Object Oriented Programming Course Notes

Felipe Balbi

May 15, 2020

Contents

Week 1 **5**

1.004 Demonstrating the first example application: merkelsim currency exchange simulation . . . 5
1.005 Demonstrating the second example application: otodecks DJ application . 5
1.006 Reading for Topic 1 . . . 5
1.101 Introduction to Topic 1 . . . 6
1.102 C++ is a low-level language . . . 6
1.105 Writing our first program . . . 6
1.107 Development environments . . . 6

Week 2 **7**

1.601 What is refactoring? . . . 7
1.603 Write a print menu function and a processOption function . . . 7
1.605 Write menu functions . . . 8

Week 3 **9**

2.001 Introduction to Topic 2 . . . 9
2.003 Reading for Topic 2 . . . 9
2.101 C++ is strongly typed . . . 10
2.103 How are floating points values represented? . . . 10
2.106 The order book data set . . . 11
2.301 Introduction to classes . . . 11
2.303 Defining our first class: OrderBookEntry . . . 11
2.305 Constructors: passing initialisation data into objects . . . 12
2.307 Constructor initialisation list . . . 12
2.309 Create a vector of objects . . . 13
2.311 Three ways to iterate over a vector . . . 14

Week 4 **15**

2.501 Convert OrderBookEntry to header and cpp . . . 15
2.503 Use the OrderBookEntry header in the main cpp and compile . . . 15
2.505 Create the Merkelmain class . . . 17
2.507 Implement the init function . . . 17
2.509 Limiting exposure . . . 17
2.511 Load orderBook function . . . 18

Week 5 **19**

3.003 Reading for Topic 3 . . . 19

3.101 The tokenise algorithm in pseudocode . . . 19
3.103 The tokenise algorithm in C++ . . . 19
3.105 Testing tokenise . . . 20
3.201 Open a file . . . 23
3.203 Read a file using getline . . . 25
3.205 Tokenise then translate the data into the correct format . . . 26
3.207 Dealing with exceptions . . . 28
3.209 Programming exercise . . . 29

Week 6 **31**
3.401 The plan . . . 31
3.403 Create the CSVReader class . . . 31
3.405 Make it compile . . . 32
3.407 Implement the OrderBookEntry making function . . . 33
3.409 Implement the tokenise and file parsing functions . . . 34
3.411 Integrate it into MerkleMain init function and compute some statistics . . . 36

Week 7 **38**
4.003 Reading for Topic 4 . . . 38
4.101 Introduction to separating orders by type, product and time . . . 38
4.103 Set up the OrderBook class . . . 39
4.105 Implement the constructor and integrate to MerkelMain . . . 39
4.107 Implement getKnownProducts with a map . . . 39
4.109 Implement getOrders . . . 40
4.111 Implement get low and high stats . . . 41
4.201 Introduction to how the simulation needs to work with time . . . 42
4.203 Setting up earliest time . . . 43
4.205 Moving through time . . . 44
4.207 Printing stats for current time window . . . 45

Week 8 **46**
4.401 How will we retrieve data from the user for an OrderBookEntry? . . . 46
4.403 Read a line from the user, part 1 . . . 46
4.405 Read a line from the user, part 2 . . . 46
4.407 Making it robust . . . 49
4.409 Introduction to inserting the order into the order book . . . 51
4.411 The code to insert the order into the order book . . . 51
4.502 Pseudocode . . . 53
4.504 Converting the pseudocode into C++ . . . 54
4.506 Preparing for testing . . . 57
4.508 Doing testing . . . 57

Week 9 **58**
5.003 Reading for Topic 5 . . . 58

5.101 The plan for the wallet 58
5.103 The Wallet class 58
5.105 Getting to a compile-ready state – linker error 59
5.107 Implement insert and contains functions 60
5.109 Implement print and remove functions 61
5.111 How much cash do we need for bids and asks? 62
5.113 Implement cash check on bids and asks 63
5.115 Integrate to the main app enterAsk 63
5.117 Complete the enter bid function 64
5.301 Complete the trade: the theory 65
5.303 Adding the username and special types 65
5.305 Putting the money in their wallet 68

Week 1

Key Concepts

- Write, compile and run a C++ program that prints messages to the console
- Use the standard library to do text I/O in the console
- Write and call simple functions

1.004 Demonstrating the first example application: merkelsim currency exchange simulation

Merklerex is a currency trading simulator. It allows us to work on a snapshot of real trading data downloaded from an exchange service. We can keep track of our wallet, trade currencies by buying and selling them and try to make "money".

1.005 Demonstrating the second example application: otodecks DJ application

OtoDecks is a DJ application. The second half of the course will be building this application. It allows us to load different tracks and mix them together on the fly.

1.006 Reading for Topic 1

The textbook for this course is:

Horton, I. and P. Van Weert Beginning C++17: From novice to professional. (Berkeley, CA: Apress, 2018) 5th edition [9781484233665].

We will point you to the specific sections of the textbook you should read in the lesson worksheets. In this week's worksheets, you will see the following sections:

- Chapter 2, p.38: std::cin
- Chapter 5, p.138: while loops
- Chapter 8, p.259: functions

Accessible from here.

1.101 Introduction to Topic 1

During topic 1 we learn to write, compile and run C++ programs.

1.102 C++ is a low-level language

But high-level and low-level we refer to the level of abstraction between the program we write and the HW it runs on.

When we write SW in a high-level language, such as JavaScript or Python, we don't need to worry much about how the underlying machine is carrying out its work. Details such as memory management is taken care of by the language itself.

Conversely, a low-level language, such as C/C++ or Assembly, give us access to very small details of the machine. We must handle memory manually, sometimes control specific machine registers and so on.

1.105 Writing our first program

In order to convert a C++ source code into an executable, a compiler will parse the C++ source and generate a target machine assembly, from there we generate object code which is later linked to produce the final executable.

```
g++ -S main.cpp  # produces assembly
g++ -c main.cpp  # produces object code
g++ main.cpp     # compile, assemble, and link
```

1.107 Development environments

During this course we will use an IDE, or an Integrated Development Environment, which is basically a text editor combined with all other tools needed during the development phase: a debugger, compiler, linker, file manager, console, and so on.

The course provides a web-based version of Visual Studio Code for our consumption. It's accessible from here.

Week 2

Key Concepts

- Write, compile and run a C++ program that prints messages to the console
- Use the standard library to do text I/O in the console
- Write and call simple functions

1.601 What is refactoring?

Refactoring is modifying code without modifying behavior. In other words, it's the practice of extracting "ideas" from code and splitting it into a separate function, either for aesthetics or so the code can be reused elsewhere.

1.603 Write a print menu function and a processOption function

With the knowledge of refactoring, we can refactor our big `main()` function by splitting it into smaller functions.

We introduce `printMenu()`, `getUserOption()`, and `processUserOption()`. Each of these functions is concerned with a single thing. They know the minimum amount of details they need to know.

For example, all the `std::cout` statements that make up our menu, can be moved to `printMenu()`.

```
1  void printMenu(void)
2  {
3      std::cout << "1. Show Help" << std::endl;
4      std::cout << "2. Stats" << std::endl;
5      std::cout << "3. Offer" << std::endl;
6      std::cout << "4. Bid" << std::endl;
7      std::cout << "5. Wallet" << std::endl;
8      std::cout << "6. Continue" << std::endl;
9  }
```

All that code can be removed from `main()` and replaced with a call to `printMenu()`. The same idea is applied to `getUserOption()` and `processUserOption()`.

1.605 Write menu functions

Following the same idea as the previous section, let's augment `processUserOption()` with smaller helper of its own. At the moment, these helper functions will only print out the relevant message but eventually more functionality will be added.

We have a total of 6 menu options and, therefore, will add 6 functions. They are:

1. `printHelp()`
2. `printMarketStats()`
3. `enterOffer()`
4. `enterBid()`
5. `printWallet()`
6. `gotoNextTimeFrame()`

Week 3

Key Concepts

- Select appropriate data types to represent a dataset in a C++ program
- Describe how a class can be used to combine multiple pieces of data into one unit
- Write a class with functions

2.001 Introduction to Topic 2

The main content of topic 2 is "data". We learn about data types and how C++ is a strongly typed language. We also learn about classes.

2.003 Reading for Topic 2

Your Essential reading textbook is:
Horton, I. and P. Van Weert Beginning CC++17: From novice to professional. (Berkeley, CA: Apress, 2018) 5th edition [9781484233665].

- Chapter 7 on p.219 introduces the std::string data type. Page 223 includes a summary of the various ways you can create std::string objects in C++.
- The textbook discusses enum class in Chapter 3, p.77. They use the example of days of the week.
- The textbook covers std::vector in Chapter 5, p.169. Note the difference between initialising with curly braces and normal braces, covered on p.170.
- The textbook discusses the syntax for a class on Chapter 11 p.386. It includes information about how to add function members to classes. Do not worry too much about that for now, we will get into that later!
- See Chapter 11 p.388 in the textbook for more information about constructors, including an explanation of why we were able to use the class even before we had written a constructor (i.e. because the compiler provides a default constructor).
- Member initialiser lists are covered in Chapter 11 p.394.
- Chapter 6, p.216 provides examples and further explanation about using references and range loops.

