Rust Quick Guide

Ecosystem • Language • Concepts • Crates • More

By Joel Parker Henderson & ChatGPT

2024-09-09

Contents

What is the Rust Quick Guide? 6
What are the Rust Quick Guide projects? 7
Who is this for? 8
How can I contribute? 9
What makes Rust good? 10
What is the Rust ecosystem? 11
Who might benefit from learning Rust? 12
What are good ways to learn Rust? 13
What are good projects to learn Rust? 14
Is Rust a good first language? 15
What are the hardest parts of Rust? 16
Rust stable versus Rust nightly 17
What is Rust missing? 18
What are some non-goals of Rust? 19
Why do companies not use Rust? 20
The borrow checker - example 21
Channels for thread communication 22
Concurrency and parallelism 23
Error messages 24
Foreign Function Interface (FFI) 25
Futures for asynchronous operations 26
Monomorphisation 27
Resource Acquisition Is Initialization 28
Unsafe code 29
WebAssembly - example 30
Zero-cost abstractions - example 31
Scalar types 32
Compound types 33
Sum types and product types 34
str type versus String type 35
String types for UTF-8, C, OS, and paths 36
Box type for a smart pointer 37

Cow type for clone-on-write 38
RefCell type for dynamic borrowing 39
Rc type for single-thread sharing 40
Arc type for multi-thread sharing 41
Pin type - example 42
Copy trait and Clone trait for duplication 43
Debug trait for debugging and printing 44
Display trait for formatting 45
dyn trait for dynamic dispatch 46
dyn trait in a type position 47
Eq, PartialEq, Ord, PartialOrd traits 48
From and Into traits for conversions 49
Send trait for sending among threads 50
Sync trait for syncing among threads 51
Send & Sync - implementations 52
Sealed traits 53
enum keyword for enumerations 54
struct keyword for custom data types 55
union keyword for multi-type memory 56
match keyword for control flow 57
mod keyword for module namespaces 58
mod keyword for nested hierarchies 59
Nested-or-pattern for matching 60
async/await keywords for futures 61
trait keyword for polymorphism 62
println! macro for printing output 63
assert! macro and friends for testing 64
regex! macro for lazy static optimization 65
catch_unwind! macro to handle panic 66
macro_rules! for declarative macros 67
Annotations for compiler directives 68
Destructuring into components 69
Iterators for traversing collections 70
Closures for anonymous functions 71

Macros for metaprogramming 72
Panic and how to handle it with a hook 73
Pass by value or reference 74
Range syntax for a sequence of values 75
Tuples for ordered collections 76
Memory lifetimes 77
Implicit lifetimes 78
Explict lifetimes 79
Memory on the stack or the heap 80
Memory ownership - example 81
Mutability and immutability 82
No garbage collection 83
Borrow splitting 84
Test framework 85
Test assertions 86
Unit testing 87
Integration testing 88
Documentation testing 89
Documentation testing annotations 90
Source-based code coverage 91
Liskov substitution principle - example 92
rustup command-line tool 93
Cargo package manager and crates 94
cargo-install-favorites 95
Blessed recommendations 96
Clippy linting 97
Helix text editor 98
Rustfmt - examples 99
Rust mdBook for documentation 100
Cross-compiling for multiple platforms 101
Rhai script 102
Abstract syntax tree (AST) 103
Tree-sitter parsing library 104
Language Server Protocol (LSP) 105

Static analysis for error detection 106
Design patterns: introduction 107
Design patterns: adapter 108
Design patterns: builder 109
Design patterns: observer 110
Design patterns: singleton 111
cargo-cache crate for caching builds 112
cargo-crev for community-driven trust 113
cargo-dist crate for distribution archives 114
cargo-release crate - examples 115
cargo-make crate - example 116
Criterion crate - example 117
Rust governance 118
The Rust Foundation 119
The Rust RFC process 120
The Rust roadmap 121
About the author 122
About the ebook PDF 123
About related projects 124

What is the Rust Quick Guide?

Rust Quick Guide provides quick topic explanations to help people learn about the Rust programming language, ecosystem, concepts, crates, and more.

You can try any topic, in any order, at any time, akin to a frequently asked questions resource.

Link: https://github.com/sixarm/rust-quick-guide

Rust Quick Guide work best as an adjunct to a comprehensive Rust book, such as the The Rust Programming Language.

A comprehensive book is valuable to read, cover to cover, for thorough technical explanations.

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

We welcome constructive advice, new topic ideas, pull requests, open issues, and the like. See CONTRIBUTING.md. Rust Quick Guide provides continually evolving, with ongoing additions, corrections, and optimizations.

What are the Rust Quick Guide projects?

https://github.com/sixarm/rust-quick-guide

Rust Quick Guide provides sample projects. These projects are small Rust programs that you can read, build, and run. Each project demonstrates one quick topic summary, or demonstrates one crate. The projects are in the Rust Quick Guide repository, in the `projects` directory.

Some of the projects for topics are:

- from_and_into_traits
- closures_for_iterators
- test_driven_development
- pass_by_value_or_reference
- the_borrow_checker

Some of the projects for crates are:

- assertables/values_strings_sets
- csv/read_a_spreadsheet_file
- reqwest/make_http_requests
- serde/parse_json_data
- sqlx/create_table_insert_into_select

Example command to run a project:

```
cd projects/topics/hello_world
cargo run
```

Many of the projects include a simple integration test:

```
cd projects/topics/hello_world
cargo test
```

Who is this for?

Rust Quick Guide is for anyone who wants to learn about Rust, and who likes to browse quick topic explanations. We're creating it because we are very excited about Rust, and very excited about more people learning Rust.

We're creating these summaries to help teach students and junior-intermediate developers who are generally familiar with introductory programming concepts and programming languages.

Questions that come up frequently with our students and developers:

- Questions about Rust in context, such as how Rust relates to their own interests, and to other programming languages.
- Questions about Rust language concepts, such as memory management, traits, and futures.
- Questions about Rust in practice, such as how to access a database, what tools do, and which libraries to choose.

These summaries are trying to answer these kinds of questions quickly and simply. Browse the topics you want, and skip the rest. And if you have questions, you can let us know, because we're continually adding topics and improving these summaries.

For Teachers

If you're a teacher, we'd love to hear from you about how you're teaching Rust, how your students are learning, and how we can improve this project for your students. Email joel@joelparkerhenderson.com.

For Professionals

If you're a professional, we'd love to learn about how you and your company are working with Rust, and how we can improve this project for your coworkers. Email joel@joelparkerhenderson.com.

How can I contribute?

Rust Quick Guide provides a free community resource, and we welcome help.

Anyone can edit these summaries to improve them, such as to edit an existing page, or add a new page, or refactor how the pages are organized. You can do this a variety of ways.

- You can create a GitHub pull request that provides the changes you want.
- You can open a GitHub issue that provides your suggestions, and we can converse about what you want.
- You can contact the maintainer via email at joel@joelparkerhenderson.com.

Git repository

All the work is available via our GitHub repostory.

Link: https://github.com/sixarm/rust-quick-guide

All the projects are available in the `projects` directory.

Open source license

If you would like to contribute work, then your work will use the project license.

The project license is Creative Commons Attribution Non Commercial Share Alike 4.0 International (CC-BY-NC-SA-4.0).

The license is a free libre open source license. We use the license because it enables people to share improvements with everyone.

What makes Rust good?

There are a variety reasons why Rust is considered a good programming language and good for developers.

- Performance: Rust is designed to provide high performance and low-level control, making it an excellent choice for systems programming. Its memory safety guarantees, achieved through its ownership and borrowing system, allow it to be fast without sacrificing safety.
- Safety: Rust's ownership and borrowing system ensures that programs are safe and free from common programming errors such as null pointer dereferences, buffer overflows, and data races. Rust is designed to prevent undefined behavior and make it difficult to write unsafe code.
- Concurrency: Rust's design is well-suited for concurrent and parallel programming. Its ownership and borrowing system make it easier to write code that is thread-safe and can be run in parallel.
- Community: Rust has a large and active community that is dedicated to improving the language and its ecosystem. The community provides excellent documentation, libraries, and tools that make it easier to learn and use Rust.
- Cross-platform support: Rust can be used to write code for a wide range of platforms, including desktop, mobile, and web applications. Rust also has excellent support for compiling to WebAssembly, which makes it possible to write high-performance web applications in Rust.

Overall, Rust's combination of performance, safety, concurrency, community, and cross-platform support make it an excellent programming language for a wide range of applications.

What is the Rust ecosystem?

The Rust programming language has a growing and vibrant ecosystem that includes a wide range of tools, libraries, and frameworks to support development in various domains. Here are some key components:

- Rust standard library: The Rust standard library provides a set of essential data types and functions that are included in every Rust project.
- Rust tooling: Rust has a growing ecosystem of development tools, including IDEs, code editors, linters, and debuggers.
- Cargo: Cargo is Rust's package manager and build tool. It provides an easy way to manage dependencies, build projects, and publish to the community.
- Rust crates: Rust crates are packages that can be managed by Cargo. The Rust community maintains a large repository of open-source crates, covering a wide range of functionality.
- System programming libraries: Rust is well-suited for system programming, and the language has many libraries to support this, such as low-level C libraries, and ergonomic interface to Unix system calls, and more.
- Web frameworks: Rust has several web development frameworks, including Actix, Rocket, and Warp. These frameworks provide abstractions and tools to build web applications in Rust.
- Embedded development libraries: Rust is increasingly being used for embedded development, and the language has libraries to support this, such as crates for microcontrollers, and libriaries for hardware abstraction layers, and more.

These are just a few examples of the tools and libraries available in the Rust ecosystem. The Rust community is active and collaborative, with ongoing efforts to improve and expand the language's ecosystem.

Who might benefit from learning Rust?

Rust is a versatile language that can be used in a variety of domains, from systems programming to web development. Here are some groups of people who might benefit from learning Rust:

- Systems programmers: Rust's combination of high performance and memory safety make it an excellent choice for systems programming, including operating systems, device drivers, and low-level network programming.
- Web developers: Rust's ability to compile to WebAssembly and its focus on performance make it a good fit for building high-performance web applications.
- Game developers: Rust's low-level control and performance make it a good choice for game development, particularly for real-time games that require fast processing and efficient memory usage.
- Security researchers: Rust's memory safety guarantees make it an excellent choice for writing secure software and for performing security research.
- Embedded systems developers: Rust's low-level control and efficient memory usage make it a good choice for embedded systems development, including robotics, Internet of Things (IoT) devices, and microcontrollers.
- Developers interested in learning new programming paradigms: Rust's ownership and borrowing system and its emphasis on functional programming concepts such as immutability make it an interesting language to learn for those interested in exploring new programming paradigms.

Overall, Rust can be a good fit for a wide range of developers, depending on their interests and needs.

