

IB9N7 C++ for Quantitative Finance

Project 2015: Central limit order book implementation

26 February 2015

Deadline: 15 April 2015 at Noon.

1 Overview

(This section has been copied from http://en.wikipedia.org/wiki/Central_limit_order_book.)

A Central Limit Order Book or ("CLOB") is a trading method used by most exchanges globally. It is a transparent system that accepts customer orders for particular securities and matches them on a 'price-time priority' basis. Each order is either a bid or an offer **limit** order: an order to respectively buy or sell a security at a price "no worse" than the specified limit price (in the case of bid orders, to buy at a price no more than the limit, and for offer orders, to sell at a price no less than the limit). The highest bid order and the lowest offer order constitutes the **best** market or "the touch" in a given security contract. Customers can routinely cross the bid/ask spread to effect low cost execution. They also can see market depth or the "stack", the aggregate order book showing bid orders for various sizes and prices on one side vs. offer orders at various sizes and prices on the other side.

The CLOB is by definition fully transparent, real-time, anonymous and low cost in execution. The "central" aspect refers to the consolidation of all trades in the given securities to be via the single limit order book (rather than a fragmented collection of limit order books). For the purposes here, all orders can be considered to be basic limit orders.

For this project, students are to implement a simplified version of a CLOB according to the given specifications.

1.1 Orders

During the stock exchange opening times, every market participant (trader) can submit a limit order where each of the following is specified:

- Security identifier;
- Buy or Sell;
- Price ("the limit");
- Quantity (volume).

The stock exchange tries to execute this order immediately if possible. If the order cannot be fully executed immediately, the stock exchange puts this order into a database called the order book. (For the purposes of this assignment, the term "database" should be interpreted as a container object.) The order can be executed later according to "price-time priority", upon which it is removed from the order book. Also, each order in the order book can be cancelled by the trader who submitted it.

1.2 Order execution and price-time priority

(The content of this section is adapted from what the XETRA exchange says about the price-time priority principle: <http://www.boerse-frankfurt.de/en/help/orderbook/xetra+order+book+howto/price+time+priority>.)

When an order is accepted:

- It must have an order identifier (which must be unique across all orders, whether buy or sell) and must be given a timestamp.

This timestamp is used (only) to prioritise orders in the book with the same price — the order accepted earliest at a given price limit gets executed first. It therefore need not necessarily describe a conventional date and time, as long as it can be used to apply the price-time priority principle.

- The exchange first compares the new order against the limits of all orders contained in the order book.

If the incoming order (or part of it) is immediately executable, meaning it is capable of being matched against an existing order or orders, one or more transactions are generated.

To be immediately (partially) executable, the order must be:

- an order to buy at a price at or above the lowest offer in the order book;
- an order to sell at a price at or below the highest bid in the order book.

If the order is immediately (partially) executable, it is matched with an order in the order book according to the price-time priority: Orders with the best possible prices (highest price limit for buy orders, lowest price limit for sell orders) always take precedence in the matching process over other orders with worse prices. Again, if multiple orders have the same price limit, the criterion used for establishing matching priority is the order timestamp.

Orders may not necessarily be executed at a single price, but may generate several transactions at different prices. The price at which each transaction takes place is determined by the price of the matched order already present in the order book (**not** that of the newly accepted order — clients should therefore check the order book before submitting orders to ensure they are not overpaying).

If the matched order in the order book is fully executed as a result of the transaction, it is removed from the order book, otherwise it is updated to indicate the quantity executed and the volume still open. When a large order executes against the total available quantity at a given price level, the next best price level becomes best. This process continues as long as the incoming order remains executable. If not completely executed upon entry, the order is held in the order book, with fields denoting the quantity of the order that has been executed and the volume still open.

It is possible for a single order to generate multiple (partial) executions at different points in time. For example, an order may generate a partial execution upon entry, while the remaining open order remains in the order book.

The example section later should help to illustrate these principles.

