

# The Little CPler



# Todo list

idea: at some point in the code explain why is the book written in English and no any other language . . . . .	3
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) . . . . .	3
A phrase telling them what to do with the book . . . . .	5
Tools necessary to follow the book (for everything to work), note: the source code can be found in a repo, though . . .	7
Ask to open a file with a text editor . . . . .	7
Extend explanation on how to compile . . . . .	7
ask to modify the file and see what happens . . . . .	7
add std option to make it C++11 compilant . . . . .	7
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) . . . . .	9
Add interludes asking whoever is reading to pause for a while to recover from so much info . . . . .	11
Add a space before footnotes . . . . .	11
remember to explain how to initialize using {} . . . . .	14
ADD explanation on the architecture of computers . . . . .	15
mm..., doesn't sound that good, rewrite . . . . .	17
Add whole file here . . . . .	17
add exercises about all of this . . . . .	18
add one exercise that ask students to write a program with a single compilation error and ask another to find it and correct it . . . . .	18
explain at some point what do you mean by precision, maybe another interlude could be helpful here . . . . .	22
fill me! . . . . .	25
add exercises using <code>while</code> . . . . .	28
ask them first exercise from project euler, and reference it . . . . .	32
ask them to give us the sequence of fibonacci to an arbitrary number, using loops and variables on some way (this is the exercise that I remember the clearest doing as I took the class, it was so gratifying to solve the puzzle) . . .	32

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 necessary 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;
}
```

```
: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 compile. 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;
}
```

```
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 output of this program:

```
#include <iostream>

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

    return 0;
}
```

```
Adding two numbers: 5
```

Nice<sup>†</sup>

<sup>†</sup>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<sup>†</sup>

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

A simple operation between 3 5 20: 160

<sup>†</sup>Here you can see only a snippet of the whole code. The complete code the snippet represents can be found in the source accompanying 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 awful, 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:<sup>†</sup>

```
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.

<sup>†</sup>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<sup>†</sup> useful here, it is telling us that we forgot a ;!

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

<sup>†</sup>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  $-2147483648$  and  $2147483647$ <sup>†</sup>

What if we put  $-12$  in the variable `num_1`?

The output is:

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

<sup>†</sup>This numbers are based on a program compiled for a 32bit computer, the values may vary between different computers.

## 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:<sup>1</sup>

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

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

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!

---

<sup>1</sup>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 spaghetti, then Peter clean his teeth” than “Peter clean his teeth, then Peter eats spaghetti”.

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:

```
int var1 = 6;
std::cout << var1 << " ";
var1 = 20;
std::cout << var1 << " ";
var1 = -5;
std::cout << var1 << std::endl;
```

The output is:

6 20 -5



018.

What is the output of:

```
int var1 = 6;
std::cout << var1 << " ";
int var1 = 20;
std::cout << var1 << " ";
var1 = -5;
std::cout << var1 << std::endl;
```

It doesn't compile because you cannot ask for more space in memory (declare) with the same name variable, you gotta use a different name.<sup>†</sup>

<sup>†</sup>this will be truth until we learn how to "shadow" a variable name with help of scopes

mm..., doesn't sound that good, rewrite

019.

What is the output of:<sup>†</sup>

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

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

<sup>†</sup>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);
```

020.

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.,  $21 = 7 \times 3 + 0$ .

021.

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
```

022.

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

```
8 28 8
```

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

<sup>†</sup>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

023.

What is the output of:<sup>†</sup>

```
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;
```

<sup>†</sup>`double` allows us to declare “real” numbers (they are actually rational). AND we can operate with them as we did with `int`’s

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<sup>2</sup><sup>†</sup>, and we know the weight of the object, so we can calculate too its momentum.

<sup>†</sup>free fall ;)

024.

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
```

025.

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**.

026.

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<sup>†</sup>, 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 `!=`.

<sup>†</sup>take care, don't confuse it with the assignment operator `=`!

027.

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.

028.

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;
    }
}
```

029.

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-commerce"
    << 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-commerce
```

030.

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

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

```
Every integer is a divisor of 0
```

031.

What is the output of:

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

It doesn't compile! You wanna know why? Well, keep guessing with the following exercises.