What are good ways to learn Rust?

There are a variety of good ways to learn Rust, depending on your learning style and preferences.

Here are a few suggestions:

- Read the official Rust book: The Rust Programming Language is an excellent resource for learning Rust. It covers all the fundamentals of the language, including ownership, borrowing, lifetimes, and more. The book is well-written, easy to follow, and includes plenty of examples and exercises.
- Work through Rust exercises and tutorials: There are several online resources that provide Rust exercises and tutorials, such as Rustlings, Exercism, and Rust by Example. These resources provide hands-on experience with Rust and help you solidify your understanding of the language.
- Join the Rust community: Rust has a vibrant and welcoming community that can provide valuable support and resources as you learn. Joining the Rust community can help you find answers to your questions, connect with other Rust programmers, and stay up-to-date on the latest developments in the language.
- Contribute to open source projects: Contributing to open source projects is a great way to learn Rust and gain experience with real-world projects. You can start by finding a Rust project that interests you and submitting a pull request to fix a bug or add a feature.
- Build your own Rust projects: Building your own Rust projects is a great way to practice your skills and explore the language's features. Start with a simple project, such as a command-line tool, and gradually work your way up to more complex applications.

Whichever approach you choose, learning Rust takes time and effort. With dedication and persistence, you can become a proficient Rust programmer and take advantage of the language's many benefits.

What are good projects to learn Rust?

Learning Rust can be a rewarding experience, and contributing to open-source projects can be a great way to develop your skills while making meaningful contributions to the community.

- Rustlings is a collection of small exercises designed to help you learn Rust syntax and concepts. It covers topics like ownership, borrowing, and macros, and is a great way to start.
- Rust Game Development Working Group is a community of programmers working on game libraries and tools. This group can be a great way to learn about Rust's capabilties for gaming and graphics programming.
- Servo is a modern, high-performance browser engine written in Rust. It is a complex project that touches on many different aspects of systems programming, including concurrency, memory management, and performance optimization.
- Tokio is a runtime for writing asynchronous Rust applications. It provides abstractions and tools for writing scalable and efficient network applications. Contributing to it can be a good way to learn about concurrency, async/await features, and futures.
- RustCrypto provides cryptographic libraries. It includes implementations of cryptographic algorithms, as well as higher-level libraries for building secure systems. This project can be a great way to learn about Rust memory safety, security, and low-level systems programming.
- The Rust language itself is a great way to learn Rust. Dive into the source code. Learning it and contributing to it can be a great way to learn about the language and its internals.

These are just a few examples of the many open-source Rust projects available for learning and contributing. Whatever your interests, there is likely a Rust project out there that can help you develop your skills.

Is Rust a good first language?

https://www.reddit.com/r/rust/comments/owmxhr - paraphrased

Question: Is rust a good first language for a complete beginner? I have a little programming experience and I want to try low-level stuff.

Opinion 1: your goals

If your goal is to get deep into programming then Rust is a solid choice. It will force you to recognize how things work from low to high level and why they are built the way they are. Unlike other low-level options (like C/C++) Rust also pushes you towards good practices and modern ideas/approaches which is good.

If your goal is to dabble a little then Python is great. It sweeps a lot of inconvenient details under the rug which does make it way easier to use. But the most important part is its wide ecosystem: whatever you do, from web services to games, from machine learning to video processing, there is likely a library that can help you.

Opinion 2: learning curve

On the one hand, Rust forces you to "program well" from the start, or things simply won't work. This steers you away from many mistakes you can make in other langues, especially other low level langues. And if you can avoid getting into lifetimes too much, the base language is very nice to work with compared to most older languages.

On the other hand, Rust forces you to think about things that some other languages handle automatically. Additionally, Rust is compiled and has less immediate ability to give you feedback outside of printing to the screen and warnings/errors. Many tools are also in 3rd party libraries, many which are still WIP, so you'll need to learn more than just the language to do "cooler" stuff.

What are the hardest parts of Rust?

While Rust is a powerful programming language with many benefits, it can also have some challenges. Here are some of the hardest parts:

- Ownership and Borrowing: Rust's ownership and borrowing system is one of its most unique and powerful features, but it can also be one of the most challenging to learn. Understanding ownership, borrowing, and lifetimes can take time and practice, especially for those who are used to garbage-collected or reference-counted languages.
- Error messages: While Rust's error messages are known for being helpful and informative, they can also be overwhelming for new users. Rust's borrow checker is very strict, and its error messages can sometimes be difficult to understand, especially when dealing with complex borrowing situations.
- Macros: Rust's macro system is a powerful tool for metaprogramming, but it can also be challenging to use. Macros require a deep understanding of Rust's syntax and type system, and they can be difficult to debug when something goes wrong.
- Syntax: Rust's syntax can be verbose and sometimes difficult to read, especially for those who are used to more concise or expressive languages. This can make it harder to write clean, readable code, especially for beginners.
- Limited ecosystem: While Rust's ecosystem is growing rapidly, it is still relatively small compared to other languages. This can make it harder to find libraries and tools for certain tasks, and it can also make it harder to find experienced Rust developers to work with.

Rust stable versus Rust nightly

In Rust, there are two main channels of development for the compiler and language: the Rust stable channel and the Rust nightly channel.

The Rust stable channel is the main release channel for Rust, where only stable and well-tested features are included. The goal of this channel is to provide a stable and reliable Rust experience for most users. The stable channel has a predictable release schedule and is recommended for most users.

The Rust nightly channel is a more experimental channel that contains bleeding-edge features that are still under development. The nightly channel is updated more frequently than the stable channel, and it may contain features that are not yet stable or well-tested. The nightly channel is intended for developers who want to experiment with new features, contribute to the Rust project, or provide early feedback on new features.

Some features are only available on the nightly channel, while others are only available on the stable channel. In general, the Rust team works to stabilize features as quickly as possible and move them to the stable channel.

To switch between the stable and nightly channels, you can use the rustup tool.

To switch to the latest stable version of Rust, you can run:

```
rustup default stable
```

To switch to the latest nightly version of Rust, you can run:

```
rustup default nightly
```

Overall, the choice between the stable and nightly channels depends on your needs. If you want a stable and reliable Rust experience, you should use the stable channel. If you want to experiment with new features or contribute to the Rust project, you may want to use the nightly channel.

What is Rust missing?

While Rust is a powerful and versatile language, there are still some areas where it may be lacking in comparison to other languages. Here are a few things that Rust may be missing:

- Mature ecosystem: Rust is still a relatively new language, and as a result, its ecosystem is still developing. Some libraries or tools may not be as fully-featured or mature as those available in more established languages.
- Slower compilation times: Rust's powerful type system and borrow checker can result in longer compilation times compared to other languages. This can be a drawback for developers who require faster feedback cycles.
- Limited support for garbage collection: Rust does not have built-in support for garbage collection, which can make it more difficult to manage memory in some cases. While Rust's ownership and borrowing system provides safety guarantees and avoids issues such as memory leaks, it can also require more careful management of memory allocation and deallocation.
- Learning curve: Rust has a steep learning curve, especially for developers who are not familiar with low-level systems programming or functional programming concepts. This can make it challenging for developers to become proficient in the language quickly.
- Limited support for some platforms: While Rust has good support for Linux and other popular platforms, support for some niche platforms or hardware may be limited. This can be a concern for developers working on specialized projects that require support for these platforms.

It's worth noting that many of these limitations are being actively addressed by the Rust community, and the language continues to evolve and improve over time.

What are some non-goals of Rust?

These non-goals are listed in a previous official Rust FAQ.

1. We do not employ any particularly cutting-edge technologies. Old, established techniques are better.
2. We do not prize expressiveness, minimalism or elegance above other goals. These are desirable but subordinate goals.
3. We do not intend to cover the complete feature-set of C++, or any other language. Rust should provide majority-case features.
4. We do not intend to be 100% static, 100% safe, 100% reflective, or too dogmatic in any other sense. Trade-offs exist.
5. We do not demand that Rust run on “every possible platform”. It must eventually work without unnecessary compromises on widely-used hardware and software platforms.

Why do companies not use Rust?

Rust has gained a lot of popularity and adoption in recent years, but some companies are hesitant to adopt the language. Here are some potential reasons why companies may avoid Rust:

- Lack of expertise: Rust is a relatively new language and may not yet have a large pool of experienced developers compared to more established languages. Companies may be hesitant to adopt Rust if they do not have the in-house expertise or resources to adopt Rust.
- Risk aversion: Some companies may be risk-averse and may prefer to stick with more established languages that have a proven track record of success. Rust still a relatively new, and may be perceived as less stable or less reliable compared to more established languages.
- Learning curve: Rust's syntax and concepts can be challenging for developers who are unfamiliar with systems programming or functional programming. Companies may be hesitant to adopt Rust if they anticipate a steep developer learning curve.
- Limited ecosystem: While Rust's ecosystem is growing rapidly, it may not yet have the same level of library support or tooling as more established languages. This can make it more difficult or time-consuming for companies to develop and maintain Rust code.
- Legacy code: Many companies have existing codebases written in other languages, and transitioning to Rust may require significant time and resources. Companies may be hesitant to make this investment if the benefits of transitioning are not clear.

Many of these concerns are actively being addressed by the Rust community, with ongoing efforts to improve the language's ecosystem and make it more accessible to developers of all backgrounds.

The borrow checker - example

The borrow checker guarantees that an immutable borrow never changes data. This guarantee enables you to have multiple immutable borrows of the same object simultaneously.

```
let mut a = ['a', 'b', 'c'];

let b = &a; // Borrow data immutably
//b[0] = 'x'; // Changing data won't compile
println!("{:?}", b[0]); // Reading data is fine

let c = &mut a; // Borrow data mutably
c[0] = 'x'; // Changing data is fine.
println!("{:?}", c[0]); // Reading data is fine.
```

The borrow checker guarantees that mutable borrows of the same object never overlap. This guarantee protects you from accidentally doing conflicting mutations in mutable borrows.

```
let mut a = ['a', 'b', 'c'];

// Valid code because the mutable borrows are one at a time.
let b = &mut a;
b[0] = 'x';
let c = &mut a;
c[0] = 'y';

// Invalid code because the mutable borrows overlap.
//let b = &mut a;
//let c = &mut a;
//b[0] = 'x';
//c[0] = 'y';
```

Channels for thread communication

Rust channels are a way to facilitate communication between threads in Rust. They allow threads to send messages to each other in a synchronized and safe manner, without the need for explicit locking or other synchronization primitives.

In Rust, channels are created using the `std::sync::mpsc` module, which stands for "multiple producer, single consumer." This means that multiple threads can send messages into a channel, but there will only be one thread receiving those messages.

