MWC64X\_A, MWC64X\_M, distance);
94. s->x=tmp.x;
95. s->c=tmp.y;
96. }
98. **void** MWC64X\_SeedStreams(mwc64x\_state\_t \*s, ulong baseOffset, ulong perStreamOffset)
99. {
100. uint2 tmp=MWC\_SeedImpl\_Mod64(MWC64X\_A, MWC64X\_M, 1, 0, baseOffset, perStreamOffset);
101. s->x=tmp.x;
102. s->c=tmp.y;
103. }
105. //! Return a 32-bit integer in the range [0..2^32)
106. uint MWC64X\_NextUint(mwc64x\_state\_t \*s)
107. {
108. uint res=s->x ^ s->c;
109. MWC64X\_Step(s);
110. **return** res;
111. }

114. //CPL code here
115. //
116. //
117. ///////////////////////////////////////////////////////////////////////////
118. #define UINTMAX 4294967295
120. **typedef**
121. **struct**
122. {
123. **double** price;
124. **int** time;
125. } PastPrice;
127. **typedef**
128. **struct**
129. {
130. **int** buyVolume;
131. **int** sellVolume;
132. **double** lastPrice;
133. **int** numPastPrices;
134. } MarketDataSmallCL;
136. **typedef**
137. **struct**
138. {
139. **double** cashPos;
140. **double** cashPosWO;
141. **int** volPos;
142. **int** volPosWO;
143. **double** startCash;
144. **int** startVol;
145. **int** id;
146. **int** type;
147. **bool** isMarket;
148. **double** price;
149. **int** volume;
150. } TraderCL;

