

SISTEMAS COMPUTACIONAIS AVANÇADOS (SISTCA)

ADVANCED COMPUTING SYSTEMS

LICENCIATURA EM ENGENHARIA ELECTROTÉCNICA E DE
COMPUTADORES

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
POLITÉCNICO DO PORTO

Lab Classes Script:

Introduction to Rust Programming Language

Version Control

Version number	Date issued	Authors	Update information
V1.0	May 2021	Francisco Pires (1181808) Beatriz Gomes (1190422) David Ribeiro (1190493) Ricardo Macedo (1190998) Supervision: Prof. Luis Vilaça	Original version of the script.

Table of Contents

1.	Introduction.....	1
1.1	Scientific/Technological Context	1
1.2	Motivation for this topic/lab script.....	1
1.3	Objectives (of this lab script)	2
1.4	Structure (of this lab script).....	3
2.	Theoretical (scientific/technological background)	3
2.1	Main theoretical concepts/terminology	3
2.2	State-of-The-Art	4
3.	Outlook of the technology.....	7
3.1	Setup/Installation (SW/HW)	7
3.2	First steps	8
4.	Tutorial/Functionality	8
5.	Exercise(s)	24
5.1	Problem A	24
5.1.1	(re)solution A	25
5.2	Problem B	26
5.2.1	(re)solution B	27
6.	Challenge	29
7.	References	31

1. Introduction

1.1 Scientific/Technological Context

Rust was officially launched in 2012 by Mozilla, however, the idea originated in 2006 by an engineer named Graydon Hoare. Since then, Hoare has begun working on the programming language, catching Mozilla's attention, who came together to develop the language and built it in the best possible way [1].

With Rust it is possible to build reliable and efficient systems software. Rust is also used by developers for networking software, such as web servers, mail servers and web browsers. In addition, you can use this programming language to build games, web-assembly programs, applications for embedded devices and command line programs [2]. It is also present in compilers and interpreters, databases, operating systems and cryptography.

One of the Rust's great advantages is to ensure memory security and try to fix many of the bugs related to the incorrect memory usage that occurs in C and C++. Additionally to these advantages, Rust has relevance in the field of processing large amounts of data.

1.2 Motivation for this topic/lab script

Rust is one of the most diversified languages to learn and start using to replace bloated code on other applications. Since 2018, the community around Rust has been improving the experience on various domains, such as CLI (Command Line Interface) Tools, WebAssembly, Networking and even Embedded devices with necessities for low-level control, developing various crates and guides to make the experience smoother.

In the course of LEEC, that means you can use the same language to build the structure to a database accessible on the web, programm microcontrollers, and even build your cross-platform programs. If you are interested, you can also use Rust to build your own kernel, with bare metal support for the hardware.

It's important to note that Rust is also the most loved language the sixth-year in a row according to the annual survey from Stack Overflow [3].

Despite this, Rust is not yet among the five most used programming languages [1], since the learning process can be a little more time consuming.

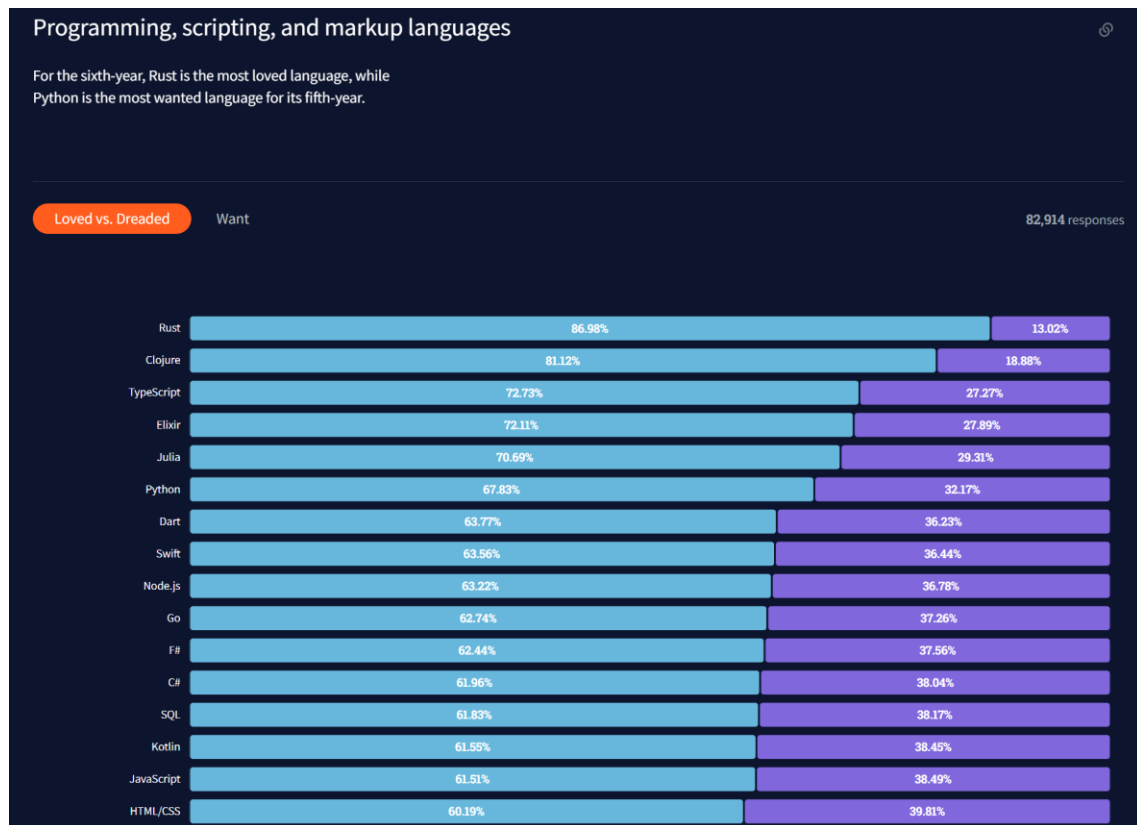


Figure 1 – Stack Overflow most loved programming languages [3]

1.3 Objectives (of this lab script)

The main goal of this report is to present an introduction of the Rust language and to make known its main applications in current technologies.

After reading the script, the student should be able to solve the proposed exercises that are at the end of the report. In this way, the main concepts of this programming language will be assimilated.

In addition, the following objectives are expected to be achieved:

- Understanding of the main theoretical concepts of language;
- Advantages and Disadvantages of Rust compared to other programming languages;
- Implementation and development of basic applications based on Rust.

1.4 Structure (of this lab script)

This lab script is structured as follows:

- In the first section, an introduction to the Rust language is made, as well as its technological context, followed by the motivation and main objectives of the report.
- In Chapter 2, the theoretical concepts of Rust will be deepened, and a comparison will be made with Go and C++, ending with the presentation of its Advantages and Disadvantages.
- Chapter 3 demonstrates the process of setting up and installing the tools needed to develop projects with Rust, as well as the first steps for these projects.
- Chapter 4 goes into depth on all the important features for working with Rust, such as variables, data types, functions, test conditions and loops.
- Chapter 5 presents two exercises to be solved with Rust and their solutions.
- In Chapter 6 a challenge is proposed for the student to solve with the skills they have acquired throughout the script.

2. Theoretical (scientific/technological background)

2.1 Main theoretical concepts/terminology

There are a few important concepts around the Rust Language that one should be aware of, for example, the Ownership feature, which makes Rust to be memory safe without needs for a garbage collector [4]. It is Rust's most unique feature, and, in contrary to other languages, with garbage collections or explicit need to allocate and free memory, Rust uses a set of rules that the compiler checks, and, if any are violated, the program won't compile. On the other side, it won't slow down your program while running.

Traits allow type classes to adopt different behaviors, inspired by the Haskell language. Basically, floats and integers can both implement the 'Add' trait and any type that can be printed implement the 'Display' or 'Debug' traits [5].