032.

Does this compile? If yes, then what is its output?

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

```
Every integer is a divisor of -2
```

But wait, what? -2 is not always divisible by anyother integer!

It does compile, and its output is:

033.

Maybe, 2 and 3 are really the same thing.

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

The output is:

```
Every integer is a divisor of -2
```

But is the same, why? What have we changed from before? What is the difference with the code that gives us a zero?

034.

Ok, let's stop with the silliness and try with an example that actually give us some clue of the situation.

```
int num = 20;
std::cout << num << " ";
{
    int num = 42;
    std::cout << num << " ";
    num = 3;
    std::cout << num << " ";
}
std::cout << num << " ";
num = 0;
std::cout << num << std::endl;
```

Well, the code inside the brackets ({} ) acts as if it was being run alone without the intervention of the code of the outside.

```
20 42 3 20 0
```

It is effectively as if this was the code being run:

```
int num = 20;
std::cout << num << " ";
{
    int var = 42;
    std::cout << var << " ";
    var = 3;
    std::cout << var << " ";
}
std::cout << num << " ";
num = 0;
std::cout << num << std::endl;
```

035.

Any time we enclose code between brackets ({} ) we are defining a new **scope**. Any variable we declare inside a scope lives only in that scope, the variable *dies* once the scope is closed, is this the reason why this code

```
{
    int num = 42;
    std::cout << num << " ";
}
std::cout << num << std::endl;
```

You guessed it right! Given that **num** is a variable outside the inner scope, the inner scope can read and modify the variable content.

```
42 1 1
```

doesn't compile. There is no **num** variable in the bigger scope when it wants to show it.

A rule about scopes is that they can access to variables from the outside, scopes that enclose them. For example this code, does indeed compile:

```
int num = 42;
std::cout << num << " ";
{
    num = 1;
    std::cout << num << " ";
}
std::cout << num << std::endl;
```

What is its output?

036.

What is the output of:

```
int num = 42;
std::cout << num << std::endl;
{
    int num = 1;
    std::cout << num << std::endl;
}
std::cout << num << std::endl;
{
    num = 1;
    std::cout << num << std::endl;
}
std::cout << num << std::endl;
```

```
42
1
42
1
1
```

Right, in the first inner scope, we declare a new space and shadow the access to the outside variable `num`, and in the other inner scope, we use the variable accessible from the outside scope.

037.

What is the output of:<sup>†</sup>

```
int a = 20;
int b = 21;
if (b-a > 0) {
    float num = -12.2;
    std::cout << num * 3 << std::endl;
} else {
    double num = -12.2;
    std::cout << num * -3 << std::endl;
}
```

The output is:

```
-36.6
```

<sup>†</sup>yet again another type of data. `floats` are like `doubles` but they represent numbers with less precision. Well I haven't talked what precision is, forgive me, for the time being just assume that using `double` in your code is better than using `float`.

explain at some point what do you mean by precision, maybe another interlude could be helpful here

038.

What does this output:

```
int n = 15;
bool either = n<3;
if (either) {
    std::cout << "P" << std::endl;
} else {
    std::cout << ":( " << std::endl;
}
```

```
:(
```

The output is:

with `bool`<sup>†</sup> we tell the compiler that it should interpret `either` as a boolean (either `true` or `false`).

This means that you can in fact store integer values inside a `bool` (i.e., `bool bad = 23;`) but it's consider bad practice and may result in undefined behavior.

<sup>†</sup>True and False are the only two possible values that a statement can take in traditional logic, and so it does for us, there are only two options for `bool`. But modern computers are build on blocks of 32 or 64 bits and operations, therefore, with values of 32 and 64 bits are cheap to perform. Operations on single bits are not simple when you manipulate multiple bits, it is slightly more expensive to use a single bit to represent truth or false. Compilers usually use a block of memory (32 or 64 bits) to represent a boolean, the convention is for zero (all 32-64 bits in zero) to be false and anything else (eg, all bits in zero, or only one bit in zero, etc) to be true.

039.

