SAFETECH

Rust Programming Language

Day 3

00. Whoami

Quentin Texier

g0h4n

Pentester & Redteamer

@randorisec

https://www.safetech.red/
https://github.com/g0h4n
https://twitter.com/g0h4n_0
https://www.htwmcl.fr

Table of contents

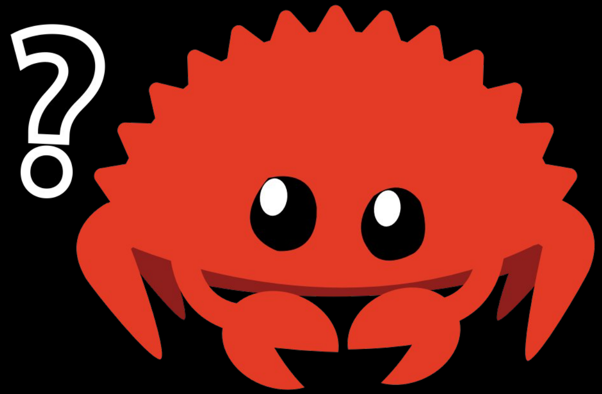
Comprehensive Rust course from Google:
https://github.com/google/comprehensive-rust

Day 3 : Morning

20. Modules
21. Testing
22. Error Handling
23. Unsafe Rust
24. TP - rustbuster

Day 3 : Afternoon

EXAM 2 – to be completed in the afternoon in 3 hours in a group of 2 people



Envoyer les TP à l'adresse email : safetech.red@protonmail.fr
Objet du mail : ***NOM-PRENOM-NOMPROJET***
Supprimer les dossiers « target/ » des projets
Et nommer le zip : ***NOM-PRENOM-NOMPROJET.zip***

20. Modules

20a. Modules

We have seen how impl blocks let us namespace functions to a type.
Similarly, mod lets us namespace types and functions:

```rust
mod foo {
    pub fn do_something() {
        println!("In the foo module");
    }
}

mod bar {
    pub fn do_something() {
        println!("In the bar module");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

```
$ cargo new modules
$ cd modules
$ cargo build
```

20b. Filesystem Hierarchy

Omitting the module content will tell Rust to look for it in another file:

```rust
mod garden;
```

This tells rust that the **garden module content** is found at **./src/garden.rs**.
Similarly, a **garden::vegetables** module can be found at ./src/garden/vegetables.rs.

The crate root is in:

- **./src/lib.rs** (for a library crate)
- **./src/main.rs** (for a binary crate)

Modules defined in files can be documented, too, using “inner doc comments”.
These document the item that contains them – in this case, a module.

```rust
//! This module implements the garden,
//! including a highly performant germination
//! implementation.

// Re-export types from this module.
pub use garden::Garden;
pub use seeds::SeedPacket;

/// Sow the given seed packets.
pub fn sow(seeds: Vec<SeedPacket>) {
    todo!()
}

/// Harvest the produce in the garden that is ready.
pub fn harvest(garden: &mut Garden) {
    todo!()
}
```

20c. Visibility

Modules are a privacy boundary:

- Module items are private by default (hides implementation details).
- Parent and sibling items are always visible.
- In other words, if an item is visible in module foo, it's visible in all the descendants of foo.

```rust
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

```
$ cargo new visibility
$ cd visibility
$ cargo build
```

20d. use, super, self

A module can bring symbols from another module into scope with use.
You will typically see something like this at the top of each module:

```
use std::collections::HashSet;
use std::process::abort;
```

Paths are resolved as follows:

- As a relative path:
  foo or **self::foo** refers to **foo** in the **current** module,
  super::foo refers to **foo** in the **parent** module.

- As an absolute path:
  crate::foo refers to **foo** in the **root** of the current crate,
  bar::foo refers to foo in the **bar** crate.

20e1. Exercice: Modules for a GUI Lib

In this exercise, you will **reorganize a small GUI Library implementation.**
This library defines a Widget trait and a few implementations of that trait, as well as a main function.

It is typical to put each type or set of closely-related types into its own module, so each widget type should get its own module.

In the examples folder use “exercice-gui-modules”.

Edit the resulting src/main.rs to add mod statements, and add additional files in the src directory.

```
$ cargo new exercice-gui-modules
$ cd exercice-gui-modules
$ cargo build
```

20e2. Solution: Modules for a GUI Lib

Structure for optimized code:

```
src
├── main.rs
├── widgets
│   ├── button.rs
│   ├── label.rs
│   └── window.rs
└── widgets.rs
```

21. Testing

21a. Test Modules
21b. Other Types of Tests
21c. Compiler Lints and Clippy

21a. Test Modules

Rust and Cargo come with a simple unit test framework:

- Unit tests are supported throughout your code.
- Integration tests are supported via the **tests/** directory.

