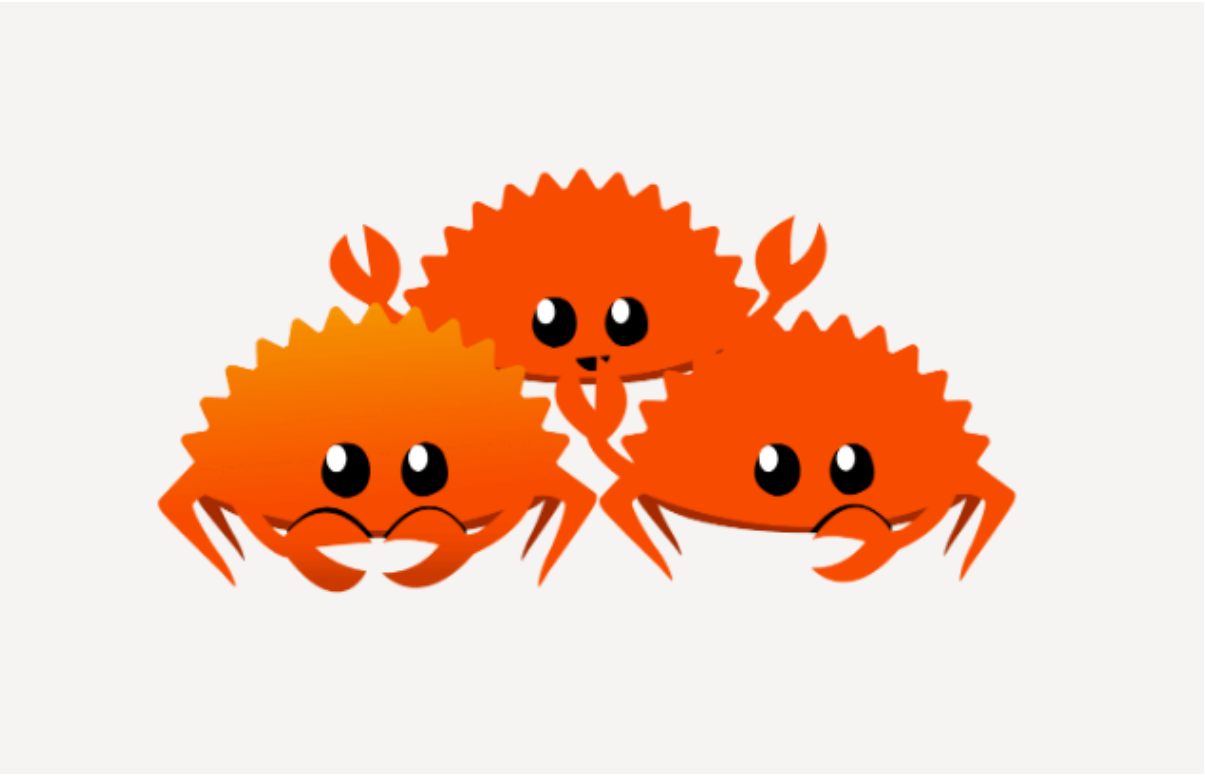
WORKBOOK

PVD816F Programming in Rust

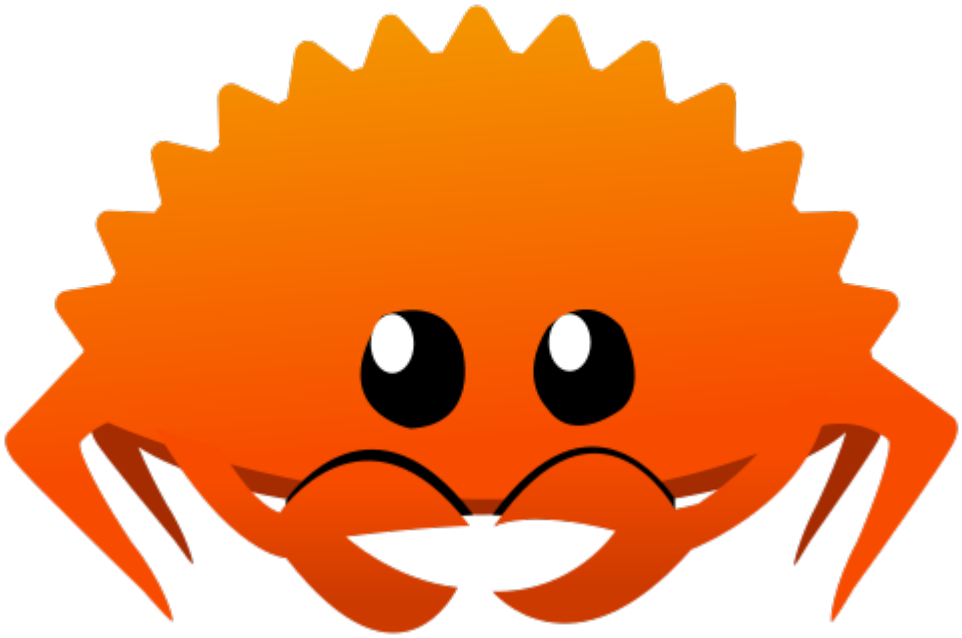
Pascal Van Dam
"Let us orchestrate your success!" #K8SMastery

Author(s):

Contents:



1 Rust Setup



1.1 Introduction

In this lab we will setup the Rust SDK and write and run the famous helloworld program in Rust.

1.1.1 Requirements

To be able to execute this lab, you need one Ubuntu Noble 24.04 (LTS) or compatible machine. The virtual machines must be provided with at least 4 GB Memory and 2 CPUs.

Furthermore, you need an internet connection to be able to install the Rust SDK and to install additional Rust `crates` when needed.

1.1.2 Installing the Rust SDK

Rust offers a handy tool called `rustup` that is used to install the Rust SDK and to keep it up-to-date. Use the following command as your normal unprivileged user to install the rustup tool and to install the Rust SDK.

Listing 1: Linux

```
[~ $> curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Listing 2: Mac/OSX

```
[~ $> curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Listing 3: Windows

```
Download and execute rustup-init.exe from https://static.rust-lang.org/rustup/dist/i686-pc-
↪windows-gnu/rustup-init.exe
```

When asked for to choose an installation option, please select `1) Proceed with installation (default)`

After the `rustup` has finished, please `log out` and `log in` again to activate the Rust environment.

Now it's time to validate our Rust SDK installation:

```
[~ $> rustc --version
```

This reports the version of the Rust compiler.

You should be able to get a response like:

```
rustc 1.93.0 (254b59607 2026-01-19)
```

Kindly create a directory called `temp-rust` and change into it:

```
[~ $> mkdir temp-rust
[~ $> cd temp-rust
```

In this `temp-rust` directory, create a program callled helloworld.rs with the following content:

Listing 4: helloworld.rs

```
1 // helloworld.rs
2
3 // This is the main function
4 fn main() {
5
6     println!("Hello World!");
7
8 }
```

You can compile the program using the `rustc` command:

```
[~ $> rustc helloworld.rs
```

This will produce a binary called `helloworld` which can be executed like this:

```
[~ $> ./helloworld
```

Observer the size of the binary using the `ls -alh helloworld` command.

```
[~ $> ls -alh helloworld
```

```
-rwxrwxr-x 1 pascal pascal 3.8M Oct  8 11:57 helloworld
```

The binary is setting you back on 3.8MiB of storage. A beefy binary, don't you think? There's a lot of extra information compiled in this binary. We can slim it down for now using the UNIX/Linux `strip` utility. Later in the course we will use Rust tool `cargo` to make a binary `release ready` for us.

```
[~ $> strip helloworld
[~ $> ls -alh helloworld
```

```
-rwxrwxr-x 1 pascal pascal 315K Oct  8 12:01 helloworld
```

We have stripped it down to 315KiB, that's better.

Now, let's remove the directory and it's contents as we are going to use another way to build Rust programs a little later in this lab:

```
[~ $> rm -rf temp-rust
```

1.1.3 Additional rustup functionality

With `rustup` we can do more things:

1. Update the installed release to the latest version
2. Update itself (`rustup`)
3. Remove your installation of Rust
4. Add/Remove extra rust components
5. Configure Rust to compile to different targets (Cross-compiling)
6. Install a rust toolchain from default, beta or nightly channels
7. Choose which toolchain default, beta or nightly should be used

Cross-compiling is a different module in our course, so we will save that for later.

Let's now first make sure that `rustup` is of the latest version:

```
[~ $> rustup self update
```

We will either get a conformation that we do have the latest version, or that a new version is going to be installed.

Next, let's make sure our toolchain in the default channel is fresh:

```
[~ $> rustup update
```

The tool `rustup` will either confirm that the toolchain is up-to-date, or upgrade to the most recent version in the stable channel.

For Rust extensions in editors like Vim, Neovim or VScode, we need to have the so called `rust-analyzer` installed. Let's check which components are available for installation and install the `rust-analyzer` if it's available for our platform.

First, list the available components with the following command:

```
[~ $> rustup component list
```

The components that are listed in BOLD are already installed. We need to install `rust-analyzer-x86_64-unknown-linux-gnu` given that we are on Linux on an x86_64 architecture.

```
[~ $> rustup component add rust-analyzer-x86_64-unknown-linux-gnu
```

BTW; the other components listed give an indication how many platforms are currently supported by Rust, and the list is still growing.

To remove your rust setup you can use the `rustup self uninstall` command. But we are not going to practice that now, we have a whole course ahead of us where we need this Rust environment.

1.1.4 Cargo

We almost never invoke the `rustc` compiler ourselves. Instead Rust provides us with the `cargo tool`. This tool helps us performing a lot of tasks like:

1. Setting up a work environment for our new Rust program
2. Compiling/building a Rust program in debug mode
3. Compiling/building a Rust program in release mode
4. Testing a Rust program
5. Formatting a Rust program (pretty printing)
6. Linting our Rust program using Clippy

But also some of the more advanced tasks like `cross-compiling` our Rust program for different architectures.

Detailed information about the use of `Cargo` will follow in the rest of the course.

First let's use `Cargo` to setup the environment for our `helloworld` program. For this we use the `cargo init` command.

```
[~ $> cargo init helloworld
```

A new directory `helloworld` will be created with the following contents:

In this directory you will see that the following contents have been created:

```
helloworld
├── Cargo.toml
└── src
    └── main.rs
```

The Cargo.toml is the so called `Manifest` file and contains all the info needed for building the program.

Take a look at the contents of the file:

Listing 5: Cargo.toml

```
[package]
name = "helloworld"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.
↪html

[dependencies]
```

In the section `[package]` you can see the name of the program/package, called helloworld. The version is the version of our program/package. The `edition` parameter states the edition of Rust our program is supposed to be compatible with, in this case the Rust 2021 edition.

There are no dependencies yet, but if they were there, you would be able to find them underneath the `[dependencies]` section.

In the `src` directory you will find a predefined `main.rs` file containing `helloworld` functionality.

Listing 6: helloworld.rs

```
1 fn main() {
2     println!("Hello, world!");
3 }
```

We can build the program by switching to the helloworld (`cd helloworld`) directory and, in this directory, using the `cargo build` command like this:

```
[~ $> cargo build
```

The resulting output will be something like this:

```
Compiling helloworld v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/helloworld)
 Finished dev [unoptimized + debuginfo] target(s) in 0.89s
```

Please note that you can see that the executable gets run from another directory `target/debug`.

Check whether you can file the `helloworld` executable in the `target/debug` directory.

With the `cargo run` command the binary will be build from the source and run directory. As we already have compiled our program in the previous step, the compilation step will be omiited.

```
[~ $> cargo run
```

As seen above, the default build will build an executable suitable for debugging, but it's very large. To build for an optimized executable for production, we use the so called 'release' mode of the `cargo build` command like this:

```
[~ $> cargo build --release
```

```
Compiling helloworld v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/helloworld)
 Finished release [optimized] target(s) in 0.42s
```

You can find the resulting optimized binary in the `target/release` directory. Here, `optimized` doesn't mean a significantly slimmer binary. Optimized for release means that in contrast to the debug build, the compiler will take more effort in optimizing for performance whilst taking a longer time to compile. The builds in `debug` mode (the default mode) will result in less optimized executables but faster compilation times.

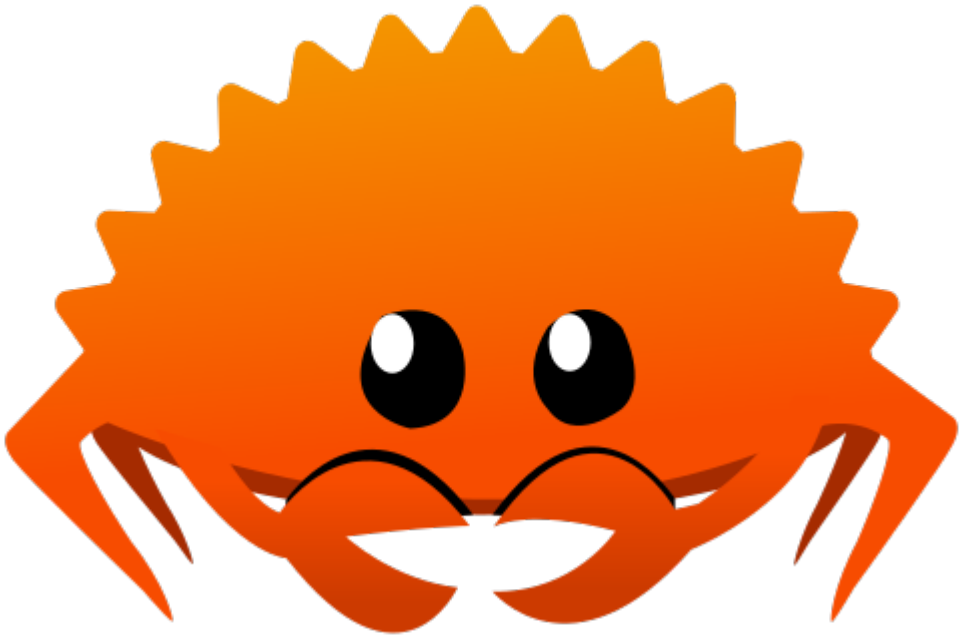
To clean up the build environment (removing build artifacts), one can use the following command:

```
[~ $> cargo clean
```

1.2 Summary

- The tool `rustup` is used to install the Rust SDK and maintain it.
- The tool `cargo` is used to scaffold our Rust projects, build, analyze, test and and run Rust code.
- The toml file `cargo.toml` contains information about our project.