To create a channel, you first need to import the module, then you can send messages and receive messages.

```
use std::sync::mpsc;
fn main() {
    let (sender, receiver) = mpsc::channel(); // create channel
    sender.send("Hello, World!").unwrap(); // send message
    let message = receiver.recv().unwrap(); // receive message
}
```

If there are no messages in the channel, the `recv` method will block until a message is available. Alternatively, you can use the `try_recv` method to receive a message without blocking:

```
match receiver.try_recv() {
    Ok(message) => println!("Received message: {}", message),
    Err(_) => println!("No message received"),
}
```

It's important to note that sending and receiving messages through a Rust channel takes ownership of the values being sent. This means that the value being sent is moved into the channel, and can no longer be used by the sender after the send operation. Similarly, the value received from the channel is moved out of the channel, and can no longer be received by any other threads. This ownership model ensures that Rust channels are safe and thread-safe.

Concurrency and parallelism

In Rust, concurrency refers to the ability of a program to perform multiple tasks or operations at the same time, while parallelism refers to the ability of a program to perform multiple tasks or operations simultaneously, using multiple processors or cores.

Rust provides mechanisms for concurrency and parallelism:

- Threads: Rust's standard library provides a low-level interface for creating and managing threads. Threads allow a program to execute multiple tasks in parallel, but require careful synchronization to avoid data races and other concurrency issues.
- Channels: Rust's channels provide a high-level mechanism for communication between threads. Channels allow multiple threads to send and receive data, and ensure that the data is transmitted in a synchronized and safe manner.
- Futures: Rust's futures provide a mechanism for asynchronous programming, allowing a program to perform non-blocking I/O and other operations without blocking the main thread. Futures are composable and can be combined to create complex asynchronous workflows.
- Atomic types: Rust's atomic types provide a safe and efficient way to share data between threads. Atomic types are designed to be thread-safe, and provide operations that ensure that data is updated atomically, without the need for locks or other synchronization mechanisms.

Rust's concurrency and parallelism mechanisms are designed to be safe and efficient, and take advantage of Rust's ownership and borrowing system to prevent data races and other concurrency issues. Additionally, Rust's compiler provides powerful static analysis and optimization tools that can help identify and eliminate potential issues in concurrent and parallel code.

Error messages

Rust is known for having particularly helpful and informative error messages compared to other programming languages. Rust's error messages are designed to be both human-readable and actionable, providing developers with clear guidance on how to fix issues in their code.

Here are some key features of Rust error messages:

- Contextual information: Rust's error messages typically include contextual information such as the location and type of the error, as well as relevant code snippets and variable values.
- Suggested fixes: In many cases, Rust will provide suggested fixes for common errors, such as missing semicolons or incorrect variable types. These suggestions can save developers time and make it easier to correct errors.
- Explanation of the problem: Rust's error messages often include detailed explanations of the problem, helping developers to understand the underlying issue and how to avoid it in the future.
- Help with complex concepts: Rust's error messages can also help with complex concepts like ownership and borrowing. The messages will often explain how Rust's ownership system works and suggest ways to restructure code to avoid common pitfalls.
- Clear formatting: Rust's error messages are designed to be easy to read and understand, with clear formatting and helpful color coding.

Overall, Rust's error messages are a powerful tool for developers, helping them to identify and fix issues in their code quickly and efficiently. They are a testament to Rust's focus on developer experience and the language's commitment to making it easy to write safe and efficient code.

Foreign Function Interface (FFI)

In Rust, the Foreign Function Interface (FFI) allows Rust code to interoperate with code written in other languages, such as C or C++. This enables Rust to be used in mixed-language projects or to use existing libraries that are written in other languages.

To use the FFI in Rust, you first need to declare an external function or type from another language using the `extern` keyword:

```
extern "C" {
    fn some_function(arg1: i32, arg2: *mut i32) -> i32;
}
```

This declares a function called `some_function` that takes an `i32` and a pointer to an `i32` as arguments and returns an `i32`. The "C" string in the `extern` declaration specifies the calling convention, which tells the Rust compiler how to interact with the external function.

To call this function from Rust, you can use the `unsafe` keyword to tell the Rust compiler that the function call is unsafe and may have side effects:

```
let arg1 = 42;
let mut arg2 = 0;
let result = unsafe { some_function(arg1, &mut arg2) };
```

This calls the `some_function` function with the specified arguments, passing a mutable reference to `arg2` using the `&mut` operator.

Rust also provides a `#[no_mangle]` attribute that can be used to disable Rust's name mangling, which can be useful when interacting with external libraries. For example, you can declare a Rust function with the `#[no_mangle]` attribute and call it from C code.

In summary, the Rust FFI enables Rust code to interoperate with code written in other languages, and can be used to call external functions from Rust or to expose Rust functions to other languages.

Futures for asynchronous operations

In Rust, a future is a type that represents an asynchronous operation that may not have completed yet. Futures are for writing non-blocking code, such as to read a file, make a web request, or query a database.

Rust's futures are composable, which means that multiple futures can be combined to create more complex workflows. Futures can be chained together to form a pipeline, with each future as a step in the pipeline. When a future completes, it can trigger the next future to execute.

Futures are executed by an executor, which is responsible for scheduling and running the futures. Rust provides several built-in executors.

Example of a Rust future for an asynchronous HTTP request:

```
use futures::Future;
use reqwest::Url;

async fn fetch_url(url: Url) -> Result<String, reqwest::Error> {
    let response = reqwest::get(url).await?;
    let text = response.text().await?;
    Ok(text)
}

fn main() {
    let url = Url::parse("https://example.com").unwrap();
    let future = fetch_url(url);
    let runtime = tokio::runtime::Runtime::new().unwrap();
    let text = runtime.block_on(future).unwrap();
    println!("response text is {}", text)
}
```

This example defines an asynchronous function `fetch_url`. The function accepts a URL, then uses the `reqwest` crate to make an HTTP GET request to the URL, then returns the response text as a `String`.

The `fetch_url` function is async, so returns a `Future` that we store in a variable. We use the tokio runtime to run the `Future`. This blocks until it completes. Finally, we print the result.

Monomorphisation

Rust monomorphization is a process where generic code is transformed into specific code for each concrete type used in the program. In other words, it is the process of generating specialized code for each type that is used in a generic function or struct.

This is different from traditional dynamic dispatch, where a function or method call is resolved at runtime, based on the type of the object or value being operated on. With monomorphization, the specific implementation of a generic function is determined at compile-time, and there is no runtime overhead associated with dynamic dispatch. Monomorphization makes Rust code faster and more efficient than code that relies on dynamic dispatch.

Here's an example of monomorphism in Rust:

```
fn add<T: std::ops::Add<Output=T>>(a: T, b: T) -> T {
    a + b
}

fn main() {
    let int_sum = add(1, 2);
    let float_sum = add(1.0, 2.0);
    println!("Integer sum: {}", int_sum);
    println!("Float sum: {}", float_sum);
}
```

In this example, the add function takes two arguments of type `T`, which must implement the `std::ops::Add trait`, and returns their sum of the same type `T`. Because the type parameter `T` is constrained to implement `std::ops::Add`, the compiler can statically determine the concrete type of `T` at compile-time, resulting in monomorphic code that is optimized for the specific types used.

In the `main` function, we call add twice: once with integers and once with floats. Since Rust uses monomorphization, the compiler generates two separate versions of the add function, one for integers and one for floats. This results in efficient optimized code.

Resource Acquisition Is Initialization

Resource Acquisition Is Initialization (RAII) is a fundamental concept in many programming languages, and helps memory safety.

RAII is a way of managing resources such as memory, files, connections, etc. The core idea: when you acquire a resource, then you initialize an object that represents that resource; when that object is no longer needed, then its destructor is called, which releases the resource.

In Rust, RAII is implemented through the use of ownership and the `Drop` trait. Whenever an object is created in Rust, it is associated with an owner that is responsible for managing its memory and resources. When the owner goes out of scope, Rust automatically calls the `Drop` trait implementation for that object, which allows the object to clean up any resources it may have acquired.

Example of RAII with the standard library File type:

```
use std::fs::File;

fn main() -> std::io::Result<()> {
    let file = File::create("example.txt")?;
    // Do some work with the file variable ...
    Ok(())
}
```

In this example, we create a new `File` object using the `File::create()` method, which opens a new file for writing. When the file variable goes out of scope at the end of the function, Rust automatically calls the file's destructor, which closes the file handle and frees the file's resources.

RAII for managing resources and it helps ensure that your programs are both safe and reliable. By relying on RAII and the ownership system, Rust programs can avoid many common problems such as resource leaks, null pointer dereferences, and other forms of undefined behavior.

Unsafe code

Rust is a programming language that prioritizes safety and correctness. However, there are situations where you may need to bypass Rust's built-in safety checks to perform certain operations. In these cases, Rust provides a way to write unsafe code within a safe Rust program.

Unsafe code is Rust code that the compiler cannot verify for safety at compile-time. This code is typically used when working with low-level operations that require direct access to system resources or when interacting with code written in other programming languages.

In unsafe code, Rust allows the use of several features that are not permitted in safe code, including:

- Dereferencing raw pointers: Raw pointers are unmanaged pointers that do not have any safety guarantees. Dereferencing raw pointers can lead to undefined behavior, such as null pointer dereferences, use-after-free errors, and other memory-related bugs.
- Calling unsafe functions: Unsafe functions are Rust functions that are marked with the unsafe keyword. These functions can perform operations that are not safe to perform in safe Rust code, such as accessing memory directly or performing system-level operations.
- Modifying global state: Rust's ownership and borrowing system ensures that data is accessed safely. However, unsafe code can bypass these guarantees, and modify global state directly, which can lead to race conditions and other bugs.

Code marked as unsafe doesn't mean it's inherently dangerous or incorrect. Unsafe code is often necessary for performance-critical code, interfacing with external systems, or implementing low-level abstractions. However, writing and working with unsafe code requires a deep understanding of Rust's memory and ownership model. Rust also provides several tools, such as unsafe blocks, to help ensure that unsafe code is written and used correctly.

WebAssembly - example

Create a new Rust project, such as running:

```
cargo new wasm-example --lib
```

Add the `wasm-bindgen` dependency to your `Cargo.toml` file.

In your `lib.rs` file, add the `wasm_bindgen` macro to the top of the file, and define a simple Rust function that takes two numbers and returns their sum:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Build your Rust code as a WebAssembly module by running the following command, which creates a WASM file called `wasm-example.wasm` in the `target/wasm32-unknown-unknown/release/` directory:

```
cargo +nightly build --target wasm32-unknown-unknown --release
```

Finally, create a JavaScript file that loads the WASM module and calls your Rust function:

```
import("./wasm_example_bg.wasm").then((module) => {
  const { add } = module;
  console.log(add(1, 2)); // outputs 3
});
```

This JavaScript code loads the WASM module using the `import()` function, which is a new feature in JavaScript that allows you to dynamically load modules at runtime. Once the module is loaded, you can call your Rust function using the `add` variable.

