Object Oriented Programming Course Notes

Felipe Balbi

April 28, 2020

Contents

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:

```
std::vector<OrderBookEntry> entries;

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

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.

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

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:

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

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
```

```
#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 1: Header 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)
{
}
```

Listing 2: Implementation File

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.

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

Listing 3: `input()` method

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.