1.3 Market orders

In addition to limit orders, clients may submit market orders. A market order is a buy or sell order to be executed immediately at current market prices. This order type does not give the client any control over the price. As long as there are willing sellers or buyers, market orders are executed. Market orders are therefore used when certainty of execution is a priority

over price of execution. A specific type of market order which will be used in this project is the “market or expire” order (sometimes referred to as a “market or cancel” order). Here, if there are not enough willing sellers or buyers, then the market or expire order is filled to the maximum extent possible, and the remainder of the order is deactivated.

2 Project Outline

You are required to implement the `OrderBook` class, encapsulating (some of the features of) a CLOB. This class should maintain an order book for just one (anonymous) security and will not distinguish between different market participants.

You are provided with a starting point on `my.wbs`, consisting of:

- a Dev-C++ project file `CLOB.dev`;
- `CLOB_shared.hpp`, a header file containing declarations in the global namespace;
- `stub.cpp` and `.hpp` files for the `OrderBook` class;
- `stub.cpp` and `.hpp` files for the `AggregatedQuoteType` class;
- a `.cpp`, `.hpp` and pre-compiled object `.o` file for the `visualise` translation unit, which you will not need to edit;
- a `.cpp` and `.hpp` file for the `visualise_helper` translation unit, which you may (but need not) edit;
- the `CImg.h` header-only library, which is used by the `visualise` translation unit, but which you do not need to otherwise use or understand;
- a main translation unit, containing:
 - a sample `CLOB_simple_test` function that creates an `OrderBook` instance and invokes certain methods on it.
 - a sample `main` function that calls `CLOB_simple_test`.

The files you are provided with only define a **minimum public interface** for the `OrderBook` class. The implementations of the methods are missing, and no member variables are declared. You may like to define additional (`public`, `private` or other) members.

You are encouraged to think about the structure of your program before you start to code. Your `OrderBook` class needs to preserve, but may add to, the minimum interface provided. You are free to modify the `main` and `CLOB_simple_test` functions as you wish, but you should be aware that your class will be tested against the specification more thoroughly using a more sophisticated `main` translation unit than is provided to you. In reality, the main translation unit for this application would not have order information hard-coded into it. Instead, it might present a menu system to the user, and/or read submitted orders from a file/database. For simplicity, you are **not** asked to implement either for this assignment.

Time can be defined relatively by the order in which orders are received. For the purposes here, all orders will be received by the same machine, so this relative sense of time will be sufficient to apply the price-time priority principle. You need not and should not attempt to use any external notion of time.

The following types are defined for you in the provided `CLOB_shared.hpp` file:

```
1 enum SellBuyType {SELL, BUY};
2 enum OrderStatusType {ACTIVE, FILLED, CANCELLED, EXPIRED};
3
4 typedef unsigned long OrderIdentifierType;
5 typedef double PriceType;
6 typedef unsigned long VolumeType;
```

7 typedef unsigned long TimeType;

Note that the type `unsigned long` is used in this specification to refer to counts and identifiers of orders. You are welcome to narrow this type if your implementation cannot cope with counts this large, but you are not expected to worry about this.

Your class should support the following `public` operations:

- Default constructor (possibly implicit).

```
OrderBook ();
```

The order book starts off closed.

You may assume that the order book will always be closed before the object instance is destroyed; you do not need to provide a destructor to ensure this.

- `bool open(PriceType tick_size, PriceType tolerance, std::ostream & log);`
`bool close();`

Reject all orders which are submitted before the order book is open or after it is closed (though note that it may re-open again later for the next trading day).

At close, deactivate the remaining parts of any orders that remain active in the order book (but note that it should still be possible to inspect the state of all orders). Specifically, unexecuted orders should be considered as expired and orders that were partially executed before the order book was closed should now be truncated and considered as filled.

If `open` or `close` are executed when the order book is already open or closed, or invalid arguments are supplied to `open`, `false` should be returned (no message should be printed to `std::cout`, but you may print an appropriate message to `std::cerr`).