There are also Macros, like 'println!' that can take form of function calls but operate on distinct terms, it can take various numbers of parameters and can implement traits before compilation [6]. The main downside, is that you need to write more complex code, and you must define macros or have them called before using them, whether functions can be defined and called anywhere.

2.2 State-of-The-Art

- **Comparison with Go and C++**

→ Rust in comparison to Go

Rust and Go are two very similar programming languages. Both are recent and widely used these days. In addition, both are compiled languages and open source and both are designed for the development of microservices oriented parallel computing environments.

Table 1 - Rust vs Go [7][8]

	Rust	Go
Memory Management	The developer has complete control of memory management. Memory management in Rust is known at compile time.	Go uses the garbage collector concept, which means that the data is not released immediately. Go's garbage collector is well optimized, but it is less predictable than Rust memory management.
Memory Safety	Rust resolves common memory errors without using a garbage collector, which makes its performance superior to Go.	Uses garbage collector to solve the problem of memory safety.
Error handling	Rust introduces a new type: enum with two variants, result type and error type.	In Go, several function values are returned along with the error.
Learning	The syntax and documentation are harder to understand when using it for the first time.	It takes a few hours to learn Go and a few days to start using it in projects.

→ Rust in comparison to C++

Rust is often compared to C++. C++ is an object-oriented programming language known for its contribution in operating systems for the creation of games [9].

It was developed in 1985 with the aim of improving the design of the C language. Rust and C++ are often compared nowadays due to their direct link between hardware configuration, performance speed and low level to memory. In the table below we see the comparison of these two languages regarding their learning, performance and memory safety.

Table 2 - Rust vs C++ [10][11]

	Rust	C++
Performance	Due to its better security standards that decrease the development process cost, Rust allows to achieve a higher level performance compared to C++.	C++ can achieve its optimal performance and produce very fast applications with less time spent compiling and executing code.
Learning	Rust syntax is complicated for those starting to learn it, whether they are beginners or already experienced developers in another language.	C++ is not a simple programming language either. However, developers with experience in C, Java or C# find it easier to understand the syntax of C++
Memory Safety	Rust is "safe-by-default". It balances preventing undesirable behaviors and allowing for them if the loss will not be too great.	C++ leaves memory safety to the developer. In C++, an error might slip through the review process and cause application crashes or even security vulnerabilities.

- **Advantages and Disadvantages**

Rust is a statically-typed programming language which has been much appreciated by programmers in recent years. It is mainly used for creating robust and efficient software, as well as for file systems, operating systems and simulation engines for virtual reality. It was designed for performance and security, especially for memory management. Its growing popularity is due to its many advantages:

- Faster than languages like C++ and C;
- One of the great advantages of Rust is its robust memory management system: it can assign each bit of memory to a single owner and determines who can access that bit of memory. In addition to that, it also allows secure execution of software on several processors, guaranteeing that code runs in parallel [12];
- A multi-purpose language, bearing excellent communities [12];
- Having an inbuilt dependency builds management called Cargo: Unlike languages like C, Rust has a tool called Cargo that allows you to compile, run, download libraries and auto-generate documentation [12];
- Rust has two writing modes: Safe Rust and Unsafe Rust. Safe Rust places additional restrictions on the programmer to make sure the code works correctly. On the other hand, Unsafe Rust gives the programmer more autonomy but the code may break. This mode unlocks more options but care must be taken to ensure that the code is truly safe [13].

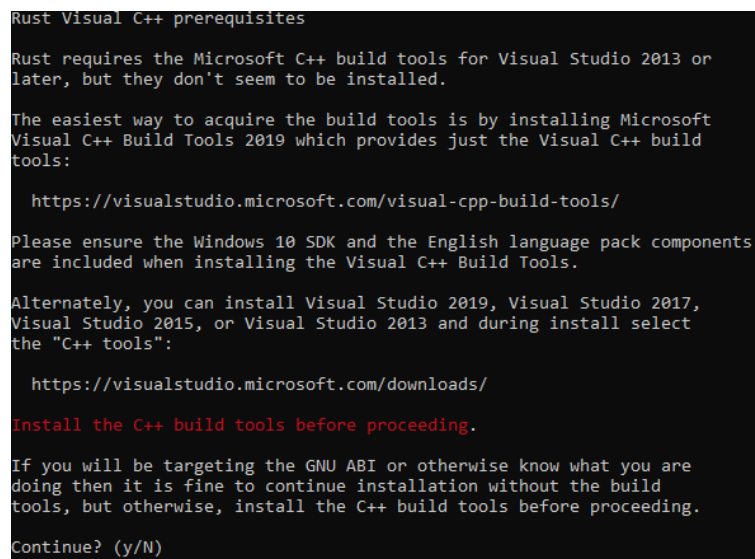
The fact that Rust is a relatively recent language makes that some desired libraries are not yet available. Thus, despite having all these benefits, the following disadvantages are highlighted:

- Learning Rust may not be very straightforward at first, as it requires more learning effort. It can require a solid knowledge of C++ or any object oriented language [14];
- For programmers who are not used to working with programming language when errors are detected in the code, it can be frustrating to receive many error messages [13];
- In Rust, code cannot be developed as quickly as in other programming languages such as Python or Ruby [14];
- The Rust compiler is very slow compared to those of other programming languages. If we have a script with many lines of code the compile time can be long [14].

3. Outlook of the technology

3.1 Setup/Installation (SW/HW)

To get started with Rust, the first thing you should do is to go to the official website (<https://www.rust-lang.org/>) and head to the "Getting Started" page. Here you can choose to install Rust through the tool called Rustup, which is supported on various systems and architectures, but for our convenience it will only show the version recommended for your system. During the installation it is strongly recommended to install the Visual C++ prerequisites on Windows from the [link](#) provided. Follow the instructions and continue with the installation. If you just want to try Rust, there is a [browser interface](#) to experiment with.

A screenshot of a text-based interface showing instructions for installing Visual C++ prerequisites for Rust. The text is white on a dark background. It explains that Rust requires the Microsoft C++ build tools for Visual Studio 2013 or later, and provides a link to the Visual C++ Build Tools 2019. It also mentions that the Windows 10 SDK and the English language pack components should be included. An alternative path is provided for installing Visual Studio 2019, 2017, 2015, or 2013. The text concludes with a prompt to continue (y/N).

```
Rust Visual C++ prerequisites

Rust requires the Microsoft C++ build tools for Visual Studio 2013 or
later, but they don't seem to be installed.

The easiest way to acquire the build tools is by installing Microsoft
Visual C++ Build Tools 2019 which provides just the Visual C++ build
tools:

    https://visualstudio.microsoft.com/visual-cpp-build-tools/

Please ensure the Windows 10 SDK and the English language pack components
are included when installing the Visual C++ Build Tools.

Alternately, you can install Visual Studio 2019, Visual Studio 2017,
Visual Studio 2015, or Visual Studio 2013 and during install select
the "C++ tools":

    https://visualstudio.microsoft.com/downloads/

Install the C++ build tools before proceeding.

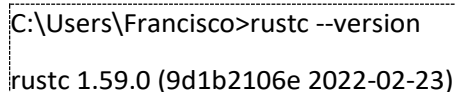
If you will be targeting the GNU ABI or otherwise know what you are
doing then it is fine to continue installation without the build
tools, but otherwise, install the C++ build tools before proceeding.

Continue? (y/N)
```

Figure 2 - Rust Setup/Installation

There is another prompt to see if the user wants to change directory of installation but it's recommended to use the default settings.

After this step, Rust is installed on the system. To confirm this, just write in the console "rustc --version" and check if a similar line got printed.