Tests are marked with **#[test]**.

Unit tests are often put in a nested tests module, using **#[cfg(test)]** to conditionally compile them only when building tests

- This lets you unit test private helpers.
- The **#[cfg(test)]** attribute is only active when you run cargo test..

```
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_empty() {
        assert_eq!(first_word(""), "");
    }

    #[test]
    fn test_single_word() {
        assert_eq!(first_word("Hello"), "Hello");
    }

    #[test]
    fn test_multiple_words() {
        assert_eq!(first_word("Hello World"), "Hello");
    }
}
```

```
$ cargo new test-modules
$ cd test-modules
$ cargo build
```

21b. Other Types of Tests

SAFETECH

Integration Tests

If you want to test your library as a client, use an integration test.

Create a **.rs** file under **tests/**:

```rust
// tests/my_library.rs
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

These tests only have access to the public API of your crate.

Documentation Tests

Rust has built-in support for documentation tests:

```rust
/// Shortens a string to the given length.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5),
"Hello");
/// assert_eq!(shorten_string("Hello World", 20),
"Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) ->
&str {
    &s[..std::cmp::min(length, s.len())]
}
```

21b. Other Types of Tests

Documentation Tests

```
/// Shortens a string to the given length.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
```

- Code blocks in *///* comments are automatically seen as Rust code.
- The code will be compiled and executed as part of **cargo test**.
- Adding # in the code will hide it from the docs, but will still compile/run it.
- Test the above code on the Rust Playground.

21c. Compiler Lints and Clippy

The Rust compiler produces fantastic error messages, as well as helpful built-in lints.
Clippy provides even more lints, organized into groups that can be enabled per-project.

```rust
#[deny(clippy::cast_possible_truncation)]
fn main() {
    let x = 3;
    while (x < 70000) {
        x *= 2;
    }
    println!("X probably fits in a u16, right? {}", x as u16);
}
```

```
$ cargo new compiler-lints-and-clippy
$ cd compiler-lints-and-clippy
$ cargo build
```

22. Error Handling

22a. Panics

SAFETECH

Rust handles fatal errors with a “**panic**”.
Rust will trigger a panic if a fatal error happens at runtime:

```
fn main() {
    let v = vec![10, 20, 30];
    println!("v[100]: {}", v[100]);
}
```

```
$ cargo new panics
$ cd  panics
$ cargo build
```

- Panics are for **unrecoverable and unexpected errors**.
  - Panics are **symptoms of bugs** in the program.
  - **Runtime failures** like failed bounds checks can panic
  - Assertions (such as **assert!**) panic on failure
  - Purpose-specific panics can use the **panic! macro**.
- A panic will “unwind” the stack, dropping values just as if the functions had returned.
  - Use non-panicking APIs (such as Vec::get) if crashing is not acceptable.

22b. Result

Our primary mechanism for error handling in Rust is the **Result** enum, which we briefly saw when discussing standard library types.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Dear diary: {contents} ({bytes} bytes)");
            } else {
                println!("Could not read file content");
            }
        }
        Err(err) => {
            println!("The diary could not be opened: {err}");
        }
    }
}
```

```
$ cargo new result
$ cd  result
$ cargo build
```

22c1. Try Operator

Runtime errors like connection-refused or file-not-found are handled with the Result type, but matching this type on every call can be cumbersome.

The try-operator **?** is used to return errors to the caller. It lets you turn the common:

```
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

Into the much simpler:

```
some_expression?
```

22c2. Try Operator

SAFETECH

We can use this to simplify our error handling code:

```
use std::io::Read;
use std::{fs, io};

fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(err) => return Err(err),
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

```
$ cargo new try-operator
$ cd  try-operator
$ cargo build
```

22d1. Try Conversions

The effective expansion of **?** is a little more complicated than previously indicated:

```
expression?
```

Works the same as:

```
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

The **From::from** call here means we attempt to convert the error type to the type returned by the function.
This makes it easy to encapsulate errors into higher-level errors.

22d2. Try Conversions

```
use std::error::Error;
use std::fmt::{self, Display, Formatter};
use std::fs::File;
use std::io::{self, Read};

#[derive(Debug)]
enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
}

impl Error for ReadUsernameError {}

impl Display for ReadUsernameError {
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "IO error: {e}"),
            Self::EmptyUsername(path) => write!(f, "Found no username in {path}"),
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> Self {
        Self::IoError(err)
    }
}
```

```
$ cargo new try-convertions
$ cd  try-conversions
$ cargo build
```

```
fn read_username(path: &str) -> Result<String,
ReadUsernameError> {
    let mut username = String::with_capacity(100);
    File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return
Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //std::fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}}
```

22e. Dynamic Error Types

Sometimes we want to allow any type of error to be returned without writing our own enum covering all the different possibilities. The **std::error::Error** trait makes it easy to create a trait object that can contain any error.