`tick_size` specifies the tick size: all prices of submitted orders should be a multiple of the tick size, or within the appropriate tolerance of such a multiple. Specifically, if an order is submitted with the price p and $|p - N \text{tick_size}| > \text{tolerance}$ for all natural numbers N , such an order should be rejected because the price is invalid. On the other hand, if there exists an N such that $|p - N \text{tick_size}| \leq \text{tolerance}$ then the price p can be accepted (it is approximately a multiple of `tick_size`). In such case, you may choose to either store p or store $N \text{tick_size}$. Hint: you cannot use the integer remainder operator with floating point types. Additionally, the floating point version `fmod` does not work well.

The order book should treat prices which differ by $\leq 2 * \text{tolerance}$ as being equal. This applies for both the price-time priority principle and the `get_aggregated_order_book` function.

The `tolerance` can always be assumed to be less than or equal to `tick_size/10`. Note that the `tolerance` and `tick_size` may change on the next call to `open`.

`log` is an open stream where the order book should write messages about submitted/-cancelled orders and executed trades (transactions). In the event that the order book needs to write a message before the first call to `open`, `log` should default to `std::cerr`. Hint: you will need to save a pointer to `log` as a member variable within the class.

Each trade message should contain the identifiers (`OrderIdentifierType`) of orders involved in the trade, the price and the traded volume, formatted as per the examples later.

Whenever outputting prices, format them through the `format_price` function first.

If the order book successfully opens/closes, the messages `"Order_book_open."` or `"Order_book_closed."` should respectively be printed to `log`.

- `bool submit_order(OrderIdentifierType order_id, SellBuyType sb, PriceType price, VolumeType volume);`

Check if the order can be accepted; recall the conditions for acceptance in terms of the `tick_size` etc, and also that order volumes must be strictly positive. A price of zero indicates that the order is to be a market or expire order. Check also that the identifier supplied is unique (you do not need to worry about generating such unique identifiers, these are hardcoded into the `CLOB_simple_test` function). Note that order identifiers need not be sequential.

If the order is accepted, execute it if possible according to the price-time priority, otherwise add it to the order book for potential execution later. If the order is accepted but is never added to the active order book (because it is immediately fully executed), information about the order still needs to be retained so that it can be queried later by `print_order_info`.

Return `true` if the order was accepted, or `false` otherwise.

Write the order message to `log`. Write the trade messages to `log` if the order was (at least partially) executed.

Once an order has been submitted, it cannot be changed (but it can be cancelled completely — clients should therefore place a new order to effect a change, but in doing so will lose their place in the queue).

- `bool cancel_order(OrderIdentifierType order_id);`

Remove the order from the active order book (but again note that information concerning the order must still be retained). Reject the cancellation and return `false` if the order has already been cancelled, has already expired, is already fully executed, or the order book is closed. Also, return `false` if the order is not found. Otherwise, return `true`.

Write the order message to `log`.

If the order is already partially executed, the total volume should be reduced to the already executed volume, and the order should be considered filled instead of cancelled. (The order message need not reflect the change in status to filled in this instance.)

- `void print_order_info (std::ostream & where, OrderIdentifierType order_id) const;`

Display the information about the order corresponding to the given order identifier on the given output stream. The information should include:

- the order status,
 - * ACTIVE, if the order is in the order book and not fully executed;
 - * FILLED, if the order was (essentially) fully executed;
 - * CANCELLED, if the order was cancelled;
 - * EXPIRED, if the order was submitted on a previous trading day.
- The type of the order: whether it is a sell or buy order;
- The current order price;
- The total order volume;
- The matched volume of the order (the volume that has already been executed);

If no corresponding order is found, an appropriate message should be printed.

The format of the output should correspond exactly with those in the examples below.

Notice that even if the order was fully executed, expired or cancelled, the order book should maintain the information about the order. Despite this requirement, such orders are not conceptually part of the order book any longer, and are not available for participating in further trades etc; they are **inactive**. Orders that have been deactivated never become active again.

