This assignment is worth 30% of the module mark.

This assignment may be attempted in groups of size up to 3. Marking criteria will be different depending on your group size. Further instructions are given below.

This assignment assesses your knowledge in file handling, STL containers, and general C++ programming skills and program design.

Clarifications and amendments may be announced on the Blackboard discussion forum for this assignment. **You are expected to read the forum.**

The Problem

You will write a program to simulate how trade orders in a stock market are processed. (Please note what we describe here is a simplified, and sometimes totally different, version of what happens in the "real world". Your program should follow the specification given here, not whatever you know about real world stock exchanges.)

Overview

In a stock market, shares of many different companies are traded. Here we assume we are trading the shares of one specific company only. Investors put in *orders*, which are instructions on how many shares they want to buy/sell and at what price. The orders are *pending* until they are *matched*: a buying order can be matched with a selling order following certain rules. The two orders are then said to be *executed*.

In real life, the orders are placed in real-time in response to market situations, but in this assignment we assume the orders are all in a file, to be read in and processed by your program one by one; more on this later.

Orders

An order is either a buying order (investor wants to buy some shares) or a selling order (investor wants to sell). Each order comes with a target trade quantity (number of shares they want to buy/sell) and possibly a target price.

Orders can be a limit order or a market order. A *limit order* is one that specifies a target price (also known as *limit price*), which is the highest price that the buyer is willing to pay per share (for buying orders), or the lowest price that the seller is willing to sell per share (for selling orders). Trades must not happen at a price that contradicts these limits, but the actual execution price may be in a "more favourable" direction for the traders. For example, a selling order with a limit price of 10.00 may be executed at a price of 10.25, if some buyer is desperate enough to pay a high price. A *market order* is one that doesn't have any price limit; in other words the buyer/seller is willing to pay whatever price the market currently offers.

An order may also be specified as *indivisible*, which means the target quantity must either be traded in full or not at all; or *divisible*, which means the order can be matched with some other order with a smaller target quantity, and the remaining quantity will still be pending, possibly matched with some other orders at the same or different prices, or not executed at all. For example, suppose there is a selling order which is a market order and has a target quantity of 50. It may be matched with a buying order with a limit price of 10.00 and a quantity of 30. The remaining quantity 20 becomes a new selling order, which may match (immediately or later) another buying order with a price limit 9.75 and a quantity of 10. The remaining quantity 10 again becomes a new selling order, but it may not match with anything. In this case the seller sold 30 shares at a price of 10.00, 10 shares at a price of 9.75, and the final 10 shares are not sold.

Matching orders

At any one point, your program may have some pending orders previously read. Upon reading each new order, the program should try to match it to some pending order of the opposite (buying vs. selling) type. A buying order b can be matched with a selling order s if and only if:

- The limit price of b is higher than or equal to the limit price of s, or if one or both of them are market orders; and
- If both b and s are indivisible orders, than they must have the same target quantity; if one of them is divisible and the other indivisible, then the divisible order's target quantity must be at least as large as the indivisible order's quantity (so the indivisible order can be executed in full); if both are divisible then there are no restrictions on quantity.
  For example, a divisible order buying 50 shares can be matched to an indivisible order selling 30 shares, so the indivisible one is executed in full and the divisible one has 20 shares left to be matched. But in the opposite case, a divisible order buying 30 shares cannot be matched to an indivisible order selling 50 shares. Note that in the latter case, it may be that a number of (divisible or indivisible) buy orders can give a combined total of exactly 50 shares to fit what is required by the indivisible selling order, but you are not required to (and should not) find and execute such "combo" matches.

Order priorities

If in the above process, the new order can be matched with more than one order of the opposite type, then it will be matched to the one with the highest priority, as defined below:

- For buying orders, those with a higher limit price have higher priority; for selling orders those with a lower limit price have higher priority.
- Market orders have higher priority than any limit orders.
- If two orders have equal limit prices (or both are market orders), priority is given to the one arriving earlier.

Executing orders

Once a pair of orders are matched, they are "executed" (fully or partially). The execution price (the price at which the transaction takes place) is determined as follows:

- If both the buying and selling orders are limit orders, then the limit price of the "older" order (the one arriving earlier) is the execution price;
- If one of them is a market order and the other is a limit order, then the limit order's limit price is the execution price;
- If both are market orders, then the execution price of the previous transaction is used as the execution price. (There will always be a "previous transaction price"; see file input below.)