```
$ cargo new dynamic-error-types
$ cd  dynamic-error-types
$ cargo build
```

```rust
use std::error::Error;
use std::fs;
use std::io::Read;

fn read_count(path: &str) -> Result<i32, Box<dyn Error>> {
    let mut count_str = String::new();
    fs::File::open(path)?.read_to_string(&mut count_str)?;
    let count: i32 = count_str.parse()?;
    Ok(count)
}

fn main() {
    fs::write("count.dat", "1i3").unwrap();
    match read_count("count.dat") {
        Ok(count) => println!("Count: {count}"),
        Err(err) => println!("Error: {err}"),
    }
}
```

22f. thiserror and anyhow

The **thiserror** and **anyhow** crates are widely used to simplify error handling.

- **thiserror** is often used in libraries to create custom error types that implement From<T>.
- **anyhow** is often used by applications to help with error handling in functions, including adding contextual information to your errors.

```
$ cargo new thiserror-and-anyhow
$ cd  thiserror-and-anyhow
$ cargo build
```

```rust
use anyhow::{bail, Context, Result};
use std::fs;
use std::io::Read;
use thiserror::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)]
#[error("Found no username in {0}")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .with_context(|| format!("Failed to open {path}"))?
        .read_to_string(&mut username)
        .context("Failed to read")?;
    if username.is_empty() {
        bail!(EmptyUsernameError(path.to_string()));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err) => println!("Error: {err:?}"),
    }
}
```

23. Unsafe Rust

23a. Unsafe

The Rust language has two parts:

- **Safe Rust**: memory safe, no undefined behavior possible.
- **Unsafe Rust**: can trigger undefined behavior if preconditions are violated.

We saw mostly safe Rust in this course, but it's important to know what Unsafe Rust is.

Unsafe code is usually small and isolated, and its correctness should be carefully documented. It is usually wrapped in a safe abstraction layer. Unsafe Rust gives you access to five new capabilities:

- Dereference raw pointers.
- Access or modify mutable static variables.
- Access **union** fields.
- Call **unsafe** functions, including **extern** functions.
- Implement **unsafe** traits.

We will briefly cover unsafe capabilities next.
For full details, please see [Chapter 19.1 in the Rust Book] and the [Rustonomicon].

23b. Dereferencing Raw Pointers

Creating pointers is safe, but dereferencing them requires **unsafe**:

```
$ cargo new dereferencing-raw-pointer
$ cd  dereferencing-raw-pointer
$ cargo build
```

```rust
fn main() {
    let mut s = String::from("careful!");

    let r1 = &mut s as *mut String;
    let r2 = r1 as *const String;

    // SAFETY: r1 and r2 were obtained from references and so are guaranteed to
    // be non-null and properly aligned, the objects underlying the references
    // from which they were obtained are live throughout the whole unsafe
    // block, and they are not accessed either through the references or
    // concurrently through any other pointers.
    unsafe {
        println!("r1 is: {}", *r1);
        *r1 = String::from("uhoh");
        println!("r2 is: {}", *r2);
    }

    // NOT SAFE. DO NOT DO THIS.
    /*
    let r3: &String = unsafe { &*r1 };
    drop(s);
    println!("r3 is: {}", *r3);
    */
}
```

23c. Mutable Static Variables

It is safe to read an immutable static variable:

```rust
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

However, since data races can occur, it is unsafe to read and write mutable static variables:

```rust
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    // SAFETY: There are no other threads which could be accessing `COUNTER`.
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_counter(42);

    // SAFETY: There are no other threads which could be accessing `COUNTER`.
    unsafe {
        println!("COUNTER: {COUNTER}");
    }
}
```

```
$ cargo new mutable-static-variables
$ cd  mutable-static-variables
$ cargo build
```

23d. Unions

SAFETECH

Unions are like enums, but you need to track the active field yourself:

```
$ cargo new unions
$ cd unions
$ cargo build
```

```rust
#[repr(C)]
union MyUnion {
    i: u8,
    b: bool,
}

fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("bool: {}", unsafe { u.b }); // Undefined behavior!
}
```