A screenshot of a terminal window showing the command 'rustc --version' being executed in a Windows command prompt. The output shows the version number 1.59.0 and the commit hash 9d1b2106e, dated 2022-02-23.

```
C:\Users\Francisco>rustc --version

rustc 1.59.0 (9d1b2106e 2022-02-23)
```

If the output was different, you may need to restart the shell for occur the changes.

It is also advisable to install a IDE like [Visual Studio Code](#) or [VSCodium](#) where you can install extensions for many languages and helps with code completion and highlights, but you can use a text editor like [Notepad++](#), or the default on your system. For reference, in this guide we will be using Visual Studio Code, but VSCodium should have the same instructions.

3.2 First steps (Hello, world!)

Now that we are in conditions of starting programming, open up VS Code and start by choosing a folder in system where the files will be placed (File -> Open Folder) and start by opening the built-in terminal and type "cargo new hello". This will create a new folder with the necessary files. Using the sidebar file explorer, navigate to hello/src/main.rs and open it. As you can see, the default file is already ready with the "Hello, world" ready to be printed in the terminal. To do this, navigate in the terminal to the hello folder (using "cd hello") and then do "cargo run".

If everything works as intended, an "Hello, world!" was printed in the terminal!

```
PS C:\Users\Francisco\Documents\SISTCA> cd hello
PS C:\Users\Francisco\Documents\SISTCA\hello> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
    Running `target\debug\hello.exe`
Hello, world!
PS C:\Users\Francisco\Documents\SISTCA\hello>
```

Also, if you pay close attention to the output of the terminal, you can see that the program was made into an executable file and it can be executed even with a double click (but you won't see this message unless you execute it on a command line!).

4. Tutorial/Functionality

This section introduces the basic concepts for programming in Rust, which are common to many programming languages, namely: functions, variables, data types, if/else conditions and for, while and loop expressions.

- What is a function in Rust?

All programs in Rust must have a main function. The code inside this function is always the first code to be executed in a program. To declare a function in Rust, we use the fn keyword [15]. The println! macro expects one or more input arguments, which it displays to the screen or standard output.

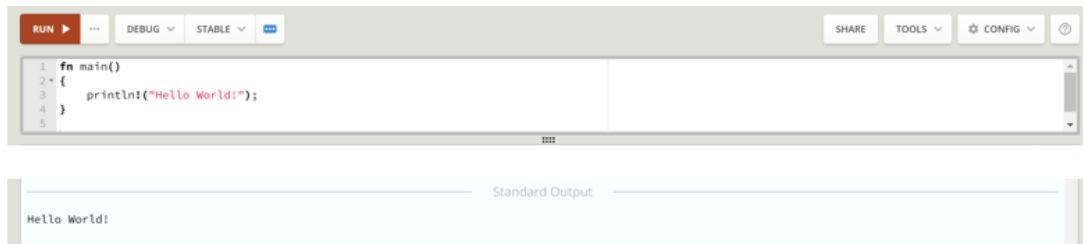


Figure 3 - Main function – “Hello World!”

- The todo! Macro

A macro in Rust is a function that requires a variable number of input arguments. By using the todo! Macro we are identifying unfinished code in our program [16].

When you compile code that uses the todo! macro, the compiler can return a panic message where it hopes to find completed functionality.



Figure 4 - todo! Macro

- Value substitution for {} arguments

The println! macro replaces each instance of curly brackets {} inside a text string with the value of the next argument in the list.

Each instance of curly brackets {} inside the text string is replaced with the value of the next argument in the list.

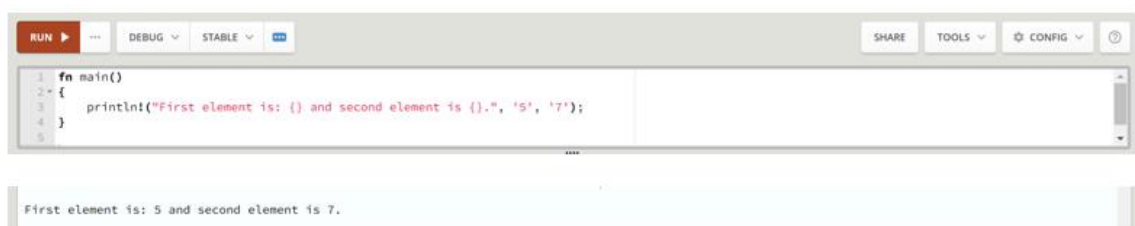


Figure 5 - {} substitution

- Comments

In Rust, the idiomatic comment style starts a comment with two slashes. To write more than a single line of code, follow the second example:

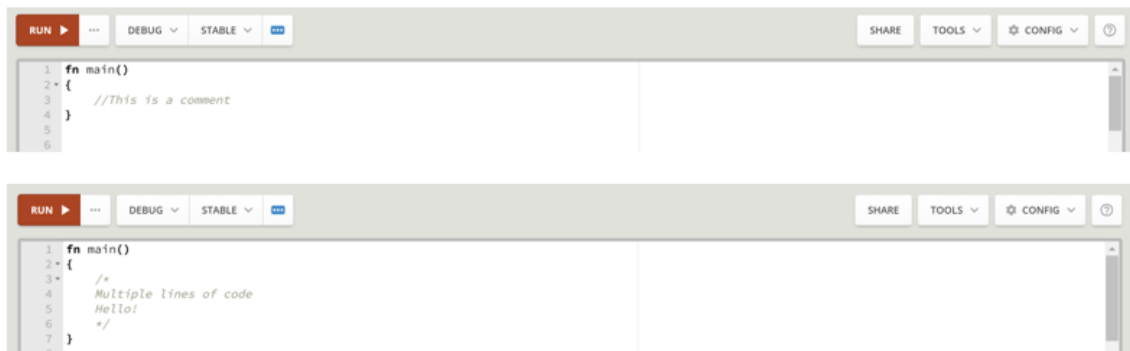


Figure 6 - Comments

Creating and using variables in Rust

We use *variables* to store our data in a named reference that we can refer to later in our code.

- Declaration of variables

In Rust, a variable is declared with the keyword `let`. Each variable has a unique name.

Note: certain keywords, like *fn* and *let* are reserved for use only by Rust, so they can't be used as names of functions or variables.

The following code declares two variables. The first variable is declared but not bound to a value. The second variable is declared and bound to a value.

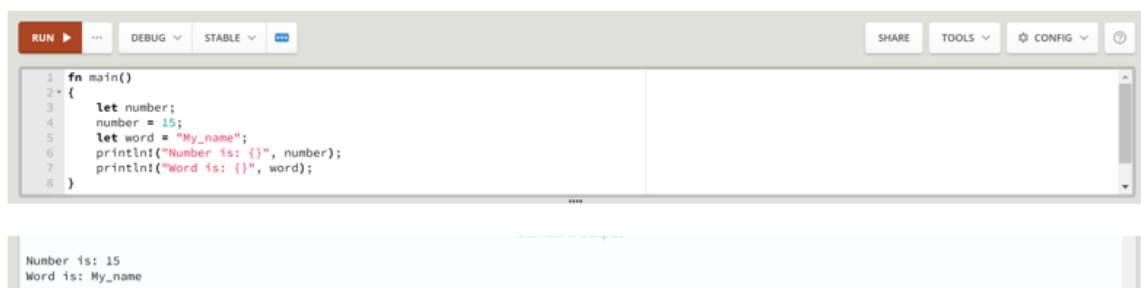
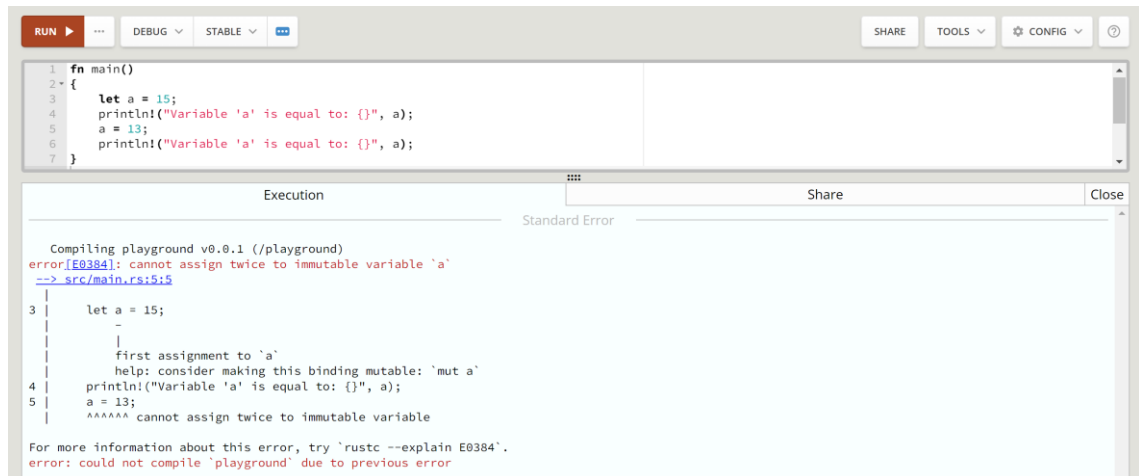


