The Little CPPler

Todo list

idea: at some point in the code explain why is the book written in English and no any other language

idea: have a couple of exercises that could only be done in groups, and make students do an activity at the start and middle of the class with totally random people to solve those activities (talleres)

Chapter 1

What to expect from this book

A phrase telling them what to do with the book

This is NOT a reference book, this is a teaching book, it is for anybody to acquire experience writing code in C++.

Learn like you do, don't learn just to accumulate. Learning is rewarding, but teach/apply what you learn, from day one.

Chapter 2

Prerequisites

Tools

Tools neccesary to follow the book (for everything to work), note: the source code can be found in a repo, though

Using the tools

Ask to open a file with a text editor

Extend explanation on how to compile

ask to modify the file and see what happens

Given the file `000-hello-world.cc`:

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

compile the file into a executable with either `clang++` or `g++`:

add std option to make it C++11 compilant

```
clang++ 000-hello-world.cc -o 000-hello-world.exe
g++ 000-hello-world.cc -o 000-hello-world.exe
```

and run it with:

```
./000-hello-world.exe
```

the output of the program in terminal will be:

```
Hello World!
```

Chapter 3

How to read this book

Yeah, you should learn first how to read this book! :P

Explain how the book is supposed to be read (try to answer question with answer covered, then reveal answer and try understand what is it doing)

Chapter 4

Block 1: Basics

Getting to know C++(11)

Add interludes asking whoever is reading to pause for a while to recover from so much info

Add a space before footnotes

000.
What do you think the following code will output after compiling and running it?

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

The output is:

```
Hello World!
```

001.
Now, what if you compile this other file:

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  std::cout << "I'm a program example and I'm "
            << "in English."
            << std::endl;
  return 0;
}
```

The output is:

```
Hello World!
I'm a program example and I'm in English.
```

Pay close attention to the output, there are three sentences surrounded by quotation marks ("), but there are only two lines in the output. Why?

002.

If we change our example slightly (notice the semicolon (;)) what do you think it will happen?

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  std::cout << "I'm a program example and I'm "
            << "in English."
            << std::endl;
  return 0;
}
```

Well, it doesn't compiles! We get an error similar to:

```
:7:13: error: expected expression
            << "in English."
            ^
```

It is telling us that it was expecting something (a `std::cout` for example) before `<<`.
Try removing or adding random characters (anywhere) to the example and you will find that the compiler just admits a certain arrangement of characters and not much more. But, why? Well, the compiler just understands the grammar of C++ as we just understand the grammar of our human languages. Going a little further with the analogy, we can understand the grammar of any human language (its parts (verbs, prepositions, …) and how are they connected) but we can only understand the meaning (semantics) of those languages we have studied (or our mother tongues).

003.

Does the following program compiles. If yes, what is its output?

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  std::cout << "I'm a program"
            << std::endl
            << "example and I'm"
            << std::endl
            << "in English."
            << std::endl;
  return 0;
}
```

Yep, it in fact compiles, and its output is:

```
Hello World!
I'm a program
example and I'm
in English.
```

Notice how `std::endl` puts text in a new line, that's in fact its whole job.

004.

Well that's getting boring. What if we try something different for a change. What is the otput of this program:

```
#include <iostream>

int main()
{
  std::cout << "Adding two numbers: "
            << 2 + 3
            << std::endl;
  return 0;
}
```

Nice[†]

```
Adding two numbers: 5
```

[†]this is a footnote, read all of them, they may tell you little things that the main text won't.

005.
Let's try something a little more complex[†]

```
std::cout
  << "A simple operation between " << 3
  << " " << 5 << " " << 20 << ": "
  << (3+5)*20 << std::endl;
```

```
A simple operation between 3 5 20: 160
```

[†]Here you can see only a snippet of the whole code. The complete code the snippet represents can be found in the source accompaning this book.

From now on, all code will be given on snippets for simplicity but remember that they are that, snippets, uncomplete pieces of code that need your help to get complete.

006.
What is the purpose of `<< " " <<` in the code?

`<< " " <<` adds an space between the numbers otherwise the output would look weird.

007.
What if we remove all spaces from the last example?