- Chapter 11 p.388 provides more reading about default constructors.

Accessible from here.

2.101 C++ is strongly typed

C++, much like C, is a strongly typed language. There is an urban legend floating around that the `++` in the language name means that something was added to it on top of C and because of it, it's better.

That, however, is far from the truth. What was added to the language is just countless extra bytes to support the broken idea of multiple-inheritance and operator overloading while also maintaining C++ developers employed. That is because once you write code in C++, it's impossible to debug it and, therefore, projects never come to completion, so developers remain employed, forever re-writing the same piece of code.

Also, note that C++ was never "invented" as it started as a set of pre-processor macros which ran before the C compiler and still produced C code. What this tells us is that whatever C++ can do, C can also achieve.

With that out of the way, let's continue with the regular schedule.

Here are some of the C++ types:

Group	Type Names

Character Types	`char`
	`char16+t`
	`char32_t`
	`wchar_t`

Integer Types (signed)	`signed char`
	`signed short int`
	`signed int`
	`signed long int`
	`signed long long int`

Integer Types (unsigned)	`unsigned char`
	`unsigned short int`
	`unsigned int`
	`unsigned long int`
	`unsigned long long int`

Floating-point Types	`float`
	`double`
	`long double`

Boolean Types	`bool`

2.103 How are floating points values represented?

Floating point numbers are represented in IEEE 754 format. For 32-bit variables (single precision floating point numbers) the format is:

SXXXXXXX XMMMMMMM MMMMMMMM MMMMMMMM

Where:

S Sign bit, 1 means negative and 0 means positive

X Exponent bits for a power-of-two

M Mantissa bits. There are 23 bits represented in memory, plus an implicit 1 bit making it the 24^{th} bit. All the explicitly represented bits are fraction bits.

The numbers are, then, of the form $(-1)^{Sign} \times 1.Mantissa \times 2^{Exponent}$.

2.106 The order book data set

The order book is composed of people trying to order things and people trying to sell things. We put those two sets of people together using a matching engine.

The dataset provided is a CSV file where each line is one entry.

Each entry is composed of a timestamp, a trading pair, a trade type, a price, and an amount.

2.301 Introduction to classes

A class is a way of organizing data and functions that are related to each other.

We have already used the standard library `std::string` and `std::vector` classes previously. Now let's define our own `OrderBookEntry` class.

2.303 Defining our first class: OrderBookEntry

To define a class we use the keyword `class` followed by the name of the class and an opening curly brace `{`. This follows with a `public:` keyword to tell the compiler that what follows should be visible for users of the class, then we put our fields.

Below we can see the current version of the class.

```
1 class OrderBookEntry {
2 public:
3     double price;
4     double amount;
5     std::string timestamp;
6     std::string product;
7     OrderBookType orderType;
8 };
```

2.305 Constructors: passing initialisation data into objects

Constructors help us pass the initial values of the properties of objects during instance creation.

The constructor is a function that has the same name as the class and initializes the class' properties from arguments passed into the constructor. Like below:

```
class OrderBookEntry {
public:
    OrderBookEntry(double price, double amount, std::string timestamp,
                   std::string product, OrderBookType orderType)
    {
        this->price = price;
        this->amount = amount;
        this->timestamp = timestamp;
        this->product = product;
        this->orderType = orderType;
    }

    double price;
    double amount;
    std::string timestamp;
    std::string product;
    OrderBookType orderType;
};
```

When creating an instance, we pass the arguments in curly braces, like so:

```
OrderBookEntry entry{5234.12341234, 0.00233432,
                     "2020/03/17 17:01:24.884492",
                     "BTC/USDT", OrderBookType::bid};
```

2.307 Constructor initialisation list

We can simplify our constructor by using an initialization list. Code would look like the one below.

```
class OrderBookEntry {
public:
    OrderBookEntry(double price, double amount, std::string timestamp,
                   std::string product, OrderBookType orderType)
        : price(price),
          amount(amount),
          timestamp(timestamp),
```

```
8             product(product),
9             orderType(orderType)
10    {
11    }
12
13    double price;
14    double amount;
15    std::string timestamp;
16    std::string product;
17    OrderBookType orderType;
18 };
```

2.309 Create a vector of objects

Now that we have our own class, we can combine our 5 vectors into a single vector of `OrderBookEntry` instances.

It would look like so:

```
1 std::vector<OrderBookEntry> entries;
2
3 OrderBookEntry entry{5234.12341234, 0.00233432,
4                      "2020/03/17 17:01:24.884492",
5                      "BTC/USDT", OrderBookType::bid};
6
7 entries.push_back(entry);
```

Assuming we have several entries on this vector, how can iterate over the vector and operate on each item? We can use a loop as shown below.

```
1 for (OrderBookEntry entry : entries) {
2     std::cout << "The price is " << entry.price << std::endl;
3 }
```

Note that the construct above will generate a copy of each entry for us to operate. This will prevent us from modifying our vector while operating on the entries with the added cost of extra overhead for the copying. If we don't want to copy, the change is simple:

```
1 for (OrderBookEntry& entry : entries) {
2     std::cout << "The price is " << entry.price << std::endl;
3 }
```

What the `&` states is that we will operate on references to each of the entries, rather than on copies.

2.311 Three ways to iterate over a vector

Here they are:

```
/* Iterators with copies or indexes */
for (OrderBookEntry entry : entries) {
    std::cout << "The price is " << entry.price << std::endl;
}

for (OrderBookEntry& entry : entries) {
    std::cout << "The price is " << entry.price << std::endl;
}

/* Array-style Indexing */
for (unsigned int i = 0; i < entries.size(); ++i) {
    std::cout << "The price is " << entries[i].price << std::endl;
}

/* Object-style Indexing */
for (unsigned int i = 0; i < entries.size(); ++i) {
    std::cout << "The price is " << entries.at(i).price << std::endl;
}
```

Week 4

Key Concepts

- Select appropriate data types to represent a dataset in a C++ program
- Describe how a class can be used to combine multiple pieces of data into one unit
- Write a class with functions

2.501 Convert OrderBookEntry to header and cpp

During this video we break the `OrderBookEntry` class into specification and implementation.

We start by creating two new files:

`OrderBookEntry.h` The header file where the specification will live.

`OrderBookEntry.cpp` The implementation file

We can move all the `OrderBookEntry` class to the header file, then remove the implementation of `OrderBookEntry()` constructor, this means deleting the initializer list and the curly braces `{}`.

After this is done, the header should look something like the one shown in 1.

And the implementation should look like the one shown in 2.

2.503 Use the OrderBookEntry header in the main cpp and compile

With the new class added, we need to tell the rest of the Software how to use it.

When compiling we need to include all necessary files. Instead of compiling with:

```
1 g++ --std=c++11 main.cpp
```

We need to also compile the new file:

```
1 g++ --std=c++11 main.cpp OrderBookEntry.cpp
```

Listing 1 Header File

```
#include <string>

enum class OrderBookType{bid, ask};

class OrderBookEntry {
public:
    OrderBookEntry(double price, double amount, std::string timestamp,
                   std::string product, OrderBookType orderType);

    double price;
    double amount;
    std::string timestamp;
    std::string product;
    OrderBookType orderType;
};
```

Listing 2 Implementation File

```
#include <OrderBookEntry.h>

OrderBookEntry::OrderBookEntry(double price, double amount,
                               std::string timestamp, std::string product,
                               OrderBookType orderType)
    :
    price(price),
    amount(amount),
    timestamp(timestamp),
    product(product),
    orderType(orderType)
{
}
```

2.505 Create the Merkelmain class

What would we put in a class to represent the entire application? Currently, in our `main.cpp` we have several flat functions that operate on the menu of the application.

We would like to package all of those functions into a new class which gets instantiated from the `main()` function in order to start the application.

Just like in previous section, we start by creating two new files: the header (`MerkleMain.h`) and the implementation file (`MerkleMain.cpp`).

We will move all the flat functions into a new class `MerkelMain`. With those done, we change our `main.cpp` to simply instantiate `MerkelMain` and call its `init()` method.

2.507 Implement the init function

To have a functional application again, we should implement the `init()` method in the `MerkelMain` class.

In summary, the code that was part of `main()` as a set of flat functions, will now be called from `MerkelMain::input()`. Like shown in 3.

Listing 3 `input()` method

```
1  MerkelMain::input()
2  {
3      int input;
4
5      while (true) {
6          printMenu();
7          input = getUserOption();
8          processUserOption(input);
9      }
10 }
```

2.509 Limiting exposure

We want to limit the amount of details from our class that's exposed to the outside. Ideally, we should expose only the bare minimum necessary to have the required functionality.

This means that there is a single function users need and that is `MerkelMain::init()` (well, and the constructor).

To limit exposure, we add the `private:` specifier on the functions.