- `std::string format_price (PriceType price) const;`

Return a string containing the formatted version of the price `price`.

The formatting should be such that `price` always has exactly $1 + \lceil -\log_{10}(\text{tick_size}) \rceil$ decimal places, where $\lceil \cdot \rceil$ denotes the ceiling function.

- `std::vector<AggregatedQuoteType> get_aggregated_order_book(SellBuyType which_side) const;`

Get the information on (the active and visible portions of orders in) the current order book. Return a vector representing the requested sides of the order book (bid or ask). Each element of the vector should be of type `AggregatedQuoteType`, a simple class providing the following member functions (you may or may not need/want to add more):

- `PriceType get_price()const;` — price of the level;
- `VolumeType get_volume()const;` — volume available for the level;
- `unsigned long get_number_of_orders()const;` — number of active orders of the level.

The elements in the vector should be ordered according to price, so the best price comes first. That is, in decreasing order for bid side and in increasing order for the ask side.

Note: this function should NOT generate any output (in your final submission). You may use the provided `display_aggregate_order_book` free function in `visualise.cpp` to generate corresponding output, though you are also free to write (and submit) another function that outputs this information to aid with your testing if you find the provided function too cumbersome. There is no prescribed formatting of the output that you need to follow in this instance.

The output on the given `log` stream should correspond exactly to the output in the examples (next section).

Any additional information you wish to output (e.g. debugging information) should be done on the `cerr` stream, and you should consider commenting out such code before submitting. In particular, you might like to provide a function that outputs the entire order book (possibly also including inactive orders), rather than an aggregated form of it.

2.1 Example

All the examples refer to the same security X, and prices will be assumed to be in units of pence (note that the prices can involve fractions of pence as shares are often traded in large volumes).

Let us trace through the expected output of the `CLOB_simple_test` function you are provided with (called by `main`), line-by-line:

```
12     OrderBook book;
```

Creates a new `OrderBook` object called `book`. (Note that, although only one instance of the `OrderBook` class is created by the provided `main` translation unit, you should not generally assume that only one instance of the `OrderBook` class will ever exist.) This line is not associated with any output. The order book starts off empty.

```
13     display_aggregate_order_book(book);
```

This displays the aggregate order book. It will be empty initially. This is present primarily to help you trace through your code, and is disabled by default. Refer to the lecture for more on this. There are many identical lines in the `main` translation unit and they will not be discussed further.

```
15     book.open(0.1, 0.01, std::cout); // return values ignored in this example
```

The order book is opened for trading.

The `tick_size` is set to 0.1 here, and the `tolerance` to 0.01. `std::cout` is used as **log in this instance**, but your implementation should not assume this is always the case (i.e. you should not directly refer to `std::cout` in your `OrderBook.cpp`).

This prints the following message on **log**:

```
1  Order book open.
```

```
18     std::cerr << "*****" << std::endl;
19     std::cerr << "Block_A_-_there_should_be_no_matches_yet" << std::endl;
20     std::cerr << "*****" << std::endl;
21     book.submit_order(1u, SellBuyType::SELL, 10.1, 100u);
```

A sell order for 100 of security X is submitted with limit price 10.1 p. There is no order in the order book for this to match with, so it is saved in the order book for potential execution later.

```
23     book.submit_order(2u, SellBuyType::SELL, 10.3, 75u);
```






```
25     book.submit_order(3u, SellBuyType::BUY, 9.8, 200u);
```

```
27     book.submit_order(4u, SellBuyType::BUY, 10.0, 150u);
```

```
29     book.submit_order(5u, SellBuyType::BUY, 9.9, 100u);
```



```
31     book.submit_order(6u, SellBuyType::SELL, 10.1, 50u); // this entry shares the same price as a
    previous entry
```

For each of these orders, no matches are made. Hence, the order book for security X then looks like this:

Bid (buy) side				Ask (sell/offer) side			
	Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level	
	10.00	150	1	10.10	150	2	 6 1
	9.90	100	1	10.30	75	1	 2
	9.80	200	1				

The best price levels for each side are shown by the top row, and the filled bars just help to visualise the active volumes of orders. The overlaid numbers are the respective order identifiers. Here, orders 1 and 6 are both active sell orders for the same price, so they are combined onto the same level of the aggregated order book. I have chosen to represent orders with a higher price-time priority at the same level further to the outside.