2 Setting up an IDE for Rust



2.1 Introduction

In this lab we will help you setup your editor for programming in Rust. As there's phletora of editors and Integrated Development Environments (IDE) out here we only focus on two of them and make a mention of a third one.

We will discuss setting up:

- Vim
- VScode

2.1.1 VIM

Vim is an improved version of the trusted UNIX VI editor. It has evolved into a complete IDE and has a loyal flock of followers, but there are also people that have difficulties getting a hold of it. A pro of Vim is that you can literally get if for any environment and it's very lightweight. It doesn't require a graphical environment and it works easilly in resource deprived environments. Currently, there are many plugins available that make programming in Rust with VIM as your editor a pleaseant and productive experience.

We assume you have Vim installed.

To install a set of plugins for Rust in Vim, kindly execute the following recipe:

```
[~ $> git clone https://thegitcave.org/pascal/pvd836-rust-vim.git
[~ $> curl -sLO install-node.vercel.app/lts
[~ $> sudo sh ./lts -y
[~ $> cp itg836-rust-vim/vimrc ~/.vimrc
[~ $> vim +'PlugInstall --sync' +qa
[~ $> vim +'CocInstall coc-rust-analyzer coc-yaml coc-json -sync' +qa
```

It might be the case that you need to hit the ENTER key after you have executed the 5th line due to a at that time missing color-scheme.

2.1.2 VScode

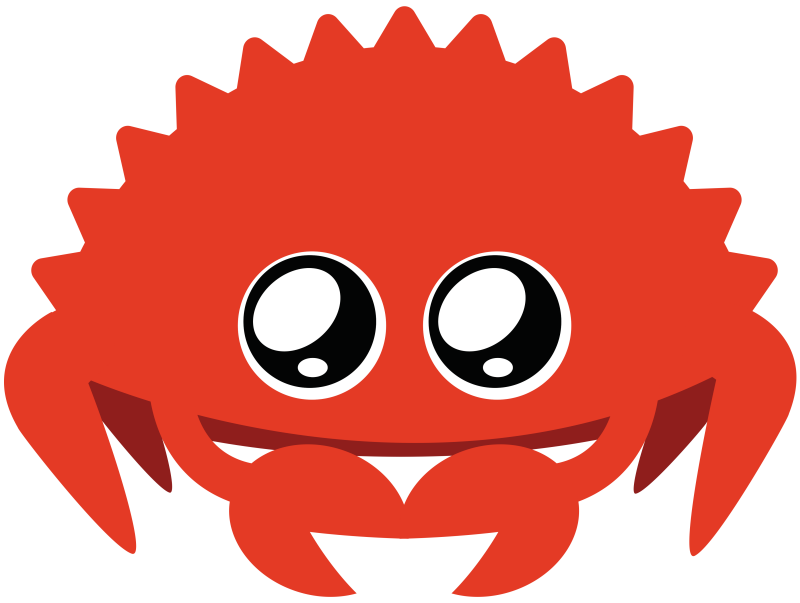
For VScode from Microsoft, it is important to install the proper 'extensions' for Rust. It is assumed you have VScode already installed on your platform. To install the advised extensions for rust please use the following commands from the commandline or look up the extensions in the VScode themselves:

```
[~ $> code --install-extension vadimcn.vscode-lldb
[~ $> code --install-extension serayuzgur.crates
[~ $> code --install-extension bungcip.better-toml
[~ $> code --install-extension rust-lang.rust-analyzer
```

2.1.3 Jetbrains IDE

Many developers use Jetbrains set of IDEs for their programming jobs. For the IntelliJ IDEA and CLion offerings you can download an OpenSource Rust plugin. This plugin can be found at: https://www.jetbrains.com/rust/download/

3 Introduction to Rust



3.1 Introduction

In this lab, we will get a gentle introduction into the Rust language.

3.1.1 Requirements

To be able to execute this lab, you need to have your Rust environment setup with `rustup`

3.1.2 A simple rust program

We are going to create a Rust program called `greeting`. Of course, we are going to setup the environment for our code using Cargo.

```
[~ $> cargo new greeting
[~ $> cd greeting
```

Using your favorite editor, replace the contents of the `src/main.rs` with the following program:

Listing 1: greeting

```
//
// greeting
//

fn greeting() {
    println!("Hello World!");
}

fn main() {
    greeting();
}
```

Here, `//` denote comments. `greeting` is a function prefixed as such with `fn`, its body enclosed in curly braces. The `entry` point of the program is the `main` function.

In the body of `greeting` function, you will see we will call `println!("Hello World!");`. Mind that this is NOT a function but a `macro`. It's code that generates code. We will use these `macro's` a lot in Rust programming and discuss how to create them ourselves later in the course. Anyhow this `println!` macro enables us to print something to `stdout` and finish it with a `newline`.

We already wrote that Cargo is a very useful tool while writing Rust programs. Now that we have written the code for our `greeting` program, let's consult `clippy` about any issues it has with our code, wether it is syntax errors or logical erros that could cause the program to do the things we say, but not the things we want.

Run `clippy` using the following command.

Listing 2: clippy

```
[~ $> cargo clippy
```

Listing 3: Output

```
Checking greeting v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/greeting)
Finished dev [unoptimized + debuginfo] target(s) in 0.60s
```

In this case, `Clippy` has no constructive criticism on our code.

Let's look at another example.

Listing 4: clippy-2

```
fn sum(a: i64, b: i64) -> i64 {
    return a + b;
}

fn main() {
    let a: i64 = 10;
```

(continues on next page)

(continued from previous page)

```
    let b: i64 = 20;

    let s = sum(a, b);
    println!("Sum of {} and {} is {}", a, b, s);
}
```

We see some things we haven't learned yet, but that will be discussed in the the modules further in the course. Key is this code will compile without any issue. Please try it.

Create the `clippy-2` project with `cargo new clippy-2` and put above program text in `clippy-2/src/main.rs` Then build the program using `cargo build`.

Listing 5: compile clippy-2

```
[~ $> cd clippy-2
[~ $> cargo build
```

Listing 6: Output

```
Compiling clippy-2 v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/clippy-2)
Finished dev [unoptimized + debuginfo] target(s) in 0.58s
```

The code is correct and if you execute it with `cargo run`, it will give you the expected and correct results. However, it is NOT `idiomatic` rust. Kindly consult `clippy` with `cargo clippy` and see what constructive advise it has for us.

Listing 7: cargo clippy

```
[~ $> cargo clippy
```

Listing 8: Output

```
Checking clippy-2 v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/clippy-2)
warning: unneeded `return` statement
 --> src/main.rs:2:5
    |
  2 |     return a+b ;
    |     ^^^^^^^^^^^ help: remove `return`: `a+b`
    |
  = note: `#[warn(clippy::needless_return)]` on by default
  = help: for further information visit https://rust-lang.github.io/rust-clippy/master/index.
↪html#needless_return

warning: `clippy-2` (bin "clippy-2") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.27s
```

What is the meaning of this? Well, when we will learn how to use functions and expressions in generic we will learn that Rust has the concept of tail expressions that can make Rust code more readable. What this means for a function is that if the last line of a function is an expression and it's not ended with a `;`, the value of that expression will be used as the return value of the function. E.g. no need to use the `return` statement explicitly. The use of the `return` statement at the end of a function is correct and compilable Rust, however, it's not Idiomatic Rust. Your friendly Code Coach Clippy helps you in crafting Idiomatic Rust code.

You can 'rewrite' the code yourself or ask Clippy to do that for you:

Listing 9: clippy fixit

```
[~ $> cargo clippy --fix
```

Listing 10: Output

```
  Checking clippy-2 v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/clippy-2)
     Fixed src/main.rs (1 fix)
```

(continues on next page)

(continued from previous page)

```
   Finished dev [unoptimized + debuginfo] target(s) in 0.20s
```

3.1.3 Displaying output in Rust

To learn Rust or any language, it is very handy to know how to print output on the screen to see what results our programs produce. As seen in a previous lab Rust introduces the `println!` macro for this.

In this section we will learn how to work with this `println!` macro to produce orderly output from out Rust programs.

Kindly create a new project called `printing-1` with `cargo new` and put the following program text in `printing-1/src/main.rs`:

Listing 11: printing-1

```
fn main() {
    let a: i64 = 42 ;
    let s = "Hello Rustaceans" ;
    let h: u32 = 4207849484 ;
    let fname = "Larry" ;
    let lname = "Wall" ;

    println!() ;
    println!("a == {}",a) ;
    println!("s == {}",s) ;
    println!("x == {:X}",h) ;
    println!("x == {:x}",h) ;

    println!("{0} {1}",fname,lname)
}
```

The `{}` are so called placeholders that will be replaced with the value(s) of the expression(s) supplied. These placeholders can also be `numbered`, after which they refer to the first, second, third etc expression specified in the `println!` macro. This macro can also do more nice things like formatting our output or printing it in a certain base like hex, octal, decimal, binary etc. For more info please refer to: https://doc.rust-lang.org/rust-by-example/hello/print.html

Now let's do an excercise. We are going to print a first and last name in normal and lastname, firstname order. Kindly use two methods to accomplish this.

Create a new project called `printing-2` and use the following code as basis for your printing-2/src/main.rs file:

Listing 12: printing-2

```
// printing-2

fn main() {
    let fname = "Linus" ;
    let lname = "Torvalds" ;

    // Add printing of firstname lastname and lastname, firstname in two ways
    // 1) Using {} placeholders
    // 2) Using positional placeholders {n}
    //

    println!("Change me!")
}
```

3.2 Challenges: Mastering the `println!` macro

Before we dive into the unique aspects of Rust, let's first get comfortable with the `println!` macro and its formatting capabilities. Being able to produce well-formatted output will be invaluable as we explore more complex Rust concepts.

For each challenge, create a new Cargo project, write the code, and make sure it compiles and runs.

3.2.1 Challenge P1: Number bases — hex, octal, and binary

As systems programmers, we often need to inspect values in different number bases. The `println!` macro has built-in format specifiers for this.

Create a new project called `fmt-bases`:

```
[~ $> cargo new fmt-bases
[~ $> cd fmt-bases
```

Put the following code in `fmt-bases/src/main.rs`:

Listing 13: fmt-bases/src/main.rs

```
fn main() {
    let value: u32 = 255;

    println!("Decimal     : {}", value);
    println!("Hex (lower) : {:x}", value);
    println!("Hex (upper) : {:X}", value);
    println!("Octal       : {:o}", value);
    println!("Binary      : {:b}", value);

    // With the 0x / 0o / 0b prefixes using the # flag
    println!("Hex prefixed    : {:#x}", value);
    println!("Octal prefixed  : {:#o}", value);
    println!("Binary prefixed : {:#b}", value);
}
```

Tasks:

1. Compile and run the program. Write down each line of output.
2. Change `value` to `48879` (which is `0xBEEF` in hex). Run the program again and verify the hex output.
3. Add a new line that prints the value `0xDEAD_BEEF` as a 32-bit binary string with the `0b` prefix. How many bits do you see?

Key takeaway

Use `:x`, `:o`, and `:b` for hex, octal, and binary output. The `#` flag adds the conventional prefix (`0x`, `0o`, `0b`). These are essential when working with bitwise operations, memory addresses, or register values.

3.2.2 Challenge P2: Padding and alignment

When printing tables or aligned output, we need to control the width of our fields and the alignment of text within them.

Create a new project called `fmt-align`:

```
[~ $> cargo new fmt-align
[~ $> cd fmt-align
```

Put the following code in `fmt-align/src/main.rs`:

Listing 14: fmt-align/src/main.rs

```
fn main() {
    // Right-aligned (default for numbers), width 10
    println!("Right : [{:>10}]", "Rust");
    println!("Left  : [{:<10}]", "Rust");
    println!("Center: [{:^10}]", "Rust");

    // Padding with a custom fill character
    println!("Dashed: [{:->10}]", "Rust");
    println!("Dotted: [{:.>10}]", "Rust");
    println!("Stars : [{:*^10}]", "Rust");

    // Numbers with zero-padding
    println!("Zero-padded: [{:05}]", 42);
    println!("Zero-padded: [{:08b}]", 42);
}
```

Tasks:

1. Compile and run the program. Observe the alignment and fill characters.
2. Create a small "table" that prints the following three items right-aligned in a 15-character column with dot-fill:

   ```
   Item...........Price
   Keyboard.........€49
   Monitor..........€299
   Mouse............€25
   ```

   Hint: you can use two fields per line, one left-aligned and one right-aligned.
3. Print the numbers 1 through 16 in binary, each zero-padded to 8 bits wide. (Hint: use a `for` loop: `for i in 1..=16 { ... }`)

Key takeaway

Format specifiers `<`, `>`, and `^` control left, right, and center alignment. A fill character can be placed before the alignment symbol. Use `0` before the width for zero-padding numbers. Combining these gives you full control over tabular output.

3.2.3 Challenge P3: Floating point precision

When dealing with floating point numbers, controlling the number of decimal places is essential for clean output.

Create a new project called `fmt-precision`:

```
[~ $> cargo new fmt-precision
[~ $> cd fmt-precision
```

Put the following code in `fmt-precision/src/main.rs`:

Listing 15: fmt-precision/src/main.rs

```
fn main() {
    let pi = std::f64::consts::PI;

    println!("Default    : {}", pi);
    println!("2 decimals : {:.2}", pi);
    println!("5 decimals : {:.5}", pi);
    println!("8 decimals : {:.8}", pi);

    // Combining width and precision
    println!("Padded     : [{:>12.4}]", pi);
    println!("Zero-padded: [{:012.4}]", pi);

    // Scientific notation
    println!("Sci (lower): {:e}", pi);
    println!("Sci (upper): {:E}", pi);
    println!("Sci precise: {:.3e}", pi * 1000.0);
}
```

Tasks:

1. Compile and run the program. Note how different precision values affect the output.
2. Euler's number is available as `std::f64::consts::E`. Print it with exactly 10 decimal places.
3. The speed of light is approximately `299_792_458.0` m/s. Print it in scientific notation with 2 decimal places.
4. Create a small conversion table that prints temperatures from 0°C to 100°C in steps of 10, each with their Fahrenheit equivalent formatted to 1 decimal place:

   ```
   Celsius | Fahrenheit
   --------+-----------
      0.0  |      32.0
     10.0  |      50.0
     ...
   ```

   The formula is: `F = C * 9.0/5.0 + 32.0`

Key takeaway

Use `:.N` for N decimal places. Use `:e` or `:E` for scientific notation. Width and precision can be combined as `:W.P` where W is the total width and P is the number of decimal places. Rust pulls common mathematical constants from `std::f64::consts`.