```
std::cout
  << "A simple operation between " << 3
  << 5 << 20 << ": "
  << (3+5)*20 << std::endl;
```

```
A simple operation between 3520: 160
```

It looks aweful, doesn't it? Spaces are important as formatting!

008.
Let's try it now multiline:

```
std::cout
  << "A simple operation between " << std::endl
  << 3 << std::endl
  << 5 << std::endl
  << 20 << ": " << std::endl
  << (3+5)*20 << std::endl;
```

```
A simple operation between
3
5
20:
160
```

009.
Did you noticed that we only used a `std::cout`?
What is then the output of the code below?

```
std::cout
  << "A simple operation between " << std::endl;
std::cout << 3 << std::endl;
std::cout << 5 << std::endl;
std::cout << 20 << ": " << std::endl;
std::cout << (3+5)*20 << std::endl;
```

```
A simple operation between
3
5
20:
160
```

Yeah, it's the same as before!
Notice how the semicolon (`;`) indicates the ending of a statement in code. The code above could all be written in a single line (and not in 6 lines) and it would output the same:[†]

```
std::cout << ... << (3+5)*20 << std::endl;
```

Remember every `std::cout` is always paired with a semicolon (`;`) which indicates the ending of its effects, like a *dot* indicates the ending of a sentence or paragraph.

[†]sorry, the single line is too long to show in here all at once.

010.
What happens if you try to compile and run this faulty code?

```
std::cout
  << "A simple operation between " << std::endl;
std::cout << 3 << std::endl;
std::cout << 5 << std::endl
std::cout << 20 << ": " << std::endl;
std::cout << (3+5)*20 << std::endl;
```

It fails to compile because there is a semicolon missing in the code!
The error shown by the compiler is actually† useful here, it is telling us that we forgot a ;!

```
:8:30: error: expected ';' after expression
  std::cout << 5 << std::endl
                               ^
                               ;
1 error generated.
```

†Why "actually"? Well, you will find that most of the time errors thrown by the compiler are hard to understand. It is often something that we programmers need to learn to do. We learn to understand the confusing error messages compilers give us.

011.
But what if we want to not input 5 or 20 twice?
What is the output of the code below?
remember to explain how to initialize using {}

```
int num_1 = 3;
int num_2 = 5;
int num_3 = 20;

std::cout
  << "A simple operation between "
  << num_1 << " "
  << num_2 << " "
  << num_3 << ": "
  << (num_1+num_2)*num_3
  << std::endl;
```

The output is:

```
A simple operation between 3 5 20: 160
```

012.
what is the output if you change the value 5 for 7?

The output is:

```
A simple operation between 3 7 20: 200
```

013.
`num_1` is a **variable** and it allows us to save integers on it, you can try changing it's value for any number between -2147483649 and 2147483648†
What if we put -12 in the variable `num_1`?

†This numbers are based on a program compiled for a 32bit computer, the real values dependent between different computers.

The output is:

```
A simple operation between -12 5 20: -140
```

Interlude: Variables

Now, it's time to explain what are **variables** and what happens when we write `int name = 0`.

`int name = 0` is equivalent to:[1]

```
int name;
name = 0;
```

The first instruction **declares** a space for an `int` in memory (RAM memomry).

So, let's study the architecture of computers, mainly RAM and CPU.

ADD explanation on the architecture of computers

`int name;` then is telling the compiler to reserve (*declare*) some space that nobody else should use. This space can have any value we want. Because this space in memory could have been used by anybody else in the past, its value is *nondefined*, meaning that it can be anything. Therefore we use the next line `name = 0;` to save a zero in the space declared.

BEWARE! `name = 0;` is NOT an equation!

I repeat, `name = 0;` is NOT an equation!, you are **assigning** a value to a variable, you could easily assign many different values to a variable, though just the last one will stay in memory:

```
int name;
name = 0;
name = 12;
```

The procedure of *declaring* and then *assigning* a value to a variable is so common that the designers of the language have made a shortcut:

```
int name = 0;
```

Now, let's go back to the code!

[1] only for the simplest values `int`, `double`, ..., but not for objects. Objects are out of the scope of this book, but it is important to know they exist.

014.
What is the output of:

```
int var1 = 6;
int var2 = 3;
int var3 = 10;

std::cout
  << (var1+var2) * var3 - var2
  << std::endl;
```

The output is:

```
87
```

015.
What is the output of:

```
int var1 = 6;
int var2 = 3;

std::cout
  << (var1+var2) * var3 - var2
  << std::endl;

int var3 = 10;
```

Yeah, it doesn't compile, you are trying to use a variable *before* declaring it (asking for a space in memory to use it). The compiler gives you the answer:

```
:9:20: error: use of undeclared identifier 'var3'
  << (var1+var2) * var3 - var2
                   ^
1 error generated.
```

ORDER (of sentences) is key! It is not the same to say "Peter eats spaguetti, then Peter clean his teeth" than "Peter clean his teeth, then Peter eats spaguetti".
A program runs sequentially from the first line of code to the last

016.
What is the output of:

```
int var1 = 6;
int var2 = 3;
int var3 = 10;

var2 = var1*3;

std::cout
  << (var1+var2) * var3 - var2
  << std::endl;
```

The output is:

```
222
```

We can in fact assign to the variable (at the left of `=`) any `int` value result of any computation. In this case, the computation `var1*3` is assigned to `var2`

017.
What is the output of:[†]

```
#include <cmath>
...
int var1 = 6;
int var2 = 3;
int var3 = var1 * pow(var2, 3);

std::cout << "var1 * pow(var2, 3) => "
          << var3 << std::endl;
```

[†]Notice the dots (...) in the code above. This dots are just for notation, they aren't meant to be written in the source code. The dots represent a division between two different parts of code.

Add whole file here

The output is:

```
var1 * pow(var2, 3) => 162
```

Remember that
`int var3 = var1 * pow(var2, 3);`
is equivalent to

```
int var3;
var3 = var1 * pow(var2, 3);
```

018.

Till now we've seen just two operations (`*` and `+`), but there are plenty more:

```
int var1 = (2 + 18 - 6 * 2) * 5;
int var2 = var1 / 3;
int var3 = var1 % 3;

std::cout << "var1 => " << var1 << std::endl;
std::cout << "var2 => " << var2 << std::endl;
std::cout << "var3 => " << var3 << std::endl;
```

The output is:

```
var1 => 40
var2 => 13
var3 => 1
```

You probably know all operations here, but maybe not `%`. It is the *modulus* operation, or residue operation, and it symbolises the result of the residue of dividing integer numbers. For example, the result of dividing 50 by 3 can be written as:

$$50 = 3 \times 16 + 2$$

Where 50 is the dividend, 3 is the divisor, 16 the (integer) result, and 2 the modulus/remainder.
If an integer is divisible by another then the modulus of operating them must be 0, e.g., $21 = 7 \times 3 + 0$.

019.

What is the output of:

```
std::cout << 8 % 3 << " "
          << 8 % 2 << " "
          << 7 % 2 << " "
          << 17 % 1 << " "
          << 17 % 7 << " "
          << std::endl;
```

The output is:

```
2 0 1 0 3
```

020.

```
std::cout << 20 - 6 * 2 << " "
          << (20 - 6) * 2 << " "
          << 20 - (6 * 2) << " "
          << std::endl;
```

```
8 28 8
```

Notice how `20 - 6 * 2` does reduce[†] to 8 and not to 28! (i.e., $20-6\times2 = 20-(6\times2) \neq (20-6)\times2$). Each operator has a specific precedence that indicates if it must be applied before another operator, `*` for example has a higher precedence than `+`.

[†] or compute

---

add exercises about all of this

add one exercise that ask students to write a program with a single compilation error and ask another to find it and correct it

---

021.
What is the output of:

```
double acction   = 9.8; // m/s^2 acceleration
double mass      = 3;   // kg    mass
double initial_v = 10;  // m/s   ini. velocity
double time      = 2.3; // s     time passed

double final_v = initial_v + acction * time;

std::cout << "Velocity after " << time
          << "s is: " << final_v << "m/s"
          << std::endl;

double momentum = mass * final_v;

std::cout << "Momentum after " << time
          << "s is: " << momentum << "kg*m/s"
          << std::endl;
```

The output is:

```
Velocity after 2.3s is: 32.54m/s
Momentum after 2.3s is: 97.62kg*m/s
```

We are using here the equation $v = u + a * t$ to determine the final velocity of an object (in a line) after $2.3s$. The object starts with a velocity $2.3m/s$, has a constant acceleration of $9.8m/s^2$[†], and we know the weight of the object, so we can calculate too its momentum.

[†] free fall ;)

022.
What is the output of:

```
int var1 = 8;
if (var1 < 10) {
  std::cout << "var1 smaller than 10"
            << std::endl;
} else {
  std::cout
    << "var1 greater than or equal to 10"
    << std::endl;
}
```

The output is:

```
var1 smaller than 10
```

023.
What is the output of the code above if we change `var1`'s assignment from `8` to `19`?

The output is:

```
var1 greater than or equal to 10
```

Precisely, the second line runs but not the first.
The structure:

```
if (statement) {
  true branch
} else {
  false branch
}
```

runs the `true branch` if the statement `statement` evaluates to `true` otherwise it runs the `false branch`.

024.
What about the output of

```
int var1 = 10;
if (var1 == 7 + 3) {
  std::cout << "var1 is equal to 7 + 3"
            << std::endl;
} else {
  std::cout << "var1 does not equal 10"
    << std::endl;
}
```

Well, that's easy:

```
var1 is equal to 7 + 3
```

Note, `==` is the equality operation[†], and it compares any two statements as `var1` or `3+2*12` for equality.
As you expect it, there are many operations to compare between different statements, these are `==`, `<=`, `>=`, `<`, `>`, and `!=`.

[†] take care, don't confuse it with the assignment operator `=`!

025.

What is the output of:

```
int var1 = 2*2+2;
if (var1 != 7 + 3) {
  std::cout << "2*2+2 != 7+3" << std::endl;
} else {
  std::cout << "2*2+2 == 7+3" << std::endl;
}
```

The output is:

```
2*2+2 != 7+3
```

Yeah, `!=` is the unequality operator.

026.

What is the output of:

```
int var1 = 2*2+20;
if (var1 >= 7 + 3) {
  std::cout << "i never run :(" << std::endl;
} else {
  if (var1 > 23) {
    std::cout << "hey!!" << std::endl;
  } else {
    // nothing in this branch
  }
}
```

The output is:

```
i never run :(
```

Usually, if we only care about the true branch of an `if` statement, then we simply ignore it. The code at left is equivalent then to:

```
int var1 = 2*2+20;
if (var1 >= 7 + 3) {
  std::cout << "i never run :(" << std::endl;
} else {
  if (var1 > 23) {
    std::cout << "hey!!" << std::endl;
  }
}
```

027.

It's possible to put more than one statement inside the `if` statement. For example:

```
int var1 = 2*2+20;
if (var1 >= 7 + 3) {
  std::cout
    << "you are our visitor number 3889"
    << std::endl;
  int magicnumber = 999999;
  std::cout
    << "and your our winner! please "
    << "deposit " << magicnumber
    << " into our account and you will "
    << " receive 20x what you deposited "
    << std::endl;
}
std::cout
  << "Welcome to e-safe-comerce"
  << std::endl;
```

The output as you expected.

```
you are our visitor number 3889
and your our winner! please deposit 999999 into our acc
Welcome to e-safe-comerce
```

028.

```
int magicnumber = -2;
if (2!=3) {
  magicnumber = 0;
} else {
  magicnumber = 42;
}
std::cout << "Every number is a divisor of "
          << magicnumber << std::endl;
```

Of course, 0 can be divided by any number, because it can be written in the form $0 = k * n$ where $k = 0$ and n is an arbitrary number.

```
Every number is a divisor of 0
```

Chapter 5

Block 2: More space to play with

Chapter 6

Block 3: deprecated stuff that you better know cos everybody uses it