Figure 7 - Declaration of variables

- Immutable vs mutable

What is the difference between the two? In Rust, variable bindings are immutable by default. When a variable is immutable, after a value is bound to a name, we can't change that value [17].

If we try to change a variable that is immutable we receive an error message from the compiler. To mutate a value, we must first use the mut keyword to make a variable binding mutable.



The screenshot shows a Rust playground interface. The code editor contains the following code:

```
1 fn main()
2 {
3     let a = 15;
4     println!("Variable 'a' is equal to: {}", a);
5     a = 13;
6     println!("Variable 'a' is equal to: {}", a);
7 }
```

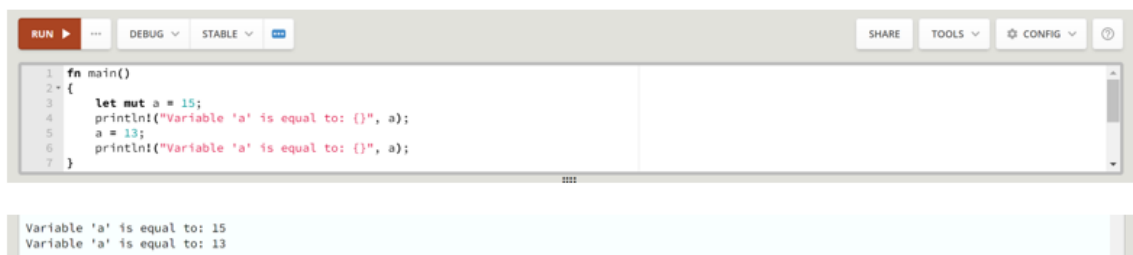
The output pane shows the following error message:

```
Compiling playground v0.0.1 (/playground)
error[E0384]: cannot assign twice to immutable variable `a`
--> src/main.rs:5:5
3 |     let a = 15;
  |         ^
  |         |
  |         first assignment to `a`
  |         help: consider making this binding mutable: `mut a`
4 |     println!("Variable 'a' is equal to: {}", a);
5 |     a = 13;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `playground` due to previous error
```

Figure 8 - Immutable error

If we try to change a variable that is mutable we no longer get an error when running the program:



The screenshot shows a Rust playground interface. The code editor contains the following code:

```
1 fn main()
2 {
3     let mut a = 15;
4     println!("Variable 'a' is equal to: {}", a);
5     a = 13;
6     println!("Variable 'a' is equal to: {}", a);
7 }
```

The output pane shows the following output:

```
Variable 'a' is equal to: 15
Variable 'a' is equal to: 13
```

Figure 9 - Mutable variable

Data Types (numbers and text) and boolean values (true and false)

Rust is a static type language, it means then that the compiler must know exactly the data type for all variables in the code. This way, the program is able to compile and execute. Rust already comes with some primitive data types built in.

Rust also offers more complex data types to work with data series, such as string and tuple values, later will appear information on the tuples.

- Integers and Float

Integers in Rust are identified by bit size and the *signed* property.

A **signed** integer can be a positive or negative number.

An **unsigned** integer can be only a positive number.

Table 3 - Length integers [18]

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
Architecture-dependent	Isize	usize

The isize and usize types depend on the kind of computer your program is running on. If you don't identify the type for an integer, and the system can't deduce the type, it assigns the i32 type (a 32-bit signed integer) by default.

Rust has two floating-point data types for decimal values: f32 (32 bits) and f64 (64 bits). The default floating-point type is f64.

Example for integers: If we have an unsigned value and assign it a negative value, the program gives an error.



```
1 fn main()
2 {
3     //unsigned value
4     let number1: u32 = 5;
5     //signed value
6     let number2: i32 = -5;
7
8     println!("Number 1 is: {}", number1);
9     println!("Number 2 is: {}", number2);
10 }
```

Number 1 is: 5
Number 2 is: -5

Figure 10 - Signed and unsigned values integers

Example for floats:



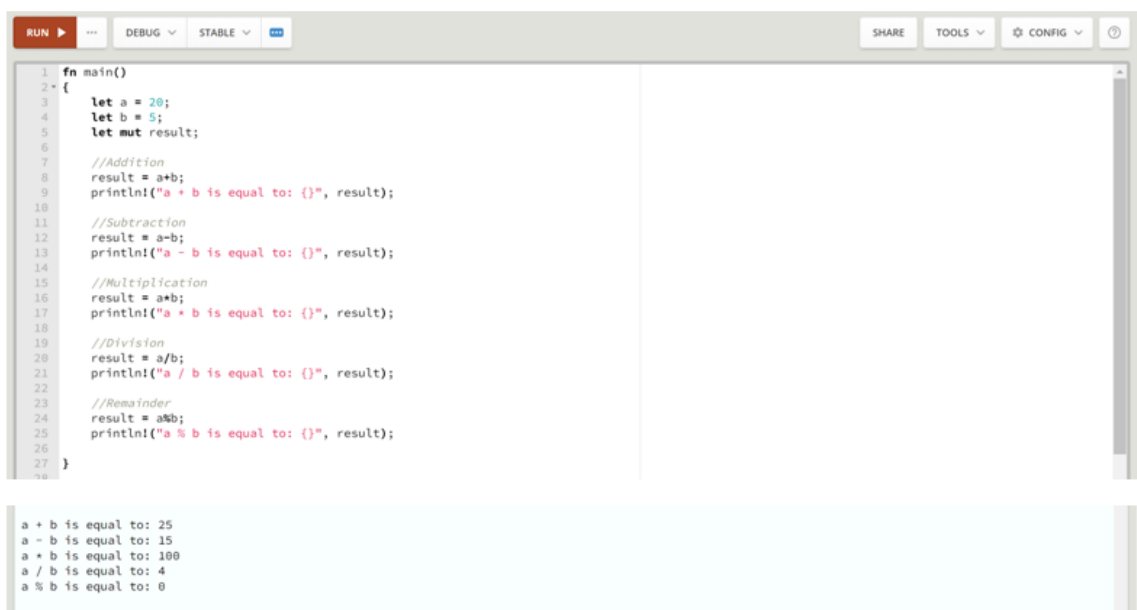
```
1 fn main()
2 {
3     //unsigned value
4     let number1 = 0.5;
5     //signed value
6     let number2: f32 = -5.32;
7
8     println!("Number 1 is: {}", number1);
9     println!("Number 2 is: {}", number2);
10 }
```

Number 1 is: 0.5
Number 2 is: -5.32

Figure 11 - Signed and unsigned values floats

- Numeric operations

It is also possible to support mathematical operations in Rust. In the example below we see examples for addition, subtraction, multiplication and division.



```
1 fn main()
2 {
3     let a = 20;
4     let b = 5;
5     let mut result;
6
7     //Addition
8     result = a+b;
9     println!("a + b is equal to: {}", result);
10
11    //Subtraction
12    result = a-b;
13    println!("a - b is equal to: {}", result);
14
15    //Multiplication
16    result = a*b;
17    println!("a * b is equal to: {}", result);
18
19    //Division
20    result = a/b;
21    println!("a / b is equal to: {}", result);
22
23    //Remainder
24    result = a%b;
25    println!("a % b is equal to: {}", result);
26
27 }
```

a + b is equal to: 25
a - b is equal to: 15
a * b is equal to: 100
a / b is equal to: 4
a % b is equal to: 0

Figure 12 - Numeric Operations

- Booleans

The boolean type in Rust is used to store truthiness. It has two possible values: true or false. A boolean value is often returned by a comparison check.

