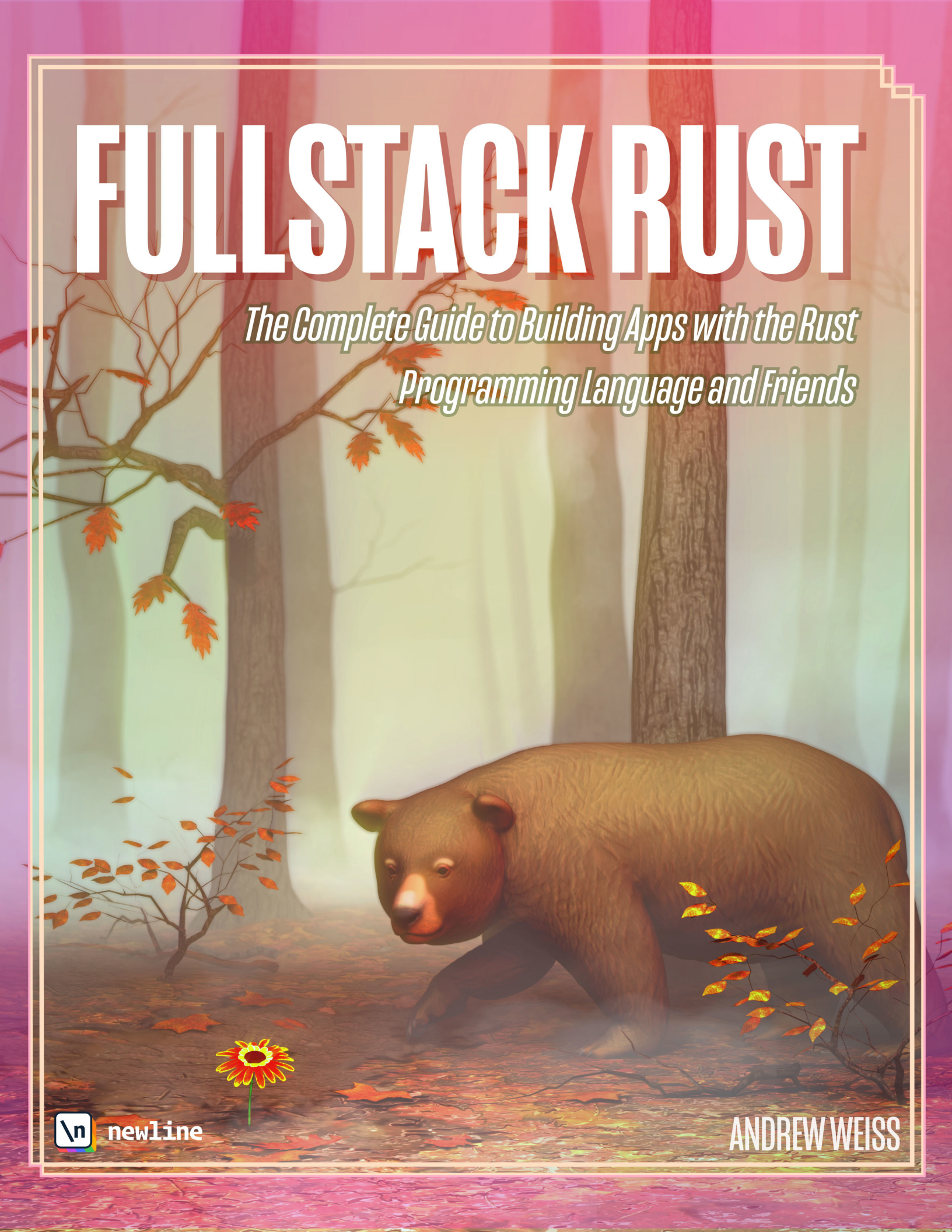
FULLSTACK RUST

The Complete Guide to Building Apps with the Rust Programming Language and Friends



newline

ANDREW WEISS

Fullstack Rust

The Complete Guide to Buildings Apps with the Rust Programming Language and Friends

Written by Andrew Weiss
Edited by Nate Murray

Published by Fullstack.io.

Contents

Book Revision

Revision 3 - EARLY RELEASE - 2020-01-29

EARLY RELEASE VERSION

This version of the book is an early release. Most, but not all, of the chapters have been edited. Contents will change before First Edition.

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/rust[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/rust
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

Introduction

There are numerous reasons to be hopeful about the future of computing, one of which is the existence and continued progression of the Rust programming language.

We are currently in the fifth era of programming language evolution. This is an era where languges have been able to take all of the learnings since the 1950s and incoporate the best parts into languages each with its own cohesive vision.

We have specialized languages cropping up for a wide variety of tasks and countless general purpose languages being actively developed and used. There are significant resources in industry to invest in language design and development which compliment the vibrant academic community. With tools like LLVM and the explosion of open source, creating a language has never been easier.

