EXERCISES

In the virtual machine (password: rust) CLion and Visual Studio Code are provided and setup to complete the tasks. The preferred way is to use Visual Studio Code, but CLion provides some automatic type annotations. Feel free to not use the VM if you already have experience with Rust and have it setup on your notebook.

CLion: *This might be very slow in Virtual Box!* Usually CLion uses CMake to build C Projects. The IDE will show an error about not finding the CMakeLists.txt. Don't worry about it. Run the program or test it by pressing the little green run triangle next to the `main` method or next to a `test`.

Visual Studio Code: You can call `cargo build` by pressing `Ctrl + Shift + P` and typing build. This will build the project associated with the currently opened file. To run the program or tests call `cargo run` or `cargo test` in the same way.

You can find the files for the exercises on the desktop of your virtual machine. Every Exercise is in a separate folder with an appropriate name.

GUESSING GAME

Implement a number guessing game. This is a binary project, so you only need to be concerned about the file `guessing_game/src/main.rs`.

- First read an input from `stdin`. The required imports are already in scope.
- Parse that input to an `u32` value. You can use the `parse` and `trim` methods for strings. The documentation might be helpful: https://doc.rust-lang.org/std/string/struct.String.html
- Match your input to the secret number. Can you figure out how to use the `Ordering` enum that has been brought into scope to match?
- Do all of this in a loop.

GETTING STARTED WITH TRAITS AND GENERICS

In this exercise you get an idea of how to use traits and generics in Rust. You can build your project after every step to make sure you didn't screw up at any point ;).

- Create a new binary project using Cargo: `cargo new trait_exercise --bin`. The project will be in the `trait_exercise` folder.
- Create a trait `TwoDimensional` which requires the methods `area(&self) -> f64` and `circumference(&self) -> f64`.
- Define at least two structs which resemble two dimensional geometric shapes with all the fields that define the shape (e.g. circle with one field radius. You may need the `std::f64::consts::PI` constant). The fields should have the type `f64` to make your life easier.
- Implement the `TwoDimensional` trait for your structs.
- Implement a function that take a `TwoDimensinal` struct as argument and return both the area and circumference of the shape in a form you see fitting.
- Implement the `Drop` trait for one of your shapes so that the name of the shape is printed after the shape is dropped. Put some of those shapes in a vector and iterate over them using a `for` loop. Print anything after the loop. What do you notice?

SIMPLE STACK

In this exercise your job is to implement a very basic stack, which only accepts `i32` values as content. For this you need the two data structures `Stack` and `StackElem`.

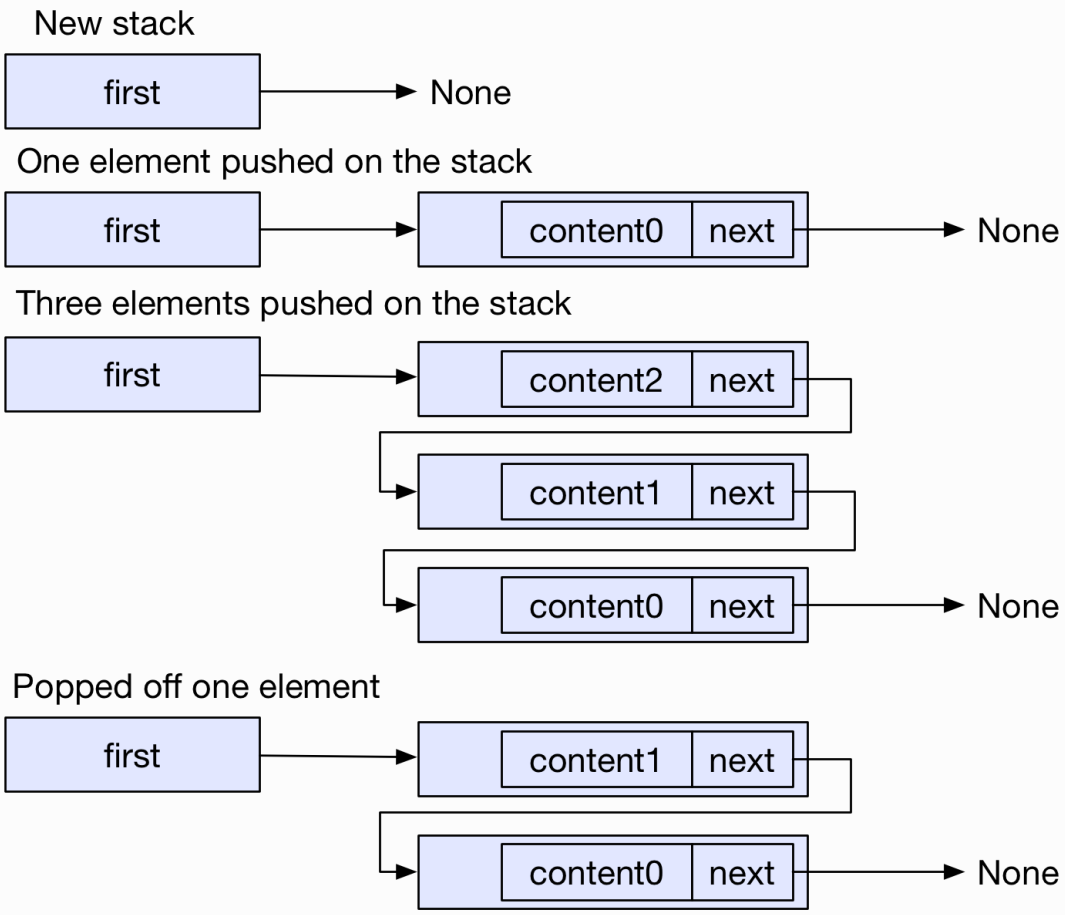
A `Stack` points to its first `StackElem` or nowhere if there is no element in the stack. A `StackElem` has some content of the type `i32` and a pointer to the element following. The `Stack` provides three methods: `new`, `push` and `pop`.

The image shows an overview of the operations. It also gives you an idea of the fields in `Stack` and `StackElem`

- To create a new Stack construct and return a Stack that doesn't point anywhere
- You have to consider that `StackElem`s may point to another `StackElem`
- To push an element on the stack the value in Stack itself needs to be changed
- To pop an element off the stack you also need to change the value in Stack itself

Additionally you can try to implement the required methods for the Iterator trait. On calling the required method, the iterator should simply pop off the first element. You can look up the *Required Methods* here: https://doc.rust-lang.org/std/iter/trait.Iterator.html In the implementation sekelton there is already the field `Item` which indicates the type of the items that is iterated over.

To run the tests for the iterator you have to uncomment the tests in the `simple_stack/tests/simple_stack.rs` file.



Changing the Simple Stack to take &str

If your Stack is working and the tests have passed, you can try to change the type of the content from `i32` to `&str`. As you have learned in the lecture, a struct containing references needs explicit lifetimes.

By simply changing `i32` to `&str` you get a bunch of really helpful compiler errors. Try to solve the exercise with those errors.

Here you also have to uncomment the tests in the `simple_stack/tests/simple_stack.rs`, and comment the tests using the integer simple_stack.