In the example below we see that the variable 'condition' will have the boolean value: false.



```
1 fn main()
2 {
3     let a = 14;
4     let b = 10;
5     let condition = a < b;
6     println!("'a' is bigger than 'b'? {}", condition);
7 }
8
9
10
```

'a' is bigger than 'b'? false

Figure 13 - Boolean type

- Characters and strings

Rust supports text values with two basic string types and one character type. A character is a single item, while a string is a series of characters.



```
1 fn main() {
2     let a = 'a';
3     let b = 'B';
4     let smiley_face = '😄';
5     println!("First: {}, Second: {}, Smiley_face: {}", a, b, smiley_face);
6 }
7
```

First: a, Second: B, Smiley_face: 😄

Figure 14 - Characters and strings

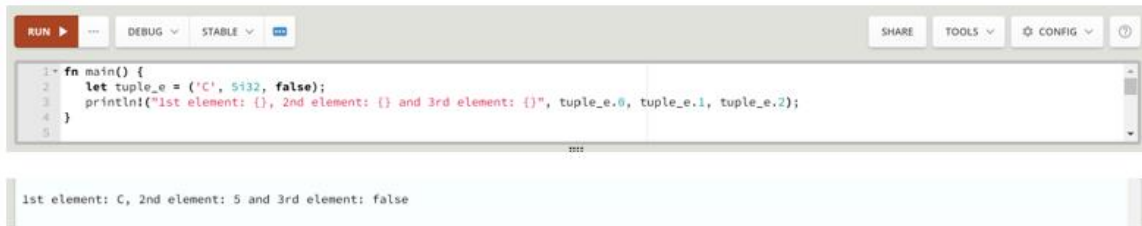
Compound Types: Tuples and Arrays

Compound types can group multiple values into one type. Rust has two primitive compound types: tuples and arrays.

- What's a Tuple?

A tuple is a grouping of values of different types collected into one compound value. The individual values in a tuple are called elements [19].

Tuples have a **fixed length**: once declared, they cannot grow or shrink in size, in other words, elements can't be added or removed so its size is equal to its number of elements. The data type of a tuple is defined by the sequence of the data types of the elements.



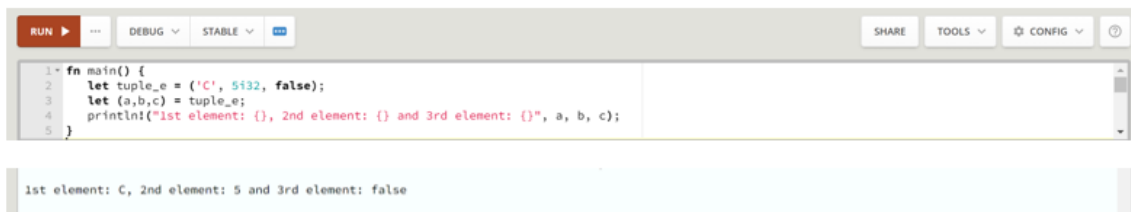
```
1- fn main() {
2-   let tuple_e = ('C', 5132, false);
3-   println!("1st element: {}, 2nd element: {} and 3rd element: {}", tuple_e.0, tuple_e.1, tuple_e.2);
4- }
5- 
```

1st element: C, 2nd element: 5 and 3rd element: false

Figure 15 - Tuple example

Tuples are useful when you want to combine different types into a single value. Functions can use tuples to return multiple values because tuples can hold any number of values.

To get the individual values out of a tuple, you can use pattern matching to destructure a tuple value. This is called destructuring, because it breaks the single tuple into three parts, as you can see in the example below:



```
1- fn main() {
2-   let tuple_e = ('C', 5132, false);
3-   let (a,b,c) = tuple_e;
4-   println!("1st element: {}, 2nd element: {} and 3rd element: {}", a, b, c);
5- }
6- 
```

1st element: C, 2nd element: 5 and 3rd element: false

Figure 16 - Destructuring a tuple

- What is an Array?

An array is a collection of objects of the same type that are stored sequentially in memory. The length of an array is equal to the number of array elements and its size can be specified in the code [20].

Contrary to arrays in some other languages, arrays in Rust have a fixed length. Unlike a tuple, each element of an array **must have the same type**.

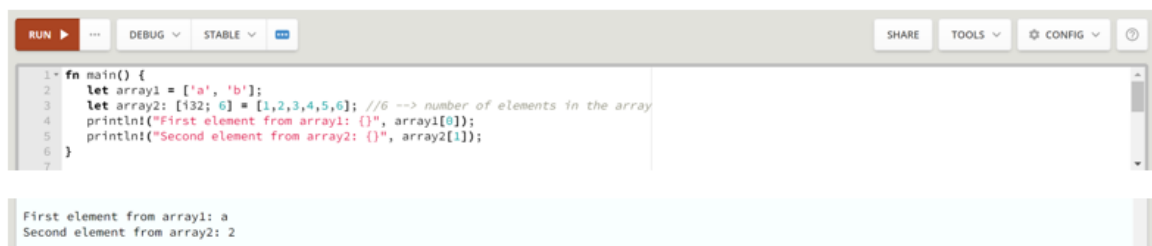
Only one thing about an array can change over time: the values of the array elements. The data type remains constant and the number of elements (length) also remains the same. Only the values can vary.

When we access an invalid position from the array Rust returns an error.

An array can be defined in two ways:

- ➔ A comma-separated list of values, where the **length isn't specified**. In the example we have array1 in this format;
- ➔ The initial value followed by a semicolon, and then the array length. In the example we have array2 in this format.

In the example below, in array2, i32 is the type of each element. After the semicolon, the number 6 indicates the array contains six elements.



```
1 fn main() {  
2     let array1 = ['a', 'b'];  
3     let array2: [i32; 6] = [1,2,3,4,5,6]; //6 --> number of elements in the array  
4     println!("First element from array1: {}", array1[0]);  
5     println!("Second element from array2: {}", array2[1]);  
6 }  
7
```

First element from array1: a
Second element from array2: 2

Figure 17 - Array example

Data collection – structs and tuple

- What are Structs?

A structure is a type that is made up of other types. The elements of a structure are called fields. Like tuples, the fields in a structure can have several data types. A significant benefit of the structure type is that you can name each field so that it is clear what the value means [21].

Struct types are often defined outside of the main function and other functions in the Rust program. To define them, we enter the keyword struct followed by the struct name.

There are 3 different types of structures in Rust: classic structures, tuple structures and unit structures. The main difference between them is that they support different ways of grouping and working with data. In the paragraphs below we see then what distinguishes them [22].

- The **classic C** structures are the most frequently used. Each structure field has a name and a data type. One advantage of the classic structure definition is that you can access the value of a structure field by name. After defining a classic structure, the structure fields can be accessed using the syntax `<structure>.<field>`.
- **Tuple structs** are similar to classic structs, but the fields don't have names. To access the fields in a tuple struct, we use the same syntax as we do for indexing a tuple: `<tuple>.<index>`.

- **Unit structs** are most commonly used as markers.