It is in this environment that Rust has been voted the "most loved programming language" in the Stack Overflow Developer Survey every year since 2016. Standing out in this increasingly crowded world of languages is enough of a reason to ask why Rust?

Why Rust?

There are a few potential readings of this question: why should I learn Rust, why are others using Rust, why should I choose Rust over language X? These are all relevant, but I want to start with a bit of a philosophical argument for Rust independent of these specific points.

There is a limit to how transformative an experience you can have when learning a language in a similar paradigm to one you already know. Every language and paradigm has an intrinsic style that is forced on you as you try to solve problems.

If you work within that style then your code will flow naturally and the language will feel like it is working with you. On the other hand, if you fight the natural style of the language you will find it hard or impossible to express your ideas.

Moreover, learning and working with a language will teach you ways to be more effective based on how the language guides you based on its natural design. How much you are able to learn is a function of how much your prior experience and mental models cover the new language.

Rust borrows a lot of ideas from other languages and is truly multi-paradigm, meaning you can write mostly functional code or mostly imperative code and still fit nicely within the language. The most unique feature of the language, the borrow checker, is a system that enforces certain invariants which allow you to make certain safety guarantees. Even this is built on prior art found in earlier languages.

All of these good ideas from the world of programming language design combine in a unique way to make Rust a language that truly makes you think about writing code from a novel perspective. It does not matter how much experience you have, learning Rust will forever change the way you write code for the better.

Okay with that philosophical argument out of the way, let's dig in to some specifics of why Rust is a exciting.

To help guide this discussion, we can break things down into a few broad categories.

On language comparisons

There is no best programming language. Almost every task has a variety of languages which could be the right tool. Every language comes with good parts and bad parts. Evaluating these tradeoffs when faced with a particular problem space is an art unto itself. Therefore, nothing in this book is intended to disparage or denigrate any particular alternative language. The primary goal of this book is to faithfully present Rust. That being said, sometimes comparisons with other languages are instructive and are meant to be instructive rather than as fuel in a flame war.

Language features

There are a lot of features of Rust which make it a great tool for a great number of tasks. Some highlights include:

- Performance
- Strong, static, expressive type system
- Great error messages
- Modern generics
- Memory safety
- Fearless concurrency
- Cross platform
- C interoperability

Let's briefly go through some of these which are probably the biggest reasons that Rust gets talked about.

Performance

Rust is exceptionally fast, in the same ballpark as C and C++. For some programs, specifically due to the lack of pointer aliasing, the Rust compiler can sometimes have enough information to optimize code to be faster than what is possible in C without directly writing assembly. For the vast majority of use cases, you should consider Rust to be fast enough.

Often the most obvious way to write a program is also the fastest. Part of this comes from the commitment to zero-cost abstractions, which are summarized by Bjarne Stroustrup, the creator of C++, as:

> What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

Most of the abstractions in Rust, for example iterators, are zero-cost by this definition. The most efficient way to traverse a vector of data is to use a for loop which uses an iterator trait. The generated assembly is usually as good as you could hope for had you written it by hand.

The other aspect of performance is memory consumption. Rust does not have a garbage collector so you can use exactly as much memory as is strictly necessary at any given time. Due to the design of the language, you start to think and see every memory allocation. Using less memory is often easier than the converse. The rest of the language is designed around making working without a garbage collector painless.

Type system

The type system of Rust is influenced by the long lineage of functional programming languages such as ML and Haskell. It is static, nominal, strong, and for the most part inferred. Don't worry if that didn't mean anything to you, but if it did then great. You encode the ideas and constraints of your problem with types. You only have to specify types in a few places with the rest able to be inferred. The compiler then checks everything for you so that you get faster feedback about potential problems. Entire classes of bugs are impossible because of static typing. Most things you encounter in practice are expressible in the type system. The compiler then checks everything for you so that you get faster feedback about potential problems. Entire classes of bugs are impossible because of static typing.

A type system is often called expressive if it is easy to encode your ideas. There are some concepts which are impossible to express in static type systems. Rust has powerful abstraction facilities like sum and product types, tuples, generics, etc. which put the type system definitely in the expressive camp.

Memory safety

A language is memory safe if certain classes of bugs related to memory access are not possible. A language can be called memory unsafe if certain bugs are possible. A non-exhaustive list of memory related bugs include: dereferencing null pointers, use-after free, dangling pointers, buffer overflows.

If you have never written code in a memory unsafe language then these might sound like gibberish to you, which is fine. The important point is this class of bugs is a consistent and large source of security vulnerabilities in systems implemented with memory unsafe languages. For example, about 20% of CVEs[4] ever filed against the Linux kernel are due to memory corruption or overflows. Linux is implemented primarily in C, a spectacularly memory unsafe language.

Memory safety bugs are bad for security and reliability. They lead to vulnerabilities and they lead to crashes. If you can rule these out at compile time then you are in a much better state of the world.