Zero-cost abstractions - example

Here's an example of zero-cost abstrations:

```
fn add<T: std::ops::Add<Output=T>>(x: T, y: T) -> T {
    x + y
}

fn main() {
    let x = 1;
    let y = 2;
    let z = add(x, y);
    println!("{}", z);
}
```

In this example, the `add` function takes two arguments of any type that implements the `Add` trait, adds them together using `+`, and returns the result.

The `add` function is generic, so it can be used with any type that implements `Add`, such as numbers, strings, or even custom objects.

Because the function is generic, it will be optimized by the Rust compiler to perform as efficiently as possible. This means that using the `add` function will not incur any additional runtime overhead, even though it uses an abstraction (the `Add` trait) to make the function more generic and reusable.

In this way, Rust demonstrates the concept of zero-cost abstraction, allowing developers to write modular, reusable code without sacrificing performance.

Scalar types

Rust has several scalar types that represent basic values and data structures. These types are built into the language and do not require any additional dependencies or libraries to use.

Boolean (bool): Represents a logical value, either true or false.

```
let a: bool = true;
```

Signed integers (i8, i16, i32, i64, i128): Represent whole numbers that can be positive or negative. The number after the 'i' represents the number of bits the integer type uses.

```
let a: i8 = 1;
let b: i16 = 1;
let c: i32 = 1;
let d: i64 = 1;
let e: i128 = 1;
```

Unsigned integers (u8, u16, u32, u64, u128): Represent whole numbers that can only be positive. The number after the 'u' represents the number of bits the integer type uses.

```
let a: u8 = 1;
let b: u16 = 1;
let c: u32 = 1;
let d: u64 = 1;
let e: u128 = 1;
```

Floating-point numbers (f32, f64): Represent decimal numbers with single or double precision.

```
let a: f32 = 1.0;
let b: f64 = 1.0;
```

Character (char): Represents a single Unicode character.

```
let a: char = 'a';
```

Compound types

In Rust, a compound type is a type that is composed of other types. There are two main compound types in Rust: tuples and arrays.

Tuples: A tuple is an ordered list of elements of different types. Tuples in Rust are declared using parentheses and the elements are separated by commas. For example, the following code creates a tuple containing a string and an integer:

```
let my_tuple = ("Hello, World!", 42);
```

We can access the individual elements of a tuple using indexing syntax:

```
let my_tuple = ("Hello, World!", 42);
let my_string = my_tuple.0;
let my_int = my_tuple.1;
```

Arrays: An array is a fixed-size collection of elements of the same type. Arrays in Rust are declared using square brackets and the elements are separated by commas. For example, the following code creates an array of integers with five elements:

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

We can access the individual elements of an array using indexing syntax:

```
let my_array = [1, 2, 3, 4, 5];
let my_element = my_array[2]; // Access the third element
```

Arrays in Rust have a fixed size, which means that they cannot be resized at runtime. However, Rust provides a more flexible compound type called a vector, which can be resized dynamically.

Compound types are useful for grouping related data together and passing them around as a single unit. They also allow for more complex data structures and algorithms to be created. By using tuples and arrays effectively, Rust developers can write more efficient and maintainable code.

Sum types and product types

In Rust, sum types and product types are two fundamental concepts in algebraic data types, which are used to define custom data structures.

Sum type

A sum type is a type that can have one of several possible values. In Rust, sum types are defined using the `enum` keyword. An enum can have one or more variants, each of which can contain different types of data.

Example sum type:

```
enum Color {
    Red,
    Green,
    Blue,
}
```

In this example, `Color` is a sum type that combines three variants into a single type.

Product type

A product type is a type that combines two or more types into a single type. In Rust, product types are defined using the tuple syntax, which looks like (T1, T2, ..., Tn). The resulting type can be thought of as a record that contains values of each of the individual types.

Example product type:

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

In this example, `Point` is a product type that combines three i32 values into a single type.

str type versus String type

In Rust, both `str` and `String` are used to represent textual data, always using Unicode UTF-8 encoded u8 bytes. But they have some differences in terms of how they are stored and accessed.

- Type: `str` is a primitive type. `String` is a standard library type.
- Memory: `str` is immutable, and can be stored in the program's binary, or stack, or heap. `String` is mutable, growable, and heap-allocated.
- Ownership: `&str` is a slice that borrows ownership from another string or a static string literal. In contrast, `String` owns the string data it contains.
- Lifetime: `str` has a static lifetime (i.e., it lives as long as the program runs) in case of string literals, or a borrowed lifetime (i.e., it lives as long as the reference it was borrowed from) in case of borrowed slices. `String` has a dynamic lifetime (i.e., it lives as long as there is a reference to it).
- Usage: `&str` is usually used for function arguments and return types, or for string literals, while `String` is typically used when you need to create or modify a string at runtime.
- Coercion: a `&String` can be coerced to a `&str`, such as when a `&String` arg is passed to a function signature with a `&str` parameter.
- Indexing: Indexing by bytes is different than counting by characters, because `str` and `String` both store Unicode characters, using the UTF-8 variable-width encoding format, which means one character can take up more than one byte.

String types for UTF-8, C, OS, and paths

Rust provides different string types for different kinds of text.

`str` and `String`: a Unicode UTF-8 value sequence. A `str` is slice-like and immutable. A `String` is owned, mutable, and growable.

`CStr` and `CString`: a C-style null-terminated char byte sequence. A `CStr` is slice-like and immutable. A `CString` is owned, mutable, and growable.

`OsStr` and `OsString`: a platform-specific operating system string. A `OsStr` is slice-like and immutable. A `OsString` is owned, mutable, and growable.

`Path` and `PathBuf`: a platform-specific file path string. A `Path` is slice-like and immutable. A `PathBuf` is owned, mutable, and growable.

Converting between a platform-specific type (`OsStr`, `OsString`, `Path`, `PathBuf`) and a platform-independent type (`str`, `String`) may require lossy conversion, or handling conversion errors.

Examples of string types:

```
use std::ffi::{CStr, CString};
use std::ffi::{OsStr, OsString};
use std::path::{Path, PathBuf};

let a: &str = "foo";
let b: String = String::from("foo");

let c: &CStr =  CStr::from_bytes_with_nul(b"foo\0")
    .expect("Error in CStr::from_bytes_with_nul");
let d: CString = CString::from_vec_with_nul(b"foo\0".to_vec())
    .expect("Error in CString::from_vec_with_nul");

let e: &OsStr = OsStr::new("foo");
let f: OsString = OsString::from("foo");

let g: &Path = Path::new("foo");
let h: PathBuf = PathBuf::from("foo");
```

Box type for a smart pointer

In Rust, a `Box` is a smart pointer that provides a way to allocate memory on the heap and move data into that memory.

`Box` will allocate an object at runtime rather than at compile time. When a value is wrapped in a `Box`, it is moved to the heap and the `Box` itself is stored on the stack. This allows you to allocate a large object on the heap without having to worry about stack size limitations. When a `Box` goes out of scope, the memory it allocated is automatically deallocated. This eliminates the need to manually manage memory and helps prevent common memory-related bugs such as memory leaks and dangling pointers.

Another benefit of `Box` is that it enables ownership transfer. When you move a value into a `Box`, you transfer ownership of the value to the `Box`. This means that the `Box` becomes the owner of the value and is responsible for cleaning it up when it goes out of scope. This can be useful when you need to transfer ownership of a value between different parts of your program.

Usage

To use `Box`, you can create a new instance by calling the `Box::new` function and passing in the value you want to allocate on the heap. For example, to allocate a new `i32` value on the heap and store it in a `Box`:

```
let my_box = Box::new(42);
```

This creates a new `Box` that contains the value 42. When `my_box` goes out of scope, the memory it allocated will be automatically deallocated.

Overall, `Box` is a useful tool for allocating objects on the heap, transfering ownership between parts of your program, and using automatic deallocation to help prevent memory-related bugs.

Cow type for clone-on-write

https://doc.rust-lang.org/std/borrow/enum.Cow.html

A Rust `Cow` type is a clone-on-write smart pointer. When a function receives a `Cow` type as an argument, the function can modify the data without actually modifying the original data structure. Instead, the `Cow` type makes a clone of the data when it is modified, and any other references to the original data continue to point to the original data.

The `Cow` type is implemented as an enum with two variants: `Borrowed` and `Owned`, which express "either a reference, or an owned type". You choose which variant you want depending on your goal.

```
use std::borrow::Cow;

fn main() {
    let a = ['a', 'b', 'c'];
    let mut b = Cow::Borrowed(&a);

    // The `b` Cow enum is borrowed, and points to `a`.
    match b {
        Cow::Borrowed(_) => println!("Borrowed"),
        Cow::Owned(_) => println!("Owned"),
    }

    // Convert `b` to mutable i.e. clone it, then change it.
    b.to_mut()[0] = 'x';

    // Now the `b` Cow enum is Owned i.e. has its own data.
    match b {
        Cow::Borrowed(_) => println!("Borrowed"),
        Cow::Owned(_) => println!("Owned"),
    }
}
```

A typical use case for `Cow` is optimization by not doing copies. For example, you write a function that returns a String, but there are cases when you already have a &'static str containing the data; you can return `Cow::Borrowed` so you don't need to allocate and copy a new String.

RefCell type for dynamic borrowing

The Rust `RefCell` type is a container type that provides dynamic borrow checking at runtime, allowing for mutable or immutable borrows of its inner value based on certain rules. For example, there are cases where runtime borrow checking is necessary, such as when a value needs to be mutated within a shared reference.

- Mutable borrows: `RefCell` provides mutable borrows of an inner value through the use of its `borrow_mut` method. This method returns a mutable reference (`&mut`) to the inner value, which can be modified. However, `borrow_mut` will panic at runtime if there are any outstanding references (mutable or immutable) to the inner value.
- Immutable borrows: `RefCell` provides immutable borrows through its `borrow` method, which returns an immutable reference (`&`) to the inner value. Multiple immutable references can be outstanding at the same time, but attempting to call borrow_mut while there are outstanding immutable references will cause a panic.

Example:

```
use std::cell::RefCell;

fn main() {
    let x = RefCell::new(1);

    // Borrow a mutable reference to x's inner value
    let mut mutable_ref = x.borrow_mut();
    *mutable_ref = 2;

    // Borrow an immutable reference to x's inner value
    let immutable_ref = x.borrow();
    println!("The value of x is: {}", *immutable_ref);
}
```

Rc type for single-thread sharing

In Rust, `Rc` (Reference Counted) is a smart pointer that provides shared ownership of a value. `Rc` tracks the number of references to a value. If a new reference is created, `Rc` increments the reference count. If an existing reference is dropped, `Rc` decrements the reference count. When the reference count reaches zero, `Rc` drops the value.