3.2.4 Challenge P4: Debug printing with {:?} and {:#?}

The {} placeholder uses the Display trait, which is designed for user-facing output. But not every type implements Display. The **Debug** trait, invoked with {:?}, is available on most types and is invaluable for inspecting data structures during development.

Create a new project called fmt-debug:

```
[~ $> cargo new fmt-debug
[~ $> cd fmt-debug
```

Put the following code in fmt-debug/src/main.rs:

Listing 16: fmt-debug/src/main.rs

```
fn main() {
    // Tuples don't implement Display, but they implement Debug
    let point = (3, 7);
    // println!("{}", point);    // ← this would NOT compile!
    println!("Debug  : {:?}", point);

    // Arrays
    let primes = [2, 3, 5, 7, 11, 13];
    println!("Primes : {:?}", primes);

    // Vectors
    let colors = vec!["red", "green", "blue"];
    println!("Colors : {:?}", colors);

    // Pretty-print with {:#?}
    let matrix = vec![
        vec![1, 2, 3],
        vec![4, 5, 6],
        vec![7, 8, 9],
    ];
    println!("Matrix (compact) : {:?}", matrix);
    println!("Matrix (pretty)  :\n{:#?}", matrix);

    // Ranges
    let range = 1..=10;
    println!("Range  : {:?}", range);
}
```

Tasks:

1. Compile and run the program. Compare the compact {:?} output with the pretty-printed {:#?} output.
2. Uncomment line 4 (println!("{}", point);). What error do you get? What trait is missing?
3. Create a tuple with mixed types: let mixed = ("Rust", 2015, true, 3.14); and print it using {:?}. Does it work?
4. The dbg! macro is another useful debugging tool. Try the following and observe how it differs from println!:

```
let x = 5;
let y = dbg!(x * 2) + 1;
dbg!(y);
```

What extra information does dbg! print that println! does not?

Key takeaway

Use {:?} for Debug output on types that don't implement Display (tuples, arrays, vectors, etc.). Use {:#?} for pretty-printed Debug output. The dbg! macro is a handy alternative that prints the file, line number, expression, and value — perfect for quick debugging.

3.2.5 Challenge P5: Named parameters and expressions in `println!`

The `println!` macro supports **named parameters**, which make complex format strings much more readable. You can also use simple expressions directly inside curly braces (since Rust 1.58).

Create a new project called `fmt-named`:

```
[~ $> cargo new fmt-named
[~ $> cd fmt-named
```

Put the following code in `fmt-named/src/main.rs`:

Listing 17: fmt-named/src/main.rs

```
fn main() {
    // Named parameters
    println!("{language} was released in {year}",
        language = "Rust",
        year = 2015
    );

    // Captured variables (Rust 1.58+)
    let city = "Rotterdam";
    let country = "Netherlands";
    println!("{city} is in the {country}");

    // Combining named parameters with formatting
    println!("Hex: {value:#06x}", value = 255);
    println!("Bin: {value:#010b}", value = 255);

    // Numbered AND named can be mixed (though not usually recommended)
    println!("{0} loves {language}. Yes, {0} does!",
        "Ferris",
        language = "Rust"
    );
}
```

Tasks:

1. Compile and run the program.
2. Using named parameters, print the following sentence:
   `"My name is <name>, I am <age> years old and I live in <city>."`
   Use named parameters in the format string and fill in your own details.
3. Create two variables `width` and height“ of type `f64` and calculate the area. Print the result using captured variables:

   ```
   Rectangle: 12.50 x 7.30 = 91.25 m²
   ```

   Use precision formatting to show 2 decimal places.
4. Why are named parameters more readable than positional ones in long format strings?

Key takeaway

Named parameters (`{name = value}`) and captured identifiers (`{variable}`) make format strings self-documenting and less error-prone than positional arguments, especially when a format string has many placeholders.

3.3 Challenges: Variables, Ownership & Friends

Now that we know how to create projects with Cargo, produce well-formatted output, and use Clippy, it's time to explore the concepts that make Rust unique. The following 8 challenges will introduce you to **immutable** and **mutable** variables, **shadowing**, **ownership**, and **borrowing** — the cornerstones of Rust's safety guarantees.

For each challenge, create a new Cargo project, write the code, make sure it compiles and runs, and then answer the questions.

Important:

Some of these challenges contain code that will **not compile on purpose**. That is the exercise! Read the compiler error, understand what Rust is telling you, and fix the code.

3.3.1 Challenge 1: Immutable variables — the default

In Rust, variables are **immutable by default**. This means once you bind a value to a name, you cannot change it.

Create a new project called `challenge-1`:

```
[~ $> cargo new challenge-1
[~ $> cd challenge-1
```

Put the following code in `challenge-1/src/main.rs`:

Listing 18: challenge-1/src/main.rs

```
fn main() {
    let speed = 88;
    println!("The DeLorean needs to reach {} mph!", speed);

    speed = 100;
    println!("Actually, let's go {} mph!", speed);
}
```

Tasks:

1. Try to compile the program with `cargo build`. What error do you get?
2. Read the compiler message carefully. Rust tells you *exactly* what is wrong and how to fix it.
3. Fix the program so that it compiles and runs. What keyword did you add?

Key takeaway

Variables in Rust are immutable by default. The compiler protects you from accidentally changing values you didn't intend to change. To opt-in to mutability, use `let mut`.

3.3.2 Challenge 2: Mutable variables — opting in

Now that you know about `mut`, let's practice using mutable variables intentionally.

Create a new project called `challenge-2`:

```
[~ $> cargo new challenge-2
[~ $> cd challenge-2
```

Put the following code in `challenge-2/src/main.rs`:

Listing 19: challenge-2/src/main.rs

```
fn main() {
    let mut countdown = 10;

    println!("Launch sequence initiated!");

    while countdown > 0 {
        println!("T-minus {}...", countdown);
        countdown -= 1;
    }

    println!("🚀 Liftoff!");
}
```

Tasks:

1. Compile and run this program. Observe the output.
2. Now remove the `mut` keyword from line 2 and try to compile again. What error do you get?
3. Put `mut` back. Now add a line after `Liftoff!` that changes `countdown` to `-1` and prints `"Status: {}"` with the new value.

Key takeaway

Use `mut` when you know a variable needs to change. If you declare `mut` but never actually mutate the variable, Clippy will warn you about it — try `cargo clippy` to verify!

3.3.3 Challenge 3: Shadowing — a new binding with the same name

Rust allows you to declare a new variable with the same name as a previous one. This is called **shadowing**. It is *not* the same as mutation — you are creating a completely new binding that happens to reuse the name.

Create a new project called `challenge-3`:

```
[~ $> cargo new challenge-3
[~ $> cd challenge-3
```

Put the following code in `challenge-3/src/main.rs`:

Listing 20: challenge-3/src/main.rs

```
fn main() {
    let x = 5;
    println!("x is: {}", x);

    let x = x + 10;
    println!("x is now: {}", x);

    let x = x * 2;
    println!("x is finally: {}", x);

    // Shadowing can even change the type!
    let x = "I am no longer a number!";
    println!("x says: {}", x);
}
```

Tasks:

1. Compile and run the program. Write down each value of `x`.

2. Notice that on line 12, `x` changes from an integer to a string. This is allowed with shadowing but would NOT be allowed with `mut`. Try it: replace lines 2–12 with:

```
let mut x = 5;
x = "I am no longer a number!";
```

   What error do you get?

3. What is the difference between shadowing and mutability? Write your answer in your own words.

Key takeaway

Shadowing creates a brand-new variable that just reuses the name. Because it is a new `let` binding, it can even change the type. Mutation with `mut` can only change the value, never the type.

3.3.4 Challenge 4: Ownership with Copy types — integers are easy

Rust's **ownership** model is one of its most important features. Every value in Rust has exactly one owner. When the owner goes out of scope, the value is dropped (freed).

However, some simple types like integers, floats, booleans, and characters implement the `Copy` trait. For these types, assigning to a new variable creates a **copy** of the value, and the original variable remains usable.

Create a new project called `challenge-4`:

```
[~ $> cargo new challenge-4
[~ $> cd challenge-4
```

Put the following code in `challenge-4/src/main.rs`:

Listing 21: challenge-4/src/main.rs

```
fn main() {
    let a = 42;
    let b = a;       // a is COPIED because i32 implements Copy

    println!("a = {}", a);   // ✓ a is still valid!
    println!("b = {}", b);   // ✓ b has its own copy

    let flag = true;
    let another_flag = flag; // bool also implements Copy

    println!("flag = {}, another_flag = {}", flag, another_flag);
}
```

Tasks:

1. Compile and run the program. Both `a` and `b` are valid after the assignment. Why?
2. Add a new variable `let c: f64 = 3.14;` and assign it to `let d = c;`. Print both `c` and `d`. Does it compile? Why?
3. What types in Rust implement `Copy`? (Hint: all the simple scalar types and tuples of Copy types.)

Key takeaway

Types that implement the `Copy` trait are duplicated on assignment. Both the original and the new variable are independent and fully usable. Integers, floats, booleans, and characters are all `Copy` types.

3.3.5 Challenge 5: Ownership with non-Copy types — the String move

Not all types are `Copy`. Types that manage heap memory — like `String` — use **move semantics** instead. When you assign a `String` to a new variable, ownership **moves** and the original variable becomes invalid.

Create a new project called `challenge-5`:

```
[~ $> cargo new challenge-5
[~ $> cd challenge-5
```

Put the following code in `challenge-5/src/main.rs`:

Listing 22: challenge-5/src/main.rs

```
fn main() {
    let greeting = String::from("Hello, Rustacean!");
    let moved_greeting = greeting;

    println!("moved_greeting: {}", moved_greeting);
    println!("greeting: {}", greeting);   // ← This line is the problem!
}
```

Tasks:

1. Try to compile this program. What error does the compiler give you?
2. The compiler says value `used after move`. In your own words, explain what happened to `greeting` on line 3.
3. Fix the program in **two different ways**:
   a. **Remove** the use of `greeting` after the move (simplest fix).
   b. **Use** `.clone()` to create a deep copy of the String, so both variables own their own data.

Key takeaway

For heap-allocated types like `String`, assignment **moves** ownership. The old variable becomes invalid. If you need both variables to remain usable, explicitly `clone()` the data.

3.3.6 Challenge 6: Clone — explicit deep copies

Let's practice using `.clone()` to make deliberate copies of data that lives on the heap.

Create a new project called `challenge-6`:

```
[~ $> cargo new challenge-6
[~ $> cd challenge-6
```

Put the following code in `challenge-6/src/main.rs`:

Listing 23: challenge-6/src/main.rs

```
fn main() {
    let city = String::from("Rotterdam");
    let city_backup = city.clone();

    println!("Original : {}", city);
    println!("Backup   : {}", city_backup);

    // Let's prove they are independent
    let mut city = city;          // shadow into a mutable binding
    city.push_str(" Centraal");

    println!("Modified : {}", city);
```

(continues on next page)

(continued from previous page)

```
13      println!("Backup   : {}", city_backup);  // unchanged!
14  }
```

Tasks:

1. Compile and run the program. Verify that modifying `city` does not affect `city_backup`.
2. On line 9, we shadow `city` into a `mut` binding. Why do we need `mut` here?
3. Change line 3 to `let city_backup = city;` (remove `.clone()`). What happens when you compile? Why?
4. Think about performance: why doesn't Rust clone automatically for heap types?

Key takeaway

`.clone()` is an explicit opt-in to deep copying. Rust forces you to be deliberate about copies of heap data so you are always aware of the performance cost. Compare this to languages where copies happen silently behind the scenes.

3.3.7 Challenge 7: Borrowing with references — reading without taking

Instead of transferring ownership, you can **borrow** a value by creating a reference with `&`. A reference lets you read (or write, with `&mut`) a value without taking ownership. The original owner keeps its data.

Create a new project called `challenge-7`:

```
[~ $> cargo new challenge-7
[~ $> cd challenge-7
```

Put the following code in `challenge-7/src/main.rs`:

Listing 24: challenge-7/src/main.rs

```
1   fn print_length(s: &String) {
2       println!("'{}' has {} characters", s, s.len());
3   }
4
5   fn main() {
6       let message = String::from("Rust is memory safe!");
7
8       print_length(&message);       // borrow message
9       print_length(&message);       // borrow again — still valid!
10
11      println!("I still own: {}", message);  // ✓ message was never moved
12  }
```

Tasks:

1. Compile and run the program. Notice that `message` is still usable after being passed to the function twice.
2. Change the function signature to `fn print_length(s: String)` (remove the `&` from both the parameter and the calls on lines 8-9). What happens? Why can't you call the function twice anymore?
3. Revert to using `&String`. Now, inside `print_length`, try to add `s.push_str("!!!");`. What error do you get? Why?

Key takeaway

An immutable reference `&T` lets you look at data without taking ownership. You can have as many immutable references as you want at the same time. But you cannot modify data through an immutable reference — for that you need `&mut`.

3.3.8 Challenge 8: Mutable borrowing — one writer at a time

Rust allows **mutable references** with `&mut`, but enforces a strict rule: you can have **either** one mutable reference **or** any number of immutable references, but never both at the same time. This prevents data races at compile time.

Create a new project called `challenge-8`:

```
[~ $> cargo new challenge-8
[~ $> cd challenge-8
```

Put the following code in `challenge-8/src/main.rs`:

Listing 25: challenge-8/src/main.rs

```
1  fn add_excitement(s: &mut String) {
2      s.push_str("!!!");
3  }
4
5  fn main() {
6      let mut slogan = String::from("Fearless concurrency");
7
8      println!("Before: {}", slogan);
9
10     add_excitement(&mut slogan);
11
12     println!("After : {}", slogan);
13 }
```

Tasks:

1. Compile and run the program. Observe how the function modifies the original `slogan` through a mutable reference.
2. The variable `slogan` must be declared `mut`. Remove `mut` from line 6 and try to compile. What error do you get?
3. Now try to break the borrowing rules. Add the following code **before** the `println!("After")` line:

   ```
   let r1 = &slogan;
   let r2 = &mut slogan;
   println!("{} {}", r1, r2);
   ```

   What error does the compiler give you? What rule is being enforced?
4. Fix the code from task 3 by ensuring immutable and mutable borrows don't overlap. (Hint: use `r1` before creating `r2`.)