Let's take a look at another type of variable, `char`:<sup>†</sup>

```
char apples = 23;
std::cout << "A small integer: " << (int)apples
          << std::endl;
```

The output isn't surprising:

```
A small integer: 23
```

`char` may not seem different to `int`, but it is. `int`, depending on the system, has a size of 32 or 64, but `char` has always the same size 8 bits. With 8 bits we can represent  $2^8$  different states, that is 256 different numbers.

<sup>†</sup>you can ignore the weird `(int)` thing for now, it is called **cast** if you are curious, we will get to them later on.

040.

What is the output of:

```
int i = 2;
std::cout << i << " ";
i = i + 1;
std::cout << i << " ";
i = i + 1;
std::cout << i << " ";
i = i * 3;
std::cout << i << std::endl;
```

The output is:

```
2 3 4 12
```

041.

Every compiler may define the size of `int`, `char`, `double`, ..., differently depending on the architecture. If you want to know how many bytes<sup>†</sup> are assigned to any variable type, you can use `sizeof`. An example of use:

```
std::cout << "A char   is " << sizeof(char)
          << " bytes" << std::endl;
std::cout << "A int    is " << sizeof(int)
          << " bytes" << std::endl;
std::cout << "A double is " << sizeof(double)
          << " bytes" << std::endl;
std::cout << "A float  is " << sizeof(float)
          << " bytes" << std::endl;
std::cout << "A bool   is " << sizeof(bool)
          << " bytes" << std::endl;
```

This may look different in your computer, but mine runs on 64 bits, therefore `double` has 64 bits and `int` half of that.

```
A char   is 1 bytes
A int    is 4 bytes
A double is 8 bytes
A float  is 4 bytes
A bool   is 1 bytes
```

(Hint: if a byte is 8 bits, a `char` is 8 bits, how many bytes are a `char`?)

<sup>†</sup>one byte is 8 bits

042.

What is the output of:

```
char a = 100;
char b = 20;
char c = a + b;
std::cout << "a : " << (int)a << std::endl
          << "b : " << (int)b << std::endl
          << "a+b : " << (int)c << std::endl;
```

Not surprising, that is the output.

```
a : 100
b : 20
a+b : 120
```

043.

What is the output of:

```
char a = 100;
char b = 30;
char c = a + b;
std::cout << "a : " << (int)a << std::endl
          << "b : " << (int)b << std::endl
          << "a+b : " << (int)c << std::endl;
```

The output is:<sup>†</sup>

```
a : 100
b : 30
a+b : -126
```

well, that's surprising! What the heck happened?

The answer lies in the 8 bit part that I was talking about. `char` can only hold 256 different numbers, but usually we use one of those bits to indicate the sign<sup>‡</sup>, therefore we have left only 7 bits for the number.  $2^7$  is 128, so we can store 128 positive numbers and 128 negative numbers. If we did it naïvely we could represent the numbers from 0 to 127 with 7 bits plus one bit for the sign, but that's rarely used, we would be representing 0 in two ways +0 and -0, the answer is to count from 0 to 127 and from -128 to -1, i.e., we can lay down all the representable numbers by 8 bits in the following way: -128, -127, -126, ..., -2, -1, 0, 1, 2, ..., 125, 126, 127.

When you pass over the limit of what 7 bits<sup>§</sup> can store you need to go somewhere, and by convention that is going back to the first number, i.e., adding numbers one by one will lead you to the beginning no matter where you start:  $1 + 1 = 2$ ,  $2 + 1 = 3$ , ...,  $125 + 1 = 126$ ,  $127 + 1 = -128$ ,  $-128 + 1 = -127$ , ...,  $-1 + 1 = 0$ ,  $0 + 1 = 1$ .

<sup>†</sup>This output may change depending on the compiler you are using, it could happened that you don't see anything wrong at all. In that case, try changing 100 for 220.

<sup>‡</sup>For more details look at "two's complement binary representation"

<sup>§</sup>this is called overflow

044.

What is the output of:

```
char i = 126;
std::cout << (int)i << " ";
i = i + 1;
std::cout << (int)i << " ";
i = i + 1;
std::cout << (int)i << " ";
i = i + 1;
std::cout << (int)i << std::endl;
```

The output, as you may have easily guessed is:

```
126 127 -128 -127
```



## Interlude: How are numbers represented?

fill me!