Example:

```
// Classic struct
struct User { name: String, age: u8, email: String, active: bool }

// Tuple struct with data types only
struct Usernames(char, char, char, char, f32);

// Unit struct
struct Unit;

//we can give values inside the main function
fn main(){
    let mut user = User{
        name: String::from("SISTCA"),
        age: 24,
        email: String::from("sistca@example.com"),
        active: true,
    };
}

//Or outside the main function
fn build_user(name: String, email: String) -> User {
    User {
        name: email,
        email: username,
        age: 23,
        active: true ,
    }
}
```

Figure 18 - Structs types

- Enums variants to compound data

Enums are a way of defining custom data types in a different way than we do with structs [23].

We use the enum keyword to create an enum type, which can have any combination of the enum variants. Like structs, enum variants can have named fields, but they can also have fields without names, or no fields at all.

Enums allow you to define a type by enumerating its possible *variants*. Each variant in the enum is independent and stores different amounts and types of values.

The enum in our example has three variants of different types:

- First has no associated data type or data.
- Second has two fields with data types String and char.
- Third contains an anonymous struct with named fields x and y, and their data types (i64).

```

1 enum Example {
2     // An enum variant can be like a unit struct without fields or data types
3     First,
4     // An enum variant can be like a tuple struct with data types but no named fields
5     Second(String, char),
6     // An enum variant can be like a classic struct with named fields and their data types
7     Third { x: i64, y: i64 }
8 }
9

```

Figure 19 - Enum example

We can also work with enums in structs. For that we need to define a separate struct for each variant in the enum. Then, each variant in the enum uses the corresponding struct. The struct holds the same data that was held by the corresponding enum variant. This style of definition allows us to refer to each logical variant on its own.

Example:

```

1 // Define a tuple struct
2 struct Tuple(String, char);
3 // Define a classic struct
4 struct Classic { x: i64, y: i64 }
5
6 enum Example { First(bool), Second(Tuple), Third(Classic)}
7
8 fn main(){
9
10     let new = Classic { x: 100, y: 250 };
11     println!("{}", new.x, new.y);
12
13 }
14

```

Figure 20 - Enums and Structs

- Vector data type

As with arrays, vectors store multiple values that **have the same data type**. Unlike arrays, the size or length of a vector can grow or shrink at any time [24].

A common way to declare and initialize a vector is with the `vec!` macro. This macro also accepts the same syntax as the array constructor.

```

1 let three_nums = vec![15, 3, 30];
2 let zeroes = vec![0; 5];
3
4

```

Figure 21 - Initializing a vector

Vectors can also be created by using the `Vec::new()` method. This method of vector creation lets us add and remove values at the end of the vector. To support this behavior, we declare the vector variable as **mutable** with the `mut` keyword.

```
1 let mut fruit = Vec::new();  
2  
3
```

Figure 22 - Another vector declaration

To add a value to the end of the vector, we use the `push(<value>)` method and to remove the value at the end of the vector, we use the `pop()` method.

```
1 fn main() {  
2     let mut v = Vec::new();  
3     v.push(1);  
4     v.push(2);  
5     v.push(3);  
6     v.push(4);  
7     v.pop();  
8  
9     println!("{:?}", v)  
10 }
```

[1, 2, 3]

Figure 23 - Push and Pop

Vectors support indexing in the same manner as arrays. We can access element values in the vector by using an index. The first element is at index 0 and the last element is at vector length - 1.

- Working with functions

Functions are the primary way code is executed within Rust.

Function definitions in Rust start with the `fn` keyword. After the function name, we specify the function's input arguments as a comma-separated list of data types inside parentheses.



```

1 fn main() {
2     println!("Hello, world!");
3     presentation();
4 }
5
6 fn presentation() {
7     println!("My name is:");
8 }

```

Hello, world!
My name is:

Figure 24 - Function declaration

When a function has input arguments, we name each argument and specify the data type at the start of the function declaration.



```

1 fn main() {
2     let name = "Nick";
3     println!("Hello, world!");
4     presentation(name);
5 }
6
7 fn presentation(name: &str) {
8     println!("My name is {}", name);
9 }

```

Hello, world!
My name is Nick

Figure 25 - Function with input arguments

When a function returns a value, we add the syntax `-> <type>` after the list of function arguments and before the opening curly bracket for the function body. The arrow syntax `->` indicates that the function returns a value to the caller (fig 26).

In Rust, the usual practice is to return a value at the end of a function by having the final line of code in the function equal the value to be returned.



```

1 fn divide_by_4(number: u32) -> u32 {
2     number / 4
3 }
4
5 fn main() {
6     let number = 64;
7     println!("{}", number, divide_by_4(number));
8 }

```

64 divided by 4 = 16

Figure 26 - How to return a value

When explicitly using the `return` keyword, we end the statement with a semicolon. But, if we send a return value without using the `return` keyword, we do not end the statement with a semicolon.

Use if/else conditions

We can create conditional branches in our code by using the if and else keywords.

The if and else keywords are used with expressions to test values and do actions based on the test result. All conditional expressions result to a boolean: true or false.

```
1 fn main() {  
2     let number = 3;  
3  
4     if number < 5 {  
5         println!("condition was true");  
6     } else {  
7         println!("condition was false");  
8     }  
9 }
```

condition was true

Figure 27 - If and else conditions

Unlike most other languages, if blocks in Rust can also act as expressions. All execution blocks in the condition branches must return the same type for the code to compile.

```
1 fn main(){  
2     let formal = true;  
3     let greeting = if formal {  
4         "Good day to you."  
5     } else {  
6         "Hey!"  
7     };  
8     println!("{}", greeting)  
9 }
```

Good day to you.

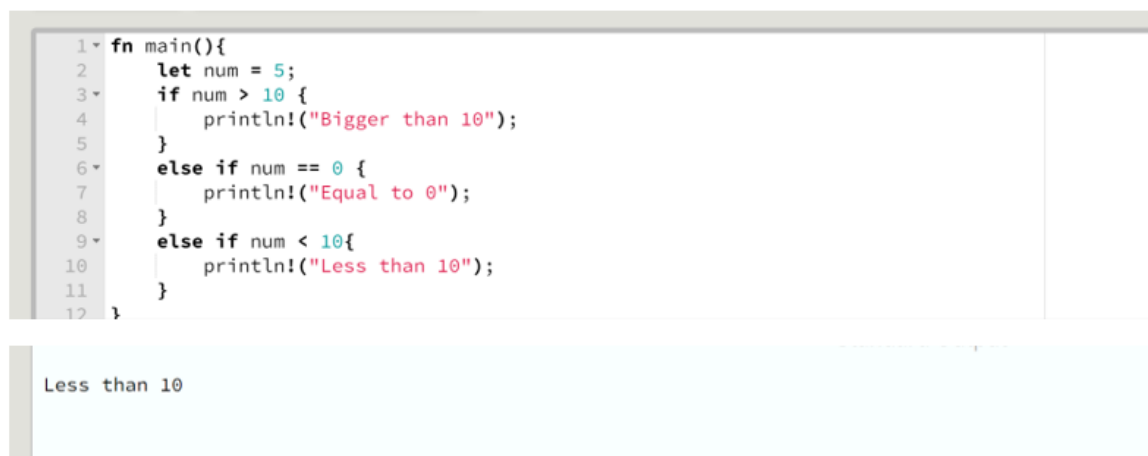
Figure 28 – If blocks as expressions

In this example, we assign a value to the greeting variable based on the result of the if formal expression. When the expression if formal is true, the greeting value is set to

the string "Good day to you." When the expression is false, the greeting value is set to the string "Hey!".

We can combine if and else together to form an else if expression. Several else if conditions can be used after the starting if condition, and before a closing else condition, which is optional.