2.511 Load orderBook function

Let's create a `loadOrderBook()` method in our `MerkelMain` class which will be responsible for loading data into the application.

This will be a `private` method to be called by `init()`.

After adding the function and calling it from `init()` we need to give it some data. We start with some sample static data for now. The next step is move the `orders` vector to the class itself as `private` member of the class. As a final step, we add `#include` guards to prevent the same header from being included twice. We achieve this by adding `#pragma once` to the beginning of the header file.

Week 5

Key Concepts

- Convert pseudocode algorithms involving iteration, logic and string processing into working C++ code
- Use exception handling to gracefully recover when processing unreliable data
- Read text data from a file using the getline function

3.003 Reading for Topic 3

Your Essential reading textbook is:
Horton, I. and P. Van Weert Beginning CC++17: From novice to professional. (Berkeley, CA: Apress, 2018) 5th edition [9781484233665].
Accessible from here.

- Exceptions
  - Chapter 15, including a try-catch example on p.577.
- Static functions
  - Chapter 11, p.423 For more information on File I/O in C++, we recommend this reference:

    'C++ files', w3schools (no date) `https://www.w3schools.com/cpp/cpp_files.asp` accessed 16 April 2020.

3.101 The tokenise algorithm in pseudocode

We're going to build a tokenizer for processing the CSV input file.
In summary, we walk the input stream character-by-character until we find a separator, which for is a comma `,` character.

3.103 The tokenise algorithm in C++

To make things simple, we start the tokenizer in a new, empty c++ source file. We start with including `string`, `vector`, and `iostream` which we will use.
The tokeniser, is a function that takes as input one line in CSV format and a separator character, and returns a vector of strings.

The function is essentially a `while` loop which runs for as long as there are characters to be consumed. Below the entire source code can be seen.

```
#include <iostream>
#include <string>
#include <vector>

std::vector<std::string> tokenise(std::string line, char separator)
{
    std::vector<std::string> tokens;
    signed int start;
    signed int end;

    start = line.find_first_not_of(separator, 0);

    do {
        std::string token;

        end = line.find_first_of(separator, start);

        if (start == line.length() || start == end)
            break;

        if (end >= 0)
            token = line.substr(start, end - start);
        else
            token = line.substr(start, line.length() - start);

        tokens.push_back(token);
        start = end + 1;
    } while(end > 0);

    return tokens;
}
```

3.105 Testing tokenise

Now that we have an implementation of a tokeniser, we should test it with a few different inputs to verify that it returns what we expect it to return.

We achieve that by calling the `tokenise()` function and printing out the resulting vector.

```
#include <iostream>
#include <string>
```

```cpp
#include <vector>

std::vector<std::string> tokenise(std::string line, char separator)
{
    std::vector<std::string> tokens;
    signed int start;
    signed int end;

    start = line.find_first_not_of(separator, 0);

    do {
        std::string token;

        end = line.find_first_of(separator, start);

        if (start == line.length() || start == end)
            break;

        if (end >= 0)
            token = line.substr(start, end - start);
        else
            token = line.substr(start, line.length() - start);

        tokens.push_back(token);
        start = end + 1;
    } while(end > 0);

    return tokens;
}

int main(void)
{
    std::vector<std::string> tokens;
    std::string s = "thing,thing1,thing2";

    tokens = tokenise(s, ',');

    for (std::string& tok : tokens) {
        std::cout << tok << std::endl;
    }

    return 0;
}
```

The code above yields this result:

```
thing
thing1
thing2
```

Which is exactly what we expected. To make sure this really works, let's test some possible corner cases:

```
#include <iostream>
#include <string>
#include <vector>

std::vector<std::string> tokenise(std::string line, char separator)
{
    std::vector<std::string> tokens;
    signed int start;
    signed int end;

    start = line.find_first_not_of(separator, 0);

    do {
        std::string token;

        end = line.find_first_of(separator, start);

        if (start == line.length() || start == end)
            break;

        if (end >= 0)
            token = line.substr(start, end - start);
        else
            token = line.substr(start, line.length() - start);

        tokens.push_back(token);
        start = end + 1;
    } while(end > 0);

    return tokens;
}

int main(void)
{
    std::vector<std::string> tokens;
```

```
36      std::string t1 = "t1,test1,test1";
37      std::string t2 = "t2,test2,test2,";
38      std::string t3 = "t3";
39
40      tokens = tokenise(t1, ',');
41
42      for (std::string& tok : tokens) {
43          std::cout << tok << std::endl;
44      }
45
46      tokens = tokenise(t2, ',');
47
48      for (std::string& tok : tokens) {
49          std::cout << tok << std::endl;
50      }
51
52      tokens = tokenise(t3, ',');
53
54      for (std::string& tok : tokens) {
55          std::cout << tok << std::endl;
56      }
57
58      return 0;
59  }
```

Which yields the expected results as follows:

```
t1
test1
test1
t2
test2
test2
t3
```

3.201 Open a file

Continuing with the same `tokenise()` function, let's open a file from which we will take our input to feed the tokeniser.

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4  #include <vector>
```

```
5
6  std::vector<std::string> tokenise(std::string line, char separator)
7  {
8      std::vector<std::string> tokens;
9      signed int start;
10     signed int end;
11
12     start = line.find_first_not_of(separator, 0);
13
14     do {
15         std::string token;
16
17         end = line.find_first_of(separator, start);
18
19         if (start == line.length() || start == end)
20             break;
21
22         if (end >= 0)
23             token = line.substr(start, end - start);
24         else
25             token = line.substr(start, line.length() - start);
26
27         tokens.push_back(token);
28         start = end + 1;
29     } while(end > 0);
30
31     return tokens;
32 }
33
34 int main(void)
35 {
36     std::vector<std::string> tokens;
37     std::ifstream file{"20200317.csv"};
38
39     if (!file.is_open()) {
40         std::cout << "Could not open file" << std::endl;
41         exit(-1);
42     }
43
44     file.close();
45
46     return 0;
47 }
```

3.203 Read a file using getline

Now it's time to iterate over the file line by line and see if we can read the correct output.

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

std::vector<std::string> tokenise(std::string line, char separator)
{
    std::vector<std::string> tokens;
    signed int start;
    signed int end;

    start = line.find_first_not_of(separator, 0);

    do {
        std::string token;

        end = line.find_first_of(separator, start);

        if (start == line.length() || start == end)
            break;

        if (end >= 0)
            token = line.substr(start, end - start);
        else
            token = line.substr(start, line.length() - start);

        tokens.push_back(token);
        start = end + 1;
    } while(end > 0);

    return tokens;
}

int main(void)
{
    std::vector<std::string> tokens;
    std::ifstream file{"20200317.csv"};
    std::string line;

    if (!file.is_open()) {
```

```
41          std::cout << "Could not open file" << std::endl;
42          exit(-1);
43      }
44
45      while(std::getline(file, line)) {
46          std::cout << line << std::endl;
47      }
48
49      file.close();
50
51      return 0;
52  }
```

3.205 Tokenise then translate the data into the correct format

To tokenise every line, we should call `tokenise()` on each line read from the file.

```
1   #include <fstream>
2   #include <iostream>
3   #include <string>
4   #include <vector>
5
6   std::vector<std::string> tokenise(std::string line, char separator)
7   {
8       std::vector<std::string> tokens;
9       signed int start;
10      signed int end;
11
12      start = line.find_first_not_of(separator, 0);
13
14      do {
15          std::string token;
16
17          end = line.find_first_of(separator, start);
18
19          if (start == line.length() || start == end)
20              break;
21
22          if (end >= 0)
23              token = line.substr(start, end - start);
24          else
25              token = line.substr(start, line.length() - start);
```

```
26
27          tokens.push_back(token);
28          start = end + 1;
29      } while(end > 0);
30
31      return tokens;
32  }
33
34  int main(void)
35  {
36      std::vector<std::string> tokens;
37      std::ifstream file{"20200317.csv"};
38      std::string line;
39
40      if (!file.is_open()) {
41          std::cout << "Could not open file" << std::endl;
42          exit(-1);
43      }
44
45      while(std::getline(file, line)) {
46          double amount;
47          double price;
48
49          tokens = tokenise(line, ',');
50
51          if (tokens.size() != 5) {
52              std::cout << "Bad line" << std::endl;
53              continue;
54          }
55
56          amount = std::stod(tokens[4]);
57          price = std::stod(tokens[3]);
58
59          for (std::string& tok : tokens) {
60              std::cout << tok << std::endl;
61          }
62      }
63
64      file.close();
65
66      return 0;
67  }
```

3.207 Dealing with exceptions

As it turns out, `std::stod()` can raise an exception if fed bogus data. If we want our application to actually work, we should handle these exceptions.