Key takeaway

`&mut T` gives you a mutable reference — you can modify the borrowed data. But Rust's **borrowing rules** guarantee that at any point in time you have either **one** `&mut` or **many** `&`, never both. This is how Rust prevents data races without a garbage collector.

3.4 Summary of the Rust rules

Here is a quick recap of the rules you have explored in these challenges:

Table 1: Rust Variables & Ownership Cheat Sheet

Concept	Rule
Immutability	Variables are immutable by default. Use `let mut` to opt in to mutability.
Shadowing	`let x = ...;` can be used again to create a new binding with the same name, even with a different type.
Ownership	Every value has exactly one owner. When the owner goes out of scope, the value is dropped.
Copy types	Simple scalar types (`i32`, `f64`, `bool`, `char`, etc.) are copied on assignment. Both variables remain valid.
Move semantics	Heap types like `String` are **moved** on assignment. The original variable becomes invalid.
Clone	Use `.clone()` to explicitly deep-copy heap data when you need both variables to remain valid.
Immutable borrowing	`&T` borrows data without taking ownership. You can have many `&T` references at the same time.
Mutable borrowing	`&mut T` borrows data for modification. You can have only **one** `&mut T` at a time, and no `&T` references may coexist.

These rules are enforced at **compile time** by the Rust compiler. There is no runtime overhead and no garbage collector. This is how Rust achieves memory safety with zero-cost abstractions.

4 Rust Strings

4.1 Introduction

There are two datatypes for strings in Rust:

1. The string literal (&str) or slice.
2. The String as a vector of bytes, also called the `owned` string.

This lab will help you practive with the differences.

4.1.1 Requirements

A well-set-up Rust environment.

4.1.2 String literal or slice

Use string literals when you need to have a 'view' on a string. E.g. the string is not expected to expand.

So concatenation of two `&str` is not directly possible. You can do so however by using the `format!` macro which is closely related to `println!` and friends. `format!` will return a string slice containing the formatted string.

Exercise: Create a Rust program that will will have the following 2 string literals:

let s1 = "Hello "; let s2 = "Rustaceans";

Create a third string literal called s3 as s1 concatenaded with s2 using the format! macro.

4.1.3 String vector

A String is a different kind of beast. Strings need to be created like this:

let s1 = String::from(“Hello “) ; let s2 = String::from(“Rustaceans”) ;

Exercise: Create a Rust program that will make s1 the value of String s1 and String s2 concatenated.

5 Borrowing

5.1 Introduction

Understanding the concepts of `Ownership`, `Borrowing` and `Lifetimes` are essential for programming in Rust. Recall that Rust ensures memory safety by allowing code only to compile when it is able to ensure this safety. In this lab we will discover and practice with `Ownership` and `Borrowing`. We will discuss and practice `Lifetimes` later in this course.

5.1.1 Requirements

A well-set-up Rust environment and the memorization of the following Rust `Ownership` rules:

`1. Every variable owns its initialized value`

```
// Only owner owns the value "There can only be one"

let owner = String::from("There can only be one")
```

`2. Every value can only have one owner at a time`

```
// new_owner will now be the new owner

let new_owner = owner;

// owner is not 'valid' anymore
```

`3. Once a variable is out of declared scope, the owned value will be dropped and memory will be deallocated`

```
// a variable in a separate scope
{
```

(continues on next page)

(continued from previous page)

```
    let example = String::from("Here's a new scope");
}

// Will not compile as 'example' is dropped as soon as it leaved the innerscope
// and is not availble in the outerscope.

print!("{}", example)
```

In Rust, the compiler utilizes the `Borrowchecker` to analyze and verify whether no unsafe memory accesses would be made and if so, breaks off the compilation of the code.

5.1.2 Ownership

Create a new project called `ownership-1` with the following code:

Listing 1: borrowing-1/main.rs

```
1  // ownership_1
2
3  fn main() {
4
5      let a: i64 = 42 ;
6      let b = a;
7
```

(continues on next page)

(continued from previous page)

```
    println!("a == {0} and b == {1}",a,b) ;
}
```

Try to build/run it. Why does this compile, while the next example doesn't?

Listing 2: borrowing-3/main.rs

```
// ownership_3

fn main() {

    let a = String::from("one");

    {
      let b = a;
      println!("b == {0}",b);
    }

    println!("a == {0}",a);

    // println!("a == {0} and b == {1}",a,b) ;
}
```

The compiler reminds you in a friendly way that this does not compile:

The Rust compiler notifies us that the ownership of the value formerly owned by the variabele a has moved to the variable b. Ergo, a is not valid anymore, it does not own the value anymore and has no existence anymore.

And to make sure you truly understand the concept of ownership in Rust, why does the next example also compile without issues? They are both about strings, are they not?

Listing 3: borrowing-2/main.rs

```
// ownership_2

fn main() {

    let a = "one";
    let b = a;

    println!("a == {0} and b == {1}",a,b) ;
}
```

5.1.3 Borrowing

Borrowing is the concept of accesing the value of a variable without actually taking ownership of it. In an analogy we are lending the BMW of our friend, but we do not get to own it and we are expected to return it in exactly the same state as it is now.

Borrowing is referencing the variable and it's only needed for datatypes that do not fit on the stack and need to be allocated on the heap.

Let's get the example with the 'Strings' compiling. We need to have variable b to reference variable a. We do this with the & operator like this:

Listing 4: borrowing-3/main.rs

```
// ownership_3a

fn main() {

    let a = String::from("one");
```

(continues on next page)

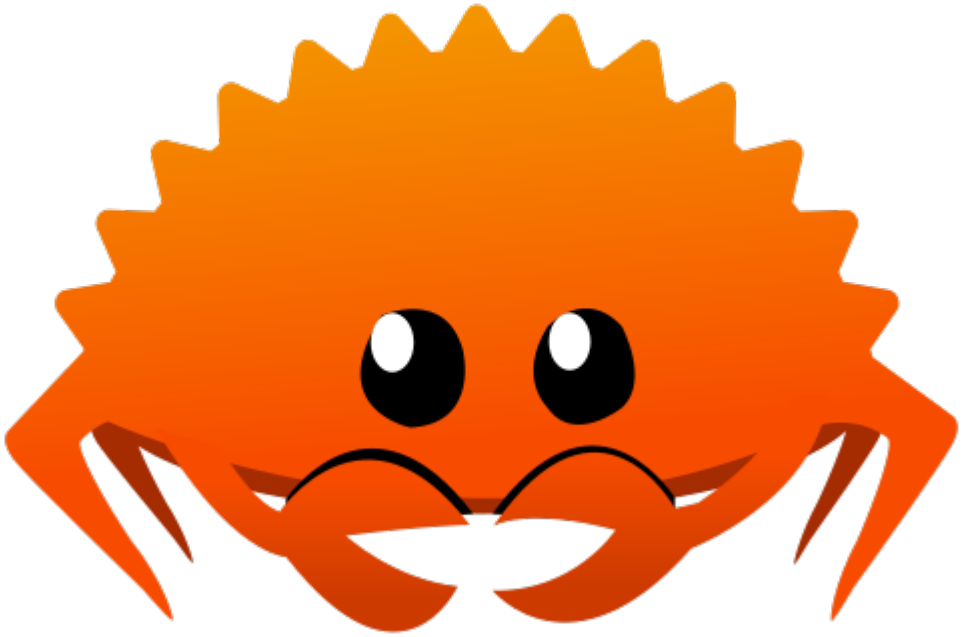
(continued from previous page)

```
    // let b = a.clone() ;
    let b = &a ;

    println!("a == {0} and b == {1}",a,*b) ;
}
```

Now compile/build this program.

It should pose no problems.

6 Arrays in Rust



6.1 Introduction

This lab will help you practice with arrays and tuples in Rust

6.1.1 Requirements

A well setup Rust environment

6.1.2 Exercise 1

Write a Rust program that uses an array of tuples that relate the following information:

1. Month name
2. Nr of days in the month

E.g. you should get an array of 12 elements which each element being a tuple like ("January",31)

In your program print per month it's name and the nr of days in it. If a boolean variable `is_leapyear` you should add 1 to the days of the month of February.

6.1.3 Exercise 2

The following code defines a multi dimension array representing a Tic-Tac-Toe board. The board is initialized to all zeroes. (Empty)

Listing 1: arrays-3/main.rs

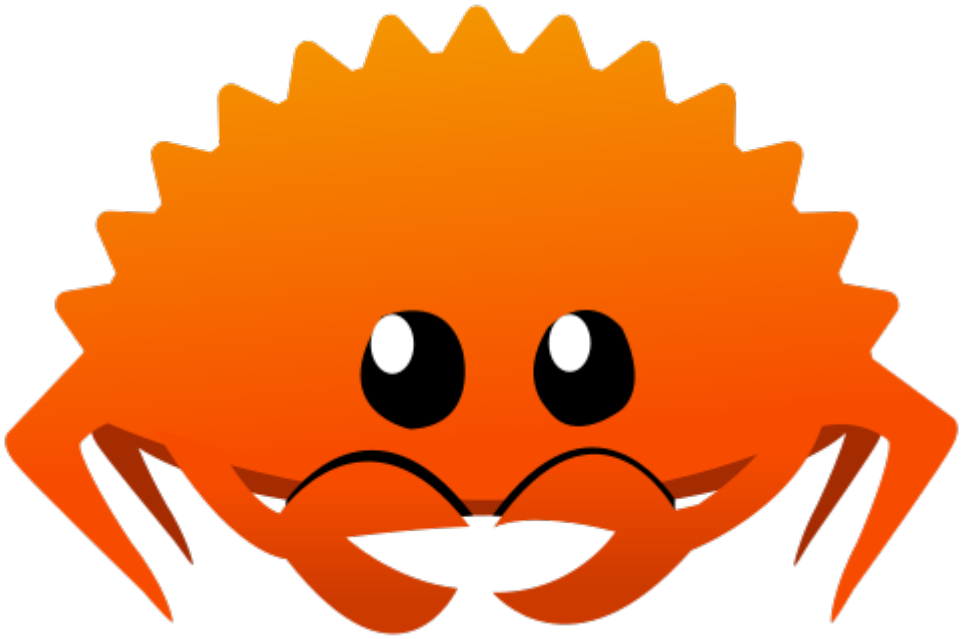
```
fn main() {
    let tictactoe = [[0;3];3];

    // print_board(tictactoe);
}
```

Task: write code that will display the 'value' of each position on the board. The printed layout should represent the appearance of the board.

E.g:

0 0 0
0 0 0
0 0 0

7 Conditionals



7.1 Introduction

This lab will help you practice with conditionals in Rust

7.1.1 Requirements

A well setup Rust environment.

7.1.2 Exercise 1

Listing 1: borrowing-3/main.rs

```
use std::io;

fn main() {
    println!("Enter a number: ");
    let mut guess = String::new();

    io::stdin().read_line(&mut guess).expect("failed to readline");

    let my_nr: i32 = guess.trim().parse().unwrap();

    print!("You entered {}", my_nr);
}
```

Above program will request input from the user. In later modules we will explain how it works. This input is converted to a number.

Kindly extend the program to:

1. Print that the nr is zero if a 0 is keyed in.
2. Print if the nr is odd or even

For this excercise use the if conditional statements

7.1.3 Exercise 2

The same tasks as for excercise 1, but now implement it using `match`. Please do not forget the default `arm`.

7.1.4 Exercise 3

Implement a basic calculator for the four primary arithmetic operations: addition, subtraction, multiplication, and division. The user provides an operation in the form of a string ("add", "subtract", "multiply", or "divide") and two numbers. Your program should then perform the requested operation on the numbers.

Steps:

1. Define an enum named Operation with variants for each of the 4 operations.
2. Implement a function that converts a string into the Operation enum.
3. Implement the calculator using a match expression to choose the appropriate arithmetic operation based on the enum variant.
4. Test your calculator in the main function with a few examples.

E.g. to help you with some scaffolding:

Listing 2: calc/main.rs

```
enum Operation {
    Add,
    Subtract,
    Multiply,
    Divide,
```

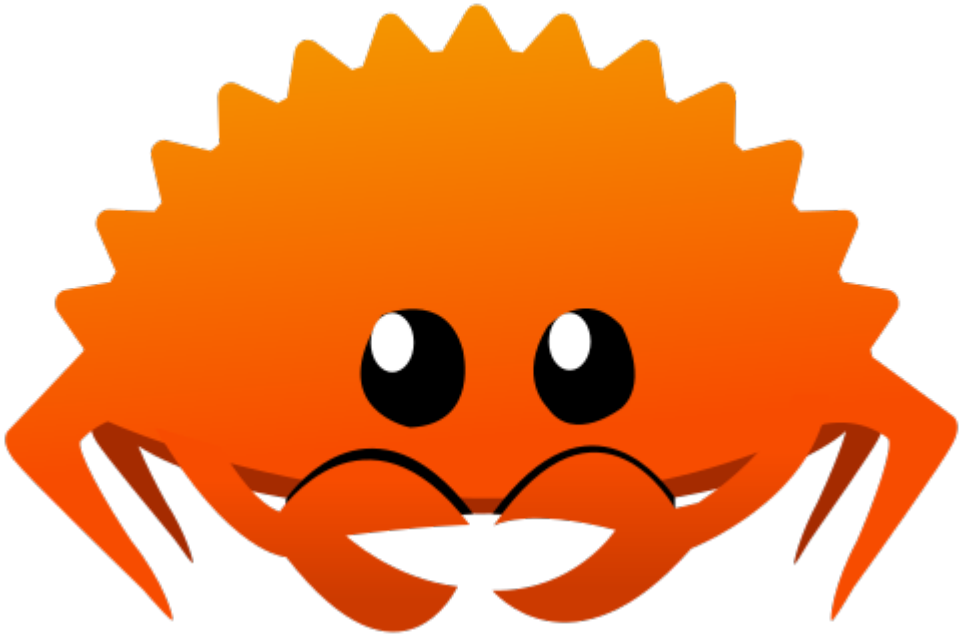
(continues on next page)

(continued from previous page)

```
}

fn str_to_operation(s: &str) -> Option<Operation> {}

fn calculate(op: Operation, a: f64, b: f64) -> f64 {}

fn main() {
    let operation_str = "add";
    let x = 5.0;
    let y = 3.0;

    match str_to_operation(operation_str) {
        Some(op) => {
            let result = calculate(op, x, y);
            println!("Result of {}ing {} and {}: {}", operation_str, x, y, result);
        }
        None => println!("Unknown operation: {}", operation_str),
    }
}
```

8

Loops in Rust



8.1 Introduction

This lab will help you familiarize yourself with loops in Rust.