[4] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

Rust is designed to be memory safe, and thus it does not permit null pointers, dangling pointers, or data races in safe code. There are many interacting features which allow this guarantee. The primary one is the unique system of ownership combined with the borrow checker. This is part of the compiler that ensures pieces of data live at least as long as they need to in order to be alive when they are used.

One other feature is the builtin `Option` type. This is used to replace the concept of null found in many other languages. In some languages, every type is secretly the union of that type with null. This means that you can always end up with bugs where you assume some variable had a value and it actually was inhabited by the dreaded null. Rust disallows this by not having null and instead having a type which can explicitly wrap other types. For example, consider this Rust code:

```
fn print_number(num: Option<i32>) {
  match num {
    Some(n) => println!("I see {}!", n),
    None => println!("I see nothing!"),
  }
}

fn main() {
  let x = Some(42);
  let y = None;

  print_number(x);
  print_number(y);
}
```

The function `print_number` must handle the case where `num` is `None`, meaning the `Option` has no value. There are a few different ways to handle that case but you must explicitly do something for that case or else your code will not compile.

The one caveat here is that Rust does allow blocks of code to be marked `unsafe` and within those blocks it is possible to violate memory safety. Some things are impossible for the compiler to verify are safe and therefore it refuses to do so. It requires you to use `unsafe` regions of code to ensure that you understand the invariants required to make sure your code truly is safe.

This does not defeat the purpose, rather in isolates the areas of auditability to just those sections of code which are specifically marked. Nothing you do in normal Rust, also called safe Rust, can result in a memory safety violation, unless something in unsafe code did something wrong ahead of you.

As an example, calling C functions from Rust is unsafe. This is because Rust has no way of knowing what the C code is doing, and C is inherently unsafe, therefore the compiler cannot uphold its guarantees if you call out to C. However, can it be safe to call C? Yes, provided you fill in the visibility gap for the compiler with your own logic.

Fearless concurrency

Concurrency in programming means that multiple tasks can be worked on at the same time. This is possible even for a single thread of execution by interleaving the work for different tasks in chunks rather than only working on tasks as entire chunks.

Parallelism in programming means multiple tasks executing at the exact same time. True parallelism requires multiple threads of execution (or the equivalent).

The Rust language describes its facilities for concurrent and parallel computing as fearless concurrency with a bit of conflation of terms. I will continue in this tradition and use concurrency to mean concurrency and/or parallelism.

Most modern, high level languages have chosen how they want to support concurrency and mostly force you down that path. Some more general purpose languages provide the tools to handle concurrency however you see fit. For example, Go is designed around Communicating Sequential Processes (CSP) and therefore concurrency is most easily achieved using channels and goroutines. Python, on the other hand, has libraries for threads, multiprocesses, message passing actors, etc.

Rust is a low-level language by design and therefore provides tools that allow you to use the model of your choice to achieve your particular goals. Therefore, there are facilities for threads but also channels and message passing.

Regardless of what technique you choose to tackle concurrency and/or parallelism, the same ownership model and type system that ensures memory safety also ensures thread safety. This means that it is a compile time error to write to the same memory from different threads without some form of synchronization. The details are less important than the concept that entire classes of problems that are notoriously difficult to debug in other languages are completely eliminated at compile time while, importantly, retaining all of the performance benefits.

C interoperability

Rust is foremost a systems programming language. That means it is designed for building low level systems with strict performance requirements and reliability constraints. Frequently in this world, C is the glue that binds many disparate systems. Therefore being able to interoperate with C is an absolute necessity to be able to have a serious systems language. Luckily it is straightforward to interact with C both by calling into C from Rust, as well as exposing Rust as a C library.

You might be saying that sounds great but I don't plan on writing an operating system anytime soon so why should I care? C is also the most common mechanism for making dynamic languages faster. Typically, when parts of your Python or Ruby code are showing performance problems, you can reach for an extension written in C to speed things up. Well, now you can write that extension in Rust and get all of the high level benefits of Rust and still make your Python or Ruby code think it is talking to C. This is also quite an interesting area for interacting with the JVM.

Ecosystem

Software is not constructed in a vacuum, the practice of programming is often a community driven endeavor. Every language has a community whether it actively cultivates it or not. The ecosystem around a language includes the community of people, but also the tooling or lack thereof.

Rust has grown quite a lot in its short life and has gone through some growing pains as a result. However, the community has always been very welcoming and importantly the culture is a first-class citizen. Rust specifically has a community team as part of the governance structure of the language. This goes a long way to helping the language and ecosystem grow and mature.

We will cover much of the useful tooling that exists around Rust in detail below. However, suffice it to say that the tooling around the language is some of the best that exists. There have been a lot of learnings over the past twenty years about how to manage toolchains and dependencies and Rust has incorporated all of this quite well.

The nature of programming

The systems and applications we are building today are different than 50 years ago, they are even different than 10 years ago. Therefore, it should not be too much of a stretch to say that the tools we use should also be different.