The corresponding aggregated order book can be visualised by removing features of individual orders, like this:

Bid (buy) side				Ask (sell/offer) side			
	Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level	
	10.00	150	1	10.10	150	2	
	9.90	100	1	10.30	75	1	
	9.80	200	1				

The **log** output generated by submitting these orders is:

```

2 Order submitted: ID=1, type=SELL, price=10.10, volume=100.
3 Order submitted: ID=2, type=SELL, price=10.30, volume=75.
4 Order submitted: ID=3, type=BUY, price=9.80, volume=200.
5 Order submitted: ID=4, type=BUY, price=10.00, volume=150.
6 Order submitted: ID=5, type=BUY, price=9.90, volume=100.
7 Order submitted: ID=6, type=SELL, price=10.10, volume=50.
```

```

33 book.print_order_info(std::cout, 1u);
```

This line yields the following output on **cout**:

```

8 Order information: ID=1, type=SELL, price=10.10, total volume=100, executed volume=0,
  status=ACTIVE.
```

```

35 std::cerr << "*****" << std::endl;
36 std::cerr << "Block_B_-_there_should_be_a_match" << std::endl;
37 std::cerr << "*****" << std::endl;
38 book.submit_order(7u, SellBuyType::BUY, 10.2, 200u);
```

Now, a buy order for 200 of security X is submitted with limit price 10.2 p. A transaction occurs upon submission of this order because it gets matched with orders on the sell side. According to price-time priority, order 1 is the best order on the sell side. The price of the transaction is the price of the order already present in the order book (order 1), so 10.1 p. There are only 100 units to match against, however. Therefore, sell order 1 becomes fully executed and the buy order becomes partially executed. Other searches for matches are made against the remainder of the buy order. Order 6 is also a match, and all 50 units of this order get matched. The price is again 10.1 p. The buy order is still not fully executed, and there are no more matches, so the remaining volume of 50 is saved in the order book for potential execution later.

The following messages are generated on the **log**:


```

9 Order submitted: ID=7, type=BUY, price=10.20, volume=200.
10 Transaction: SELL=1, BUY=7, price=10.10, volume=100.
11 Transaction: SELL=6, BUY=7, price=10.10, volume=50.

```

In the transaction lines, the details of the order in the book are printed first, followed by the details of the newly submitted order.

And the resulting order book is now:

Bid (buy) side				Ask (sell/offer) side		
	Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level
7	10.20	50	1	10.30	75	1
4	10.00	150	1			
5	9.90	100	1			
3	9.80	200	1			

```

39 book.print_order_info(std::cout, 1u);
40 book.print_order_info(std::cout, 7u);

```

These lines yield the following output on `cout`:

```

12 Order information: ID=1, type=SELL, price=10.10, total volume=100, executed volume=100,
    status=FILLED.
13 Order information: ID=7, type=BUY, price=10.20, total volume=200, executed volume=150,
    status=ACTIVE.

```

```

43 std::cerr << "*****" << std::endl;
44 std::cerr << "Block_C_-_market_orders" << std::endl;
45 std::cerr << "*****" << std::endl;
46 book.submit_order(10u, SellBuyType::SELL, 0.0, 200u);

```

The price of this SELL order is the special value 0.0, which signifies a “market or expire” order. As there are orders on the opposing side of the order book, this order can be (at least partially) executed. In fact, the total volume of orders on the BUY side exceed 200, so the market or expire order can be fully executed.