045.

What is the output of:

```
int i = 0;
if (i<3) {
    std::cout << "There is no else statement";
    i = i * 2;
}
std::cout << std::endl;
```

The output is:

There is no else statement

Notice how we ignored the `else` statement, well that's ok, we can just do stuff for when something is true otherwise we don't do anything.

046.

What is the output of:

```
int i = 0;
if (i<3) {
    std::cout << i << " ";
    i = i + 1;
}
if (i<3) {
    std::cout << i << " ";
    i = i + 1;
}
std::cout << std::endl;
```

The output is:

0 1

047.

What is the output of:

```
int i = 0;
if (i<3) {
    std::cout << i << " ";
    i = i + 1;
}
if (i<3) {
    std::cout << i << " ";
    i = i + 1;
}
if (i<3) {
    std::cout << i << " ";
    i = i + 1;
}
if (i<3) {
    std::cout << i << " ";
    i = i + 1;
}
std::cout << std::endl;
```

The output is:

0 1 2

048.

What is the output the above if we change all appearances of `i<3` for `i<2` or `i<4`.

The outputs are:

0 1

for `i<2` and

0 1 2 3

for `i<4`.

049.

What is the output if we change `i<3` for `i<5`?

Well there are only 4 `ifs`, the limit is 4 numbers printed on the screen:

```
0 1 2 3
```

050.

What should we do to print five numbers?

Well, there are many options. We could add a `std::cout << i << std::endl;` line after the last `if` statement, or we could copy an `if` (and this is what I wanted you to answer)

051.

What do you think this outputs?

```
int i = 0;
while (i<4) {
    std::cout << i << " ";
    i = i + 1;
}
std::cout << std::endl;
```

The output is:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

Nice, no repeated code!

052.

What happens if we change `i<4` above for `i<5`?

Well, now we can see five numbers:

```
0 1 2 3
```

053.

What is the output of:

```
int i = 1;
while (i<4) {
    std::cout << i << " ";
    i = i + 1;
}
std::cout << std::endl;
```

The output is:

```
1 2 3
```

054.

What is the output of:

```
int i = -2;
while (i<4) {
    std::cout << i*2 << std::endl;
    i = i + 1;
}
```

The output is:

```
-4
-2
0
2
4
6
```

055.

Something a little more elaborated:

```
int i = -2;
while (i<3) {
    std::cout << i << " squared is: " << i*i
              << std::endl;
    i = i + 1;
}
```

The output is:

```
-2 squared is: 4
-1 squared is: 1
0 squared is: 0
1 squared is: 1
2 squared is: 4
```

But I'm getting bored with all those `i = i + 1`, fortunately that operation is so common in day to day work that it can be written shorter: `i++`.

056.

What is the output of:

```
int i = -6;
while (i<=-2) {
    std::cout << i << " squared is: " << i*i
               << std::endl;
    i++;
}
```

```
-6 squared is: 36
-5 squared is: 25
-4 squared is: 16
-3 squared is: 9
-2 squared is: 4
```

add exercises using while

057.

What is happening here?

```
int a = 13;
int b = 6;
int c = 8;
while (b<a && b<c) {
    std::cout << "Both conditions met"
               << std::endl;
    b++;
}
```

```
Both conditions met
Both conditions met
```

058.

And here?

```
int a = 13;
int b = 6;
int c = 8;
while (b<a || b<c) {
    std::cout << "At least one condition met"
               << std::endl;
    b++;
}
```

```
At least one condition met
At least one condition met
At least one condition met
At least one condition met
At least one condition met
At least one condition met
At least one condition met
```

Well, `&&` is called the **and** operator, and `||` is called the **or** operator, they operate on **bools** only.<sup>†</sup>

<sup>†</sup>Well, they can actually operate in any number, any number not zero is treated as **true** otherwise **false**. This means that `a && b` is a valid statement above, but it is a weird one, its result is **true**. Such expressions, even though totally legal, are discouraged.

059.