Unlike `Arc` smart pointer, `Rc` cannot be safely shared between threads and is used for single-threaded scenarios.

For example, consider the following code:

```
use std::rc::Rc;

fn main() {
    let shared_data = Rc::new(vec![1, 2, 3]);
    let data1 = shared_data.clone();
    let data2 = shared_data.clone();

    println!("{:?}", shared_data);
    println!("{:?}", data1);
    println!("{:?}", data2);
}
```

Here, an `Rc` shares ownership of a vector between multiple references. The `Rc::new()` function creates a new `Rc` that points to a vector of [1, 2, 3]. The `clone()` method creates two new `Rc`s that point to the same vector, and the reference count is incremented. The `println!()` macro prints the values of each reference to the console.

`Rc` is a useful tool for scenarios where shared ownership of a value is needed in a single-threaded environment. By using reference counting to manage the lifetime of the value, `Rc` ensures that the value is not dropped until all references to it have been dropped.

Arc type for multi-thread sharing

In Rust, `Arc` (Atomically Reference Counted) is a smart pointer that provides shared ownership of a value, similar to `Rc` (Reference Counted) smart pointer. The difference is that `Arc` can be safely shared between threads, for concurrent programming; this is because `Arc` uses atomic operations to increment and decrement the reference count.

`Arc` works by keeping track of the number of references to a value. When a new reference to the value is created, `Arc` increments the reference count. When an existing reference is dropped, `Arc` decrements the reference count. When the reference count reaches zero, `Arc` drops the value. When an `Arc` is cloned, a new pointer to the same value is created, and the reference count is incremented.

```
use std::sync::Arc;
use std::thread;

fn main() {
    let shared_data = Arc::new(vec![1, 2, 3]);
    for i in 0..3 {
        let data = shared_data.clone();
        thread::spawn(move || {
            let vec = data.iter()
            .map(|x| x + i).collect::<Vec<_>>();
            println!("{:?}", vec);
        });
    }
}
```

Here, an `Arc` shares ownership of a vector between multiple threads. The `Arc::new()` function creates a new `Arc` that points to a vector of `[1, 2, 3]`. The `clone()` method creates a new `Arc` that points to the same vector, and the reference count is incremented. The `thread::spawn()` function creates three threads, each of which iterates over the vector and adds the current loop index to each element. The results are collected into a new vector, which is printed to the console.

Pin type - example

Here's an example of how to use the Rust Pin type:

```
use std::pin::Pin;

struct Data {
    value: i32,
}

impl Data {
    fn new(value: i32) -> Self {
        Self { value }
    }
}

fn main() {
    let data = Data::new(1);
    let pinned_data = Pin::new(&data);

    // Invalid move of `data`:
    // let moved_data = data;

    // Invalid move of `pinned_data`:
    // let moved_pinned_data = pinned_data;

    // We can access the value of `data` through `pinned_data`
    println!("{}", pinned_data.value);
}
```

In this example, we define a Data struct that holds a single integer value. We then create a new instance of this struct, and use the `Pin::new` function to create a Pin wrapper around a reference to this instance.

Once `pinned_data` is created, trying to move data results in a compile-time error. Similarly, attempting to move `pinned_data` results in a compile-time error, because it is a wrapper around a pinned reference.

We can still access the value of data through `pinned_data`, as shown by the `assert_eq!` statement. The reference remains valid, even if the data structure itself is moved.

Copy trait and Clone trait for duplication

In Rust, the Copy trait controls how values are copied, while the Clone trait controls how values are cloned.

The `Copy` trait is used for types that can be safely copied bit-by-bit, without any special consideration for ownership or memory management. When a value with the `Copy` trait is assigned to a new variable or passed to a function, a bitwise copy of the original value is made. This means that the original value remains unchanged, and any changes made to the copied value do not affect the original.

Examples of types that implement the `Copy` trait include simple scalar types like integers and booleans, as well as tuples and arrays that only contain types that implement the `Copy` trait.

The `Clone trait`, on the other hand, is used for types that need to be explicitly cloned in order to make a copy. When a value with the `Clone` trait is cloned, a new instance of the value is created, and any owned data is also cloned. This means that changes made to the cloned value do not affect the original, and vice versa.

To implement the `Clone` trait for a type, you need to provide an implementation of the `clone` method, which creates a new instance of the type and clones any owned data. Rust also provides a default implementation of `Clone` for types that implement the `Copy` trait, which simply returns a bitwise copy of the value.

```
#[derive(Copy, Clone)]
struct Point { x: i32, y: i32 }

fn main() {
    let a = Point { x: 10, y: 20 };
    let b = a; // This does a  copy
    let c = a.clone(); This does a clone
}
```

Debug trait for debugging and printing

In Rust, the `Debug` trait is a built-in trait that allows developers to print and debug Rust types. It provides a basic representation of a type suitable for debugging purposes.

When a type implements the `Debug` trait, it can be printed using the println! macro with the `{:?}` format specifier. This will print a debug representation of the type, which is often more informative than the default string representation.

To implement the `Debug` trait for a custom type, developers need to define a `debug` method on the type that returns a `fmt::Debug` trait object. This method should return a formatter that describes the structure of the type in a way that is suitable for debugging.

For example, let's consider a simple `Point` struct:

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

We use the `derive` attribute to automatically generate an implementation of the `Debug` trait for our `Point` struct. This creates a `debug` method that returns a formatter that prints the `x` and `y` fields.

With this implementation, we can use the `println!` macro to print a `Point` value like this:

```
let p = Point { x: 10, y: 20 };
println!("Point: {:?}", p);
```

This will output:

```
Point: Point { x: 10, y: 20 }
```

Display trait for formatting

The Rust `Display` trait is a built-in trait that allows developers to format a value as a string for display purposes. It provides a human-readable representation of a type.

When a type implements the `Display` trait, it can be formatted as a string using the `format!` macro or the `println!` macro with the `{}` format specifier.

To implement the `Display` trait for a custom type, we define a `fmt` method on the type that takes a formatter object. The formatter object implements the `fmt::Write` trait, which provides methods for writing to a string buffer.

Example:

```
use std::fmt;

struct Point {
    x: i32,
    y: i32,
}

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        // The `write!` macro writes fields into the formatter
        write!(f, "x is {} and y is {}", self.x, self.y)
    }
}

fn main() {
    let p = Point { x: 1, y: 2 };
    println!("Point {}", p); // "Point x is 1 and y is 2"
}
```

Here, we define a `fmt` method for the `Display` trait on our `Point` struct. This method takes a formatter object, and writes the `x` field and `y` field of the struct into a user-friendly string.

dyn trait for dynamic dispatch

In Rust, a `dyn trait` is a way to specify a trait object with dynamic dispatch.

A `trait` object is a pointer to an object that implements a trait, and is used when the concrete type of an object is not known at compile time. In other words, it allows you to write code that can work with different types that implement a particular trait without knowing the exact type at compile time.

When defining a `trait` object in Rust, you can use the `dyn` keyword to indicate that the trait object should be dynamically dispatched. This means that the specific implementation of the trait for a given object will be determined at runtime rather than at compile time.

For example, consider the following trait definition:

```
trait MyTrait {
    fn my_method(&self);
}
```

To define a trait object with dynamic dispatch, use the dyn keyword:

```
fn my_function(obj: &dyn MyTrait) {
    obj.my_method();
}
```

In this example, `my_function` takes a reference to a trait object that implements the `MyTrait` trait, with dynamic dispatch specified using the `dyn` keyword. This means that at runtime, the specific implementation of `my_method` for the given object will be determined dynamically.

Using `dyn trait` allows Rust to provide runtime polymorphism, which is useful in situations where the concrete type of an object is not known at compile time, but needs to be determined at runtime. However, it can come at a performance cost compared to static dispatch, which is resolved at compile time.

dyn trait in a type position

Runnable project

You can use `&dyn` with a trait name in a type position. This is useful to abstract over a variety of implementations.

Example:

```
trait Speak {
    fn speak(&self);
}

type Cat;

impl Speak for Cat {
    fn speak(&self) {
        println!("meow");
    }
}

type Dog;

impl Speak for Dog {
    fn speak(&self) {
        println!("woof");
    }
}

fn main() {
    let pets: Vec<&dyn Speak> = vec![&Cat, &Dog];
    for pet in pets {
        pet.speak();
    }
}
```

Eq, PartialEq, Ord, PartialOrd traits

In Rust, traits are used to define shared behavior for types. The following are commonly used traits for comparing types:

- Eq trait: This trait defines the equality relation between two values of a given type. The Eq trait requires that the type implements the PartialEq trait, which defines the partial equality relation. If two values of a type are equal according to the Eq trait, they must be considered indistinguishable in every way.
- PartialEq trait: This trait defines the partial equality relation between two values of a given type. The PartialEq trait requires that the type implements an eq method that takes another value of the same type as an argument, and returns a bool indicating whether the two values are equal. If two values of a type are equal according to the PartialEq trait, they must be considered indistinguishable for the purposes of the Eq trait as well.
- Ord trait: This trait defines the total order relation between two values of a given type. The Ord trait requires that the type implements the PartialOrd trait, which defines the partial order relation. If two values of a type are compared using the Ord trait, they must be completely ordered in a consistent way.
- PartialOrd trait: This trait defines the partial order relation between two values of a given type. The PartialOrd trait requires that the type implements a partial_cmp method that takes another value of the same type as an argument, and returns an Option indicating the ordering relationship between the two values. If two values of a type are compared using the PartialOrd trait, they must be partially ordered in a consistent way.

These traits are important for comparing types in Rust, and are used extensively in Rust's standard library.

From and Into traits for conversions

The Rust `From` trait and `Into` trait are used to convert between types.

The `From` trait provides a `from` method that takes an argument of a different type and returns an instance of the implementing type. This allows for easy conversion between different types, especially when converting from a type that is not owned by the implementing type.

The `Into` trait provides an `into` method that takes no arguments and returns an instance of a different type. This allows for easy conversion between different types, especially when converting from a type that is owned by the implementing type.

Example:

```
struct MyStruct(i32);

// Convert from i32
impl From<i32> for MyStruct {
    fn from(val: i32) -> Self {
        MyStruct(val)
    }
}

// Convert into i32
impl Into<i32> for MyStruct {
    fn into(self) -> i32 {
        self.0
    }
}

fn main() {
    let my_struct = MyStruct::from(42);
    let i: i32 = my_struct.into();
}
```

This example defines a simple `MyStruct` struct. We implement the `From` trait `from` method. We implement the `Into` trait `into` method.

Send trait for sending among threads

The Rust `Send` trait indicates that a type is safe to be sent across thread boundaries. This means that if a type implements the `Send` trait, it can be safely passed from one thread to another without causing any data races or undefined behavior. For example, the `String` type in Rust implements the `Send` trait, which means it can be safely shared across multiple threads.