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

std::vector<std::string> tokenise(std::string line, char separator)
{
    std::vector<std::string> tokens;
    signed int start;
    signed int end;

    start = line.find_first_not_of(separator, 0);

    do {
        std::string token;

        end = line.find_first_of(separator, start);

        if (start == line.length() || start == end)
            break;

        if (end >= 0)
            token = line.substr(start, end - start);
        else
            token = line.substr(start, line.length() - start);

        tokens.push_back(token);
        start = end + 1;
    } while(end > 0);

    return tokens;
}

int main(void)
{
    std::vector<std::string> tokens;
    std::ifstream file{"20200317.csv"};
    std::string line;

```

```
40      if (!file.is_open()) {
41          std::cout << "Could not open file" << std::endl;
42          exit(-1);
43      }
44
45      while(std::getline(file, line)) {
46          double amount;
47          double price;
48
49          tokens = tokenise(line, ',');
50
51          if (tokens.size() != 5) {
52              std::cout << "Bad line" << std::endl;
53              continue;
54          }
55
56          try {
57              amount = std::stod(tokens[4]);
58              price = std::stod(tokens[3]);
59          } catch(std::exception& e) {
60              std::cout << "Bad float!" << std::endl;
61              continue;
62          }
63
64          for (std::string& tok : tokens) {
65              std::cout << tok << std::endl;
66          }
67      }
68
69      file.close();
70
71      return 0;
72  }
```

3.209 Programming exercise

Write some exception handling code to read in a CSV data file in a different format to the order book data. See if you can convert the fields in the data file into various types such as doubles, etc.

There are many sources of open CSV data, for example:

- 'London underground geographic maps/CSV', Wikimedia Commons (updated December 2019) `https://commons.wikimedia.org/wiki/London_Underground_geographic_maps/CSV` accessed 16 April 2020.

- European data portal (no date) `https://www.europeandataportal.eu/data/catalogues?locale=en` accessed 16 April 2020.

Week 6

Key Concepts

- Convert psudeocode algorithms involving iteration, logic and string processing into working C++ code
- Use exception handling to gracefully recover when processing unreliable data
- Read text data from a file using the getline function

3.401 The plan

Now that we have a working implementation of a simple CSV reader, we need to integrate it into the `merkelrex` codebase. As we see in the data, there are different data types on each row, which means we can simply use a vector.

We need a new class. By defining our custom class, we can group these 5 different data points into a concise object and hold all "rows" of our data into a single vector.

3.403 Create the CSVReader class

We start by creating the header file, `CSVReader.h`:

```
#pragma once

#include "OrderBookEntry.h"
#include <vector>
#include <string>

class CSVReader {
public:
    CSVReader();
    static std::vector<OrderBookEntry> readCSV(std::string csvFile);

private:
    static std::vector<std::string> tokenise(std::string csvLine, char separator);
    static OrderBookEntry stringsToOBE(std::vector<std::string> strings);
}
```

3.405 Make it compile

We have the declaration of everything we want to do, now we need to implement it in `CSVReader.cpp`:

```
#include "CSVReader.h"
#include "OrderBookType.h"

CSVReader::CSVReader()
{
}

std::vector<OrderBookEntry> CSVReader::readCSV(std::string csvFile)
{
    std::vector<OrderBookEntry> entries;

    return entries;
}

std::vector<std::string> CSVReader::tokenise(std::string csvLine, char separator)
{
    std::vector<std::string> tokens;

    return tokens;
}

OrderBookEntry CSVReader::stringsToOBE(std::vector<std::string> tokens)
{
    OrderBookEntry entry{1, 1, "", "", OrderBookType::bid};

    return entry;
}
```

Now we can tie this into `main.cpp`:

```
/* ... */
#include "CSVReader.h"
/* ... */

int main(void)
{
    CSVReader reader;
}
```

With that, we can try to compile our program.

3.407 Implement the OrderBookEntry making function

At this moment, we have an application that compiles, but doesn't do anything useful. We need to move more code from our test implementation to this new class.

We will start with `stringsToOBE()`.

```
#include <iostream>

/* ... */

OrderBookEntry CSVReader::stringsToOBE(std::vector<std::string> tokens)
{
    double amount;
    double price;

    if (tokens.size() != 5) {
        std::cout << "Bad line" << std::endl;
        throw std::exception{};
    }

    try {
        amount = std::stod(tokens[4]);
        price = std::stod(tokens[3]);
    } catch(const std::exception& e) {
        std::cout << "Bad float!" << std::endl;
        throw;
    }

    OrderBookEntry entry{price, amount, tokens[0], tokens[1],
            OrderBookEntry::stringsToOrderBookType(tokens[2])};

    return entry;
}
```

Now we need to declare and implement `stringToOrderBookType()`:

```
/* ... */

enum class OrderBookType {ask, bid, unknown};

/* ... */

class OrderBookEntry {
  /* ... */

```

```
10    static OrderBookType stringToOrderBookType(std::string type);
11
12    /* ... */
13  };
```

And the implementation:

```
1   /* ... */
2
3   OrderBookType OrderBookEntry::stringToOrderBookType(std::string type)
4   {
5       if (s == "ask")
6           return OrderBookType::ask;
7
8       if (s == "bid")
9           return OrderBookType::bid;
10
11      return OrderBookType::unknown;
12  }
```

3.409 Implement the tokenise and file parsing functions

Continuing with our refactoring, we work on `tokenise()` and `readCSV()`.

Starting with `tokenise()`, we get:

```
1   std::vector<std::string> CSVReader::tokenise(std::string line,
2                                                char separator)
3   {
4       std::vector<std::string> tokens;
5       signed int start;
6       signed int end;
7
8       start = line.find_first_not_of(separator, 0);
9
10      do {
11          std::string token;
12
13          end = line.find_first_of(separator, start);
14
15          if (start == line.length() || start == end)
16              break;
17
18          if (end >= 0)
19              token = line.substr(start, end - start);
```

```
20          else
21              token = line.substr(start, line.length() - start);
22
23          tokens.push_back(token);
24          start = end + 1;
25      } while(end > 0);
26
27      return tokens;
28  }
```

The only missing bit is `readCSV()`:

```
1   /* ... */
2
3   #include <fstream>
4
5   /* ... */
6
7   std::vector<OrderBookEntry> CSVReader::readCSV(std::string csvFilename)
8   {
9       std::vector<OrderBookEntry> entries;
10      std::ifstream csvFile{csvFilename};
11      std::string line;
12
13      if (!file.is_open()) {
14          std::cout << "Could not open file" << std::endl;
15          exit(-1);
16      }
17
18      while (std::getLine(csvFile, line) {
19              try {
20                  OrderBookEntry obe = stringsToOBE(tokenise(line, ','));
21                  entries.push_back(obe);
22              } catch(const std::exception& e) {
23                  std::cout << "Bad data" << std::endl;
24              }
25          }
26
27          return entries;
28  }
```

With that all done, we finally complete our `main()` function to use it:

```
1   /* ... */
2   #include "CSVReader.h"
```

```
/* ... */

int main(void)
{
    CSVReader reader;
    CSVReader::readCSV("20200317.csv");
}
```

3.411 Integrate it into MerkleMain init function and compute some statistics

To make the code better, we're going to move the code to load and read the CSV file from `main()` to the `init()` function part of the `MerkelMain` object.

We start by reverting our change in `main()`:

```
int main(void)
{
    MerkelMain app{};
    app.init();
}
```

Looking at the implementation for `init()` we notice that it calls `loadOrderBook()`:

```
void MerkelMain::init()
{
    loadOrderBook();
    int input;
    while (true) {
        printMenu();
        input = getUserOption();
        processUserOption(input);
    }
}
```

That function, `loadOrderBook()`, should be reading the CSV file but, currently, it contains hard-coded data we used for testing. Fixing that we get:

```
/* ... */

#include "CSVReader.h"

/* ... */

void MerkelMain::loadOrderBook()
```

```
{
    orders = CSVReader::readCSV("20200317.csv");
}
```

Continuing with this exercise we can modify `printMarketStats()` to print one extra statistical data, now that we're reading the CSV file:

```
MerkelMain::printMarketStats()
{
    std::cout << "OrderBook contains: " << orders.size() << " entries" << std::endl;
    unsigned int bids = 0;
    unsigned int asks = 0;

    for (OrderBookEntry& e : orders) {
        if (e.orderType == OrderBookType::ask)
            asks++;

        if (e.orderType == OrderBookType::bid)
            bids++;
    }

    std::cout << "OrderBook asks: " << asks << " bids: " << bids << std::endl;
}
```

Week 7

Key Concepts

- Write functions that calculate basic statistics by iterating over vectors of objects
- Use test data to evaluate the correctness of an algorithm
- Use exception handling to write robust user input processing code

4.003 Reading for Topic 4

Your Essential reading textbook is:
Horton, I. and P. Van Weert Beginning CC++17: From novice to professional. (Berkeley, CA: Apress, 2018) 5th edition [9781484233665].

- std::map: Chapter 19, p.730
- Operator overloading: Chapter 12 p.450
- Sorting: Chapter 19, p.755

Accessible from here

4.101 Introduction to separating orders by type, product and time

We're going to implement a matching engine which looks for the orders in the order book and matches them together. The first step would be to match by product, for example we can't match DOGE/BTC with BTC/USDT.

The idea is to collect the data by timestamp. In a particular timestamp there are potentially several bids and asks for a particular product. We use that to run our matching algorithm.

The simulation will, then, collect all the orders within a time window and process them, including our own bids and asks.

4.103 Set up the OrderBook class

Instead of poluting the `Merkelmain` class with a bunch of different methods for accessing the data we want, e.g. the data between a date interval, we will create a new class to act as a database allowing certain queries for data to be executed.

We will create a small class called `OrderBook` with a setup like below:

```
class OrderBook {
public:
    OrderBook(std::string filename);
    std::vector<std::string> getKnownProducts();
    std::vector<OrderBookEntry> getOrders(OrderBookType type,
                                          std::string product,
                                          std::string timestamp);
};
```

We can place that in a new header `OrderBook.h`.

4.105 Implement the constructor and integrate to MerkelMain

It's time to create the `OrderBook.cpp` file. First we integrate it into the `MerkelMain.h` file:

```
/* ... */
#include <OrderBook.h>
/* ... */

class MerkelMain {
    /* ... */
    OrderBook orderBook{"20200317.csv"};
    /* std::vector<OrderBookEntry> orders; should be removed */
};
```

We should also remove `loadOrderBook()` which will be done by `OrderBook` class.

4.107 Implement getKnownProducts with a map

The `getKnownProducts()` method should return a list of unique products observed in the order book. We make use of a `map` to map between a `string` and `bool` values.

Opening our `OrderBook.cpp` we make the following changes:

```
/* ... */
#include <map>
```

```
/* ... */

std::vector<std::string> OrderBook::getKnownProducts(void)
{
    std::vector<std::string> products;
    std::map<std::string, bool> prodMap;

    for (OrderBookEntry& e : orders) {
        prodMap[e.product] = true;
    }

    for (auto const& e: prodMap) {
        products.push(e.first);
    }

    return products;
}
```

To test things out, let's print the list of known products from `MerkelMain::printMarketStats()`:

```
void MerkelMain::printMarketStats(void)
{
    for (std::string const& p : orderBook.getKnownProducts()) {
        std::cout << "Product: " << p << std::endl;
    }
}
```

When running our application, we should get a list of unique known products like below:

```
Product: BTC/USDT
Product: DOGE/BTC
Product: DOGE/USDT
Product: ETH/BTC
Product: ETH/USDT
```

4.109 Implement getOrders

It's time to implement `getOrders()`. This method allows us to query the database and pull several orders from the order book based on some filters.

```
std::vector<OrderBookEntry> OrderBook::getOrders(OrderBookType type,
                                                 std::string product,
                                                 std::string timestamp)
```

```
{
    std::vector<OrderBookEntry> orders_sub;

    for (OrderBookEntry& e : orders) {
        if (e.orderType == type &&
            e.product == product &&
            e.timestamp == timestamp)
            orders_sub.push_back(e);
    }

    return orders_sub;
}
```

Then on `MerkelMain::printMarketStats()` we can call this to test it out:

```
void MerkelMain::printMarketStats(void)
{
    for (std::string const& p : orderBook.getKnownProducts()) {
        std::cout << "Product: " << p << std::endl;
        std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookType::ask,
                                                                  p,
                                                                  "2020/03/17 17:01:24.884
        std::cout << "Asks seen: " << entries.size() << std::endl;
    }
}
```

4.111 Implement get low and high stats

We want to find the low and high asks and bids on the platform, we will implement methods to help us with that.

Our decision is to have the `OrderBook` class generate statistics by itself. This decision goes in line with the Object Orientation Paradigm which wants objects to be able to operate upon themselves.

In the `OrderBook.h` header we declare two new functions `getLowPrice()` and `getHighPrice()` which will return the lowest and highest number.

```
/* ... */

class OrderBook {
public:
    /* ... */
    static double getHighPrice(std::vector<OrderBookEntry>& orders);
    static double getLowPrice(std::vector<OrderBookEntry>& orders);
    /* ... */
};
```

Then we implement it:

```
/* ... */
#include <cfloat>

double OrderBook::getHighPrice(std::vector<OrderBookEntry>& orders)
{
    double max = DBL_MIN;

    for (OrderBookEntry const& e : orders) {
        if (e.price > max)
            max = e.price;
    }

    return max;
}

double OrderBook::getLowPrice(std::vector<OrderBookEntry>& orders)
{
    double max = DBL_MAX;

    for (OrderBookEntry const& e : orders) {
        if (e.price < min)
            min = e.price;
    }

    return min;
}
```

For a quick test, we print it out from `printMarketStats()`:

```
/* ... */
std::cout << "Max ask: " << OrderBook::getHighPrice(entries) << std::endl;
std::cout << "Min ask: " << OrderBook::getLowPrice(entries) << std::endl;
/* ... */
```

4.201 Introduction to how the simulation needs to work with time

We're going to build the ability for the application to tick through time. The idea is that the application will start at the first time step and, when we're ready, it'll step to the next point in time.

4.203 Setting up earliest time

The first thing to do is implement `getEarliestTime()` in `OrderBook`. This function will return the first time step from the data file. In the header:

```
/* ... */

class OrderBook {
public:
    /* ... */
    std::string getEarliestTime(void);
    /* ... */
};
```

In the cpp:

```
std::string OrderBook::getEarliestTime(void)
{
    return orders[0].timestamp;
}
```

Now, `MerkelMain` needs to know about that, so we add a new attribute to our class in the header:

```
/* ... */
class MerkelMain {
public:
    /* ... */
private:
    /* ... */
    std::string currentTime;
};
```

and set it up in the cpp:

```
void MerkelMain::init(void)
{
    int option;
    currentTime = orderBook.getEarliestTime();
    /* ... */
}

/* ... */

void printMenu(void)
```

```
{
    /* ... */
    std::cout << "Current Time is: " << currentTime << std::endl;
    /* ... */
}
```

4.205 Moving through time

Next step we implement the time stepping functionality. We do that by, again, augmenting `OrderBook` class. In the header we declare a new function:

```
/* ... */

class OrderBook {
public:
    /* ... */
    std::string getNextTime(std::string timestamp);
    /* ... */
};
```

And we implement it in the cpp:

```
std::string OrderBook::getNextTime(std::string timestamp)
{
    std::string next_timestamp = "";

    for (OrderBookEntry const& e : orders) {
        if (e.timestamp > timestamp)
            next_timestamp = e.timestamp;
    }

    if (next_timestamp == "")
        next_timestamp = orders[0].timestamp;

    return next_timestamp;
}
```

Then we can finally implement `gotoNextTimeframe()`:

```
/* ... */

void MerkelMain::gotoNextTimeframe(void)
{
    std::cout << "Going to next time frame." << std::endl;
    currentTime = orderBook.getNextTime(currentTime);
}
```

4.207 Printing stats for current time window

Continuing from our previous topic, we will modify `printMarketStats()` to print statistics for the current time frame.

```
1  void MerkelMain::printMarketStats(void)
2  {
3      for (std::string const& p : orderBook.getKnownProducts()) {
4          std::cout << "Product: " << p << std::endl;
5          std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookType::ask,
6                                                                    p,
7                                                                    currentTime);
8      }
9      /* ... */
10 }
```

Week 8

Key Concepts

- Write functions that calculate basic statistics by iterating over vectors of objects
- Use test data to evaluate the correctness of an algorithm
- Use exception handling to write robust user input processing code

4.401 How will we retrieve data from the user for an OrderBookEntry?

To create a new `OrderBookEntry` we need five pieces of data, as shown in figure 1.

To simplify things, we can have the user type in the three required values as CSV and we will use our `CSVReader` class to parse the values and produce a new `OrderBookEntry` instance.

4.403 Read a line from the user, part 1

With the plan from previous section, we can start taking user input to generate a new `OrderBookEntry`.

In other words, we will implement `enterAsk()` and `enterBid()` functions.