8.1.1 Requirements

A well setup Rust environment

8.1.2 Exercise 1

Create a Rust program called loop-1 that will count from 0 to 100 and that will display all numbers that are divisable by 3 or 5.

8.1.3 Exercise 2

Create a Rust program called alphabet that will loop[through all 26 letters of the alphabet and print out the vowels.

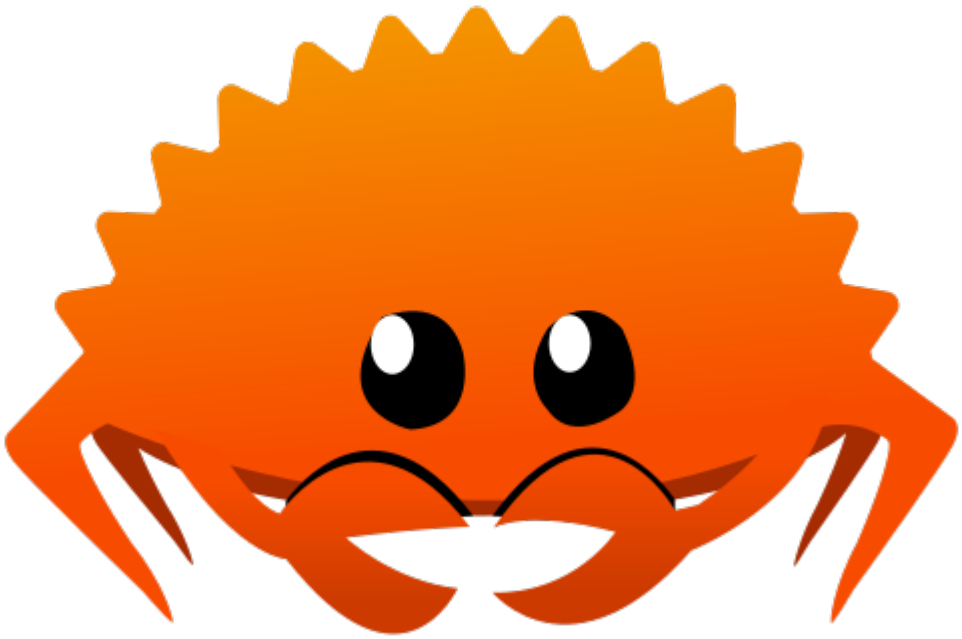
8.1.4 Excercise 3

Given the unicode string: "□ Dia dhuit, an Domhan! □"; (With some artistic freedom loosely translated as Hello World in Irish)

Kindly write a program in Rust to iterate over this string and print the individual characters in this string and insert extra spaces. E.g, the result should be:

□ D i a d h u i t , a n D o m h a n ! □

9 Enums in Rust



9.1 Introduction

The data type `enum` plays a special role in Rust. They are more powerful then the `enums` we know from other languages like C and C++. They play an important part in error handling and getting optional results from functions.

This lab will have you practice with `enums` in Rust.

9.1.1 Requirements

A well setup Rust environment, at least Rust 2021 edition

9.1.2 Exercise 1

Objective: Understand the basic usage of enums in Rust by implementing days of the week.

1. Define an enum named Day that represents the days of the week. Each day should be a variant in the enum.
2. Create a variable for each day of the week and assign the corresponding variant of the Day enum to it.
3. Using a match expression, print a small piece of information or activity you might do on each day. For instance, "Monday is Rust practice day"

9.1.3 Exercise 2

Objective: Learn to use enums in Rust that hold data by modeling a simple coffee ordering system.

Scenario: Customers can order coffee in different sizes and specify if they want milk or not. The size can be "Small", "Medium", or "Large", and milk can be "None", "Regular", or "Soy".

Steps:

1. Define the CoffeeSize and MilkOption enums:
2. Create a sample order like this:

let my_order = CoffeeOrder::Order {
size: CoffeeSize::Medium, milk: MilkOption::Soy,

}

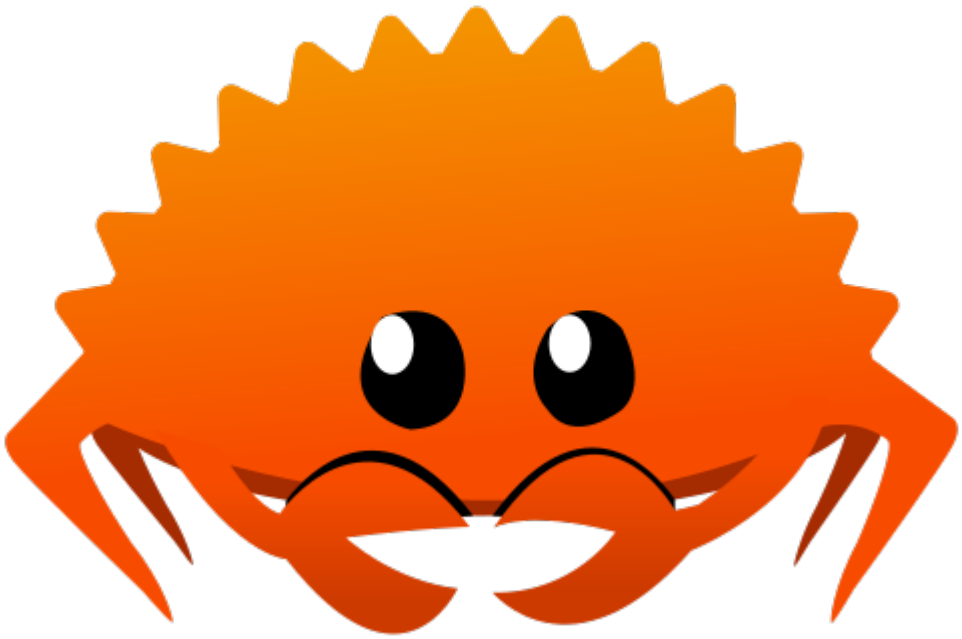
1. In the main function describe this order using a match expression.

E.g. possible output with above order:

"You've ordered a Medium coffee with soy milk."

10

Structs in Rust



10.1 Introduction

This lab will help you practice with `structs` in Rust. The data type `struct` plays an important role in Rust by create custom data types grouping together related pieces of data. Not only for structuring data but like `enums` we will see later in the course that we can implement methods on them, forming the base of OOP in Rust.

10.1.1 Requirements

A well setup Rust environment, at least Rust 2021 edition.

10.1.2 Exercise 1

1. Define a Persons struct with fields first_name, last_name and age. Choose appropriate data types for the field
2. Instantiate the struct in main (let person = Person{...};)
3. Print the contents of these structs
4. Print adult if the person has reached the adult age

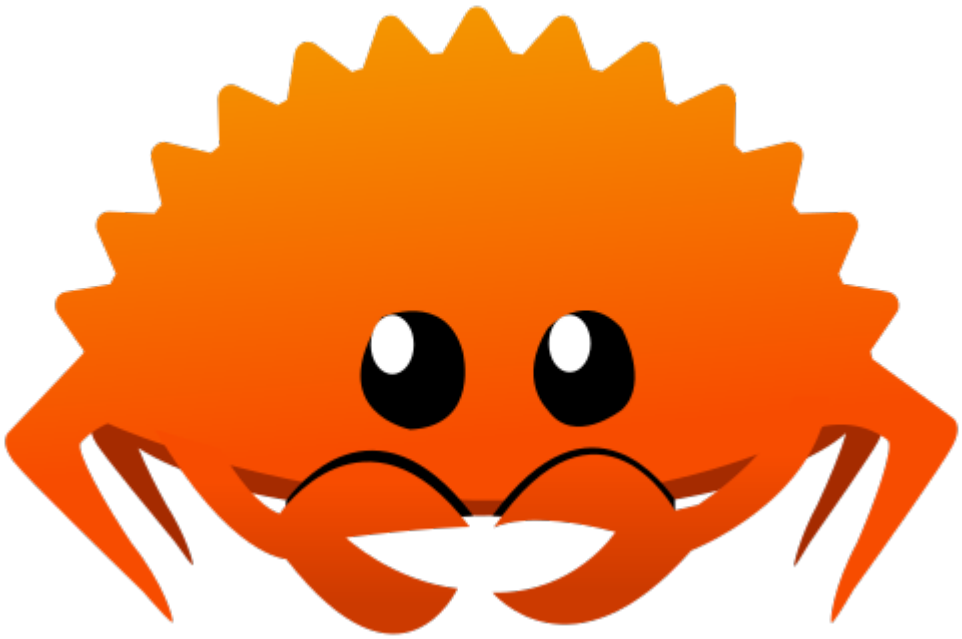
10.1.3 Exercise 2

In this exercise we are going to practice with embedded structs.

1. Define an Employee struct
   An Employee should have:
   - A first name
   - A last name
   - A job title
2. Define a Department struct
   A Department should have:
   - A department name
   - A list of the 5 (fixed) employees in that department
3. Instantiate the Department struct and 3 Employee structs.
4. Print the department info (all fields) and the Employees in that department

11

Vectors in Rust



11.1 Introduction

This lab will help you practice with defining vectors, iterating over them and updating the elements in them

11.1.1 Requirements

A well setup Rust environment

11.1.2 Exercise 1

Objective: practice with a vector of strings

1. Create a vector containing the words “hello”, “world”, and “rust”.
2. Convert the vector of words into a single string where words are separated by a space.
3. Print this string

11.1.3 Exercise 2

Write a Rust program that declares and initializes a vector of i64. Populate this vector with the nrs from 0 to 100.

After that:

Iterate over the elements in the vector and double the element’s value if the value is odd and square the elements value when the original value is even. Lastly, print out each value in the vector including their element nr;

E.g:

> Element 0 -> 0 Element 1 -> 2
> etc.

11.1.4 Exercise 3

Like the Gophers the Rustlings would like to run their own bookstore, but then full of books about Rust. They need to manage the inventory of books. Each book will have a title, author, and stock count.

Objective:

Create a basic inventory management system to add books, check stock, and sell books.

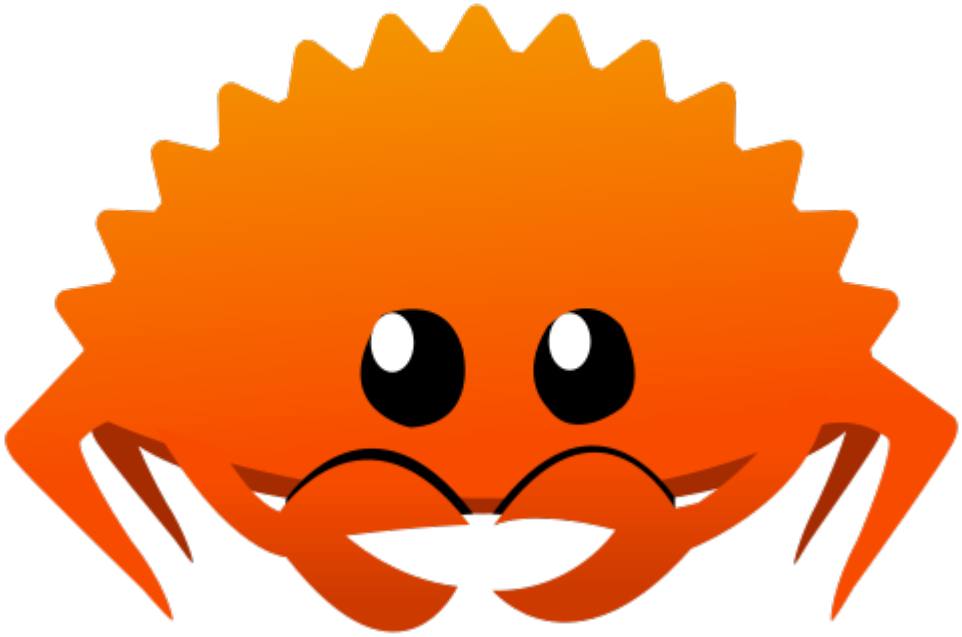
1. Structures:

   Book: Represents a book with a title, author, and stock count. Inventory: Represents the bookstore’s inventory. It uses a vector to hold the collection of books.
2. Actions/functions:

   > Add a book to the inventory. Check the stock of a specific book by title. List the entire inventory Sell a book (decrease its stock by 1).

12

Hashmaps in Rust



12.1 Introduction

This lab will help you practice with hashmaps

12.1.1 Requirements

A well setup Rust environment

12.1.2 Exercise 1

Write a Rust program that declars and initializes a hashmap `devstudents`. The hashmap should map student names to programming languages. Key of the hashmap is the name of the student, value is the name of the programming languages.

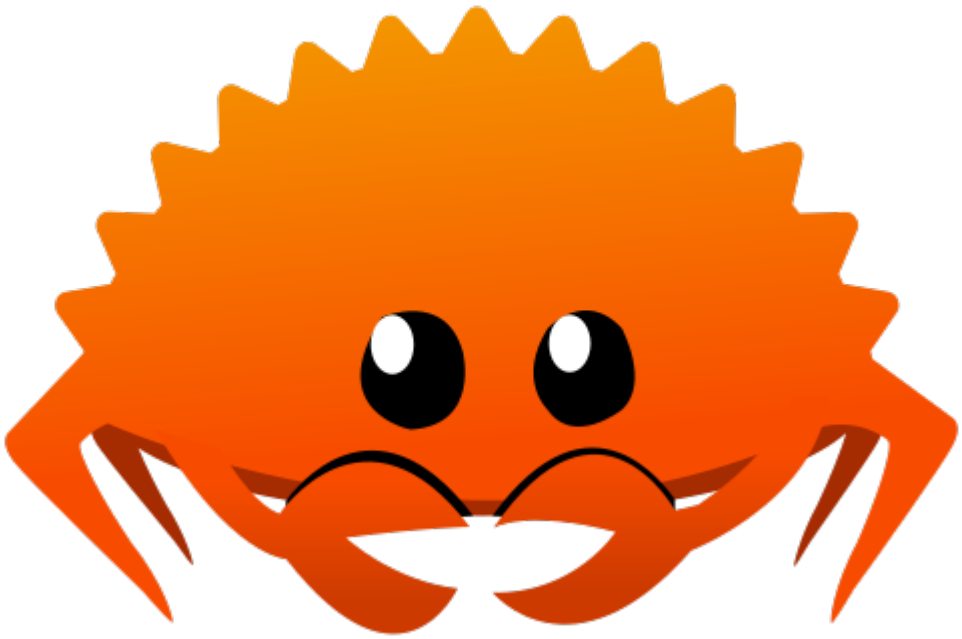
1. Populate the hashmap with the following records:

Fred,Golang Alice,Perl Kjell,Python Jarmo,Rust Sander,Zig

2. Print out all the elements in the hashmap.
3. Add a new student and language (yours to choose).
   Check before you add the student if he's not already in the hashmap. If so, print a message.
4. Sander decides to join the Rust developers. Kindly change his programming language to Rust.
5. Extra credit:
   Iterate over the HashMap and convert all programming languages to uppercase.