To implement the `Send` trait for a custom type, all of its fields must also implement the `Send` trait. This is because if a type contains non-`Send` fields, it may be possible for data races to occur when the type is shared across threads. The `Send` trait is automatically implemented for most primitive types in Rust, as well as many standard library types like `Vec` and `String`.

Here's an example of a custom type that implements the `Send` trait:

```
struct Foo {
    x: i32,
    y: String,
}

unsafe impl Send for Foo {}

fn main() {
    let foo = Foo { x: 42, y: "Hello".to_string() };
    std::thread::spawn(move || {
        println!("x = {}, y = {}", foo.x, foo.y);
    }).join().unwrap();
}
```

In this example, the `Foo` struct contains an `i32` field and a `String` field. Both `i32` and `String` implement the `Send` trait, so we can implement `Send` for `Foo` using the `unsafe impl Send for Foo {}` syntax. We can safely send a `Foo` instance to a new thread using `std::thread::spawn`, and access its fields from within the thread.

Sync trait for syncing among threads

The Rust `Sync` trait indicates that a type is safe to be shared between multiple threads. If a type implements the `Sync` trait, it can be safely accessed from multiple threads without causing any data races or undefined behavior. For example, the `Arc` type implements the `Sync` trait, so it can be safely shared between multiple threads.

Here's an example of how the Sync trait can be used:

```
use std::sync::Arc;

struct MyStruct {
    x: i32,
}

impl MyStruct {
    fn my_function(&self) {
        println!("x is {}", self.x);
    }
}

fn main() {
    let s = MyStruct { x: 1 };
    let shared = Arc::new(s);
    std::thread::spawn({
        let shared = shared.clone();
        move || { shared.my_function(); }
    }).join().unwrap();
}
```

In this example, we create a shared instance of `MyStruct` using the `Arc` type, which automatically implements the `Sync` trait. We can then safely access the shared instance from multiple threads, without worrying about synchronization issues.

Send & Sync - implementations

https://stackoverflow.com/questions/68704717/

The `Send` trait and `Sync` trait show up frequently in multi-threaded projects. Here are implementations that are especially good to know.

`&T`: since immutable references can be copied, the ability to send one to another thread would let you perform immutable access from several threads in parallel. Thus `&T` can only be `Send` if `T` is `Sync`. There is no need for `T` to be `Send` as an `&T` doesn't allow mutable access.

`&mut T`: mutable references can't be copied, so sending them to other threads doesn't allow access from several threads in parallel, thus `&mut T` can be `Send` even if `T` is not `Sync`. Of course, `T` must still be `Send`.

`MutexGuard`: destroying a `MutexGuard` on another thread is unsound, so it can't be `Send`. However if the value inside may be immutably accessed from several threads in parallel, then such an immutable access would also be safe on the `MutexGuard` itself.

`SyncWrapper`: an immutable reference to a `SyncWrapper<T>` does not allow you to perform any actions at all; it is always safe to be `Sync`.

`Rc<T>`: if you have two clones of the same `Rc<T>`, then it would be a data race to access them from different threads in parallel. This rules out both `Send` and `Sync`, since both of them would allow immutable access from other threads, and that other thread could use that to call `.clone()` remotely and obtain an `Rc<T>` on the other thread.

`Arc<T>`: this mostly behaves like `&T`. It can be cloned, so sending it to other threads requires `T: Sync`. However, it also requires `T: Send` as the last `Arc<T>` might be dropped on a different thread than where `T` was created, which you can't do without `Send`.

`RefCell<T>`: this type can never be `Sync` because you can modify the value inside with only an immutable reference, and this would be a data race if you could do it from several threads in parallel. There's no problem with `RefCell<T>` being `Send` provided that `T` is.

Sealed traits

In Rust, a sealed trait is a trait that can only be implemented within a particular module, and not outside it. This means that once a trait is marked as “sealed”, any other code outside the module where the trait was defined cannot implement it.

Sealed traits are useful when you want to limit the set of types that can implement a particular trait to a specific set of types. This can be helpful when designing APIs or libraries where you want to restrict the use of certain traits to specific contexts or modules.

To define a sealed trait in Rust, you must declare the trait as `pub` and include a private `mod` statement with the same name as the trait. This private module should contain all the implementations of the trait.

Here’s an example:

```
pub trait Sealed {}

mod private {
    use super::Sealed;

    impl Sealed for i32 {}
    impl Sealed for String {}
}

pub fn foo<T: Sealed>(val: T) {
    // do something with val
}
```

In this example, the Sealed trait is defined as pub and the implementations are placed in a private module called private. The foo function is generic over T where T must implement the Sealed trait.

Because the private module is private, no other code outside the module can implement the Sealed trait, ensuring that only the types explicitly listed within the module can be used with the trait.

enum keyword for enumerations

In Rust, an enum (short for "enumeration") is a custom data type that allows you to define a set of named values. Each value is called a variant, and you can use an enum to represent a fixed set of possible values for a particular data type.

Here's an example of an enum in Rust:

```
enum Color {
    Red,
    Green,
    Blue,
}
```

In this example, we've defined an enum called `Color` with three variants: `Red`, `Green`, and `Blue`. We can use this enum to represent a color value in our Rust program.

Enums in Rust can also include data associated with each variant. Here's an example:

```
enum IPAddress {
    V4(u8, u8, u8, u8),
    V6(String),
}
```

In this example, we've defined an enum called `IPAddress` with two variants: `V4` and `V6`. The `V4` variant includes four `u8` values representing the four octets of an IPv4 address, while the V6 variant includes a single `String` value representing an IPv6 address.

Enums in Rust can be useful for a variety of programming tasks, including defining states for a state machine, representing different types of errors, and creating custom data types for your program. Rust's enums are type-safe and flexible, making them a powerful tool for Rust programmers.

struct keyword for custom data types

A Rust `struct` is a custom data type that groups related data and functions. A struct is defined using the `struct` keyword, followed by the name of the struct, and a block of curly braces that contains the fields of the struct.

Here is an example of a struct in Rust:

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

Usage:

```
let r = Rectangle {
    width: 10,
    height: 20
};
```

Structs can also have functions associated with them, called methods. Methods are defined within the block of curly braces after the fields of the struct, and can be used to operate on the data within the struct.

Example methods:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

This example uses `impl` to define an implementation block for the `Rectangle` struct, and defines a method named `area` that calculates the area of the rectangle. The `&self` parameter indicates that the method takes a reference to the struct as its first argument.

union keyword for multi-type memory

A Rust union is a user-defined type that is similar to a struct, but instead of each field having its own memory space, a union has a single memory space that can be interpreted as different types depending on the current value of the union.

To define a union, use the union keyword, then the name of the union, then the fields of the union. For example:

```
union MyUnion {
    i: i32,
    f: f32,
}
```

In this example, `MyUnion` has two fields: `i` and `f`. The union can hold one of these fields at a time, not both. When you access a field of a union, Rust ensures the field is the active field of the union.

To change the value, use the unsafe keyword and transmute function. For example:

```
unsafe {
    let mut my_union = MyUnion { i: 42 };
    my_union.f = std::mem::transmute(3.14f32);
}
```

In this example, we use `std::mem::transmute` to convert a `f32` value into a bit pattern that can be interpreted as an `i32`. We then assign this value to `my_union.f`. Because we haven't accessed `my_union.i` since it was set, Rust considers `my_union.f` to be the active field of the union. If try to access `my_union.i` now, it would be undefined behavior (UB).

Because unions represent different types in the same memory space, it's easy to accidentally create bugs. In general, only use unions when you need to work with low-level data structures or when you need to optimize memory usage.

match keyword for control flow

The `match` keyword is a control flow construct that allows a program to match a value against a set of patterns and execute code based on the match result. The `match` keyword statement is similar to a `switch` keyword statement in other languages, but `match` provides more powerful pattern matching capabilities.

A `match` statement typically has the following syntax:

```
match <value> {
    <pattern_1> => <code_1>,
    <pattern_2> => <code_2>,
    ...
    <pattern_n> => <code_n>,
}
```

The `<value>` is the expression that is being matched against, and the `<pattern>` expressions are the patterns that are being matched. Each `<pattern>` is followed by a `=>` symbol, then a block of code that will be executed if the pattern matches the value.

In Rust, a pattern can take many forms, including literal values (e.g. `42`, "hello"), variables (e.g. `x`, `y`), wldcards (e.g. `_`), ranges (e.g. `1..=5`), enums (e.g. `Some(value)`), structs (e.g. `Point { x, y }`), tuples (e.g. `(x, y)`), and more.

The code in each match arm is executed if the pattern on the left-hand side of the `=>` operator matches the value being matched. If none of the patterns match, the `match` statement will panic at runtime.

Rust's `match` statements are powerful and flexible, allowing for complex patterns and expressions to be matched. Match statements are commonly used in Rust to handle errors, parse command-line arguments, and implement state machines, among other use cases.

Overall, match statements are a key feature of Rust's control flow syntax, and provide a powerful mechanism for pattern matching and value extraction in Rust programs.

mod keyword for module namespaces

In Rust, namespaces are a way to organize and group related items, such as functions, types, and constants, under a common name. Namespaces are implemented using modules, which are Rust's primary mechanism for organizing code into reusable components.

Modules can be defined using the `mod` keyword, followed by the name of the module and its contents enclosed in curly braces:

```
mod my_module {
    fn private_function() {
        // implementation details here
    }
    pub fn public_function() {
        // implementation details here
    }
}
```

In this example, `my_module` is a module that contains two functions: `private_function`, which is not visible outside of the module, and `public_function`, which is marked as pub and can be accessed from other modules.