23e2. Unsafe Functions - Calling

Calling Unsafe Functions

A function or method can be marked **unsafe** if it has extra preconditions you must uphold to avoid undefined behaviour:

```
$ cargo new calling-unsafe-functions
$ cd calling-unsafe-function
$ cargo build
```

```rust
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    let emojis = "🗻∈🌏";

    // SAFETY: The indices are in the correct order, within the bounds of the
    // string slice, and lie on UTF-8 sequence boundaries.
    unsafe {
        println!("emoji: {}", emojis.get_unchecked(0..4));
        println!("emoji: {}", emojis.get_unchecked(4..7));
        println!("emoji: {}", emojis.get_unchecked(7..11));
    }

    println!("char count: {}", count_chars(unsafe { emojis.get_unchecked(0..7) }));

    // SAFETY: `abs` doesn't deal with pointers and doesn't have any safety
    // requirements.
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }

    // Not upholding the UTF-8 encoding requirement breaks memory safety!
    // println!("emoji: {}", unsafe { emojis.get_unchecked(0..3) });
    // println!("char count: {}", count_chars(unsafe {
    // emojis.get_unchecked(0..3) }));
}

fn count_chars(s: &str) -> usize {
    s.chars().count()
}
```

23e1. Unsafe Functions - Writing

SAFETECH

Writing Unsafe Functions

You can mark your own functions as **unsafe** if they require particular conditions to avoid undefined behaviour.

```
$ cargo new writing-unsafe-functions
$ cd calling-unsafe-functions
$ cargo build
```

```rust
/// Swaps the values pointed to by the given pointers.
///
/// # Safety
///
/// The pointers must be valid and properly aligned.
unsafe fn swap(a: *mut u8, b: *mut u8) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // SAFETY: ...
    unsafe {
        swap(&mut a, &mut b);
    }

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

23f. Implementing Unsafe Traits

SAFETECH

Like with functions, you can mark a trait as unsafe if the implementation must guarantee particular conditions to avoid undefined behaviour.

For example, the zerocopy crate has an unsafe trait that looks something like this:

```
$ cargo new implementing-unsafe-traits
$ cd  implementing-unsafe-traits
$ cargo build
```

```
use std::mem::size_of_val;
use std::slice;

/// ...
/// # Safety
/// The type must have a defined representation and no padding.
pub unsafe trait AsBytes {
    fn as_bytes(&self) -> &[u8] {
        unsafe {
            slice::from_raw_parts(
                self as *const Self as *const u8,
                size_of_val(self),
            )
        }
    }
}

// SAFETY: `u32` has a defined representation and no padding.
unsafe impl AsBytes for u32 {}
```

24a - Exercice - rustbuster - partie 1

Créer un bruteforcer d’url pour énumérer les endpoints d’un site web:
cargo new rustbuster-1

Le programme doit prendre ne paramètre l’**URL** du serveur Web à bruteforcer et le chemin de **la wordlist** à utiliser. Il faut utiliser la librairie clap pour réaliser le parsing des arguments. Créer une structure dédiée pour la target qui va contenir l’URL, les codes erreurs autorisés, les codes erreurs non autorisés et surtout un tableau Vec<String> avec les endpoints trouvés. Ajouter la possibilité de modifier **le User-Agent** avec l’argument --ua.

Le programme doit s’utiliser de la manière suivante :

./rustbuster-1 dir -u <URL> -w <WORDLIST> --ua <USER_AGENT>

UTILISER LA WORDLIST SUIVANTE :
https://raw.githubusercontent.com/danielmiessler/SecLists/master/Discovery/Web-Content/common.txt

UTILISER LE USER-AGENT SUIVANT :
RustBuster-NOM-Prénom

24b - Exercice - rustbuster - partie 2

Créer un bruteforcer d’url pour énumérer les endpoints d’un site web:
cargo new rustbuster-2

Faire en sorte que rustbuster puisse réaliser le bruteforce de manière récursif dès qu’il trouve une dossier (code http 301). Ajouter les dossiers trouvés dans un **Vec<String>** de la structure Target pour bruteforcer avec la même wordlist les dossiers. Ajouter l’argument --recursif ou -r

/HERE
/wordpress/HERE
etc

Le programme doit s’utiliser de la manière suivante :

./rustbuster-2 dir -u <URL> -w <WORDLIST> --ua <USER_AGENT> --recursif

UTILISER LA WORDLIST SUIVANTE :
https://raw.githubusercontent.com/danielmiessler/SecLists/master/Discovery/Web-Content/common.txt

UTILISER LE USER-AGENT SUIVANT :
RustBuster-NOM-Prénom

Exam 2

Et voila!