```
1  void MerkelMain::enterAsk()
2  {
3      std::cout << "Make an ask: product,price,amount" << std::endl;
4      std::cout << "e.g. ETH/BTC,200,0.5" << std::endl;
5      std::string input;
6
7      /* ignore up to 'size-of-stream' characters stopping at the newline */
8      std::getline(std::cin, input);
9      std::cout << "You typed: " << input << std::endl;
10 }
```

4.405 Read a line from the user, part 2

Continuing from where we left off, we have to parse the user input and, for that, we will use the `CSVReader` class. Because we want to use our `tokenise()` function, we need to move it to the public space of the `CSVReader` class. After that we can call it.

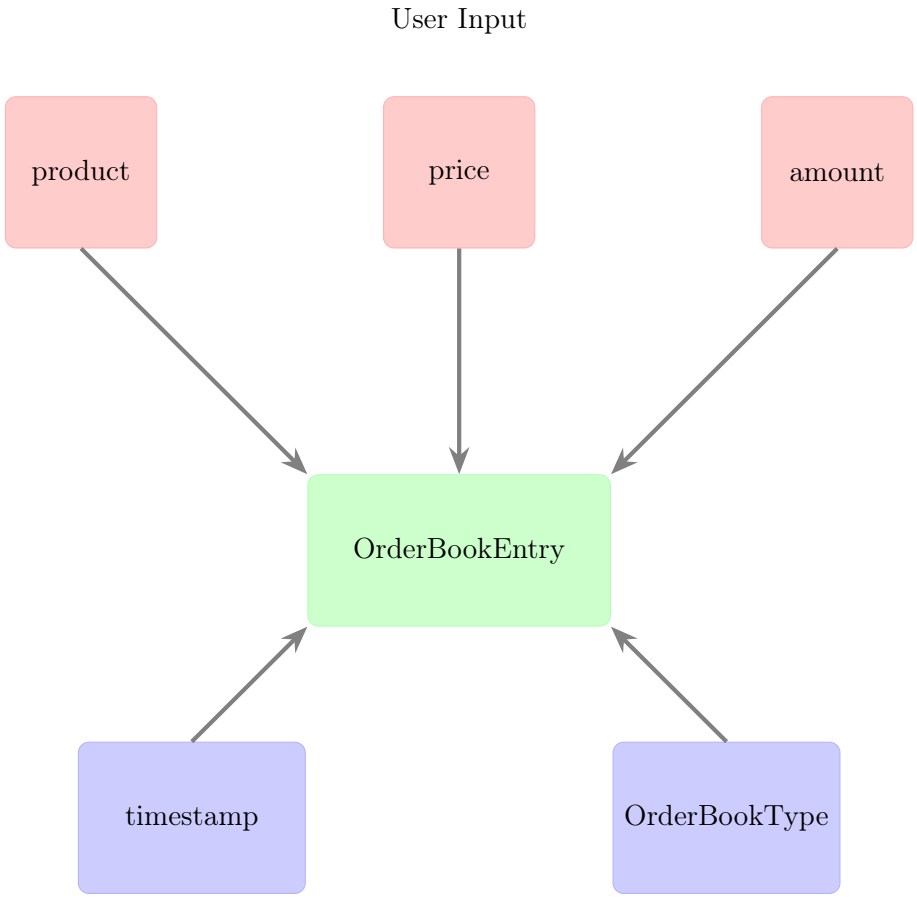

Figure 1: Creating a new OrderBookEntry

Before moving forward, we need to overload `stringsToOBE()` so we can create a new `OrderBookEntry` from the user input. Therefore:

```
/* ... */

class CSVReader {
public:
    /* ... */

    static OrderBookEntry stringsToOBE(std::string price,
                                       std::string amount,
                                       std::string timestamp,
```

```
                                        std::string product,
                                        OrderBookEntry orderType);

    /* ... */
};
```

We should implement it in `CSVReader.cpp`:

```
OrderBookEntry CSVReader::stringsToOBE(std::string priceString,
                                        std::string amountString,
                                        std::string timestamp,
                                        std::string product,
                                        OrderBookEntry orderType)
{
    double price;
    double amount;

    try {
        price = std::stod(priceString);
        amount = std::stod(amountString);
    } catch (const std::exception& e) {
        std::cout << "Bad float! " << priceString << std::endl;
        std::cout << "Bad float! " << amountString << std::endl;
        throw;
    }

    OrderBookEntry entry{price, amount, timestamp, product, orderType};

    return entry;
}
```

Now we can complete `enterAsk()`:

```
void MerkelMain::enterAsk()
{
    std::cout << "Make an ask: product,price,amount" << std::endl;
    std::cout << "e.g. ETH/BTC,200,0.5" << std::endl;
    std::string input;

    /* ignore up to 'size-of-stream' characters stopping at the newline */
    std::getline(std::cin, input);

    std::vector<std::string> tokens = CSVReader.tokenise(input, ',');
    if (tokens.size() != 3) {
        std::cout << "Bad input: " << input << std::endl;
```

```
13      } else {
14          OrderBookEntry entry = CSVReader::stringstoOBE(tokens[1],
15                                                         tokens[2],
16                                                         currentTime,
17                                                         tokens[0],
18                                                         OrderBookType::ask);
19      }
20  }
```

4.407 Making it robust

We've left a few bugs in the previous iteration which we're going to fix now.

If we pass bad input, the program may throw an exception which we need to catch. We will catch it when we try to create the OrderBookEntry after calling `stringsToOBe()`.

```
1   void MerkelMain::enterAsk()
2   {
3       std::cout << "Make an ask: product,price,amount" << std::endl;
4       std::cout << "e.g. ETH/BTC,200,0.5" << std::endl;
5       std::string input;
6
7       /* ignore up to 'size-of-stream' characters stopping at the newline */
8       std::getline(std::cin, input);
9
10      std::vector<std::string> tokens = CSVReader.tokenise(input, ',');
11      if (tokens.size() != 3) {
12          std::cout << "Bad input: " << input << std::endl;
13      } else {
14          try {
15              OrderBookEntry entry = CSVReader::stringstoOBE(tokens[1],
16                                                             tokens[2],
17                                                             currentTime,
18                                                             tokens[0],
19                                                             OrderBookType::ask);
20          } catch (const std::exception& e) {
21              std::cout << "MerkelMain::enter ask Bad input: " << std::endl;
22          }
23      }
24  }
```

This fixes one of the problems. However, there's still one more which happens if the user types garbage input on the menu selection. The culprit here is the fact that we're streaming from `std::cin` straight into a variable. Swapping that with a call to `std::getline()` should fix it.

```
int MerkelMain::getUserOption(void)
{
    int userOption = 0;
    std::string line;

    std::cout << "Type in 1-6" << std::endl;
    std::getline(std::cin, line);

    try {
        userOption = std::stoi(line);
    } catch (const exception& e) {
    }

    std::cout << "You chose: " << userOption << std::endl;

    return userOption;
}
```

When we switch over to `std::getline()` we don't need anymore to ignore input from the input stream. We should remove that line.

```
void MerkelMain::enterAsk()
{
    std::cout << "Make an ask: product,price,amount" << std::endl;
    std::cout << "e.g. ETH/BTC,200,0.5" << std::endl;
    std::string input;

    // std::getline(std::cin, input); Not needed anymore

    std::vector<std::string> tokens = CSVReader.tokenise(input, ',');
    if (tokens.size() != 3) {
        std::cout << "Bad input: " << input << std::endl;
    } else {
        try {
            OrderBookEntry entry = CSVReader::stringstoOBE(tokens[1],
                                                           tokens[2],
                                                           currentTime,
                                                           tokens[0],
                                                           OrderBookType::ask);
        } catch (const std::exception& e) {
            std::cout << "MerkelMain::enter ask Bad input: " << std::endl;
        }
    }
}
```

4.409 Introduction to inserting the order into the order book

Given a new `OrderBookEntry`, how do we insert it into the `OrderBook` in the correct position?

The way we're going to achieve this, is by appending the new entry to the end of the vector, then sorting the vector by the timestamp so it's in the correct location.

4.411 The code to insert the order into the order book

Now we have an `OrderBookEntry`, what we have to do now is insert it into the correct place. As discussed in previous section, we will insert it at the end of the vector, using `push_back()` and follow it with a call to `std::sort`.

We need to add a new method to the `OrderBook` which allows us to insert a new `OrderBookEntry`. In the header, we declare the new method:

```
/* ... */

class OrderBook {
public:
    /* ... */

    void insertOrder(OrderBookEntry& entry);

    /* ... */
};
```

And implement it in the cpp file:

```
/* ... */

void OrderBook::insertOrder(OrderBookEntry& entry)
{
    orderBook.push_back(entry);
    std::sort(orders.begin(), orders.end(),
              OrderBookEntry::compareByTimestamp);
}
```

Of course we need to implement our comparison function `compareByTimeStamp()`. Let's declare it in the `OrderBookEntry` class header:

```
/* ... */

class OrderBookEntry {
public:
```

```
    /* ... */

    static bool compareByTimestamp(OrderBookEntry& e1, OrderBookEntry& e2);
    {
        return e1.timestamp < e2.timestamp;
    }
};
```

Note that another way to achieve this would be a lambda expression, which is supported since C++11. The implementation for `insertOrder()` would look like so:

```
/* ... */

void OrderBook::insertOrder(OrderBookEntry& entry)
{
    orderBook.push_back(entry);
    std::sort(orders.begin(), orders.end(),
              [](const OrderBookEntry& e1, const OrderBookEntry& e2) {
                  return e1.timestamp < e2.timestamp;
              });
}
```

Anyway, back to the topic. In `MerkelMain` we need to call the new `insertOrder()` function:

```
void MerkelMain::enterAsk()
{
    std::cout << "Make an ask: product,price,amount" << std::endl;
    std::cout << "e.g. ETH/BTC,200,0.5" << std::endl;
    std::string input;

    // std::getline(std::cin, input); Not needed anymore

    std::vector<std::string> tokens = CSVReader.tokenise(input, ',');
    if (tokens.size() != 3) {
        std::cout << "Bad input: " << input << std::endl;
    } else {
        try {
            OrderBookEntry entry = CSVReader::stringstoOBE(tokens[1],
                                                           tokens[2],
                                                           currentTime,
                                                           tokens[0],
                                                           OrderBookType::ask);
            orderBook.insertOrder(entry);
        } catch (const std::exception& e) {
```