13

Functions in Rust



13.1 Introduction

In this lab we will practice with writing functions in Rust.

13.1.1 Requirements

A well setup Rust environment

13.1.2 Exercise 1

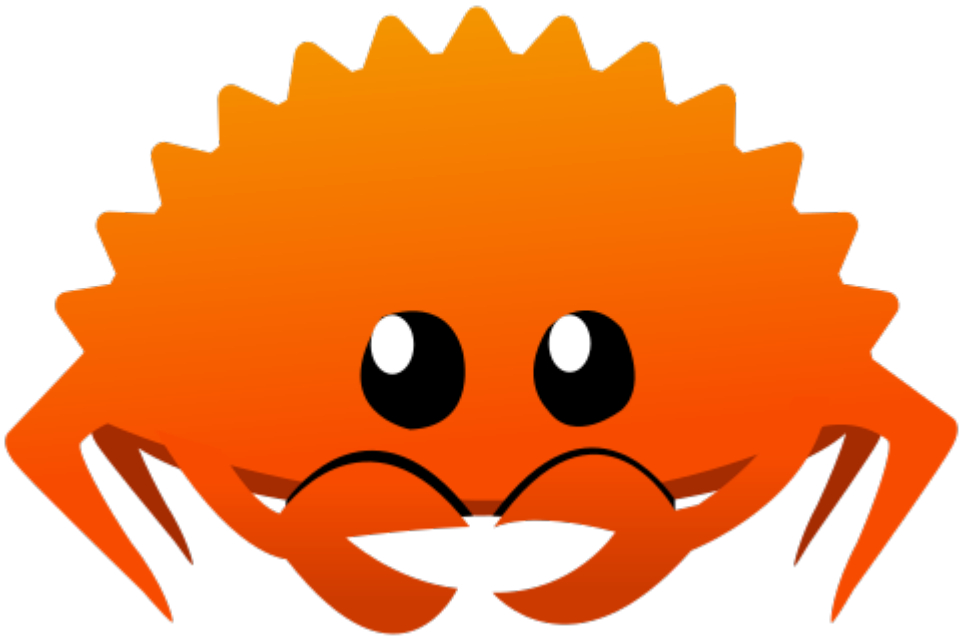
1. Write a function that returns the square root of a f64 and returns a f64. Call this function in the main body.
2. Adapt the functions so it will refuse to calculate the square root if the argument < 0 Use the Result enum to return an error condition or result.
   After calling the function, determine wether you have a valid result or an error and print this on the screen.

13.1.3 Exercise 2

1. Write a recursive funtion that will calculate the n-th fibonacci nr
   Fib(1) = 1 Fib(2) = 1
   Fib (n) = Fib(n-1) + Fib(n-2)
   Call the recursive function in the main() part.

14

Methods in Rust



14.1 Introduction

This lab focusses on using methods on structs in Rust.

14.1.1 Requirements

A well setup Rust environment.

14.1.2 Exercise 1

Define 3 structs like this:

1. Square -> having side: f64 as it's only atttribute
2. Triangle -> having side1, side2, side3 all of type f64 as attributes
3. Circle -> having radius as it's only attribute

Write for each of these 3 structs a static method called New so we can instantiate new Squares, Triangles and Circles

14.1.3 Exercise 2

Write the following 2 methods for each of the 3 structs:

1. Perimeter
   This returns the perimeter for each of the shapes (square -> 4*side, triangle -> sum of all sides, circle 2*r*PI)
2. Area
   This returns the area of each of the shapes

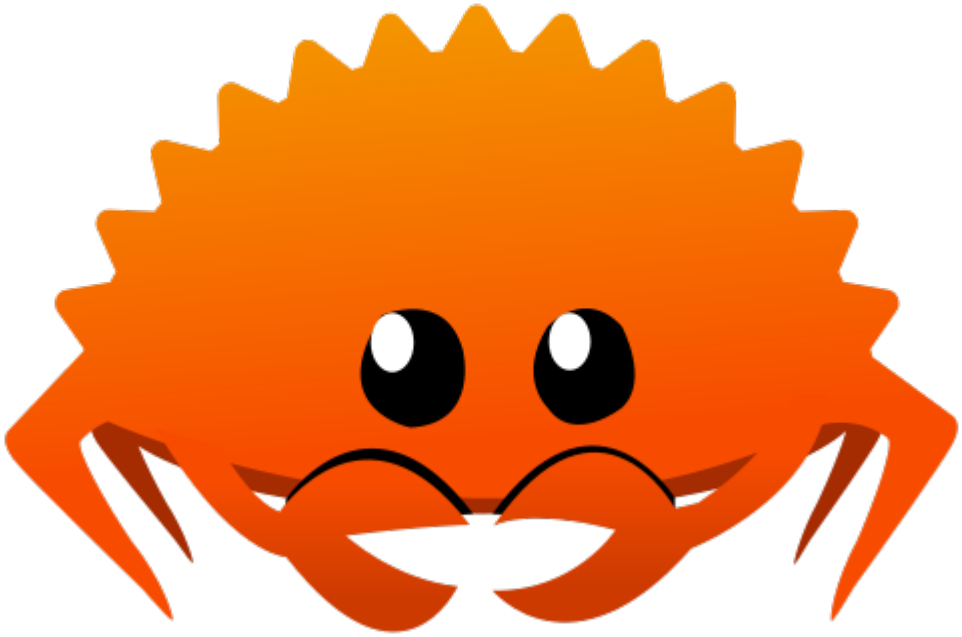
Test all the methods for the each of the shapes.

14.1.4 Exercise 3

1. Write for each of the shapes a method called `scale` that will alter the dimension of the shape. E.g. scale=2.0 will double the sides of a square, etc.
2. Display the properties of the shapes to see if indeed they are 'scaled'

15

Traits



15.1 Introduction

This labs build upon the methods we programmed in the struct methods lab. In this part we will define a trait called Shape that will define the methods perimeter and area. This means all structs/object that will implement these methods will fit the Shape trait and we can make Generic functions for area and perimeter that will accept struct/objects that will implement the Shape trait.

15.1.1 Requirements

A well installed Rust setup.

15.1.2 Exercise 1

1. Define a trait called Shape that requires the methods area() and perimeter() to be implemented.
2. Define a generic function area that takes a Shape and returns its area.
3. Define a generic function perimeter that takes a Shape and returns its perimeter.
4. Test the functions in your main() by calling it and passing it Circle, Square and Rectangle objects.
5. Add the methods scale() to the trait and define a generic function scale that takes a Shape and will scale that Shape.
6. Test the generic scale() function by passing it Circle, Square and Rectangle objects.

15.1.3 Exercise 2

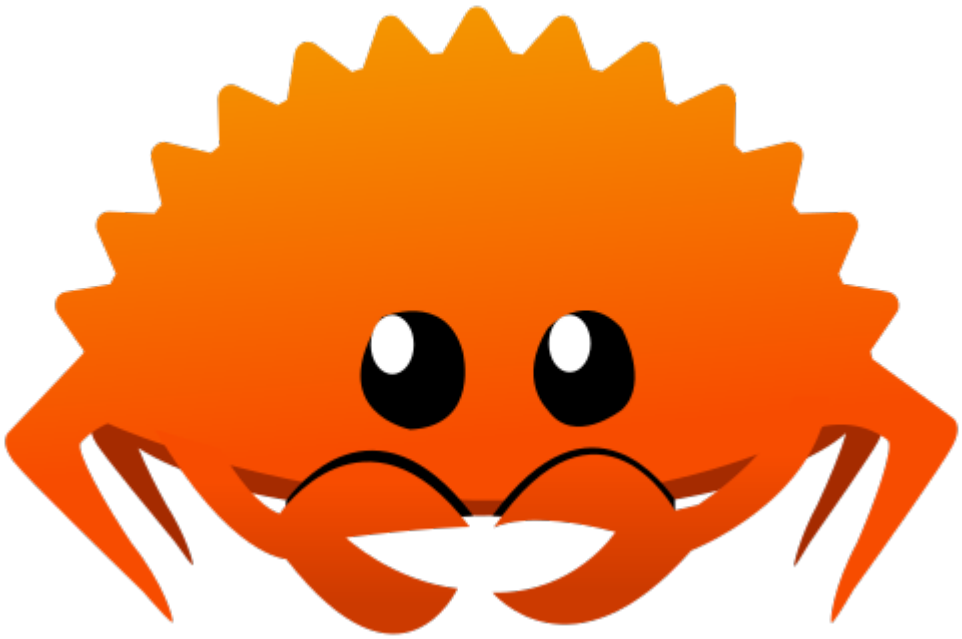
1. Design and implement a user type that represents temperature in degrees Celsius
2. Design and implement a user type that represents temperature in degrees Fahrenheit
3. Implement on both types the Display trait (method fmt) that will pretty print the degrees with the correct unit.
4. Design and implement a method that will convert from degrees Celsius to Fahrenheit and use the right return type
5. Design and implement a method that will convert from degrees Fahrenheit to Celsius and use the right return type
6. Test your program using the conversion method of C to F and vice versa. The right value and the right units should be printed.

15.1.4 Exercise 3

1. Design and implement a struct called Student that will have firstname, lastname, timestamp and a vector with all the languages this person masters.
2. Design and implement (not derive) the clone trait for this struct and add as extra functionality an update of the timestamp when the cloning is taking place.
3. Test the cloning of the struct

16

OOP in Rust



16.1 Introduction

In this lab we will practice with some of the OOP techniques in Rust

16.1.1 Requirements

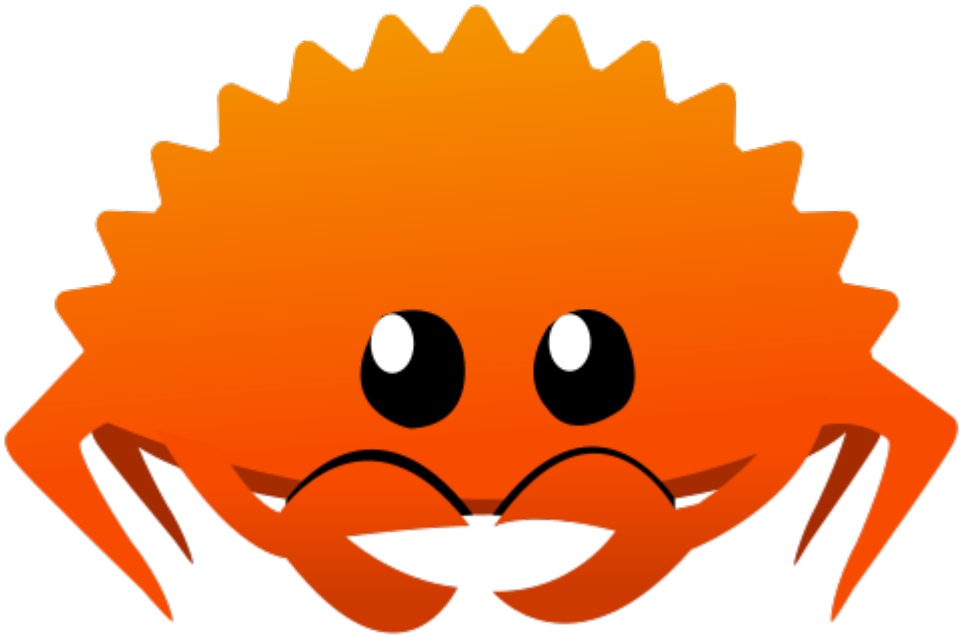
A well setup Rust environment

16.1.2 Exercise 1

1. Design and implement a datatype in Rust to hold a mathematical vector, e.g.
   3.0|
   1.5|
   |-2.0|
   The vector is one dimensional and has 3 nrs.
2. Implement the Display trait on this datatype to pretty print it. E.g. something like shown above.
3. Implement a method so that we can calculate the distance between 2 vectors using the Euclidean distance formula:
   vector A(1,1,2) vector B(2,1,2)
   Then the distance is sqrt($(x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2$)
4. Overload the * operator so that two vectors can be multiplied with each other (use the dot product to keep it simple)
5. Overload the * operator so you can multiply the vector with a scalar
6. Extra credit for adding overloading of '*'for the Cross product of 2 vectors

17

Lifetimes in Rust



17.1 Introduction

In this lab we are going to practice with lifetimes in Rust

17.1.1 Requirements

1. A fresh mind
2. Spirit
3. Coffee at arms length

17.2 Exercises

17.2.1 Exercise 1

Given the following `struct`, which represents an author and a citation:

Listing 1: lifetime-struct/main.rs

```
struct AuthorAndExcerpt {
    author: &str,
    excerpt: &str,
}

fn main() {
    let name = "Hemingway";
    let text = "The world breaks everyone, and afterward, some are strong at the broken
↪places.";
    let record = AuthorAndExcerpt::new_author_and_excerpt(name, text);
}
```

1. Modify the `AuthorAndExcerpt` struct to use appropriate lifetime annotations so that it compiles correctly.
2. Implement a function new_author_and_excerpt that accepts two string slices: one for the author's name and one for an excerpt of their writing. The function should return an instance of `AuthorAndExcerpt`.