What is the output of:

```
bool a = 5 > 2;
bool b = true;
if (a==b) {
    std::cout << "5 > 2 is equal to true"
               << std::endl;
} else {
    std::cout << "bananas!" << std::endl;
}
```

```
5 > 2 is equal to true
```

As usual.

060.

What is the output of:

```
if ((5>2)==true) {
    std::cout << "5 > 2 is equal to true"
               << std::endl;
} else {
    std::cout << "bananas!" << std::endl;
}
```

It's basically the same thing from above, isn't it.

```
5 > 2 is equal to true
```

061.

We have printed only `ints` and `doubles` to date, what would be the result of printing `bools`?

```
std::cout << "true gets printed as " << true
           << std::endl;
std::cout << "But false gets printed as "
           << false << std::endl;
```

```
true gets printed as 1
But false gets printed as 0
```

Yep, `true` is represented as 1 always!<sup>†</sup>

Moral of the fable: don't use boolean operations on variables that are not boolean! and don't use non-boolean operations on boolean variables!

<sup>†</sup>this is standard behavior, because comparing `true` with 1 will result in `true` again, but it won't if you compare it to 3 even though 3 is treated as true if you use it in an `if` expression.

062.

Let's take a look at the *truth table* for `&&`:<sup>†</sup>

```
std::cout
  << "1 && 1 == " << (true && true)
  << std::endl
  << "1 && 0 == " << (true && false)
  << std::endl
  << "0 && 1 == " << (false && true)
  << std::endl
  << "0 && 0 == " << (false && false)
  << std::endl;
```

```
1 && 1 == 1
1 && 0 == 0
0 && 1 == 0
0 && 0 == 0
```

<sup>†</sup>the parthesis around the boolean expressions are necessary, otherwise they would get confused with `<<`.

063.

And the *truth table* for `!` (not, negation):

```
std::cout
  << "!1 == " << !true << std::endl
  << "!0 == " << !false << std::endl;
```

```
!1 == 0
!0 == 1
```

All tables as you know them from logic, old stuff (actually, not that old, this tables were first used as tables in the 20th century, but they are so intuitive that one may think they're an older invention <sup>†</sup>).

<sup>†</sup>for some more history see wikipedia article *Truth table*.

064.

Now, what should be changed in the code above (for `&&`) to make the *truth table* for `||`. The output of the table should be:

```
1 || 1 == 1
1 || 0 == 1
0 || 1 == 1
0 || 0 == 0
```

And here is the code:

```
std::cout
<< "1 || 1 == " << (true || true)
<< std::endl
<< "1 || 0 == " << (true || false)
<< std::endl
<< "0 || 1 == " << (false || true)
<< std::endl
<< "0 || 0 == " << (false || false)
<< std::endl;
```

065.

What should be the value of `c` in the code below for the code to output 0 1 2 3 4?

```
int b = 10;
int c = ???;
int i = 0;

while ((i < b) && (i < c)) {
    std::cout << i << " ";
    i++;
}
std::cout << std::endl;
```

Precisely, it should be **5**. Let's take a look at the code running line by line:

1. We declare and initialize the variables `b`, `c` and `i` with the values 10, 5 and 0 respectively.
2. We ask, is `i` content smaller than `b`?, and is `i` content smaller than `c`?, and both are truth ( $0 < 10$  and  $0 < 5$ ).
3. We print in the screen the value of `i` (0)
4. We increment the value of the variable `i`
5. We ask, is `i` content smaller than `b`?, and is `i` content smaller than `c`?, and both are truth ( $1 < 10$  and  $1 < 5$ ).
6. We print in the screen the value of `i` (1)
7. We increment the value of the variable `i`
8. We ask, is `i` content smaller than `b`?, and is `i` content smaller than `c`?, and both are truth ( $2 < 10$  and  $2 < 5$ ).
9. We print in the screen the value of `i` (2)
10. We increment the value of the variable `i`
11. We ask, is `i` content smaller than `b`?, and is `i` content smaller than `c`?, and both are truth ( $3 < 10$  and  $3 < 5$ ).
12. We print in the screen the value of `i` (3)
13. We increment the value of the variable `i`
14. We ask, is `i` content smaller than `b`?, and is `i` content smaller than `c`?, and both are truth ( $4 < 10$  and  $4 < 5$ ).
15. We print in the screen the value of `i` (4)
16. We increment the value of the variable `i`
17. We ask, is `i` content smaller than `b`?, and is `i` content smaller than `c`?, one is false ( $5 < 10$  and  $5 < 5$ ).
18. We get out of the while loop and go to the next line, thus we print at last a newline.