In the case of divisible orders, any unfulfilled quantity becomes a new pending order with the "residual" quantity, but keeping all other information (including its "arrival time") the same as the original order. This residual order should immediately be matched with other pending orders and (if there is a match) executed, which may result in an even smaller residual order, which should again be immediately matched and so on, until either it is fully executed or cannot be matched. This process should be completed before another new order is read from the file and processed.

Your Program: Input and Output

Your program takes input from a file. The name of the file is given as a command line argument, so your program is to be run like

```
./main orders.txt
```

where orders.txt is the input file. Some example input files are given in a later section. The file contains a number of lines. The first line contains a single floating point number representing the last traded price (think of it as the last transaction price of the previous day). From the second line onward, each line represents an order, with whitespace-separated fields like this:

```
order001 B L I 9.75 100
order003 S M D 100
order004 S L D 10.25 120
```

- The first field is the "order ID" which is a unique alphanumeric string (without spaces) identifying the order. Note that even though it appears to come in increasing numerical order in this example, these IDs are not necessarily in increasing order and may not even have numbers in it.
- The second field is either B or S: indicate whether this is a buy or sell order.
- The third field is either L or M, indicating whether this is a limit order or market order.
- The fourth field is either D or I, indicating whether it is a divisible or indivisible order.
- For limit orders, the fifth field is the limit price, and the sixth field is the target quantity. For market orders, the fifth field is the target quantity.

The target trade quantity is a positive integer. The limit price is a floating point number. In this application, to minimise the chance of floating point inaccuracy issues, we assume all prices are multiples of 0.25.

You can ignore all input error handling. In other words, if the input file does not conform to this specification, your program is allowed to behave in any way.

Your program should read and process the orders in the file one by one. Upon reading each order, your program should display all pending orders (including the new one). There is no exact requirement on how it should look like, although you should list all of them, separate the buying ones from the selling ones, and in descending order of their priorities. Often they are displayed in a way similar to this:

```
Last trading price: 10.00
Buy                         Sell
----------------------------------------
ord001 10.00 100          ord002 M     50
ord004 10.00 50           ord007 10.25 100
ord012  9.75 60           ord003 10.25 100
ord006  9.50 200          ord019 10.50 20
ord008  9.50 20
```

(where M stands for market order)

Then, if some orders are matched, it should display information on the executed orders, and the updated list of pending orders on screen. Again, there are no exact requirements on how they should be displayed.

The details of each executed order must be written to a file called executions.txt. It should contain details of each executed order, in the order they are executed. Each execution always involve a pair of orders (one buying and one selling); the buying one should appear before the selling one. Orders that remain unexecuted at the end should also be listed, in the order they were received. Please refer to the following example and the sample outputs (in a later section) for the details of orders required and the exact wording:

```
order001 20 shares purchased at price 1.25
order003 20 shares sold at price 1.25
order001 30 shares purchased at price 1.50
order004 30 shares sold at price 1.50
order004 40 shares unexecuted
order002 100 shares unexecuted
```

Unlike the screen outputs, **this file output has to be exactly as stated here** as the marking will involve automated comparisons of your program outputs with the expected outputs.

All input and output files are in the same directory from where the program is run.

All prices should be printed to 2 decimal places (e.g. 12 should be printed as 12.00). This can be achieved by setting the output stream with `std::fixed` and `std::setprecision(2)`, e.g. for screen outputs:

```
cout << std::fixed << std::setprecision(2);
```

This only needs to be done once.

Group Sizes and Essential/Extra Functionalities

Please see this page for rules on group size, group formation, submission instructions etc.

The previous section describes the full assignment requirements. Depending on your group size, some of them may become "extras":

- For groups of size 3, you should implement everything as described above.
- For groups of size 2, you can assume all orders are limit orders. Handling of market orders count as extras.
- For groups of size 1, you can assume all orders are limit orders, furthermore you can either assume all orders are divisible, or all orders are indivisible (you can choose which of the two). In addition, only file outputs are required; screen outputs are optional. Handling of more general order types or screen outputs count as extras.
- For any group size, printing to the screen "nicely" (e.g. prices and other information "aligned" vertically etc) also counts as extras.

