# Rust - Lab 1

## Install Rust

https://www.rust-lang.org/tools/install

## Useful resources

- https://doc.rust-lang.org/book/
- https://doc.rust-lang.org/rust-by-example/
- https://doc.rust-lang.org/cargo/index.html

## Exercices

### Starter

#### 1 - To infinity and beyond!

1. Write a single line Rust function `print_all(x: u32) -> ()`, that prints all unsigned integers starting from a given value `x`.
2. Will this function eventually stop or crash? Explain your answer.
    - What happens when `u32::MAX` is reached in **debug mode**?
    - What happens in **release mode** (the default when compiling with `rustc`)?

#### 2 - Who owns the string?

1. Write the following function:

```
fn print_and_return(s: String) -> String
{
   println!("{}", s);
   s
}
```

2. Now, try to run this code:

```
fn main()
{
   let my_str = String::from("Hello world!");
   print_and_return(my_str);
   println!("{}", my_str);
}
```

   Does it compile? Why?

3. Modify the program so that:
    - The string can still be printed inside the function.
    - It can still be used after the function call.
    - Without cloning it.

    Explain your solution.

**3 - Robust error management**

1. Write a function `safe_divide(a: i32, b: i32)`
    - It should return the result if the division `a / b` is valid.
    - It should return the message `"Division by zero"` if `b == 0`.
2. In `main`, use a `match` expression to print either the result of the division or the error message.
3. Bonus: Rewrite `main` using the `?` operator to simplify error handling.
   Hint: You have to change `main`'s signature.

**4 - Handling missing values**

1. Write a function `first_even(nums: &[i32])`:
    - It should return the first even number in the slice (if it contains at least one).
    - It should return `None` if there are no even numbers.
2. Use `match` expression to print:
    - `"First even: X"` if a value was found.
    - `"No even numbers"` if not.
3. Bonus: Use the combinator `.map(...)` and `.unwrap_or(...)` instead of `match` to print the same results in a shorter style.

## Core exercices

**5 - Reverse Polish Notation (RPN) Calculator**

Reverse Polish Notation writes expressions with operands first, then the operator.
For example, `["2", "1", "+", "3", "*"]` means `(2 + 1) * 3` and evaluates to `9`.
RPN expressions are evaluated using a stack. You read the tokens from left to right:

- When you encounter a number, you push it onto the stack.
- When you encounter an operator, you pop the last two numbers, apply the operator, and push the result back onto the stack. This process continues until all tokens are processed.

The final result is the last value remaining on the stack. No parentheses are needed, because the order of operations is determined by the position of the operators.

**Create a Rust program that implements an RPN calculator.** The input should be a list of tokens, and the output should be the computed result.

**Quick hints**

- For the stack, use `Vec<i32>` to push operands and pop when you see an operator.
- Parse numbers with `token.parse::<i32>()`.
- Return rich errors (e.g. `Result<i32, Error>` or `Result<i32, String>`):

**6 - Binary Search**

Binary search is an efficient algorithm to find the position of a target value in a **sorted array**.
It works by repeatedly dividing the search interval in half:

- If the value at the middle index is equal to the target, return its index.
- If the target is smaller, search the left half.
- If the target is larger, search the right half.

Binary search has a time complexity of **O(log n)** and only works on **sorted data**.

**Create a Rust program that implements binary search.**
The input is a list of integers and a target value.
The output should be the index of the target if found, or an error message if not.

**Quick hints**

- The input list is not sorted. You must sort it without losing the original indices. One way is to define a struct `SortedValue` that stores both the value and its original index, then perform the search on a sorted list of `SortedValue`.

```
struct SortedValue {
   value: i32,
   original_index: usize,
}
```

- The time complexity of the search itself should be of `O(log n)`, where `n` is the number of elements (excluding the sorting step).