066.

Describe line by line what the following code does (and its output):

```
int b = 10;
int c = 5;
int i = 0;

while ((i < b) && (2*i < c)) {
    std::cout << i << " ";
    i++;
}
std::cout << std::endl;
```

The steps are:

1. We declare and initialize three `int` variables (`b`, `c` and `i` with 10, 7 and 0 respectively).
2. We ask, is `i` content smaller than `b`?, and is `2*i` content smaller than `c`?, and both are truth ( $0 < 10$  and  $0 < 5$ ).
3. We print in the screen the value of `i` (0)
4. We increment the value of the variable `i`
5. We ask, is `i` content smaller than `b`?, and is `2*i` content smaller than `c`?, and both are truth ( $1 < 10$  and  $2 < 5$ ).
6. We print in the screen the value of `i` (1)
7. We increment the value of the variable `i`
8. We ask, is `i` content smaller than `b`?, and is `2*i` content smaller than `c`?, and both are truth ( $2 < 10$  and  $4 < 5$ ).
9. We print in the screen the value of `i` (2)
10. We increment the value of the variable `i`
11. We ask, is `i` content smaller than `b`?, and is `2*i` content smaller than `c`?, and one is false ( $3 < 10$  and  $6 < 5$ ).
12. We get out of the while loop and go to the next line, thus we print at last a newline.

The complete output is:

```
0 1 2
```

067.

What is the output of:

```
int imzero = 0;

while (imzero < 4) {
    std::cout << imzero << " ";
}
std::cout << std::endl;
```

The output is...:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 ...
```

an unending sequence of zeros, so, what is wrong?

We forgot to add the statement `imzero++`! Without it, well, the variable `imzero` never changes its state/value, and we ask *ad infinitum* if zero is smaller than four. <sup>†</sup>

<sup>†</sup>I actually wrote faulty code like that several times in the writing of this book XD

068.

What should be the value of `c` in the code below for the code to output 0 2 4 6 8?

```
int b = 7;
int c = ???;
int i = 0;

while ((i < b) || (i+1 < c)) {
    std::cout << i << " ";
    i++;
    i++;
}
std::cout << std::endl;
```

Yep, either `c` should be either 10 or 11.

069.

What is the output of:

The output is:

```
3%2 == 1
4%2 == 0
0%2 == 0
7%2 == 1
2%2 == 0
```

Notice how the odd numbers all return 1, and all even numbers return 0. Why?

070.

What is the output:

The output is:

```
int a = 0;
while (a<=10) {
    if (a%2 == 0) {
        std::cout << a << " ";
    }
    a++;
}
std::cout << std::endl;
```

```
0 2 4 6 8 10
```

Cool, we can skip things while we count up (or down), like—in this case—odd numbers.

071.

What is the output of:

The output is:

```
int a = 0;
int b = 0;
while (a<=10) {
    if (a%2 == 0) {
        b = b + a;
    }
    std::cout << b << " ";
    a++;
}
std::cout << std::endl;
```

```
0 0 2 2 6 6 12 12 20 20 30
```

072.

What should we change in the code above to print only one time each number, i.e., what lines are needed to change or be moved (from the code above) to output:

Yep, we move the printing statement (line) to inside the true (then) branch inside the if statement:

```
0 2 6 12 20 30
```

?

```
int a = 0;
int b = 0;
while (a<=10) {
    if (a%2 == 0) {
        b = b + a;
        std::cout << b << " ";
    }
    a++;
}
std::cout << std::endl;
```

ask them first exercise from project euler, and reference it

ask them to give us the sequence of fibonacci to an arbitrary number, using loops and variables on some way (this is the exercise that I remember the clearest doing as I took the class, it was so gratifying to solve the puzzle)



## Chapter 5

### Block 2: More space to play with



## Chapter 6

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