

# Module 00 - Hello, World!

## Introduction

The Rust programming language is still fairly new: its 1.0 release is younger than 42 itself! For this reason, please understand that the knowledge acquired here may become obsolete at some point. Though this is true for most programming languages, Rust is still updated quite regularly (almost on a monthly basis) and you should be aware of that. Never stop learning.

The Rust programming language might feel a bit hard to get into, may it be its syntax, the borrow checker, or more generally, the number of details it requires the developer to think about while writing code. Trust the compiler. It is always (in 99% of cases) right. Plus it is pretty well known to provide really good error messages. Read the errors, correct your code, and get over it!

## General Rules

Any program you turn in should compile *without warnings* using the `rustc` compiler available on the school's machines without additional options. You are allowed to use attributes to modify lint levels, but you must be able to explain why you did so. You are *not* allowed to use `unsafe` code anywhere in your code.

For exercises using the `cargo` package manager, the same rule applies. In that case, no additional crate is allowed to be added in the project unless specified otherwise.

## Exercise 00: Hello, World!

```
turn-in directory:
  ex00/

files to turn in:
  hello.rs
```

What's a program without side effects?

Create a *program* that prints the string `Hello, World!`, followed by a line feed.

```
>_ ./hello
Hello, World!
```

## Exercise 01: Point Of No Return

```
turn-in directory:
  ex01/

files to turn in:
  min.rs
```

Create a `min` *function* that takes two integers, and returns the smaller one. To make the file compile and for it to be testable, you are allowed to add an optional `main`

function to prove your function is indeed correct. During the defense, you will have to write one anyway.

The function should be prototyped like this:

```
fn min(a: i32, b: i32) -> i32;
```

Oh, I almost forgot. The `return` keyword is forbidden! Good luck with that ~

## Exercise 02: yyyyyyyyyyyyyyyy

turn-in directory:

ex02/

files to turn in:

yes.rs collatz.rs print\_characters.rs

Imperative programming languages usually have some kind of statement to loop. Rust has several.

Create three *functions*. Each function must use one kind of loop supported by Rust, and you cannot use the same loop kind twice.

The functions should be prototyped as follows:

```
fn yes() -> !;
fn collatz(start: u32);
fn print_characters(s: &str);
```

The `yes` function should print the message `y`, followed by a line feed. It should do it indefinitely.

```
y
y
y
y
y
y
y
y
...
```

The `collatz` function should execute the following algorithm...

- Let `n` be any natural number.
- If `n` is even, then `n <- n/2`
- If `n` is odd, then `n <- 3n + 1`

...until `n` equals 1. On each iteration, `n` should be displayed on the standard output, followed by a line feed.

Input:

3

Output:

```
3
10
5
16
8
4
2
1
```

The `print_characters` function should print every character of the provided string, one by line. If a non-printable character is found (may it be ASCII or not), a debug representation of that character should be displayed instead. Note that using an `if` or `match` statement isn't probably going to work for *every possible* Unicode character.

```
Input:
"Hello!\n"
```

```
Output:
'H'
'e'
'l'
'l'
'o'
'!'
'\n'
```

Once again, you are allowed to add `main` functions to prove that your functions are correct.

## Exercise 03: FizzBuzz

```
turn-in directory:
  ex03/
```

```
files to turn in:
  fizzbuzz.rs
```

This is the final exam of the C piscine. This is YOUR moment. You *can* do it. Problem: your current subject is *FizzBuzz* and you're under so much stress right now that you forgot how to use loops.

Create a Rust *program* that prints a C program on the standard output. That C program must play the popular game *FizzBuzz* by itself without using any loop (`for`, `while`, `do`, etc.) statement. The only allowed function is `puts`.

The subject reads as follows:

```
Write a program that prints the numbers from 1 to 100, each separated by a
newline. If the number is a multiple of 3, it prints 'fizz' instead. If the number
is a multiple of 5, it prints 'buzz' instead. If the number is both a multiple of
3 and a multiple of 5, it prints 'fizzbuzz' instead.
```

```
>_ ./fizzbuzz
#include <stdio.h>

int main(void)
{
    puts("1");
    puts("2");
    puts("fizz");
    puts("4");
    puts("buzz");
    /* ... */
}
```

## Exercise 04: Don't Panic

```
turn-in directory:
    ex04/

files to turn in:
    dont_panic.rs
```

Unexpected events are to be expected in any computer application. Rust makes no *exception* to that rule, and provides a way to "cleanly" crash a Rust program in case something unrecoverable occurs (such as memory failing to allocate).

Create a Rust *program* that panics with the message "I DON'T KNOW WHAT IS GOING ON!!".

```
>_ ./dont_panic
thread 'main' panicked at 'I DON'T KNOW WHAT IS GOING ON!!', dont_panic.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

You can check with Valgrind that no memory has been lost (even though Rust allocates some blocks before calling the `main` function).

## Exercise 05: Cargo

```
turn-in directory:
    ex05/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
```

The Rust ecosystem relies heavily on its package manager, Cargo. Cargo is a program that is responsible for managing the dependencies of a Rust application. It has many capabilities, but this module will only make use of a few of them.

Create a Rust *program* that prints the string `Hello, Cargo!`, followed by a line feed.

```
>_ cargo run
Hello, Cargo!
```

## Exercise 06: Unit Tests

```
turn-in directory:
  ex06/

files to turn in:
  src/lib.rs  Cargo.toml

allowed dependencies:
```

Testing a program is probably at least half the work of a developer. Every single function of any numerical system should be carefully tested to avoid as many crashes and unexpected behaviors as possible.

```
pub fn fibs(n: u32) -> u32 {
    (0..n).fold((0, 1), |(a, b), _| (b, a + b)).0
}

pub fn is_prime(n: u32) -> bool {
    n >= 2 && (2..n).find(|&d| n % d == 0).is_none()
}
```

Copy the above functions, and write unit tests to determine whether those functions do work as expected.

- The `fibs` function computes the `n`-th term of the [fibonacci sequence](#).
- The `is_prime` function returns whether `n` is a prime number.

In case of an error, the test should panic with an appropriate error message, but you are *not* allowed to use the `panic!` macro.

```
>_ cargo test
running 7 tests
test zero_is_not_prime ... ok
test one_is_not_prime ... ok
test tree_is_prime ... ok
test four_is_not_prime ... ok
test first_fib_is_0 ... ok
test fifth_fib_is_3 ... ok
test seventeenth_fib_is_987 ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

...
```

Note that the above is just an example. You are allowed to do the tests you want, but they must properly check that those functions are actually correct.

## Exercise 07: The Price Is Right

turn-in directory:

ex07/

files to turn in:

src/main.rs Cargo.toml

allowed dependencies:

ftkit

To end this first module, why not try to create a simple game?

Create a *program* that allows its user to play the popular game show "The Price Is Right". The game plays as follows:

1. The program randomly chooses an object whose price is unknown to the user. The name of the object is displayed.
2. The user is prompted to bid the price of said object.
3. If the user is correct, the program ends with a message congratulating them.
4. Otherwise, a message indicates whether they overbid or underbid and the program is back to step 2.

Example:

```
>_ cargo run
Here is a fabulous 'cool ring'.
23
A 'cool ring' costs more than that!
57
A 'cool ring' isn't worth that much money!
34
Congrats! That 'cool ring' is worth $34.
```

To help you with that task, you are allowed to depend on the `ftkit` crate, which provides some basic utility functions. Documentation for that library is available [here](#).