There is an explosion of embedded systems due to what is commonly called the Internet of Things. However, is C still the best tool for that job? Mission critical software that controls real objects that could lead to serious consequences in the case of failure should be using the best tool for the job. Rust is a serious contender in this space. For example, it is easy to turn off dynamic memory allocation while still being able to use a lot of the nice parts of the language.

The other explosion is continuing on the web. We have been in a web revolution for quite a while now, but things have not slowed down. The deficits of JavaScript are well known and have been addressed along quite a few paths. We have many languages which compile to JavaScript but provide nice features like type systems or a functional paradigm. However, there are fundamental performance and security issues with JavaScript regardless of how you generate it. WebAssembly (WASM) is a step in a different direction where we can compile languages like Rust to a format natively executable in the browser.

Fun

Rust is fun to write. You will disagree with this and think I am crazy at some point while you are learning Rust. There is a learning curve which can be distinctly not fun. However, once your mental model starts to shift, you will find yourself having moments of pure joy when your code just works after the compiler gives you the okay.

Why not Rust

Rust is just another programming language and as such is just another software project. This means it has built up some legacy, it has some hairy parts, and it has some future plans which may or may not ever happen. Some of this means that for any given project, Rust might not be the right tool for the job.

One area in which Rust might not be right is when interfacing with large C++ codebases. It is possible to have C++ talk to C and then have C talk to Rust and vice versa. That is the approach you should take today if possible. However, Rust does not have a stable ABI nor a stable memory model. Hence, it is not directly compatible with C++. You can incrementally replace parts of a system with Rust and you can build new parts in Rust, but plug-and-play interoperability with C++ is not a solved problem.

Furthermore, Rust takes time to learn. Now this is often cited as a reason for sticking with some other language because one is deemed an expert in that language. However, a counter point might be that you are not as much of an expert in that language as you might believe. A further counter point is that it might not matter, the other language might be fundamentally flawed enough that being an expert is irrelevant. Nonetheless, there are times where using the tool you know is the right answer.

The gap between learning Rust and knowing it from using it in anger is a bit bigger than in some other languages. Therefore the learning curve might seem steeper than you are used to. However, this is primarily because what is safe in Rust with the borrow checker helping you can be insane in other languages.

Type systems are amazing. You tell the computer some facts about your problem domain and it continually checks that those things are true and lets you know if you screw up. Yet there are valid programs which are unexpressible in a static type system. This is both theoretically true and actually happens in practice. Moreover, dynamic languages can frequently be more productive for small, isolated tasks. Sometimes the cost of the type system is not worth it.

This book's mission

- Teach you Rust by building real applications
- Work through common scenarios
- Show normal stumbling blocks and how to deal with them
- Not to show esoteric language features
- Get you to productive with pointers where you can go deeper

Who should read this book?

- Might be hard if you have never written any code before, but maybe

- Great if you are coming from an existing systems language like C++ or Java but might actually be harder to overcome existing ways of doing things
- Great for backgrounds in dynamic languages, the type system might be a challenge if you've never worked with a strong type system before

If you have some exposure to strong, static type systems and you are comfortable with imperative programming then you'll do just fine.

Absolutely zero Rust background is assumed.

Rust has a bit of notority for having a steep learning curve, but it is actually mostly about unlearning things from other languages. Therefore, having less experience can work in your favor.

Getting your environment setup

- https://www.rust-lang.org/tools/install

Rustup

- https://rustup.rs
- https://github.com/rust-lang/rustup

Cargo

- https://crates.io/
- https://doc.rust-lang.org/cargo/guide/

IDEs, RLS, Editors

- https://www.rust-lang.org/tools
- https://github.com/rust-lang/rls

Clippy

https://github.com/rust-lang/rust-clippy

Rustfmt

https://github.com/rust-lang/rustfmt

Documentation

- Standard library: https://doc.rust-lang.org/std/index.html
- https://docs.rs/

The Book

https://doc.rust-lang.org/book/

The Nomicon

https://doc.rust-lang.org/nomicon/

Making Your First Rust App

Getting started

We are going to build an application in Rust to get a feel for the language and ecosystem. The first step for all new Rust projects is generating a new project. Let's create a new project called `numbers`:

```
cargo new numbers
```

Cargo is the package manager for Rust. It is used as a command line tool to manage dependencies, compile your code, and make packages for distribution. Running `cargo new project_name` by default is equivalent to `cargo new project_name --bin` which generates a *binary* project. Alternatively, we could have run `cargo new project_name --lib` to generate a *library* project.

Binary vs. library

A binary project is one which compiles into an executable file. For binary projects, you can execute `cargo run` at the root of your application to compile and run the executable.

A library project is one which compiles into an artifact which is shareable and can be used as a dependency in other projects. Running `cargo run` in a library project will produce an error as cargo cannot figure out what executable you want it to run (because one does not exist). Instead, you would run `cargo build` to build the library.

There are different formats which the Rust compiler can generate based on your configuration settings depending on how you wish to use your library.