17.2.2 Exercise 2

Kindly make the following Rust program compile and work by bringing on proper lifetime annotations where applicable:

Listing 2: lifetime-fstruct/main.rs

```
struct Person {
    name: &str,
}

fn make_person(name: &str) -> Person {
    Person { name }
}

fn main() {
    let name = "Alice";
    let alice = make_person(name);
    println!("Name: {}", alice.name);
}
```

17.2.3 Exercise 3

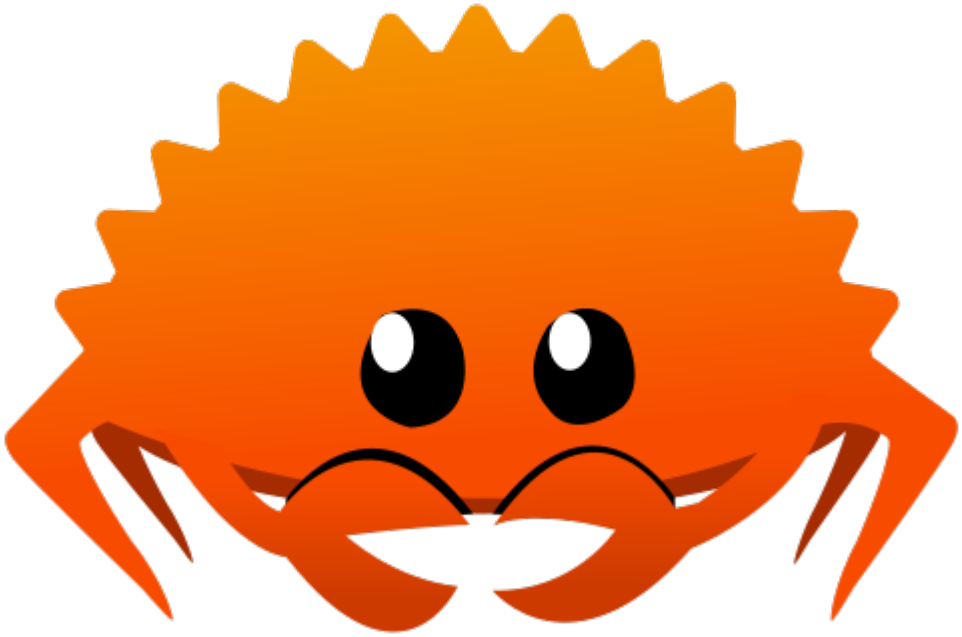
Implement a function that accepts two string slices with potentially different lifetimes and returns the slice that starts with the character 'a' or the first slice if neither starts with 'a'. However, because they can have different lifetimes, you need to handle lifetimes carefully.

Listing 3: lifetime-func/main.rs

```
fn starts_with_a<'a, 'b>(s1: &'a str, s2: &'b str) -> &'a str {}

fn main() {
    let string1 = "example";
    let string2 = "sample";
    let result = starts_with_a(string1, string2);
}
```

18

Errorhandling in Rust



18.1 Introduction

In this lab we are going to practice error handling in Rust

18.1.1 Requirements

1. A fresh mind
2. Spirit
3. Coffee at arms length

18.2 Excercices

18.2.1 Exercise 1

The first excercise concerns the quadratic equation solver again. Right now it doesn't have proper errorhandling and just displays a message when it detects that there will be no `real` solutions. But it still returns a tuple of two zeroes. You are going to transform this function so that is uses the `Result` type as output paramater/return type.

Step 1:

Please clone the rust-qeq lab again.

```
[~ $> git clone https://thegitcave.org/pascal/rust-qeq.git
```

Step 2:

Change the program in such a way that it returns a Result enum.

> In Ok() cases it should return the tuple of solutions. In Err() cases it should return the String "This equation does not have any real solutions"
>
> Test this using a main function that calls this function. Also have the calling function (main) evaluate the result act upon it. If it receives an error, it should mention that and ignore the results. If there's no error, the solutions are valid and should be printed.

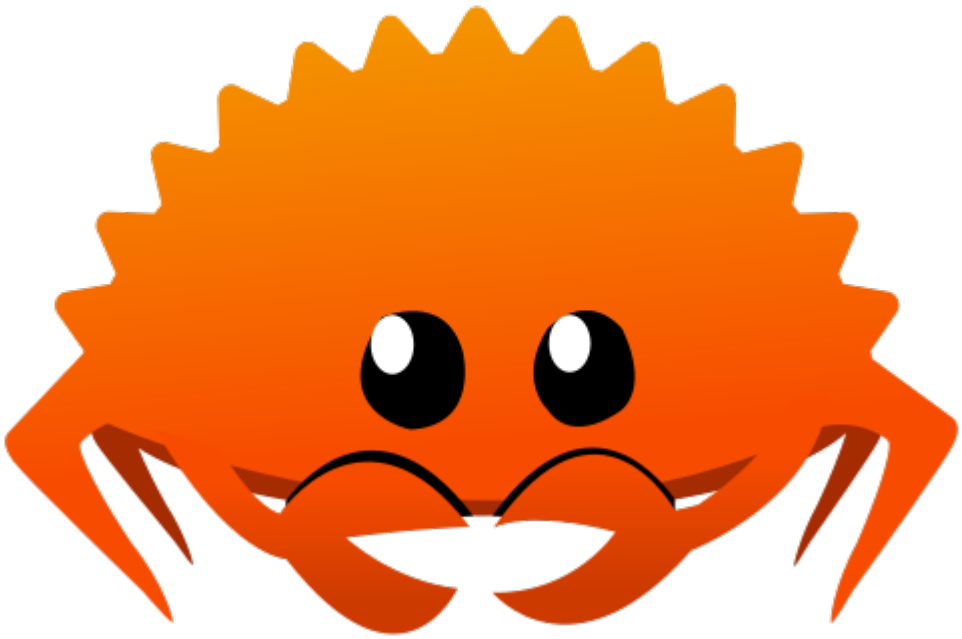
18.2.2 Exercise 2

Create a program that will open a file called `rust.txt` and read it in a string. Opening the file and reading the string will be implemented in 2 different functions. If the file cannot be openend propagate the error to the calling function/main If the string does not contain "Ferris" return also a different Error.

Have the calling function 'handle' the error.

19

Closures in Rust



19.1 Introduction

In this lab we will practice writing and using closures in Rust

19.1.1 Requirements

A well setup Rust environment

19.1.2 Exercise 1

The following struct is given:

Listing 1: city/src/main.rs

```
struct City {
    name: String,
    province: String,
    population: u64
}

fn main() {
    let c = City{ name: "Bergen".to_string(), province: "Limburg".to_string(), population: 5105 };
}
```

It contains fields name, population and province. Write a program in which you create a list of cities using a vector and create a closure to sort this vector in ascending order on province and print the results.

You can use the following data to populate your list:

City Province Population
Bergen Limburg 5105
Nijmegen Gelderland 81002
Maastricht Limburg 63201
Amsterdam Noordholland 982102
Rotterdam Zuidholland 1030101

Next, sort the list on population in descending order and print the results.

19.1.3 Exercise 2

Currying is when a function with many arguments is being made more 'accessible' by having a helper function that has fewer arguments by implementing default. Goal is to write such a 'curried' function in Rust:

Listing 2: currying/src/main.rs

```
fn add(a: u32, b: u32) -> u32 {
    a + b
}


fn main() {
    let add5 = |x| add(5, x);
    println!("{}",add5(5));
}
```

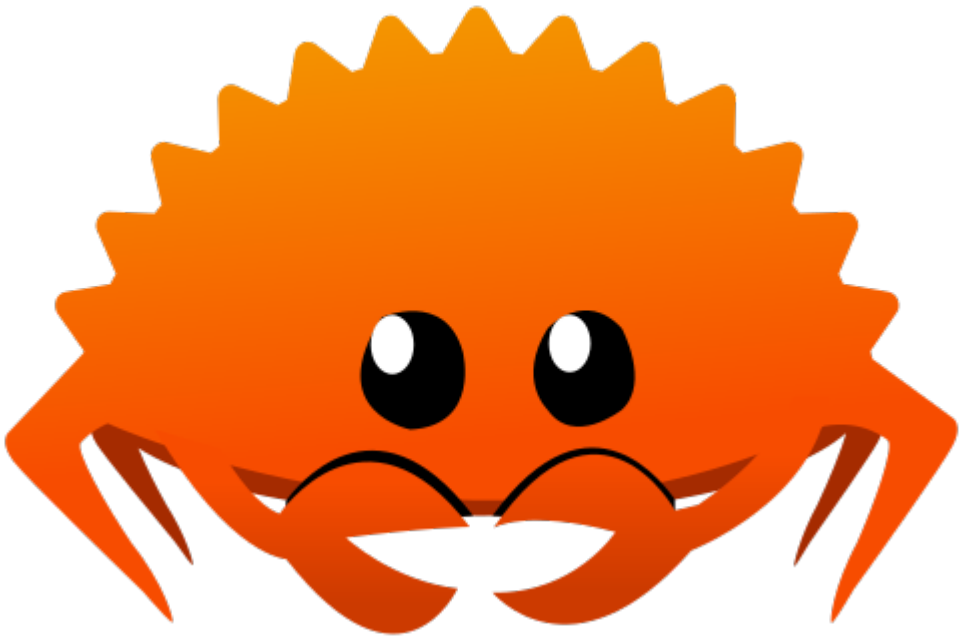
Above example shows you how to curry a addition function.

Now write a function greeting() that will have two parameters:
salutation -> that will contain for example "Gutentag", "Buenas Dias" etc name -> That will contain the name of the one to be greeted

The salutation should be a default “Hasta la vista” in the closure calling the function. E.g. your closure called `quickhi` should only have name as argument and use ‘Hasta la vista’ as the default greeting part supplied to your greeting() function.

20

Iterators in Rust



20.1 Introduction

In this lab we will practice writing and using closures in Rust

20.1.1 Requirements

A well setup Rust environment

20.1.2 Exercise 1

1. Create a vector with the following nrs: [-4,2,3,1,-7,21,42,31,-7]
2. Print all the nrs using an iterator in the vector
3. Using a adapter iterator and a closure, double the even nrs and half the odd nrs.
4. Using the filter adapter, filter out the nrs that are divisable by 5

20.1.3 Exercise 2

We are now going to create an interator ourselves.

The example show an iterator that generates a prime on each call to next():

Listing 1: iter-prime/main.rs

```rust
pub fn is_prime(n: u64) -> bool {
    if n < 4 {
        n > 1
    } else if n % 2 == 0 || n % 3 == 0 {
        false
    } else {
        let max_p = (n as f64).sqrt().ceil() as u64;
        match (5..=max_p)
            .step_by(6)
            .find(|p| n % p == 0 || n % (p + 2) == 0)
        {
            Some(_) => false,
            None => true,
        }
    }
}

pub struct Prime {
    curr: u64,
    next: u64,
}

impl Prime {
    pub fn new() -> Prime {
        Prime { curr: 2, next: 3 }
    }
}

impl Iterator for Prime {
    type Item = u64;

    fn next(&mut self) -> Option<Self::Item> {
        let prime = self.curr;
        self.curr = self.next;
        loop {
            self.next += match self.next % 6 {
                1 => 4,
                _ => 2,
            };
            if is_prime(self.next) {
```

(continues on next page)

(continued from previous page)

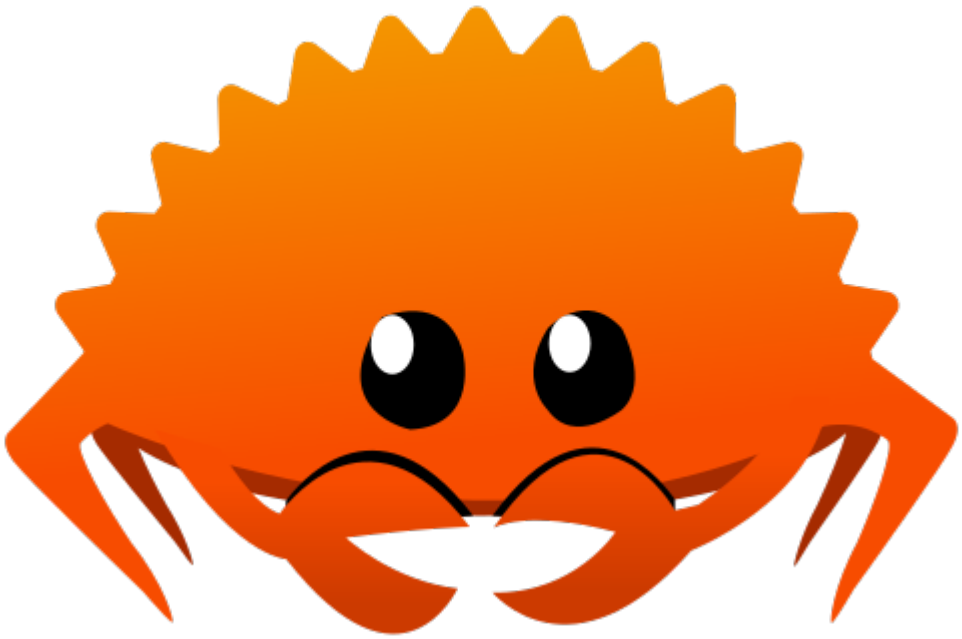
```
                break;
            }
        }
        Some(prime)
    }
}

fn main() {
    let mut p = Prime::new();

    for i in 1..=10 {
        let _ = if let Some(v) = p.next() {
            println!("Prime {} = {}", i, v);
        };
    }
}
```

1. Create an iterator that generates Faculty nrs.
2. Create an iterator that generates Fibonacci nrs.

Test your iterators.

21

Generics in Rust



21.1 Introduction

In this lab we will practice writing Generic Functions in Rust

21.1.1 Requirements

A well setup Rust environment

21.1.2 Exercise 1

1. Create a Generic function `sum()` that will calculate the sum of all types of integers and floats in a vector.
2. Use proper Trait bounds to get your generic function working.

Listing 1: generics-1/main.rs

```
fn main() {
    let v1 = vec![ 1,2,3,4,5,6,7,8,9 ];
    let v2 = vec![ 1.2,28.1,1.2,9.1 ];

    println!("{}",sum(&v1));
    println!("{}",sum(&v2));
}
```

1. Tip; assinging the 0 value to start with can be tedious if you don't now the type. You can use something like this:

Listing 2: generics-1a/main.rs

```
let mut sum: T = Default::default();
```

It provides a suitable default value for the type of var involved.

Test the Generic function.