153. //Definitions can be set in C++
154. //#define RT\_BUYSELL\_THRESH 2
155. //#define RT\_MARKET\_THRESH 8
156. //#define RT\_SIZE 1000
157. //#define RT\_PRICE\_CHANGE 0.01
158. //#define RT\_PRICE\_SIZE 10
160. TraderCL Copy(TraderCL traderCL)
161. {
162. TraderCL tcl;
163. tcl.cashPos = traderCL.cashPos;
164. tcl.cashPosWO = traderCL.cashPosWO;
165. tcl.id = traderCL.id;
166. tcl.isMarket = traderCL.isMarket;
167. tcl.price = traderCL.price;
168. tcl.startCash = traderCL.startCash;
169. tcl.startVol = traderCL.startVol;
170. tcl.type = traderCL.type;
171. tcl.volPos = traderCL.volPos;
172. tcl.volPosWO = traderCL.volPosWO;
173. tcl.volume = traderCL.volume;
175. **return** tcl;
176. }
178. **void** RandomTrader(\_\_global TraderCL\* traderCL, **const** MarketDataSmallCL data, mwc64x\_state\_t\* rng)
179. {
180. uint roll = MWC64X\_NextUint(rng);
181. TraderCL tcl = \*traderCL;
183. //Create a buy order
184. **if** (roll > UINTMAX/RT\_BUYSELL\_THRESH)
185. {
186. **bool** isMarket = **false**;
188. **if** (MWC64X\_NextUint(rng) > UINTMAX/RT\_MARKET\_THRESH)
189. isMarket = **true**;
191. //Fill order
192. tcl.isMarket = isMarket;
193. tcl.volume = (**int**)floor((**double**)MWC64X\_NextUint(rng)\*RT\_SIZE/UINTMAX) + 1;
194. tcl.price = data.lastPrice - (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
196. }
197. **else**
198. {
199. **bool** isMarket = **false**;
201. **if** (MWC64X\_NextUint(rng) > UINTMAX/RT\_MARKET\_THRESH)
202. isMarket = **true**;
204. tcl.isMarket = isMarket;
205. tcl.volume = -(**int**)floor((**double**)MWC64X\_NextUint(rng)\*RT\_SIZE/UINTMAX) + 1;
206. tcl.price = data.lastPrice + (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
207. }
209. //Update trader with this info
210. tcl.cashPosWO -= tcl.volume\*tcl.price;
211. tcl.volPosWO += tcl.volume;
212. \*traderCL = tcl;
213. //return tcl;
214. }
216. **void** LargeRandomTrader(\_\_global TraderCL\* traderCL, **const** MarketDataSmallCL data, mwc64x\_state\_t\* rng)
217. {
218. TraderCL tcl = \*traderCL;
219. tcl.isMarket = **true**;
221. **if** (data.buyVolume > data.sellVolume)
222. {
223. **bool** isMarket = **false**;
224. **if** (MWC64X\_NextUint(rng) > UINTMAX/RT\_MARKET\_THRESH)
225. isMarket = **true**;
227. //Fill order
228. tcl.isMarket = isMarket;
229. tcl.volume = -data.buyVolume;
230. //tcl.price = data.lastPrice;
231. tcl.price = data.lastPrice + (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
232. }
233. **else** **if** (data.buyVolume < data.sellVolume)
234. {
235. **bool** isMarket = **false**;
236. **if** (MWC64X\_NextUint(rng) > UINTMAX/RT\_MARKET\_THRESH)
237. isMarket = **true**;
239. tcl.isMarket = isMarket;
240. tcl.volume = data.sellVolume;
241. //tcl.price = data.lastPrice;
242. tcl.price = data.lastPrice - (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
243. }
244. **else**
245. {
246. tcl.volume = 0;
247. tcl.price = 0 ;
248. }
250. tcl.cashPosWO -= tcl.volume\*tcl.price;
251. tcl.volPosWO += tcl.volume;
252. //return tcl;
253. \*traderCL = tcl;
254. }
256. //#define PT\_SELL\_THRESH
257. //#define PT\_BUY\_THRESH
258. //#define PT\_BOUNDS
260. **void** PositionTrader(\_\_global TraderCL\* traderCL, **const** MarketDataSmallCL data, mwc64x\_state\_t\* rng)
261. {
262. TraderCL tcl = \*traderCL;
264. **if** (tcl.cashPos < tcl.startCash/PT\_BOUNDS)
265. {
266. //tcl.volume = -ceil((double)tcl.volPos/PT\_SELL\_THRESH);
267. tcl.volume = -ceil(tcl.startCash / (PT\_BOUNDS\*data.lastPrice));
268. tcl.price = data.lastPrice;
269. tcl.isMarket = **true**;
270. }
271. **else** **if** (tcl.cashPos > tcl.startCash\*PT\_BOUNDS)
272. {
273. //tcl.volume = floor((double)tcl.cashPos \* data.lastPrice/PT\_BUY\_THRESH);
274. tcl.volume = floor(tcl.startCash\*PT\_BOUNDS/data.lastPrice);
275. tcl.price = data.lastPrice;
276. tcl.isMarket = **true**;
277. }
278. **else** **if** (tcl.cashPos < tcl.startCash && tcl.cashPos >= tcl.startCash/PT\_BOUNDS)
279. {
280. //tcl.volume = -ceil((double)tcl.volPos/PT\_SELL\_THRESH);
281. tcl.volume = -ceil((tcl.startCash-tcl.cashPos)/data.lastPrice);
282. tcl.price = data.lastPrice + (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
283. tcl.isMarket = **false**;
284. }
285. **else** **if** (tcl.cashPos > tcl.startCash && tcl.cashPos <= tcl.startCash\*PT\_BOUNDS)
286. {
287. //tcl.volume = floor((double)tcl.cashPos \* data.lastPrice/PT\_BUY\_THRESH);
288. tcl.volume = floor((tcl.cashPos-tcl.startCash)/data.lastPrice);
289. tcl.price = data.lastPrice - (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
290. tcl.isMarket = **false**;
291. }
292. **else** **if** (tcl.cashPosWO <= 0)
293. {
294. tcl.volume = -ceil((**double**)tcl.volPos/PT\_SELL\_THRESH);
295. tcl.price = data.lastPrice + (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
296. tcl.isMarket = **true**;
297. }
298. **else** **if** (tcl.volPosWO == tcl.startVol)
299. {
300. tcl.volume = tcl.startVol/100;    //10;//floor((double)tcl.cashPos \* data.lastPrice/PT\_BUY\_THRESH);
301. tcl.price = data.lastPrice + (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
302. tcl.isMarket = **false**;
303. }
304. **else**
305. {
306. tcl.volume = 0;
307. tcl.price = 0;
308. tcl.isMarket = **true**;
309. }
311. tcl.cashPosWO -= tcl.volume\*tcl.price;
312. tcl.volPosWO += tcl.volume;
313. //return tcl;
314. \*traderCL = tcl;
315. }
317. **bool** checkLongTermRise(\_\_constant PastPrice\* prices, **int** n)
318. {
319. **int** riseCount = 0;
320. **int** fallCount = 0;
321. **int** count = 0;
322. **double** netChange;
324. **for** (**int** i=1; i < n; i++)
325. {
326. **if** (prices[i].price > prices[i-1].price)
327. {
328. riseCount++;
329. count++;
330. netChange += prices[i].price - prices[i-1].price;
331. }
332. **else** **if** (prices[i].price < prices[i-1].price)
333. {
334. fallCount++;
335. count++;
336. netChange += prices[i].price - prices[i-1].price;
337. }
338. **else**
339. {
340. count++;
341. }
342. }
344. **if** (netChange > 0 && riseCount > fallCount)
345. **return** **true**;
346. **else**
347. **return** **false**;
348. }
350. //#define MT\_SHORT\_RANGE
352. **bool** checkShortTermRise(\_\_constant PastPrice\* prices, **int** n)
353. {
354. **if** (n <= MT\_SHORT\_RANGE)
355. **return** **false**;
357. **int** riseCount = 0;
358. **int** fallCount = 0;
359. **int** count = 0;
360. **double** netChange = 0.0;
362. **for** (**int** i=n-MT\_SHORT\_RANGE; i < n; i++)
363. {
364. **if** (prices[i].price > prices[i-1].price)
365. {
366. riseCount++;
367. count++;
368. netChange += prices[i].price - prices[i-1].price;
369. }
370. **else** **if** (prices[i].price < prices[i-1].price)
371. {
372. fallCount++;
373. count++;
374. netChange += prices[i].price - prices[i-1].price;
375. }
376. **else**
377. {
378. count++;
379. }
380. }
382. **if** (netChange > 0 && riseCount > fallCount)
383. **return** **true**;
384. **else**
385. **return** **false**;
386. }
388. //#define MT\_SIZE\_THRESH
390. **void** MomentumTrader(\_\_global TraderCL\* traderCL, **const** MarketDataSmallCL data, **bool** ltr, **bool** str, mwc64x\_state\_t\* rng)
391. {
392. **bool** longTermRise = ltr;
393. **bool** shortTermRise = str;
395. TraderCL tcl = \*traderCL;
396. tcl.isMarket = **true**;
398. **if** (longTermRise)
399. {
400. **if** (shortTermRise)
401. {
402. tcl.volume = -80;
403. tcl.price = data.lastPrice;
404. }
405. **else**
406. {
407. **if** (tcl.volPos > MT\_SIZE\_THRESH)
408. {
409. tcl.volume = -floor((**double**)tcl.volPos/MT\_SIZE\_THRESH);
410. tcl.price = data.lastPrice - (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
411. tcl.isMarket = **false**;
412. }
413. **else**
414. {
415. tcl.volume = -80;
416. tcl.price = data.lastPrice;
417. }
418. }
419. }
420. **else**
421. {
422. **if** (shortTermRise)
423. {
424. **if** (tcl.cashPos > (MT\_SIZE\_THRESH\*data.lastPrice))
425. {
426. tcl.volume = floor((**double**)tcl.volPos/MT\_SIZE\_THRESH);
427. tcl.price = data.lastPrice + (RT\_PRICE\_CHANGE\*floor((**double**)MWC64X\_NextUint(rng)\*RT\_PRICE\_SIZE/UINTMAX));
428. tcl.isMarket = **false**;
429. }
430. **else**
431. {
432. tcl.volume = 80;
433. tcl.price = data.lastPrice;
434. }
435. }
436. **else**
437. {
438. tcl.volume = 80;
439. tcl.price = data.lastPrice;
440. }
441. }
443. tcl.cashPosWO -= tcl.volume\*tcl.price;
444. tcl.volPosWO += tcl.volume;
445. //return tcl;
446. \*traderCL = tcl;
447. }