To use a module from another module, you can use the use keyword to bring its contents into scope:

```
use my_module::public_function;

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

In this example, we bring the `public_function` from `my_module` into the scope of main, allowing us to call it directly.

Overall, namespaces in Rust provide a powerful mechanism for organizing and structuring code, enabling developers to write more modular, reusable, and maintainable software.

mod keyword for nested hierarchies

The Rust `mod` keyword can provide nested hierarchies, meaning that a modules can contain other modules:

```
pub mod outer {
    pub mod inner {
        pub fn hello() {
            println!("Hello");
        }
    }
}

fn main() {
    outer::inner::hello()
}
```

You can optionally add a `use` statement such as:

```
use outer::inner::hello;
fn main() {
    hello()
}
```

Module hierarchies can help test-driven development, because you can create an outer module `tests`, with an inner module for each function, to improve readability and encapsultation:

```
#[cfg(test)]
mod tests {
    mod my_function_1 {
        #[test]
        fn test_something() {
            assert!(/* ... */);
        }
    }
    /* ... */
}
```

Nested-or-pattern for matching

Source

The nested-or-pattern for matching combines `|` expressions.

Example `if` statement without nested-or-pattern:

```
if let Some(2) | Some(3) | Some(5) | Some(7) = value …{}
```

And with nested-or-pattern:

```
if let Some(2 | 3 | 5 | 7) = value …
```

The nested-or-pattern can be useful in many kinds of statements.

Example `match` statement:

```
match value {
    Some(n @ (2 | 3 | 5 | 7)) => println!("{n} is a prime"),…
```

Example `let` statement:

```
let (Ok(i) | Err(i)) = [1, 2, 3].binary_search(&2);
```

Example function definition:

```
fn f((Ok(i) | Err(i)): Result<i32, i32>) {}
```

async/await keywords for futures

Rust provides support for asynchronous programming through its `async/await` syntax. The `async` keyword defines a function that can be suspended and resumed later. The `await` keyword pauses execution of an `async` function until a condition is met.

When a function is declared with the `async` keyword, it becomes an asynchronous function. This means that the function can be paused at any point using the await keyword and resumed later when the awaited value becomes available. The async function returns a `Future` type that represents the result of the computation.

Example of an async function that returns a future:

```
async fn fetch(url: &str) -> Result<String, reqwest::Error> {
    let response = reqwest::get(url).await?;
    let body = response.text().await?;
    Ok(body)
}
```

The example's `fetch` function is defined with the `async` keyword. The function uses the `reqwest` crate to make an HTTP request. The first `await` waits for the response. The second `await` waits for the body text.

Example of `await` that waits for a future:

```
async fn do_something() -> i32 {
    let future = get_result_async();
    let result = await!(future);
    result + 1
}
```

The example's `await!` pauses execution of the `do_something` function until `get_result_async` is completed. Once the future completes, the result is returned and the task is resumed. The value of the result is then incremented by 1 and returned as the final result.

trait keyword for polymorphism

In Rust, a trait is a language construct that defines a set of methods that can be implemented by a type. Traits enable polymorphism, generic programming, and code reuse without sacrificing performance or safety.

Example trait that defines one method:

```
trait MyTrait {
    fn my_method(&self);
}
```

Example struct that implements the method:

```
struct MyStruct;

impl MyTrait for MyStruct {
    fn my_method(&self) {
        println!("Hello");
    }
}
```

Example function that takes the trait and calls the method:

```
fn foo<T: MyTrait>(item: T) {
    item.my_method();
}
```

To run it:

```
fn main() {
    let s = MyStruct{};
    foo(s)
}
```

Some of the common Rust traits are `Debug` and `Display` for formmating output, `Copy` and `Clone` for duplicating values, `From` and `Into` for converting values, and `Send` and `Sync` for multi-thread communication.

println! macro for printing output

The Rust `println!` macro is a built-in macro that is used to print text to stdout (standard output).

Here is an example code that uses the `println!` macro to print a simple message to the console:

```
fn main() {
    println!("Hello, World!");
}
```

In this example, we call the `println!` macro with one argument: the string `"Hello, World!"`. The macro then prints the string to the console.

The `println!` macro is similar to the `print!` macro, but adds a newline character (`\n`) to the end of the output, while the `print!` macro does not.

The `println!` macro can also accept additional arguments for string formatting. For example, we can use the `{}` placeholder to insert variables or values into the output string:

```
fn main() {
    let name = "Alice";
    let age = 30;
    println!("My name is {} and age is {}", name, age);
}
```

In this example, we use two placeholders (`{}`) in the output string to print the values of the `name` and `age` variables. When the macro is executed, it replaces the `{}` placeholders with the corresponding values (`"Alice"` and `30`, respectively). The resulting output would be:

```
My name is Alice and age is 30
```

The `println!` macro is similar to the `format!` macro for formatting strings, and the `write!` macro for writing formatted data into a buffer.

assert! macro and friends for testing

The Rust testing framework provides macros for test assertions, such as:

- `assert!(condition)`: assert `condition` is true.
- `assert_eq!(a, b)`: assert `a` is equal to `b`.
- `assert_ne!(a, b)`: assert `a` is not equal to `b`.

Example:

```
let x = 1;
let y = 2;
assert!(x < y);
```

Example with an optional message:

```
let x = 1;
let y = 2;
assert!(x < y, "We want x to be less than y");
```

Assertables crate

The Assertables crate provides more assert macros, such as:

- `assert_starts_with!(x, y)`: Does `x` start with `y`?
- `assert_contains!(array, element)`: Does `array` contains `element`?
- `assert_is_match(regex, string)`: Does `regex` match `string`?string.

Example:

```
let a = "hello world";
let b = "hello";
assert_starts_with!(&a, &b);
```

regex! macro for lazy static optimization

https://crates.io/crates/once-cell-regex

The `regex!` macro takes a string literal and returns an expression that evaluates to a `&'static Regex`. This macro can be useful to avoid the problem of compiling a regex on every loop iteration.

The `regex!` macro capabilties are provided by the `once_cell` crate and `once-cell-regex` crate.

Add to `Cargo.toml`:

```
[dependencies]
once_cell = "*"
once-cell-regex = "*"
```

Example:

```
use once_cell_regex::regex;

fn main() {
    let r = regex!("hello");
    let x = r.is_match("hello world");
    println!("{}", x); // prints "true"
}
```

once_cell crate

The `once_cell` crate can provide optimizations in many more ways, such as safe initialization of global data, general purpose lazy evaluation, runtime bytes, late initialization, and more.

There are similar crates if you need related features:

- If you want asynchronous capabilties, try the `async_once_cell` crate.
- If you want spinlocks, try the `lazy_static` crate.

catch_unwind! macro to handle panic

The Rust `panic catch_unwind!` macro is a way to catch unwinding panics that can occur when a piece of code fails at runtime. When an unwinding panic happens, Rust unwinds the stack and calls the panic handler, which can be customized to do any number of things, such as print an error message or roll back a transaction.

The `catch_unwind!` macro allows you to catch these unwinding panics and handle them in a more controlled way. It returns a Result value that lets you know if the code in the block panicked or not. If it did panic, you can then handle the error in any way you see fit, such as printing an error message or returning an alternate value.

Here's an example of how to use the `catch_unwind!` macro:

```
use std::panic;

let result = panic::catch_unwind(|| {
    // Code that might panic goes here
});

match result {
    Ok(_) => println!("Code did not panic"),
    Err(_) => println!("Code panicked!"),
}
```

In this example, we define a closure that contains the code we want to run. We then pass that closure to the `catch_unwind!` macro. If the code within the closure panics, the result value will be an `Err` value. If it doesn't panic, the result value will be an `Ok` value.

The `catch_unwind!` macro is not guaranteed to succeed, for example when using custom panics or aborting panics. Additionally, the `catch_unwind!` macro is not generally recommended outside of FFI purposes. To help prevent panics, Rust provides many non-panic functions, such as Vec `get` instead of slice, and `checked_add` instead of operator addition. To help documentation show panics, Rust Clippy provides the lint `missing_panics_doc`.

macro_rules! for declarative macros

https://doc.rust-lang.org/book/ch19-06-macros.html

The Rust `macro_rules!` macro is a powerful code generation tool that allows the developer to create custom syntax or keywords that expand into Rust code at compile time. With this macro, you can define custom syntax rules, patterns, and templates that can be used to generate code automatically.

The `macro_rules!` macro works by defining a set of rules that match the input code, similar to a regular expression. These rules are then used to generate Rust code based on the input, which can be used to reduce the amount of repetitive or boilerplate code required for a given codebase.

Syntax:

```
macro_rules! my_macro_name {
  // Define patterns and templates here that match the input code
}
```

Here's an example of a simple Rust macro that generates a for loop with a range of numbers:

```
#[macro_export]
macro_rules! number_loop {
    ($start:expr, $end:expr) => {
        for i in $start..$end {
            println!("{}", i);
        }
    }
}
```

With this macro, you can now generate a for loop by simply invoking the number_loop! macro with the desired start value and end value:

```
number_loop!(0, 10);
```

This will output the numbers from 0 to 9.

Annotations for compiler directives

In Rust, annotations are used to provide additional information to the compiler about how code should be compiled or optimized. Annotations are usually written as attributes and are placed above the item they apply to.

There are different types of annotations in Rust, such as `derive`, `allow`, `test`, `inline`, `cfg`, and more.

`#[derive]` automatically implements the given traits for a struct or enum, such as:

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u32,
}
```

`#[allow]` silences compiler warnings, such as:

```
#[allow(unused_variables)]
fn foo() {
    let x = 42;
}
```

`#[test]` marks a function as a test, so it runs with `cargo test`, such as:

```
#[test]
fn test_addition() {
    assert_eq!(2 + 2, 4);
}
```

Overall, annotations in Rust provide a way to add additional information to code that can help the compiler optimize and generate better code. They are a powerful tool for controlling the behavior of the compiler and improving the performance of Rust programs.

Destructuring into components

In Rust, destructuring is the process of taking apart a complex data structure (such as a tuple, struct, or enum) into its individual components.

Destructuring a tuple:

```
let my_tuple = (1, 2);
let (a, b) = my_tuple; // Assign a = 1, b = 2
```

Destructuring struct fields:

```
struct MyStruct {
    x: i32,
    y: String,
}

let my_struct = MyStruct { x: 42, y: String::from("hello") };
let MyStruct { a, b } = my_struct; // Assign a = 42, b = "hello"
```

Destructuring an enum variant:

```
enum MyEnum {
    Variant1(i32),
    Variant2(String),
}

let my_enum = MyEnum::Variant1(42);
match my_enum {
    MyEnum::Variant1(n) => println!("Got a number: {}", n),
    MyEnum::Variant2(s) => println!("Got a string: {}", s),
}
```

Iterators for traversing collections

In Rust, iterators are abstractions for traversing collections of data, such as arrays, vectors, and other sequences. Iterators access the elements of a collection, and can be used with many of Rust's built-in language features, such as loops and closures.

Iterators in Rust are defined by the `Iterator` trait, which provides methods for traversing and manipulating a sequence of elements. Some common methods on iterators include:

- `next()`: Return the next iterator element, or None.
- `filter()`: Return the elements that match a predicate.
- `map()`: Transform each element of the iterator.
- `fold()`: Reduce elements to a value via a function.

Example to traverse a vector and sum up its elements:

```
let v = vec![1, 2, 3, 4, 5];
let sum = v.iter().fold(0, |acc, x| acc + x);
println!("The sum is: {}", sum);
```

In this example, we create a vector `v` and use the `iter()` method to create an iterator over its elements. We then use the `fold()` method to iterate over the elements, and accumule the sum of all the elements.

Iterators can also be used in loops, as in the following example:

```
let v = vec![1, 2, 3, 4, 5];
for i in v.iter().map(|x| x * 2) {
    println!("{}", i);
}
```

In this example, we create an iterator over the vector elements, and use the `map()` method to transform each element by doubling it. We use a `for` loop to iterate over the transformed elements, to print each one.