```
21              std::cout << "MerkelMain::enter ask Bad input: " << std::endl;
22          }
23      }
24  }
```

4.502 Pseudocode

The question now is how do we match a given ask to a given bid? There are many ways to go about solving this problem. The algorithm we're using is shown in listing 1.

Algorithm 1 Trade Simulation Matching Algorithm

```
1: function MatchingAlgorithm
2:     asks ← orderBook.asks
3:     bids ← orderBook.bids
4:     sales ← new Vector
5:     SortAscending(asks)
6:     SortDescending(bids)
7:     for all ask ∈ asks do
8:         for all bid ∈ bids do
9:             if bid.price ≥ ask.price then
10:                sale ← new Order
11:                sale.price ← ask.price
12:                if bid.amount = ask.amount then
13:                    sale.amount ← ask.amount
14:                    Append(sales, sale)
15:                    bid.amount ← 0
16:                    break
17:                end if
18:                if bid.amount > ask.amount then
19:                    sale.amount ← ask.amount
20:                    Append(sales, sale)
21:                    bid.amount ← bid.amount − ask.amount
22:                    break
23:                end if
24:                if bid.amount < ask.amount then
25:                    sale.amount ← bid.amount
26:                    Append(sales, sale)
27:                    ask.amount ← ask.amount − bid.amount
28:                    bid.amount ← 0
29:                    continue
30:                end if
31:            end if
32:        end for
33:    end for
34: end function
```

4.504 Converting the pseudocode into C++

Now we can convert the matching algorithm into `C++`. We're going to implement this functionality in the `OrderBook` class. This is because the `OrderBook` is already managing all the orders. It keeps track of time and so on.

As usual, we declare the new method in the header file:

```
/* ... */

class OrderBook {
public:
    /* ... */

    std::vector<OrderBookEntry> matchAsksToBids(std::string product,
                                                std::string timestamp);

    /* ... */
};
```

And and implement it in the cpp file:

```
std::vector<OrderBookEntry> OrderBook::matchAsksToBids(std::string product,
                                                       std::string timestamp)
{
    std::vector<OrderBookEntry> asks = getOrders(OrderBookType::ask, product,
                                                 timestamp);
    std::vector<OrderBookEntry> bids = getOrders(OrderBookType::bid, product,
                                                 timestamp);
    std::vector<OrderBookEntry> sales;

    std::sort(asks.begin(), asks.end(), OrderBookEntry::compareByPriceAsc);
    std::sort(bids.begin(), bids.end(), OrderBookEntry::compareByPriceDesc);

    for (OrderBookEntry& ask : asks) {
        for (OrderBookEntry& bid : bids) {
            if (bid.price >= ask.price) {
                OrderBookEntry sale{ask.price, 0, timestamp, product,
                                    OrderBookType::sale};

                if (bid.amount == ask.amount) {
                    sale.amount = ask.amount;
                    sales.push_back(sale);
                    bid.amount = 0;
                    break;
                }

                if (bid.amount > ask.amount) {
                    sale.amount = ask.amount;
                    sales.push_back(sale);
                    bid.amount -= ask.amount;
                    break;
```

```
                }

                if (bid.amount < ask.amount) {
                    sale.amount = bid.amount;
                    sales.push_back(sale);
                    ask.amount -= bid.amount;
                    bid.amount = 0;
                    continue;
                }
            }
        }
    }

    return sales;
}
```

All that's left is implement our new comparison functions in `OrderBookEntry` and the new type in `OrderBookType`:

```
/* ... */

enum class OrderBookType { ask, bid, sale, unknown };

/* ... */

class OrderBookEntry {
public:
    /* ... */

    static bool compareByPriceAsc(OrderBookEntry& e1, OrderBookEntry& e2)
    {
        return e1.price < e2.price;
    }

    static bool compareByPriceDesc(OrderBookEntry& e1, OrderBookEntry& e2)
    {
        return e1.price > e2.price;
    }

    /* ... */
};
```

4.506 Preparing for testing

We're going to prepare the code for some testing.

In `gotoNextTimeFrame()` we will try to match asks to bids:

```
1  void MerkelMain::gotoNextTimeFrame(void)
2  {
3      std::cout << "Going to next time frame." << std::endl;
4
5      std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids("ETH/BTC",
6                                                                    currentTime);
7
8      std::cout << "Sale: " << sales.size() << std::endl;
9      for (OrderBookEntry& sale : sales) {
10         std::cout << "Sale price: " << sale.price
11                   << " amount " << sale.amount
12                   << std::endl;
13     }
14
15     currentTime = orderBook.getNextTime(currentTime);
16 }
```

4.508 Doing testing

Now we have a setup where we can easily run some test scenarios, by simply changing the contents of our `test.csv` file. We can easily test the different scenarios by changing the amounts of ask and bid and verify that the algorithm behaves as it should.

Week 9

Key Concepts

- Use object interactions to achieve complex functionality through a simple command sequence
- Explain how to model a familiar real world entity as a class with data and functions
- Decide when it is appropriate to use static or non-static functions

5.003 Reading for Topic 5

Horton, I. and P. Van Weert Beginning CC++17: From novice to professional. (Berkeley, CA: Apress, 2018) 5th edition [9781484233665].

We introduce the relevant sections from the textbook in the worksheets, but here is a list of the chapters we mention:

- Chapter 15, p.573: details on throwing exceptions
- Chapter 11, p.411: friend functions
- Chapter 12, p.449: operator overloading
- Chapter 8, p.283: setting default parameter values.

Accessible from here.

5.101 The plan for the wallet

The wallet is a class which will manage the currencies we own. The first step before using the wallet is to initialize it with our starting fund. Another operation on the wallet is place a bid. And another is possibility to sell currencies we own. To top it off, we also have the concept of a trade.

5.103 The Wallet class

As usual, we start laying out the header file, `Wallet.h`:

```
#include <string>
#include <map>

class Wallet {
public:
    Wallet();
    void insertCurrency(std::string type, double amount);
    bool containsCurrent(std::string type, double amount);
    std::string toString();

private:
    std::map<std::string, double> currencies;
};
```

The reason we're using a `map` is that the wallet may, potentially, contain different currencies in different amounts. The `map` matches very well with this setup, where we build a list of currencies and their amounts.

5.105 Getting to a compile-ready state – linker error

After making our header, we can starting implementing the methods in `Wallet.cpp`:

```
#include "Wallet.h"

#include <string>
#include <map>

Wallet::Wallet(void)
{
}

void Wallet::insertCurrency(std::string type, double amount)
{
}

bool Wallet::containsCurrent(std::string type, double amount)
{
    return false;
}

std::string Wallet::toString(void)
{
    return "";
}
```

We must add a `Wallet` to the `MerkelMain` class. Starting with the header:

```
/* ... */

#include "Wallet.h"

/* ... */

class MerkelMain {
    /* ... */

private:
    Wallet wallet;
};
```

Then in `MerkelMain.cpp` we can put some money in the wallet from the `init()` method:

```
MerkelMain::init(void)
{
    int input;

    currentTime = orderBook.getEarliestTime();
    wallet.insertMoney("BTC", 10);

    /* ... */
}
```

5.107 Implement insert and contains functions

To make it easier to test, we're going to instantiate a wallet from the `main()` function.

```
/* ... */
#include "Wallet.h"

int main(void)
{
    Wallet wallet;
    wallet.insertCurrent("BTC", 10);

    std::cout << "Wallet has BTC "
              << wallet.containsCurrency("BTC", 10)
              << std::endl;

```

```
13      std::cout << wallet.toString() << std::endl;
14  }
```

We have a simple test case, now we can implement `insertCurrency()`:

```
1   void Wallet::insertCurrency(std::string type, double amount)
2   {
3       double balance;
4
5       if (amount < 0)
6           throw std::exception{};
7
8       if (currencies.count(type) == 0)
9           balance = 0;
10      else
11          balance = currencies[type];
12
13      balance += amount;
14      currencies[type] = balance;
15  }
```

Following, we implemnt `containsCurrency()`:

```
1   bool Wallet::containsCurrency(std::string type, double amount)
2   {
3       return currencies.count(type) > 0 ? currencies[type] >= amount : false;
4   }
```

5.109 Implement print and remove functions

We'll start making a better version of `toString()`:

```
1   std::string toString(void)
2   {
3       std::string str;
4
5       for (std::pair<std::string,double> pair : currencies) {
6           std::string currency = pair.first;
7           double amount = pair.second;
8
9           str += currency + " : " + std::to_string(amont) + "\n";
10      }
11
12      return str;
13  }
```

And continue with an implementation of `removeCurrency()`. First we declare it in the header file:

```
class Wallet {
public:
    /* ... */

    bool removeCurrency(std::string type, double amount);

    /* ... */
};
```

And implement it in the cpp file:

```
bool Wallet::removeCurrency(std::string type, double amount)
{
    if (amount < 0)
        return false;

    if (currencies.count(type) == 0)
        return false;

    if (!containsCurrency(type, amount))
        return false;

    currencies[type] -= amount;

    return true;
}
```

5.111 How much cash do we need for bids and asks?

We're going to calculate how much money we need for a given bid/ask.

The format for our input string is:

$$C_1/C_2,\text{price},\text{amount}$$

The formula for an ask is:

$$amountInWallet = C_1 \cdot amount$$

The formula for a bid is:

$$amountInWallet = C_2 \cdot price \cdot amount$$

5.113 Implement cash check on bids and asks

With our plan in place, we can start implementing it in C++. Let's add a new method to our `Wallet` class:

```
/* ... */

#include "OrderBookEntry.h"

/* ... */

class Wallet {
public:
    /* ... */

    bool canFullfillOrder(OrderBookEntry order);
};
```

And implement it:

```
bool Wallet::canFullfillOrder(OrderBookEntry order)
{
    std::vector<std::string> currs = CSVReader::tokenise(order.product, '/');

    if (order.orderType == OrderBookType::ask) {
        double amount = order.amount;
        std::string currency = currs[0];

        return containsCurrency(currency, amount);
    }

    if (order.orderType == OrderBookType::bid) {
        double amount = order.amount * order.price;
        std::string currency = currs[1];

        return containsCurrency(currency, amount);
    }

    return false;
}
```

5.115 Integrate to the main app enterAsk

Next step, we're integrating the wallet back into the main application to enable the user to place asks or bids and allow the system to check if their wallet contain enough money

for the transaction.

Let's modify `enterAsk()` to integrate the wallet checking features into it.

```
void MerkelMain::enterAsk(void)
{
    /* ... */
    else {
        try {
            OrderBookEntry obe = CSVReader::stringsToOBE(tokens[1],
                                                         tokens[2],
                                                         currentTime,
                                                         tokens[0],
                                                         OrderBookType::ask);
            if (wallet.canFullfillOrder(obe)) {
                orderBook.insertOrder(obe);
            } else {
                std::cout << "Insufficient funds" << std::endl;
            }
        } catch (const std::exception& e) {
            std::cout << " MerkelAsk::enterBid Bad input " << std::endl;
        }
    }

    /* ... */
}
```

5.117 Complete the enter bid function

At the moment, `enterBid()` doesn't do anything. Let's implement it:

```
void MerkelMain::enterBid(void)
{
    std::string input;

    std::getline(std::cin, input);

    std::vector<std::string> tokens = CSVReader::tokenise(input, ',');
    if (tokens.size() != 3) {
        std::cout << "MerkelMain::enterBid Bad input! " << input << std::end;
    } else {
        try {
            OrderBookEntry obe = CSVReader::stringsToOBE(tokens[1],
                                                         tokens[2],
                                                         currentTime,
```

```
                                                                tokens[0],
                                                                OrderBookType::bid);
            if (wallet.canFullfillOrder(obe)) {
                orderBook.insertOrder(obe);
            } else {
                std::cout << "Insufficient funds" << std::endl;
            }
        } catch (const std::exception& e) {
            std::cout << " MerkelMain::enterBid Bad input " << std::endl;
        }
    }
}
```

5.301 Complete the trade: the theory

We will now have a look at the requirements for completing a trade.

A trade is essentially a two-party process where both agree to exchange something for something else.

We already have sales as a separate `OrderBookType`, but we have no way of knowing who placed the order. The reason we need to know that is because they're going to come from two different places.

There's a simple solution to this problem which is to add a `username` field to the `OrderBookEntry`. For all the orders that are already in the dataset, we can have a default `username` of `dataset` and for the cases where the order came from user input, from the user interacting with the simulation, we can use a different `username`, perhaps `simuser`.

This raises another issue: we need to know who placed a `bid` and who placed an `ask`. A simple solution is to split our `sale` type into `bidsale` and `asksale` which would, therefore, allow us to know if the user referred to in the sale placed a bid or an ask.

5.303 Adding the username and special types

We're getting close to completing the simulation. Let's continue by adding the `username` field and the new special types. Starting in the `OrderBookEntry.h` header:

```
/* ... */

enum class OrderBookType { bid, ask, asksale, bidsale, unknown };

/* ... */

class OrderBookEntry {
public:
```

```
    OrderBookEntry(double price,
                   double amount,
                   std::string timestamp,
                   std::string product,
                   OrderBookType orderType,
                   std::string username = "dataset");
    /* ... */

    std::string username;
};
```

In `OrderBookEntry.cpp` we need to update the implementation:

```
OrderBookEntry::OrderBookEntry(double price,
                               double amount,
                               std::string timestamp,
                               std::string product,
                               OrderBookType orderType,
                               std::string username)
    :
    price(price),
    amount(amount),
    timestamp(timestamp),
    product(product),
    orderType(orderType)
    username(username)
{
}
```

Now we need to update `MerkelMain.cpp` for the two cases where the `username` should be different. Those are `enterAsk()` and `enterBid()`. Updating `enterAsk()` we get:

```
MerkelMain::enterAsk(void)
{
    /* ... */
    if (tokens.size() != 3) {
        std::cout << "MerkelMain::enterAsk Bad Input! " << input << std::endl;
    } else {
        try {
            OrderBookEntry obe = CSVReader::stringsToOBE(tokens[1],
                                                         tokens[2],
                                                         currentTime,
                                                         tokens[0],
                                                         OrderBookType::ask);
            obe.username = "simuser";
```

```
            /* ... */
        }
        /* ... */
    }
    /* ... */
}
```

We do the same to the `enterBid()` method:

```
MerkelMain::enterBid(void)
{
    /* ... */
    if (tokens.size() != 3) {
        std::cout << "MerkelMain::enterBid Bad Input! " << input << std::endl;
    } else {
        try {
            OrderBookEntry obe = CSVReader::stringsToOBE(tokens[1],
                                                         tokens[2],
                                                         currentTime,
                                                         tokens[0],
                                                         OrderBookType::bid);
            obe.username = "simuser";
            /* ... */
        }
        /* ... */
    }
    /* ... */
}
```

The next step, is to update the sale so we keep the information of who placed the order. Let's update `OrderBook.cpp` accordingly. Let's update `matchAsksToBids()`:

```
OrderBook::matchAsksToBids(std::string product, std::string timestamp)
{
    /* ... */

    for (OrderBookEntry& bid : bids) {
        for (OrderBookEntry& ask : asks) {
            if (bid.price >= ask.price) {
                OrderBookEntry sale { ask.price,
                                      0,
                                      timestamp,
                                      product,
                                      OrderBookType::asksale };

```

```
                if (bid.username == "simuser") {
                    sale.username = "simuser";
                    sale.orderType = OrderBook::bidsale;
                }

                if (ask.username == "simuser") {
                    sale.username = "simuser";
                    sale.orderType = OrderBook::asksale;
                }

                /* ... */

            }

            /* ... */

        }

        /* ... */

    }

    /* ... */
}
```

5.305 Putting the money in their wallet

And we've reached the final iteration of the development of MerkelRex. All that's missing is putting the money in the wallet of the user buying currencies.

The working of matching asks and bids is done from `MerkelMain::gotoNextTimeframe()`. If a user has placed a bid or an ask, at this moment, we have the results of the sales, which means we can, maybe, add funds to the user's wallet.

We already have code that iterates over all the matched sales, what we want to do now is that if the username in the sale is `simuser`, we will move money to/from the user's wallet accordingly.

```
void MerkelMain::gotoNextTimeframe(void)
{
    /* ... */

    for (std::string p : orderBook.getKnownProducts()) {
        std::vector<OrderBookEntry sales = orderBook.matchAsksToBids(p, currentTime);
        for (OrderBookEntry& sale : sales) {
            if (sale.username == "simuser") {
```

```
                wallet.processSale(sale);
            }
        }
    }

    /* ... */
}
```

Of course we need to declare the new method on `Wallet`:

```
/* ... */

class Wallet {
public:
    /* ... */

    void processSale(OrderBookEntry& sale);

    /* ... */
}
```

And implement it:

```
void Wallet::processSale(OrderBookEntry& sale)
{
    std::vector<std::string> currs = CSVReader::tokenise(sale.product, '/');

    if (sale.saleType == OrderBookType::asksale) {
        double outgoingAmount = sale.amount;
        std::string outgoingCurrency = currs[0];
        double incomingAmount = sale.amount * sale.price;
        std::string incomingCurrency = currs[1];

        currencies[incomingCurrency] += incomingAmount;
        currencies[outgoingCurrency] -= outgoingAmount;
    }

    if (sale.saleType == OrderBookType::bidsale) {
        double incomingAmount = sale.amount;
        std::string incomingCurrency = currs[0];
        double outgoingAmount = sale.amount * sale.price;
        std::string outgoingCurrency = currs[1];

        currencies[incomingCurrency] += incomingAmount;
        currencies[outgoingCurrency] -= outgoingAmount;
```

```
23          }
24      }
```