450. //#define RT\_COUNT n
451. //#define LRT\_COUNT n
452. //#define PT\_COUNT n
454. \_\_kernel **void** ProcessTraders(ulong offset, \_\_global TraderCL\* traders, \_\_constant PastPrice\* prices, **const** MarketDataSmallCL data)
455. {
456. ulong perStream = (RT\_COUNT+LRT\_COUNT+PT\_COUNT+MT\_COUNT)/get\_global\_size(0);
458. **bool** ltr = checkLongTermRise(prices, data.numPastPrices);
459. **bool** str = checkShortTermRise(prices, data.numPastPrices);

462. \_\_global TraderCL \*tradersDest = traders+get\_global\_id(0)\*perStream;
463. **for** (uint i=0; i < perStream; i++)
464. {
465. mwc64x\_state\_t rng;
466. MWC64X\_SeedStreams(&rng, offset, perStream);
468. **if** (tradersDest[i].type == 0)
469. RandomTrader(&tradersDest[i], data, &rng);
470. **else** **if** (tradersDest[i].type == 1)
471. LargeRandomTrader(&tradersDest[i], data, &rng);
472. **else** **if** (tradersDest[i].type == 2)
473. PositionTrader(&tradersDest[i], data, &rng);
474. **else** **if** (tradersDest[i].type == 3)
475. MomentumTrader(&tradersDest[i], data, ltr, str, &rng);
476. }
477. }

## Data