The #12in23 Challenge - Rust - Trying Rust in Mechanical March 2023

Written by Seth Corker on Benevolent Bytes

Banner photo by Rod Long on Unsplash

Why Rust?

There was a range of languages to choose from in Mechanical March, most of them I haven't really used but have been interested in. I've dabbled in C and C++ in the past; sending packets across the network using C in university was my least favourite project, and I initially threw myself into learning C++ by printing out a massive manual to teach myself game development when I was a teenager.

So, Rust, Go and Zig are left. Go and Zig seem interesting, and I'd like to look at them in the future, but Rust is a programming language I've been shadowing for some time. There always seems to be some compelling tools that come have come out over the past few years that tout Rust, swc comes to mind for developer tools and Tauri as a faster Electron alternative. From the outside, it looks like Rust is taking over in cases where speed and safety are paramount. This is why I thought I'd finally take the plunge.

Developer experience

This is my initial impression of Rust. There are a few things that stand out the most, both good and bad.

Documentation

The documentation for Rust itself is thorough but, Elixir's hexdocs have raised the bar on what documentation for packages should look like. I found the documentation for packages, crates in Rust, a bit difficult to understand. There is module documentation that looks like it's automatically generated; however, there's not always examples of how a crate should actually be used and my understanding of the type annotations wasn't always enough to know what's going on. I often found I needed more context. The only reason I'm harsh of Rust is that Elixir does an outstanding job on this.

Have a look at the following docs sites for web frameworks in each ecosystem: - Rocket — https://api.rocket.rs/v0.5/rocket/ - Phoenix — https://hexdocs.pm/phoenix/installation.html Rocket doesn't really have anything beyond the different modules, macros, functions, etc. which are provided, there's nothing on how to use anything. Guides are hosted on a different website. Contrast this with Phoenix, the default documentation experience is similar to what's shown for crate documentation, but there's also an entire library of guides for each thing you might want to do.

Editor integration

As important as the language itself, is the integration with your editor. VSCode extension rust-analyzer is good, it works well and has useful features like a Run and Debug annotation above the main function. You can see annotations inline and several other useful features you'd expect. This was by far the most useful tool when developing to figure out type signatures as I needed them rather than sifting though the documentation.

Modern tooling

Rust has modern tooling. It's all the things that take the hassle out of development, and Rust has tried to bundle official tools for common use cases. `rustup` is used for managing Rust installations and cargo is the builtin package manager. Formatting, testing, documentation and linting is all handled through official solutions. This is undoubtedly what I want in a language, it's easy to pickup and get running without trying to figure out which formatter I should use, which compiler and how to format my code. I still have issues whenever I try to go back to large Python projects, which package manager do I use? How do I set up my environment? I like choice, but it can be overwhelming as someone unfamiliar with a new ecosystem.

Language features

Type system

I heard countless people complain about the borrow-checker, but the greatest challenge I had was with typing itself. Rust is very particular about types, which makes you think a lot more about which types you're using and how they behave. I found working with strings the most challenging because they can be pointers, literals etc. And I always seemed to choose the wrong one and have to cast between different types frequently, I put this down to my lack of experience with Rust. It's interesting to think about correct type usage at a low level, it's not something you usually think about when working with dynamic and high-level languages.

Useful compiler errors

I was put off by lower level programming languages by doing C in university for a networking assignment, as I mentioned earlier. It was frustrating how difficult it was working with strings and arrays compared with previous programs I'd written in Python and Java. I specifically remember debugging an issue where I accidentally wrote over a piece of memory. Compiler errors weren't helpful, it was a more manual process of commenting out code, rerunning or stepping through line by line in your head. Rust solves those issues for me.

An early example I ran into was when I attempted to multiply an `u8` (unsigned 8 bit integer) by 200, `u8` can only hold values from 0 to 255, so it would overflow and Rust shows an error during compilation. Nice!

```
error: this arithmetic operation will overflow
  --> src/main.rs:32:31
   |
32 |     print!("a: {:?} * 4 is ", number_a * 4)
   |                               ^^^^^^^^^^^^ attempt to compute `128_u8 * 4_u8`, which would overflo
   |
   = note: `#[deny(arithmetic_overflow)]` on by default
```

Compilation is quick and feedback from errors was really useful in debugging.

Pattern matching

One of my favourite features exists in Rust too! `match` is reminiscent of `case` in Elixir and the compiler checks exhaustiveness to make sure you're not missing unhandled cases.

```rust
/// Find the regexp given a reader
fn find_in_lines<T: BufRead + Sized>(reader: T, regexp: Regex) {
    for (index, line) in reader.lines().enumerate() {
        let extracted_line = line.unwrap();
        let contains_substring = regexp.find(&extracted_line);

        match contains_substring {
            Some(_) => println!("{}: {}", index, extracted_line.trim()),
            None => (),
        }
    }
}
```

Takeaways

I enjoyed my short time with Rust and I want to keep learning and building things. I didn't get to spend as much time with it as I would have liked. That's a nice feeling to have, it means I liked Rust enough to use it again. I don't use a language that is so low level with high-level constructs in my everyday. Not only that, but I've been really excited to discover more use cases and tinker around with.

The last project I started was building a command line app with Clap and learned how annotations work with the derive API. Rust has some powerful language features, and I can see an immediate use case when combined with existing languages I use; I think it will be cool to write some high-performance Rust utilities that can be leveraged by Elixir using NIFs. For now, I'm leaving Rust to see what other languages are out there, but I suspect I'll be back sometime soon.

Resources

- Playground - An online playground for the language
- rust-analyser for VSCode
- Learn Rust on exercism