The default is to generate an `rlib` which is a format for use in other Rust projects. This allows your library to have a reduced size for further distribution to other Rust projects while still being able to rely on the standard library and maintain enough information to allow the Rust compiler to type check and link to your code.

Alternative library formats exist for more specialized purposes. For example, the `cdylib` format is useful for when you want to produce a dynamic library which can be linked with C code. This produces a `.so`, `.dylib`, or `.dll` depending on the target architecture you build for.

The generated project

Let's enter the directory for our newly generated Rust project to see what is created:

```
cd numbers
```

The generated structure is:

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

Rust code organization relies primarly on convention which can be overridden via configuration, but for most use cases the conventions are what you want.

main.rs

For a binary project, the entry point is assumed to be located at `src/main.rs`. Furthermore, inside that file, the Rust compiler looks for a function named `main` which will be executed when the binary is run. Cargo has generated a `main.rs` file which contains a simple "Hello, world!" application:

src/main.rs

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

The syntax here says define a function (`fn`) with the name `main` which takes zero arguments and returns the empty tuple `()`.

> Leaving off the return type is equivalent to writing `-> ()` after the argument list of the function. All function calls are expressions which must return a value. The empty tuple `()` is a marker for no value, which is what a function with no return type implicitly returns.

The body of the function calls a macro `println` which prints its argument `"Hello, world!"` to standard out followed by a newline.

> We will cover macros more later, but for now we will mention a few basics. We know it is a macro invocation and not a normal function call because of the trailing `!` in the name. Macros are a powerful form of metaprogramming in Rust which you will use frequently but probably rarely find the occasion to have to write. Rust implements `println` as a macro instead of as a regular function because macros can take a variable number of arguments, while a regular function cannot.

The syntax of Rust is superfically similar to C++ which we can see as curly braces are used for denoting blocks and statements are semicolon terminated. However, there is quite a bit more to the Rust grammar that we will cover as we go along.

Caro.toml

The `Cargo.toml` file is the manifest file for the project which uses the TOML[5] format. This is the entry point for describing your project as well as specifying dependencies and configuration. The initial generated file contains the bare essentials for describing your project:

Cargo.toml

```
[package]
name = "numbers"
version = "0.1.0"
authors = ["Your Name <your.name@example.com>"]
edition = "2018"

[dependencies]
```

The blank section for dependencies is included because nearly every project includes some dependencies. One feature of Rust has been to keep the core language and standard library relatively slim and defer a lot of extra functionality to the community. Therefore relying on third party dependencies is encouraged.

Crates

The primary unit of code organization in Rust is called a crate. Your code exists as a crate which can be distributed to the community via crates.io[6]. Crates in Rust are analogous to gems in Ruby or packages in JavaScript. The registry at crates.io is similar to rubygems.org or npmjs.com as the defacto community repository for distributing and sharing code.

Binary Rust projects are also called crates so they do not solely represent shared library code. Furthermore, a crate can contain both a library and an executable.

It is often difficult to forsee how other's will want to use your software. A common practice in the Rust community is to create dual library/binary crates even when the primary intention of a project is to produce an executable. This can have positive effects on the API design of your code

[5] https://github.com/toml-lang/toml
[6] https://crates.io/

knowing that it should be suitable for external consumption. The binary part of the crate is typically responsible for argument parsing and configuration, and then calls into the functionality exposed by the library part of the crate. Writing all of your code only as an executable and then trying to extract a library after the fact can be a more difficult process. Moreover, the cost of splitting code into a library is minimal.

Making our crate a library

Cargo assumes the entry point for defining a library crate is a file `src/lib.rs`. Let's convert our current binary crate into a binary and library crate. First, we create our library entry point:

src/lib.rs

```
pub fn say_hello() {
    println!("Hello, world!");
}
```

There are two differences to this code from what was in `main.rs`. First, we changed the name of the function from `main` to `say_hello`. This change is more cosmetic than anything (in fact leaving it named `main` works just fine, `main` is only special in some contexts).

The second change is the keyword `pub` before `fn`. This is a privacy identifier which specifies that this function should be publicly accessible to user's of our crate. Without the keyword, we could call this function inside of our `lib.rs` file, but user's of our crate would not be able to call it. Note that our executable sees the library crate the exact same as someone who included our library as a dependency in their `Cargo.toml` file. This ensures a proper separation of concerns between code meant to be executed as a binary and the actual functionality of your project.

We can now change our `main` function to use the functionality exposed by our library:

src/main.rs

```
fn main() {
    numbers::say_hello();
}
```

Running this code should result in the same output as before:

```
$ cargo run
   Compiling numbers v0.1.0 (...)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/numbers`
Hello, world!
```

Let's unpack this function call syntax a little bit before moving on. Even though our binary exists in the same codebase as our library, we still must refer to the functions in the crate by the name of the crate, `numbers` in this case.

We wish to call a function named `say_hello` which exists in the `numbers` crate. The double colon operator `::` is used for separating items in the hierarchy of modules. We will cover modules later, but suffice it to say that crates can contain modules, which themselves can contain more modules.