Implementing these "extra functionalities" may positively affect your mark (see the marking criteria below).

Note that the input file format will stay the same, even if your group size means you only handle a more limited range of order types. For example, for groups of size 2 you can assume your program is only tested where all orders have L in the third field. But the field is still there, and your program should not assume the input file has an updated format that removes that L/M field. I'm not going to make separate test files just for you.

Marking Criteria and Testing

This assignment is unlike the previous programming assignments; you have to develop all your program from scratch. No partially written .h or .cpp file will be provided. It is up to you to decide how to structure your program, what data structures to use, what classes to create (if any), or how the source code is split into different files (if you choose to do so). There is no requirement that the program must be very "object-oriented"; you can write the whole program in one .cpp file without defining any classes, if you so wish. Even though the marking criteria will give considerations to these design choices, it does not mean that there is only one "correct" or "best" design. Remember, C++ supports a number of programming paradigms.

You can also name your file(s) in any way. However, you must supply a makefile so that when typing 'make' the program will be compiled into an executable file called 'main'. Please refer to an earlier tutorial on how to write makefiles. There is also a very simple sample below.

There are no runnable test suites as in previous assignments, although there are some sample input and output files (see next section). As usual, your program will only be tested on the departmental linux system, and this time with your makefile.

The marking criteria is also different from previous assignments; see this separate page.

Files Provided

Here are some sample input and output files. (Note that the output filenames are numbered for convenience here. Your program should always write outputs to a fixed filename executions.txt.)

- sample input 1, sample output 1, explanation 1
- sample input 2, sample output 2, explanation 2
- sample input 3, sample output 3, explanation 3
- sample input 4, sample output 4, explanation 4
- sample input 5, sample output 5, explanation 5
- sample input 6, sample output 6, explanation 6
- sample input 7, sample output 7, explanation 7
- sample input 8, sample output 8, explanation 8
- sample input 9, sample output 9, explanation 9
- [More may be added / some may be updated later]

This is a sample makefile that only compiles one file called main.cpp into an executable called main. You need to modify it if your program source files are organised differently.

- makefile

This is a zip file containing executable programs (main for linux, main.exe for Windows) which is an example implementation of what is required. The linux one should work on the departmental linux system (and probably only on it; it probably won't work on your home linux and certainly not in Windows!) You may have to type "chmod u+x main" to make it executable before running it. The Windows one is compiled on Windows x64 architecture. You probably need to run it from a command prompt, and in a slightly different way (main input.txt, without the ./). You accept the risk that anything might happen to your computer if you decide to run a Windows exe file from an untrusted source.

They are given for your reference only. Your program does not have to mimic their every behaviour: as stated already, your screen output does not have to follow any fixed format. It is not guaranteed to be bug-free and so, in case of disagreement with the assignment specifications, alert me about it! It also does not attempt to handle any errors at all, such as wrong file formats or no command line arguments.

- sample executables

Submission Instructions

Submit your completed work on Blackboard ("Assessment and Feedback" on the left, then "Assignment 2B"). Please also see group-specific instructions (same link as before)

You should package all files in your submission in a zip archive. The file MUST be called cw2b.zip (lowercase). At least the makefile should be in the top level folder in your archive; furthermore you are strongly encouraged to put everything in only the top level folder (no subfolders). Mac users: please avoid zipping on a Mac if at all possible. It tends to create an extra level of folder making automatic extraction difficult. At least it may help if the folder containing your code (before zipping) is called "cw2b".

Your zip file should include any source files needed to compile your program. In particular, please remember the makefile, and readme.txt if you have extra/incomplete features. Do not include anything unnecessary, such as the entire project folder of whatever IDE you are using. Please note that the linux file system is case-sensitive, so main.cpp and Main.cpp are not the same thing. If the filenames referred to in your makefile and the actual filenames have different upper/lowercase, it will not work. Any such problems will incur a penalty.

This is an **group assignment**, but collaboration between groups is not permitted. Plagiarism will be treated strictly according to standard university and departmental procedures. Your submissions will be sent to a plagiarism detection service (MOSS).

In line with university policy, marking will be done anonymously. Only the Blackboard-supplied userid / student number will be visible in marking.

For the above two reasons, do not include your name, userid, student number, or any other personally identifiable information in your programs.