If a condition expression evaluates to true, the corresponding action block is executed. Any following else if or else blocks are skipped. If a condition expression evaluates to false, the corresponding action block is skipped. Any following else if condition is evaluated. If all if and else if conditions evaluate to false, then any else block is executed. We can see in the example below that the first if expression is skipped since `num > 10` is false. The first else if is also skipped given that `num` is not equal to 0.

The image shows a code editor with Rust code and its output. The code is as follows:

```
1 fn main(){
2     let num = 5;
3     if num > 10 {
4         println!("Bigger than 10");
5     }
6     else if num == 0 {
7         println!("Equal to 0");
8     }
9     else if num < 10{
10        println!("Less than 10");
11    }
12 }
```

The output of the program, displayed in a light blue box at the bottom, is "Less than 10".

Figure 29 - Multiple test conditions

Using Loops

It's often convenient to run a block of code more than once. For this task, Rust provides various *loops*, which will run through the code inside the loop body to the end and then start immediately back at the beginning [25]. We will analyse the following three loops:

- loop: Repeat, unless a manual stop occurs.
 - while: Repeat while a condition remains true.
 - for: Repeat for all values in a collection.
-
- Loop

The loop expression creates an infinite loop. This keyword lets us repeat the actions in the expression body continuously. The actions repeat until we take some direct action to make the loop stop.

To stop a loop expression we can use the break keyword to set a break point. When the program encounters the break keyword, it stops executing the actions in the body of the loop expression and continues to the next code statement.

```
1 fn main(){
2     let mut counter = 5;
3     loop{
4         counter = counter+1;
5         if counter > 10 {
6             break counter;
7         }
8     };
9     println!("Counter value is {}", counter)
10 }
```

Counter value is 11

Figure 30 - Loop

- While

The while loop uses a conditional expression. The loop repeats as long as the conditional expression remains true. This keyword lets us execute the actions in the expression body until the conditional expression is false.

```
1 fn main(){
2     let mut counter = 0;
3     while counter < 5 {
4         println!("We loop a while...");
5         counter = counter + 1;
6     }
7 }
8
```

We loop a while...
We loop a while...
We loop a while...
We loop a while...
We loop a while...

Figure 31 - While loop

- For

The for loop uses an iterator to process a collection of items. The loop repeats the actions in the expression body for each item in the collection. This type of loop repetition is called iterating. When all iterations are complete, the loop stops.

We access the items in the collection by using the `iter()` method. The `for` expression binds the current value of the iterator to the result of the `iter()` method. In the expression body, we can work with the iterator value.

```
1 fn main(){
2     let fruits = ["apple", "orange", "banana"];
3     for fruit in fruits.iter() {
4         println!("{}", fruit);
5     }
6 }
7
```

```
apple is my favourite fruit.
orange is my favourite fruit.
banana is my favourite fruit.
```

Figure 32 - For loop

5. Exercise(s)

5.1 Problem A

The first program will show all Armstrong numbers (Numbers in which the sum of each digit to the power of the number of digits of the number equals itself, for example $153 = 1^3 + 5^3 + 3^3$) between the interval specified by the user.

In short; make a program that asks the user for the upper and lower limit numbers, one at a time, and then run through all the numbers in between and do the math to check if it is an Armstrong number. If it is, print it or put it in a list to print at the end of the loop.

Hint: Use the following lines of code to read input from the user.

```
std::io::stdin().read_line(&mut inicio).expect("Falha a ler linha");
let inicio=inicio.trim().parse::<u32>().unwrap();
```

5.1.1 (re)solution A

```
fn main(){

    let mut inicio = String::new();
    let mut fim = String::new();

    println!("Insira o início da busca");
    std::io::stdin().read_line(&mut inicio).expect("Falha a ler linha");
    let inicio=inicio.trim().parse::<u32>().unwrap();

    println!("Insira o fim da busca");
    std::io::stdin().read_line(&mut fim).expect("Falha a ler linha");
    let fim=fim.trim().parse::<u32>().unwrap();

    for n in inicio..=fim {
        let mut atual = n;
        let digitos = n.to_string().len().try_into().unwrap();
        let mut soma = 0;
        while atual > 0{
            let dig=atual % 10;
            soma = soma + u32::pow(dig, digitos);
            atual = atual / 10;
        };
        if soma == n {
            print!("{}",soma);
        }
    }
    println!();
}
```

5.2 Problem B

This next exercise will be a volume calculator, where the user fills in the details on the arguments of the executable used and then get an output based on those arguments, for example "solidos.exe esfera 5" → outputs "Volume da esfera: $V = (4 / 3) * \pi * \text{raio}^3 = 523.599$ (Result with 3 decimal places).

In this case, make a program that retrieves the arguments, checks if they are valid options, and then, for each of the cases of each different solid (cube, cone, sphere, cylinder and parallel), print the different formula and result. Make sure to cover the invalid cases (wrong / insufficient arguments) and make an option which tells the user the available options for arguments.

5.2.1 (re)solution B

```
use std::env;
use std::process;
use std::str::FromStr;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() <= 1 {println!("Erro - Usar: 'solidos ajuda' para
visualizar todas as opções disponíveis");process::exit(0);}

    match args[1].to_lowercase().trim() {

        "ajuda" => println!("Opções Disponíveis:\n\tCone raio
altura\n\tCubo lado\n\tCilindro raio altura\n\tEsfera raio\n\tParalelo
comprimento largura altura"),

        "cone" =>{
            if args.len() != 4 {println!("Erro - Numero de argumentos
insuficientes. Usar: 'solidos ajuda'");process::exit(0);}
            let raio = f64::from_str(&args[2]).ok();
            let altura = f64::from_str(&args[3]).ok();
            if let Some(raio) = raio {
                if let Some(altura) = altura {
                    println!("\n\tVolume do cone:  $V = \pi * r^2 * h / 3 =$ 
{resultado:.*}\n", 3, resultado
=(std::f64::consts::PI*raio*raio*altura/3.0).abs());
                } else {
                    println!("Medida Inválida");
                }
            } else {
                println!("Medida Inválida");
            }
        },

        "cubo" => {
            if args.len() != 3 {println!("Erro - Numero de argumentos
insuficientes. Usar: 'solidos ajuda'");process::exit(0);}
            let lado = f64::from_str(&args[2]).ok();
            if let Some(lado) = lado {
                println!("\n\tVolume do cubo:  $V = lado * 3 =$ 
{resultado:.*}\n", 3, resultado =(lado*lado*lado).abs());
            } else {
                println!("Medida Inválida");
            }
        },

        "cilindro" => {
            if args.len() != 4 {println!("Erro - Numero de argumentos
insuficientes. Usar: 'solidos ajuda'");process::exit(0);}
            let raio = f64::from_str(&args[2]).ok();
```

```

        if args.len() != 4 {println!("Erro - Numero de argumentos
insuficientes. Usar: 'solidos ajuda'");process::exit(0);}
        let raio = f64::from_str(&args[2]).ok();
        let altura = f64::from_str(&args[3]).ok();
        if let Some(raio) = raio {
            if let Some(altura) = altura {
                println!("\n\tVolume do cilindro:  $V = \pi * r^2 * h =$ 
{resultado:.*}\n", 3, resultado
=(std::f64::consts::PI*raio*raio*altura).abs());
            } else {
                println!("Medida Inválida");
            }
        } else {
            println!("Medida Inválida");
        }
    },

    "esfera" => {
        if args.len() != 3 {println!("Erro - Numero de argumentos
insuficientes. Usar: 'solidos ajuda'");process::exit(0);}
        let raio = f64::from_str(&args[2]).ok();
        if let Some(raio) = raio {
            println!("\n\tVolume da esfera:  $V = (4 / 3) * \pi * raio^3$ 
= {resultado:.*}\n", 3, resultado
=4.0/3.0*std::f64::consts::PI*(raio*raio*raio).abs());
        } else {
            println!("Medida Inválida");
        }
    }

    "paralelo" => {
        if args.len() != 5 {println!("Erro - Numero de argumentos
insuficientes. Usar: 'solidos ajuda'");process::exit(0);}
        let comprimento = f64::from_str(&args[2]).ok();
        let largura = f64::from_str(&args[3]).ok();
        let altura = f64::from_str(&args[4]).ok();
        if let Some(comprimento) = comprimento {
            if let Some(largura) = largura {
                if let Some(altura) = altura {
                    println!("\n\tVolume do parapelelo:  $V = comprimento
* largura * altura =$  {resultado:.*}\n", 3, resultado
=(comprimento*largura*altura).abs());
                } else {
                    println!("Medida Inválida");
                }
            } else {
                println!("Medida Inválida");
            }
        } else {