To resolve an item, be it a type or function, you start with the name of the crate, followed by the module path to get to the item, and finally the name of the item. Each part of this path is separated by `::`. For example, to get a handle to the current thread you can call the function `std::thread::current`. The crate here is `std` which is the standard library. Then there is a module called `thread`. Finally inside the `thread` module there is an exported function called `current`.

Items can exist at the top level of a crate (i.e. not nested in any modules), which you refer to simply by the name of the crate, then `::`, then the name of the item. This is what is happening with `numbers::say_hello` because `say_hello` exists at the top level of our `numbers` crate.

Tradeoffs

Two of the big selling points of Rust are performance and reliability. Performance meaning both runtime speed and memory consumption. Reliability here means catching bugs at compile time and preventing certain classes of errors entirely through language design. These goals are often seen as classically at odds with one another. For example, C lives in a world where performance is of utmost importance, but reliability is left as an exercise for the implementor.

Rust has no garbage collector and no runtime in the traditional sense. However, most difficulties of working with manual memory management are taken care of for you by the compiler. Therefore, you will often hear "zero cost" being used to describe certain features or abstractions in the language and standard library. This is meant to imply that neither peformance nor reliability has to suffer to acheive a particular goal. You write high level code and the compiler turns it into the same thing as the "best" low level implementation.

However, in practice, what Rust really gives you is the tools to make choices about what tradeoffs you want to make. Underyling the design and construction of all software is a series of tradeoffs made knowingly or implicitly. Rust pushes more of these tradeoffs to the surface which can make the initial learning period seem a bit more daunting especially if you have experience in languages where many tradeoffs are implicit.

This will be a topic that permeates this book, but for now we will highlight some of these aspects as we make our `numbers` crate do something more interesting.

Print a list of numbers

Let's build an application that creates a list of numbers and then prints each number on a line by itself to standard out. As a first step, let's just say we want to print the numbers one through five. Therefore, our goal is the following:

```
$ cargo run
1
2
3
4
5
```

Let's change our `main` function to call the yet to be defined library function `print`:

src/main.rs

```
fn main() {
    numbers::print();
}
```

Since we want to print one through five, we can create an array with those numbers and then print them out by looping over that array. Let's create the function `print` in `lib.rs` to do that:

src/lib.rs

```
pub fn print() {
    let numbers = [1, 2, 3, 4, 5];
    for n in numbers.iter() {
        println!("{}", n);
    }
}
```

Let's unpack this from the inside out. We have already seen the `println` macro, but here we are using it with a formatted string for the first time. There are two main features of string interpolation in Rust that will take you through most of what you need. The first argument to one of the printing macros (`print`, `println`, `eprint`, `eprintln`) is a double quoted string which can contain placeholders for variables. The syntax for placeholders to be printed "nicely" is `{}`, and for debugging purposes is `{:?}`. The full syntax for these format strings can be found in the official docs[7].

[7] https://doc.rust-lang.org/std/fmt/

The “nice” format is possible when a type implements the `Display` trait. The debugging format is possible when a type implements the `Debug` trait. Not all types implement `Display`, but the standard practice is for all public types to implement `Debug`. So when in doubt, use `{:?}` to see the value of some variable and it should almost always work.

> We will cover traits in detail later, but we will give a crash course here for what is necessary. Traits are part of the type system to mark certain facts about other types. Commonly they are used to define an interface to a particular set of functions that the type in question implements. You can define your own traits as well as implement traits. Whether a type implements a trait must be stated explicitly in code rather than implicitly by satisfying the functional requirements of the trait. This is one of a few differences between Rust traits and Go interfaces.

We will see when creating types later that usually you can get a `Debug` implementation derived for free, but you must implement `Display` yourself if you want it. Most built-in types implement `Display`, including integers, so we use the format string `"{}"` to say expect one variable. Note that the following does not work:

```
println!(n);
```

The first argument to the print macros must be a literal string, it cannot be a variable, even if that variable points to a literal string. Therefore, to print out a variable you need to use the format string `"{}"` as we are doing. If you forget this the Rust compiler will suggest that as what you probably want to do.

Iteration

So assuming that `n` holds an integer from our collection, we are printing it out using the `println` macro. How does `n` get bound to the values from our collection? We loop over our collection using a `for` loop and bind `n` to each value. The syntax of a `for` loop is:

```
for variable in iterator {
  ...
}
```

Note that we are calling the method `iter` on our array. Rust abstracts the idea of iteration into yet another trait, this one called `Iterator`. We have to call `iter` here to turn an array into an `Iterator` because arrays do not automatically coerce into into an `Iterator`. We shall see shortly that this is not always necessary with other collections.

This is also the first time we are calling a method on an object. Rust types can implement functions that operate on themselves and can therefore be called using this dot syntax. This is syntactic sugar for a direct function call with the receiver object as the first argument. We will cover how these functions are defined when we construct our own types and implement methods on them.