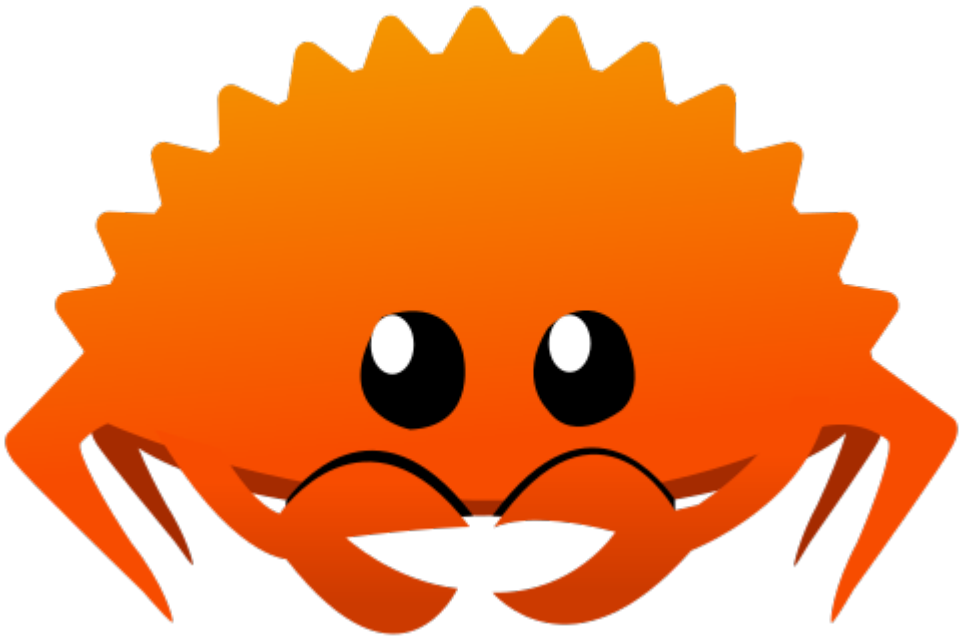
Bonus: try to write a Generic function that will calculate the average of a vector of floats or integers. (non-trivial)

21.1.3 Exercise 2

For a real challenge create generic functions that will calculate sum, avg, variance and standard deviation of a vector of generic types. E.g. the function should work for all integers and all floats.

22

Dynamic Dispatch in Rust



22.1 Introduction

In this lab we will practice with Dynamic Dispatch / Trait Objects in Rust

22.1.1 Requirements

A well setup Rust environment

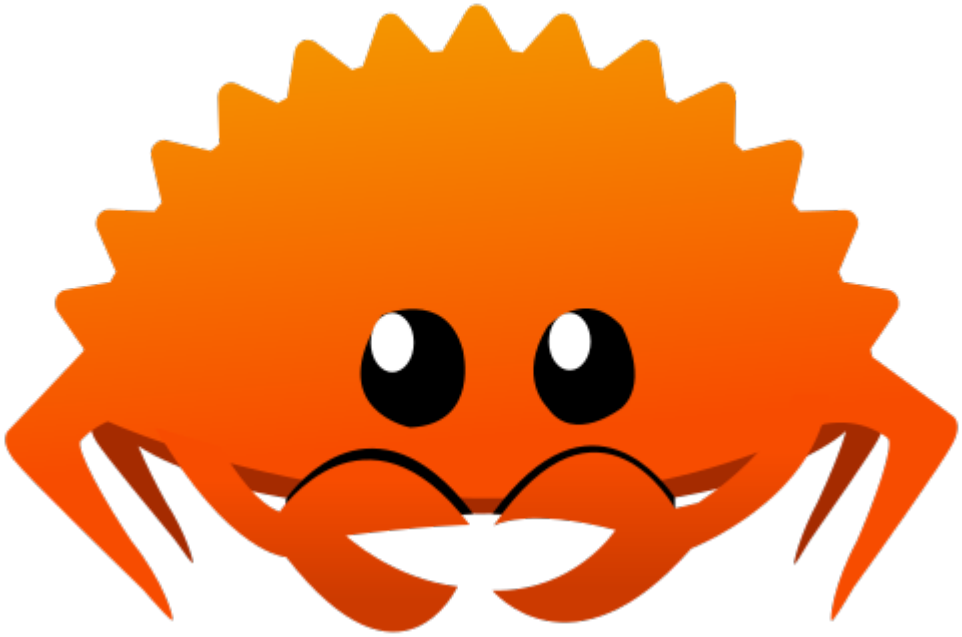
22.1.2 Exercise 1

We are going to model a Zoo of Program Language mascots. In this zoo, there are different types of animals, and each animal has its own saying. Your goal is to model this scenario:

1. Define a trait named Animal with a method named make_noise
2. Implement this trait for the animal types: Gopher (Clearness over Complexity), Crab (The safest program...) and Python (Ssssss...)
3. Create a function named animal_saying that takes in a trait object (dynamically dispatched) of the Animal trait and calls the sayit method.
4. In the main function, collect different animals in a vector and iterate through each, making them produce their respective sayings using the animal_sound function.

23

Unit testing in Rust



23.1 Introduction

In this lab we are going to practice with the testing functionality of the Rust toolchain.

23.1.1 Requirements

1. A fresh mind
2. Spirit
3. Coffee at arms length

23.1.2 What are we going to do

For this we are going to create a function that will solve quadratic equations.

The function will have a signature of:

```
fn solveqeq (a: f64, b: f64, c: f64) -> (f64, f64)
```

We will first focus on the easy way and only test with arguments a, b and c that will have solutions

You can clone the crate with git from:

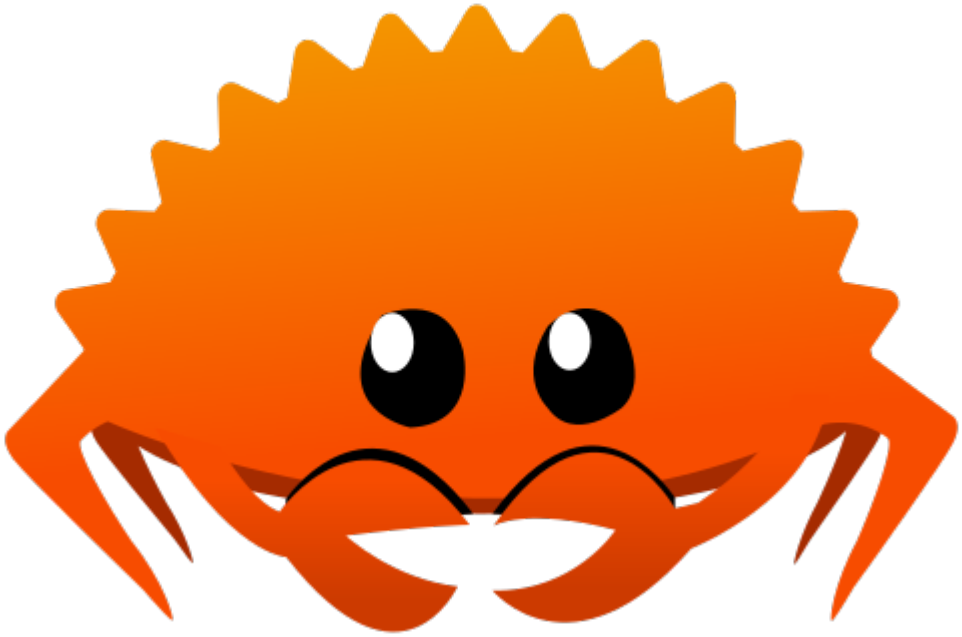
git clone https://thegitcave.org/pascal/rust-qeq.git

1. Task: Create a test case that tests the correct outcome for a certain aX^2+bx+c quadratic equation
2. Task: Check what happens if you test an equation that does not have real solutions
3. Task: Implement table driven testen that will check for at least 5 different combinations

Bonus: Alter the function to return an Result with an err if there are no real solutions. Test with this Result

24

Benchmarking in Rust



24.1 Introduction

In this lab we will practice with benchmarking functions using the external benchmark suite Criterion

24.1.1 Requirements

A well setup Rust environment

24.1.2 Exercise 1

Let's setup a new Rust library crate project with `cargo new myfac --lib`

For the lib.rs file use the following content to implement a recursive `fac(n)` function.

Listing 1: myfac/src/main.rs

```
#[inline]
pub fn fac(n: u64) -> u64 {
    match n {
        0 => 1,
        n => n*fac(n-1)
    }
}
```

To setup for Criterion benchmarking alter your `Cargo.toml` file like this:

Listing 2: code/myfac/Cargo.toml

```
[package]
name = "myfac"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.
↪html

[dependencies]

[dev-dependencies]
criterion = { version="0.4.0", features=["html_reports"] }

[[bench]]
name = "rust-fac-bench"
harness = false
```

Create a directory called `benches` in the root of your project like this:

```
├── benches
│   └── rust-fac-bench.rs
├── Cargo.lock
├── Cargo.toml
└── src
    ├── lib.rs
    └── main.rs
```

In this benches directory create a file called `rust-fac-bench.rs`

with the following content:

Listing 3: myfac/benches/rust-fac-bench.rs

```
use criterion::{black_box, criterion_group, criterion_main, Criterion};
use myfac::fac;

pub fn criterion_benchmark(c: &mut Criterion) {
```

(continues on next page)

(continued from previous page)

```
    c.bench_function("fac 20", |b| b.iter(|| fac(black_box(20))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

You can now run the Criterion benchmark like this:

```
[~ $> cargo bench
```

Observer the output.

In the directory `target/criterion/reports` you will find an index.html that containts a HTML report. You can read it with your favorite webbrowser.

Now let's add an optimization:

Please alter the `fac` function in the `lib.rs` file like this:

Listing 4: myfac/src/main.rs

```
#[inline]
pub fn fac(num: u64) -> u64 {
    (1..=num).fold(1, |acc, v| acc * v)
}
```

And run the benchmarks again. Look at the timings/numbers in the output of the `cargo bench` command and in the generated html report. What do you observe?

Can you make more optimizations that will improve the timings?

24.1.3 Exercise 2

1. Write a recursive funtion that will calculate the n-th fibonacci nr
   Fib(1) = 1 Fib(2) = 1
   Fib (n) = Fib(n-1) + Fib(n-2)
   Benchmark this function with Criterion.
2. Write an iterative implemntation and benchmark this too. Observer te results.
3. Bonus: try to optimize the recursive implementation. E.g. use memoïzation or something else that will reduce the nr of recursive calls. Use criterion to verify the impact of your optimizations.

24.1.4 Exercise 3

The king of recursion can most probably be found in the so called Ackermann function. This one explodes starting (4,0)

Listing 5: ackermann/src/main.rs

```
fn ackermann(m: u64, n: u64) -> u64 {
    match(m, n) {
        (0, n) => n+1,
        (m, 0) => ackermann(m-1, 1),
        (m, n) => ackermann(m-1, ackermann(m, n-1))
    }
}

fn main() {
    println!("__ Ackermann Function's Calculation __");

    for m in 0..5 {
        for n in 0..(16-m) {
```

(continues on next page)

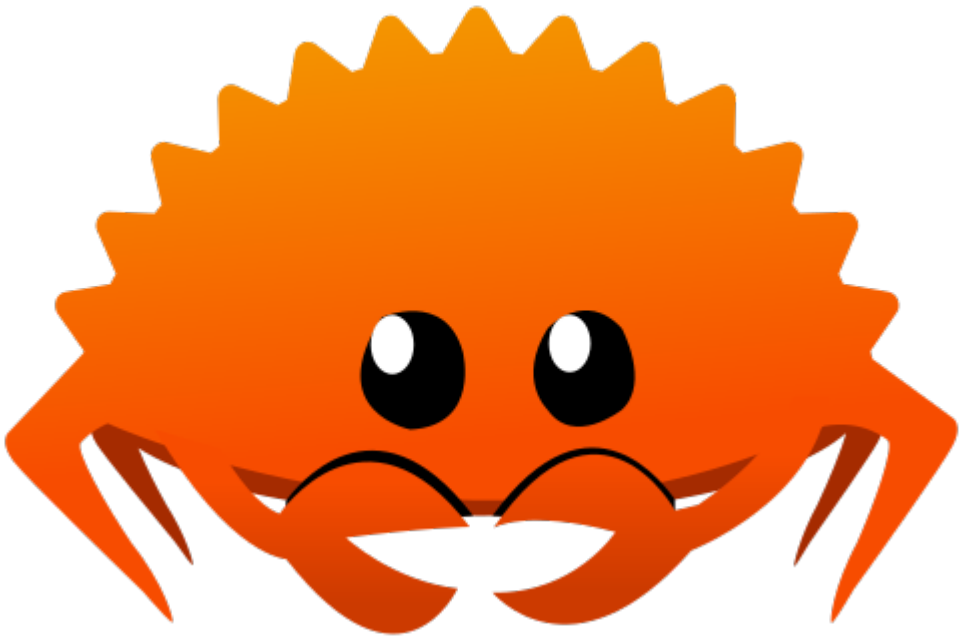
(continued from previous page)

```
            println!("ackermann({}, {}) = {}", m, n, ackermann(m, n));
        }
    }
}
```

Bonus, try to benchmark and optimize the Ackermann() function.

25

Organization of a Rust project



25.1 Introduction

In this set of labs we will practice with setting up a more complex Rust project.

We will learn to:

1. Creating a library crate and publishing it

25.1.1 Requirements

A well setup Rust environment, at least Rust 2021 edition

25.1.2 Exercise 1

Objective: Creating a binary create using multiple modules:

Remember:

- A package is the largest unit of distribution in Rust. It contains a Cargo.toml file, which is the manifest file that describes the package. A package can contain zero or one library crates and any number of binary crates. When we say "crate" in this context, we are referring to a "compilable artifact" – essentially a library or a binary. Most functionality in the Rust ecosystem is provided in the form of packages.
- A crate is a binary or library. If a package has multiple binaries (as seen in a previous example where we had multiple .rs files in a bin/ directory), each one is a separate crate. A library crate is another type of crate that provides reusable functionality and can be depended upon by other crates. Every crate has a root module, which can optionally have child modules.
- Modules are the primary way to organize and split code within a single crate. They form a hierarchical namespace system. Rust uses a file-based module system, which means that each file can correspond to a module.

So let's get started:

1. Create a new rust project called `cases` e.g. `cargo new cases`
2. In the src directory create a file called `lib.rs`
3. In this file create a function with signature `pub fn to_pascal_case(s: &str) -> String` that will accept a string slice and return that string in PascalCase. Your input string can be anything, including already in PascalCase, snake_case, SCREAMING_SNAKE_CASE or camelCase
4. Create unit tests to test the function
5. Create a `usage.rs` Rust example program in examples to `use` your library
6. If all your tests are ok and you are happy with your crate, publish it using `cargo publish`