The following `log` output is produced:

```

14 Order submitted: ID=10, type=SELL, price=0.00, volume=200.
15 Transaction: BUY=7, SELL=10, price=10.20, volume=50.
16 Transaction: BUY=4, SELL=10, price=10.00, volume=150.

```

```

47 book.print_order_info(std::cout, 10u);

```

The order information for the market or expire order is now as follows:

```

17 Order information: ID=10, type=SELL, price=0.00, total volume=200, executed volume=200,
    status=FILLED.

```

And the order book is:

Bid (buy) side				Ask (sell/offer) side		
	Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level
5	9.90	100	1	10.30	75	1
3	9.80	200	1			

```
49     book.submit_order(11u, SellBuyType::BUY, 0.0, 100u);
```

This is now a BUY market or expire order, and again results in a transaction.

The following output is produced on **log**:

```
18 Order submitted: ID=11, type=BUY, price=0.00, volume=100.
19 Transaction: SELL=2, BUY=11, price=10.30, volume=75.
```

```
50     book.print_order_info(std::cout, 11u);
```

The order could not be fully executed, but because it is a market or expire order, is not able to participate in future transactions. Therefore, the total volume is truncated and the order is marked as FILLED:

```
20 Order information: ID=11, type=BUY, price=0.00, total volume=75, executed volume=75,
    status=FILLED.
```

The order book is thus:

Bid (buy) side				Ask (sell/offer) side		
	Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level
5	9.90	100	1			
3	9.80	200	1			

```
52     book.submit_order(12u, SellBuyType::BUY, 0.0, 100u);
53     book.print_order_info(std::cout, 12u);
```

Another market or expire order is submitted, but this time there is no order available even for a partial match. The order is therefore already expired, and the order book is as it was previously.

The following is output on **log** and **cout** respectively:

```
21 Order submitted: ID=12, type=BUY, price=0.00, volume=100.
22 Order information: ID=12, type=BUY, price=0.00, total volume=100, executed volume=0,
    status=EXPIRED.
```

```
55     std::cerr << "*****" << std::endl;
56     std::cerr << "Block_D_-_manipulating_existing_orders" << std::endl;
57     std::cerr << "*****" << std::endl;
58     book.cancel_order(5u);
59     book.print_order_info(std::cout, 5u);
```

This cancels order 5 and subsequently outputs its information.

The following is output on **log** and **cout** respectively:

```
23 Order 5 cancelled.
24 Order information: ID=5, type=BUY, price=9.90, total volume=100, executed volume=0,
    status=CANCELLED.
```

The order book now looks like this:

Bid (buy) side				Ask (sell/offer) side		
	Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level
3	9.80	200	1			

```

62     std::cerr << "*****" << std::endl;
63     std::cerr << "Block_E_-_invalid_orders/actions" << std::endl;
64     std::cerr << "*****" << std::endl;
65     book.cancel_order(5u);
66     book.cancel_order(2u);
67     book.cancel_order(50u);
68     book.print_order_info(std::cout, 50u);
69     book.submit_order(15u, SellBuyType::SELL, 10.15, 50u);

```

These lines attempt invalid operations: the first because the order has already been cancelled, the second because the order is already complete, the third and forth because there is no order with identifier 50, and the last because the price is not within the allowed tolerance of a multiple of the `tick_size`.

The following is output on `log`:

```

25 Order 5 cannot be cancelled.
26 Order 2 cannot be cancelled.
27 Order 50 not found.
28 Order 50 not found.
29 Order not accepted.

```

```

71     std::cerr << "*****" << std::endl;
72     std::cerr << "Block_F_-_end_of_the_first_day's_trading" << std::endl;
73     std::cerr << "*****" << std::endl;
74     book.close();

```

The order book is closed, all remaining orders are cancelled (no cancel messages are produced for these orders), and the following is output on `log`:

```

30 Order book closed.

```

The order book is now empty.