Defining Array Types

We can move out further now to the definition of our array. Rust borrows many ideas of the ML family of languages so some concepts might be familiar if you have experience in that area. By default variables are immutable. Therefore we declare an immutable variable called `numbers` which is bound to an array with the numbers we are interested in. Rust infers the type of `numbers` based on the value we used to initialize the variable. If you want to see the type that is inferred by Rust, a trick is to write:

```
let () = numbers;
```

after the line that declares the variable `numbers`. When you try to compile this code, there will be a type mismatch in the assignment which will print out what the compiler expects:

```
$ cargo run
   Compiling numbers v0.1.0 (...)
error[E0308]: mismatched types
 --> src/lib.rs:3:9
  |
3 |     let () = numbers;
  |         ^^ expected array of 5 elements, found ()
  |
  = note: expected type `[{integer}; 5]`
             found type `()`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
error: Could not compile `numbers`.

To learn more, run the command again with --verbose.
```

We see that the compiler inferred a type of `[{integer}; 5]` for `numbers`. Arrays in Rust are a homogeneous container (all elements have the same type) with a fixed size. This allows it to be stack allocated. The ability to ensure data is stack allocated rather than heap allocated is one of the areas in which Rust allows you to decide what tradeoffs you want to make. On the other hand, because an array has a fixed size that must be known at compile time it is not useful for data which might need to grow or shrink or contain an unknown numbers of items. For this we have the `Vec` type which we will return to shortly.

You can also see that the compiler infers the type of elements of the array to be `{integer}` which is a placeholder as without any further constraints the specific type of integer is unknown. Rust has twelve integer types which depend on size and whether it is signed or unsigned. The default is `i32`

which means a signed integer that takes 32 bits of space. The equivalent unsigned type is `u32`. Let's say we wish our numbers to be `u8`, that is 8-bit unsigned integers. One way to do this is to specify directly on the numerical constant what type we want:

```
let numbers = [1u8, 2, 3, 4, 5];
```

If we do this then the compiler will infer the type `[u8; 5]` for our array. The other way is to explicitly write out the type of the variable `numbers`:

```
let numbers: [u8; 5] = [1, 2, 3, 4, 5];
```

Type annotations are written with a colon after the variable name followed by the type. We see that the size of the array (5) is part of the type. Therefore, even with the same type of elements, say `u8`, an array with four elements is a different type than an array with five elements.

Using std::vec::Vec

Rust provides a few mechanisms for alleviating some of the limitations of arrays. The first we will talk about is the vector type in the standard library, `std::vec::Vec`[8]. A vector is similar to an array in that it stores a single type of element in a contiguous memory block. However, the memory used by a vector is heap allocated and can therefore grow and shrink at runtime. Let's convert our library print function to use a vector:

src/lib.rs

```
1 pub fn print() {
2     let numbers = vec![1, 2, 3, 4, 5];
3     for n in numbers {
4         println!("{}", n);
5     }
6 }
```

We are calling another macro `vec` which this time constructs a vector with the given values. This looks very similar to the array version, but is actually quite different. Vectors own their data elements, they have a length which says how many elements are in the container, and they also have a capacity which could be larger than the length. Changing the capacity can involve quite a bit of work to allocate a new region of memory and move all of the data into that region. Therefore, as you add elements to a vector, the capacity grows by a multiplicative factor to reduce how frequently this process needs to take place. The biggest advantage is that you do not need to know upfront how large the vector needs to be; the length is not part of the type.

[8] https://doc.rust-lang.org/std/vec/struct.Vec.html

The type of a vector is `Vec<T>` where `T` is a generic type that represents the types of the elements. Therefore, `Vec<i32>` and `Vec<u8>` are different types, but a `Vec<u8>` with four elements is the same type as one with five elements.

Note also that we are no longer explicitly calling `iter` on the `numbers` variable in our for loop preamble. The reason for this is that `Vec` implements a trait that tells the compiler how to convert it into an iterator in places where that is necessary like in a for loop. Calling `iter` explicitly would not be an error and would lead to the same running code, but this implicit conversion to an iterator is common in Rust code.

Function Arguments

Let's abstract our print function into two functions. The entry point will still be print (so we don't need to change `main`) which will construct a collection, but it will then use a helper function to actually print the contents of this collection. For now we will go back to using an array for the collection:

src/lib.rs

```
pub fn print() {
    let numbers = [1, 2, 3, 4, 5];
    output_sequence(numbers);
}

fn output_sequence(numbers: [u8; 5]) {
    for n in numbers.iter() {
        println!("{}", n);
    }
}
```

This is our first function that has input or output. Type inference does not operate on function signatures so you must fully specify the types of all inputs and the output. However, we still are not returning anything so by convention we elide the `-> ()` return type which is the one exception to the rule of fully specifying the types in function signatures.

The input type of our function `output_sequence` is our five element array of `u8` values.