```

```

        println!("Medida Inválida");
    }
}
_ => println!("Opção Inválida. Usar: 'solidos ajuda'"),
}
}

```

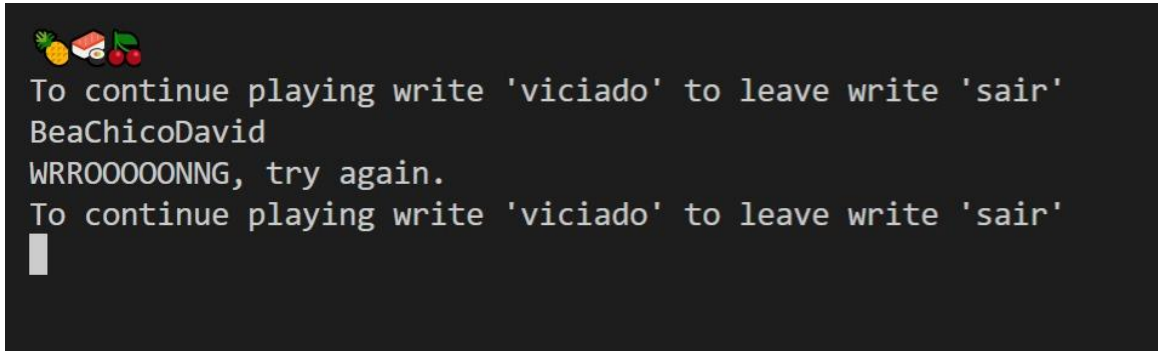
6. Challenge

Taking advantage of the num-traits, rand and std crates. Create a simple slot machine with the following specifications:

- Must have 3 displays. Each display must have at least 5 symbols/emoji.
- The resulting symbol/emoji present on each display must come from a randomly generated value that after passing through at least 2 "test" functions gives it's outputs to be computed by each display. There can't be repeated "test" functions but as long as the test mechanism is different you can test the same thing. Example of "test" function: parity test, if it's even returns 1 odd returns 0.
- You must be able to choose when the slot machine runs and every run must be started with a "simulation" of the behavior of a slot machine.
- Each display "choose" function must have a different selection structure for the symbol/emoji to be displayed.
- You need to use at least one constant and one static variable.
- At the end of each run you must request input from the player to see if he wants to play the slot again or leave the program.
- The "play again" mechanism must be activated by key words for the "play again" and "leave program" action and an error check, that, if a spelling error is detected, notifies the user and asks again.
- For the rand function use at least 2 instances of "power" for the upper limit.
- Create a mechanism for automatic jackpot after a certain amount of tries.
- Adjust the slot machine so that only after at least 30 tries can you have the chance for jackpot.

Tip: use a vector for the symbol/emoji bank and make use of the sleep functionality to create the "simulation" of the slot machine initial behavior.

Check the example below to have an idea how the program should work:

A terminal window with a dark background. At the top left, there are three emojis: a pineapple, a watermelon slice, and two cherries. Below them, the text "To continue playing write 'viciado' to leave write 'sair'" is displayed in a light gray font. This is followed by the username "BeaChicoDavid" in a standard white font. Then, the text "WRRROOOONNG, try again." appears in the same light gray font. The prompt text "To continue playing write 'viciado' to leave write 'sair'" is repeated. At the bottom, there is a single white character, possibly a cursor or a space.

```
🍍🍉🍒  
To continue playing write 'viciado' to leave write 'sair'  
BeaChicoDavid  
WRRROOOONNG, try again.  
To continue playing write 'viciado' to leave write 'sair'  
█
```

Figure 33 - Console example

7. References

- [1] "7 Reasons Why You Should Use Rust Programming For Your Next Project." [Online]. Available: <https://simpleprogrammer.com/rust-programming-benefits/>.
- [2] "What is Rust?" [Online]. Available: <https://docs.microsoft.com/en-us/learn/modules/rust-introduction/2-rust-overview>.
- [3] "2021 Developer Survey." [Online]. Available: <https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages>.
- [4] "What is Ownership?" [Online]. Available: <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>.
- [5] "Traits: Defining Shared Behavior." [Online]. Available: <https://doc.rust-lang.org/book/ch10-02-traits.html>.
- [6] "Macros." [Online]. Available: <https://doc.rust-lang.org/book/ch19-06-macros.html>.
- [7] "Rust vs Go – what do you need to know about these programming languages?" [Online]. Available: <https://codilime.com/blog/rust-vs-go-what-do-you-need-to-know-about-these-programming-languages/>.
- [8] "Go vs. Rust performance comparison: The basics." [Online]. Available: <https://www.getclockwise.com/blog/rust-vs-go>.
- [9] "Rust vs C++: Will Rust Replace C++ in Future?" [Online]. Available: <https://www.geeksforgeeks.org/rust-vs-c-will-rust-replace-c-in-future/>.
- [10] "Rust vs C++: Which Technology Should You Choose?" [Online]. Available: <https://www.ideamotive.co/blog/rust-vs-cpp-which-technology-should-you-choose>.
- [11] "Rust vs C++ and Is It Good for Enterprise?" [Online]. Available: <https://www.incredibuild.com/blog/rust-vs-c-and-is-it-good-for-enterprise>.
- [12] "In Rust We Trust." [Online]. Available: <https://analyticsindiamag.com/in-rust-we-trust/>.
- [13] "Why is Rust programming language so popular?" [Online]. Available: <https://codilime.com/blog/why-is-rust-programming-language-so-popular/>.
- [14] "Introduction to Rust." [Online]. Available: <https://chercher.tech/rust/introduction-rust>.
- [15] "Functions." [Online]. Available: <https://doc.rust-lang.org/stable/rust-by-example/fn.html>.
- [16] "Understand the basic Rust program structure." [Online]. Available: <https://docs.microsoft.com/en-us/learn/modules/rust-create-program/1-program-structure>.
- [17] "Variables and mutability." [Online]. Available: <https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html>.

- [18] "Data types." [Online]. Available: <https://doc.rust-lang.org/book/ch03-02-data-types.html>.
- [19] "Tuples." [Online]. Available: <https://doc.rust-lang.org/stable/rust-by-example/primitives/tuples.html>.
- [20] "Arrays and slices." [Online]. Available: <https://doc.rust-lang.org/stable/rust-by-example/primitives/array.html>.
- [21] "Defining and Instantiating Structs." [Online]. Available: <https://doc.rust-lang.org/book/ch05-01-defining-structs.html>.
- [22] "Define data collections by using tuples and structs." [Online]. Available: <https://docs.microsoft.com/en-us/learn/modules/rust-create-program/4-tuples-structs>.
- [23] "Use enum variants for compound data." [Online]. Available: <https://docs.microsoft.com/en-us/learn/modules/rust-create-program/5-enum-variants>.
- [24] "Storing Lists of Values with Vectors." [Online]. Available: <https://doc.rust-lang.org/book/ch08-01-vectors.html?highlight=vectors#reading-elements-of-vectors>.
- [25] "Control Flow." [Online]. Available: <https://doc.rust-lang.org/book/ch03-05-control-flow.html>.