Bid (buy) side			Ask (sell/offer) side		
Bid limit price at level	Volume of bid orders at level	Number of bid orders at level	Ask limit price at level	Volume of ask orders at level	Number of ask orders at level

```

75     book.print_order_info(std::cout, 3u);

```

Orders that were partially executed before the order book closed are now considered filled, and unexecuted orders are considered expired, as shown by the output from the above lines:

```

31 Order information: ID=3, type=BUY, price=9.80, total volume=200, executed volume=0,
    status=EXPIRED.

```

Note that this section does not demonstrate all the required functionality; you should think about other tests that are necessary to ensure full compliance with the specification.

3 Submission

You should submit your code files to my.wbs under the module's assessment item "Assessed Coursework". The deadline for submission is 15 April 2015 at Noon.

You are not required to produce a project report, but you should ensure that your code is adequately documented using comments.

Remember to back up your work: Every year, someone loses their assessment at the last moment. This does not count as an excuse for late submission.

- Your code should be provided as .cpp and .hpp files, not eg part of a Word document.
- You should not include any compiler-generated files or any of the output files generated by your program.
- You are free to create and submit new translation units and header files. You may optionally include any additional files that help to demonstrate your efforts, code files or otherwise. If your additional files are not code files or text files, they should be in PDF format.
- You should submit main.cpp even if you have not made functional changes to it, and you should submit your .dev file. If you have not used Dev-C++ as your IDE, submit whichever project/workspace/solution file that your IDE uses. If you have not used any IDE, submit a Makefile or list of compilation commands (Windows/*nix targets accepted).
- Your University number should appear in a comment at the top of each .hpp and .cpp file you submit.
- (Marking is anonymous so your name should not appear anywhere in the submission.)
- You must package all your files into a ZIP file, name the file with your 7 digit University number, and upload the ZIP file to my.wbs.

Marking

- Your program will be tested to determine if it compiles without errors or warnings.
- The output of your program will be compared against the expected output.
- Parts of the marking process may be aided by computer scripts. You should therefore be very careful in following the prescribed output formats etc.
- Your code will be looked at to examine the coding practices you have followed, and to provide partial marks where the output is not as expected.

You will be assessed on:

- The quality of your code in terms of portability, readability, maintainability, clarity, generalisability, sophistication, performance and general presentation. In particular, remember that you should aim to minimise unnecessary duplication of similar code.
- Whether the code is valid (in accordance with the C++ standards) and whether it successfully delivers the required functionality.

The specification should be sufficiently clear but you are allowed to negotiate it on an individual basis.

You should use the most advanced style you can, but note that it is significantly better to submit lower level, working code than higher level code that does not have the required functionality. You are also encouraged to consider the efficiency of your solution, but a working solution should again be the priority.

You will not be penalised for using any features that you have not been taught in the course (except where such use is ill-advisable, erroneous, unnecessarily complicated or has serious performance implications), and no extra marks will be awarded in such cases.

Group work and Plagiarism

You are encouraged to talk to your colleagues about assessments but the code you submit must be your own. Clear similarities in code will be taken as evidence of group work. A plagiarism check may be carried out on submitted work, and (if considered necessary) you may be asked certain questions about your work to verify that you understand and are familiar with the work you have submitted. Plagiarism is taken extremely seriously and any student found to be guilty of plagiarism will be severely punished. It is better to submit nothing than to submit work that is not your own.

Help

The lab assistants may be able to answer any reasonable questions you have about C++ in general or about interpretation of the specification, but will not answer questions directly relating to your project code. You should therefore try to generalise any questions you have.

Reasonable requests for assistance via e-mail to **Adam.J.Hall [at] warwick.ac.uk** will be responded to where possible (time permitting), but you should not expect your project code to be tested, debugged or otherwise examined before submission. Any requests for extensions should NOT be directed to me.

Remember the usual debugging tricks that you learnt in the course (e.g. rebuild all, breakpoints, read the compiler messages, write to `std::cerr`)!

You are free to extend the specification in ways that have not been mentioned, provided of course that such extensions are reasonable.