Rust has a few different modes of passing arguments to functions. The biggest distinction being that **Rust differentiates between**:

- a function temporarily having access to a variable (borrowing) and
- having *ownership* of a variable.

Another dimension is whether the function can mutate the input.

The default behavior is for a function to take **input by value and hence ownership** of the variable is moved into the function.

The exception to this rule being if the type implements a special trait called `Copy`, in which case the input is copied into the function and therefore the caller still maintains ownership of the variable. If the element type of an array implements the `Copy` trait, then the array type also implements the `Copy` trait.

Suppose we want to use a vector inside `print` instead, so we change the code to:

src/lib.rs

```
pub fn print() {
    let numbers = vec![1, 2, 3, 4, 5];
    output_sequence(numbers);
}

fn output_sequence(numbers: [u8; 5]) {
    for n in numbers.iter() {
        println!("{}", n);
    }
}
```

But this won't work because `[u8; 5]` and `Vec<u8>` are two different types. One possible fix is to change the input type to `Vec<u8>`:

src/lib.rs

```
pub fn print() {
    let numbers = vec![1, 2, 3, 4, 5];
    output_sequence(numbers);
}

fn output_sequence(numbers: Vec<u8>) {
    for n in numbers {
        println!("{}", n);
    }
}
```

This works for this case. It also let's us see what happens when passing a non-Copy type to a function. While arrays implement the `Copy` trait if their elements do, `Vec` does not. Hence, try adding another call to `output_sequence(numbers)` after the first one:

src/lib.rs

```
pub fn print() {
    let numbers = vec![1, 2, 3, 4, 5];
    output_sequence(numbers);
    output_sequence(numbers);
}

fn output_sequence(numbers: Vec<u8>) {
    for n in numbers {
        println!("{}", n);
    }
}
```

This gives us an error:

```
$ cargo run
   Compiling numbers v0.1.0 (...)
error[E0382]: use of moved value: `numbers`
 --> src/lib.rs:4:21
  |
3 |     output_sequence(numbers);
  |                     ------- value moved here
4 |     output_sequence(numbers);
  |                     ^^^^^^^ value used here after move
  |
  = note: move occurs because `numbers` has type `std::vec::Vec<u8>`, which does not\
 implement the `Copy` trait

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: Could not compile `numbers`.

To learn more, run the command again with --verbose.
```

We can see Rust generally has very helpful error messages. The error is that a value was used after it has been moved. The `print` function no longer owns `numbers`. The "note" in the error explains why the move happens due to vector not implementing the `Copy` trait.

Note that in the changes we have made, the body of `output_sequence` has remained the same (modulo whether we call `iter` explicitly or not), only the type signature has been changing. This is a hint that maybe there is a way to write a type signature that works for both arrays and vectors. There are again several ways to accomplish this goal.

A type signature that works for both arrays and vectors

As we have said before, Rust has a lot of power and gives you very fine-grained control over what you want to use or don't want to use. This can be frustrating when starting out because any time you ask "what is the right way to do this," you will invariably be met with the dreaded "it depends." Rather than detail every possible permutation that achieves roughly the same outcome, we are going to focus on the most common idioms. There are certain performance reasons as well as API design decisions that lead to different choices, but those are more exceptional cases than the norm. We will provide pointers to the choices we are making when it matters, but note that due to the scope of the language there is almost always more than one way to do it.

A key type that comes in handy to alleviate some of the limitations of arrays is the `std::slice`[9]. Slices are a dynamically sized view into a sequence. Therefore, **you can have a slice which references an array or a vector and treat them the same**. This is a very common abstraction tool used in Rust. This will be more clear by seeing this in action.

Let's change the signature of `output_sequence` to take a reference to a slice, and change `print` to show that it works with both arrays and vectors:

src/lib.rs

```
pub fn print() {
    let vector_numbers = vec![1, 2, 3, 4, 5];
    output_sequence(&vector_numbers);
    let array_numbers = [1, 2, 3, 4, 5];
    output_sequence(&array_numbers);
}

fn output_sequence(numbers: &[u8]) {
    for n in numbers {
        println!("{}", n);
    }
}
```

A slice of `u8` values has type `[u8]`. This represents a type with an unknown size at compile time. The Rust compilation model does not allow functions to directly take arguments of an unknown size. In order to access this slice of unknown size with something of a known size we use indirection and pass a reference to the slice rather than the slice itself. A reference to a slice of `u8` values has type `&[u8]` which has a known size at compile time. This size is known because it is equal to the size of a pointer plus the length of the slice. Note that slices convert automatically into iterators just like vectors so we again do not call `iter` explicitly in the body of our funciton. This takes care of the signature of `output_sequence` however the way we call this function from `print` has changed as well.

[9] https://doc.rust-lang.org/std/slice/index.html

GET THE FULL BOOK

This is the end of the preview!

Get the full version at:

https://newline.co/fullstack-rust

Featuring:

- 7 lessons
- 240+ pages
- Complete code

GET IT NOW