Closures for anonymous functions

Closures are a type of anonymous function that can capture variables from their surrounding environment, and create self-contained units of behavior that can be passed around and reused.

Example of a closure in Rust:

```
let add = |a, b| a + b;
let result = add(3, 4);
```

In this example, we define a closure `add` that takes `a` and `b` and returns their sum. We call the closure with `3` and `4` and print the result.

Closures in Rust are defined using the `|` symbol to specify the arguments, followed by the body in braces `{}`. Rust's type inference system allows you to omit the types of the arguments, if they can be inferred.

Example of a closure that accesses a variable outside of it:

```
let x = 5;
let add_x = |y| x + y;
let result = add_x(3);
```

In this example, we define a closure `add_x` that takes an argument `y` and adds it to the variable `x` that is already defined outside of the closure. When we call the closure with argument `3`, it captures the value of `x` and returns `8`.

Example of a closure for collection iterator map function:

```
let numbers = vec![1, 2, 3, 4];
let squares = numbers.iter().map(|x| x * x);
```

In this example, we define a vector of numbers, then use the `iter` method to create an iterator over the vector's elements. We then use the `map` method to apply a closure that squares each element of the vector.

Macros for metaprogramming

Rust macros are a powerful tool for metaprogramming, allowing you to write code that generates code at compile-time. Macros are defined using the macro_rules! macro, which allows you to match on patterns in the code and generate new code based on those patterns.

Rust macros can be used for a variety of tasks, such as creating domain-specific languages (DSLs), reducing boilerplate code, or implementing code generation tools.

There are two types of Rust macros: declarative macros and procedural macros.

Declarative macros (also known as “macro_rules! macros”) use pattern matching to transform code. They are defined using the `macro_rules!` macro and operate on the tokens that make up the code. Declarative macros can be used to create new syntax or simplify existing syntax, and they are often used to create DSLs.

Procedural macros, on the other hand, operate on the AST (abstract syntax tree) of the code. They are defined using Rust’s proc_macro API and allow you to write code that generates new code at compile-time. Procedural macros can be used to implement custom derive macros, attribute macros, and function-like macros.

Example of a declarative macro:

```
macro_rules! greet {
    (to $name:ident) => {
        println!("Hello, {}!", stringify!($name));
    };
}
```

This macro takes a value, in this case `name`, and generates a custom greeting message for it.

Panic and how to handle it with a hook

In Rust, a `panic` occurs when a program encounters a situation where it cannot continue to run safely. This can happen for a variety of reasons, such as a failed assertion, an out-of-bounds array access, or an attempt to unwrap a `None` value. When a `panic` occurs, Rust will unwind the stack and search for a `catch_unwind` block that can handle the panic. If no such block is found, the program will terminate with an error message.

By default, Rust will print an error message and terminate the program when a panic occurs. However, it is possible to customize this behavior by adding a `panic` hook. This allows you to define your own `panic` handler that can log the error, send an alert, or perform other actions before terminating the program.

You define a `panic` hook by calling the `std::panic::set_hook` function. Pass a closure that takes a `PanicInfo` struct as an argument; the struct contains useful debugging information.

Example of a panic hook that logs the message then terminates:

```
use std::panic;

fn main() {
    panic::set_hook(Box::new(|panic_info| {
        let message = panic_info
            .payload()
            .downcast_ref::<String>()
            .unwrap_or(&"Unknown error".to_string());
        eprintln!("Panic occurred: {}", message);
    }));
    panic!("Yikes!"); // Deliberately trigger a panic
}
```

This sets a `panic` hook that logs the `panic` message to the standard error stream using the `eprintln` macro. When the program encounters a `panic!` macro, it will trigger the panic hook and log the error message before terminating the program.

Pass by value or reference

Passing a value to a function can be done by value or by reference.

Pass by value means that a copy of the original value is passed to the function. Any changes made to the value inside the function will not affect the original value. In other popular langauges, pass by value is typical for passing primitive data types.

Pass by reference means that a reference to the original value is passed to the function instead of a copy. This allows the function to modify the original value, as it has access to the actual memory location of the value. In other popular langauges, pass by reference is typical for passing object data type pointers.

For example:

```
fn increment_with_pass_by_value(num: i32) {
    num + 1;
}

fn increment_with_pass_by_reference(num: &mut i32) {
    *num += 1;
}

fn main() {
    let mut x = 1;
    increment_with_pass_by_value(x);
    println!("x is {}", x); // x is still 1
    increment_with_pass_by_reference(&mut x);
    println!("x is {}", x); // x is now 2
}
```

One of the advantages of Rust is compiler warnings and help. In the pass by value function, the compiler detects that the function result is never used, and shows warnings such as “the arithmetic operation produces a value” and “note: #[warn(unused_must_use)] on by default” and “help: use let _ = … to ignore the resulting value”.

Range syntax for a sequence of values

In Rust, a range is a way to represent a sequence of values between a start and end point. A range are defined using the syntax `start..end`, where `start` is the first value in the range, and `end` is the first value not in the range.

Here are some examples of Rust ranges:

```
let a = 0..10;  // range from 0 to 9 inclusive
let b = 1..=10; // range from 1 to 10 inclusive
let c = ..5;    // range from start to 5 exclusive
let d = 5..;    // range from 5 to infinity
```

Ranges can be used in many contexts in Rust, such as in for loops:

```
for i in 0..10 {
    println!("{}", i);
}
```

This will print the numbers from 0 to 9.

Ranges can also be used with various methods provided by the `Iterator` trait, such as `map`, `filter`, `fold`, and more:

```
let nums = (0..10)
    .filter(|x| x % 2 == 0)
    .map(|x| x * 2)
    .collect::<Vec<_>>();
// nums is now [0, 4, 8, 12, 16]
```

This creates a range from 0 to 9, filters out any odd numbers, doubles the remaining even numbers, and collects them into a vector.

Overall, Rust ranges are a flexible and convenient way to represent sequences of values, and they are widely used throughout the language.

Tuples for ordered collections

In Rust, a tuple is an ordered collection of values with a fixed length. Tuples can contain values of different types and are represented using parentheses with the values separated by commas.

Example of a tuple containing a string and an integer:

```
let person = ("Alice", 30);
```

This defines a tuple called `person` containing the string “Alice” and the integer `30`. Tuples can be assigned to variables, passed as function arguments, and returned as function results, just like any other value.

You can access individual elements of a tuple using dot notation and the index of the element you want to access, starting from zero:

```
let name = person.0;
let age = person.1;
```

Tuples are often used to return multiple values from a function. For example, the `std::fs::metadata` function returns a tuple that contains information including a file’s length and permissions:

```
use std::fs;

fn main() -> std::io::Result<()> {
    let metadata = fs::metadata("file.txt")?;
    let (len, perms) = (
        metadata.len(),
        metadata.permissions(),
    );
    println!("File len:{}, permisisons:{}", len, permissions);
    Ok(())
}
```

Memory lifetimes

Rust has strict rules for memory management, which includes the concept of memory lifetimes. A memory lifetime is the duration for which a particular piece of memory is valid and can be accessed. Lifetimes can be defined in function signatures, struct definitions, and other code constructs. The borrow checker enforces rules around memory lifetimes, to ensure that memory is accessed safely and without any undefined behavior.

Memory lifetimes are determined by the ownership and borrowing system. Every value has an owner, which is responsible for allocating and freeing the memory associated with the value. When a value is borrowed, the borrower is given a reference to the memory owned by the owner. The borrower must return the reference before the owner goes out of scope, or else the program will not compile.

For example, consider the following code:

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

Here, `x` is an integer with a value of 5. The `&` operator creates a reference to `x` and assign it to `y`. The `println!()` macro prints the value of `y`. The lifetime of `x` begins when it is created and ends when it goes out of scope at the end of the `main()` function. The lifetime of `y` is the same as the lifetime of `x`, because it is a reference to the memory owned by `x`. The borrow checker ensures that `y` is returned before `x` goes out of scope.

Memory lifetimes are strict, and can be complex to learn, because they help ensure that programs are safe and free from undefined behavior, and enable high-performance memory-safe code without the need for garbage collection or other runtime memory management.

Implicit lifetimes

Lifetimes can be implicit, meaning they do not need notations, or explicit, meaning they do need notations.

Implicit lifetimes are typical, because the Rust compiler can figure out lifetimes for many typical needs, which means the developer doesn't need to write the lifetimes.

Suppose a function has one input arg and an output reference:

```
fn trim_period(s: &String) -> &str {
    s.trim_matches('.')
}
```

The compiler can figure out lifetimes by using the function signature:

- The compiler sees the output string slice, and knows that the string slice memory must come from somewhere.
- The compiler sees the function has only one input arg, so knows that the memory must come from the input arg.
- Therefore the compiler can automatically set the lifetimes: the input arg must live at least as long as the output string slice.

The compiler's implicit lifetimes are equivalent to these explicit lifetimes:

```
fn trim_period<'a>(s: &'a String) -> &'a str {
    s.trim_matches('.')
}
```

Explict lifetimes

Explicit lifetimes are atypical, and only needed when the Rust compiler can't figure out the lifetimes. For this, the developer must write explicit lifetimes so the compiler knows what to do.

Suppose a function has two input args and an output reference:

```
fn trim_period(s: &String, msg: &String) -> &str {
    println!(msg);
    s.trim_matches('.')
}
```

The compiler cannot figure out the lifetimes by using the function signature:

- The compiler sees the output string slice, and knows that the string slice memory must come from somewhere.
- The compiler sees the function has two input args, so knows that the memory must come from either of the input args, or possibly from a combination of them.
- Therefore the compiler cannot automatically set lifetimes: it's unknown which input arg must live at least as long as the output string slice, or if there's something else involved from a combination.

The compiler requires the developer to write explicit lifetimes:

```
fn trim_period<'a>(s: &'a String, msg: &String) -> &'a str {
    println!(msg);
    s.trim_matches('.')
}
```

The explicit lifetimes instruct the compiler that the output string slice memory comes from the first input arg, not the second input arg.

Memory on the stack or the heap

In Rust, memory is typically allocated either on the stack or the heap. The stack and heap are two different regions of memory that are used for different purposes.

The stack is a region of memory that is used for storing local variables and function call frames. Each time a function is called, a new stack frame is created to store the function's local variables and other data. When the function returns, its stack frame is destroyed, and the memory used by the stack frame is released.

Stack allocation is fast and efficient, because memory for a stack frame is allocated when the function is called, and released when the function returns. Stack allocation doesn't require any runtime overhead, making it an excellent choice for small, short-lived objects.

On the other hand, the heap is a region of memory that is used for dynamically allocated data. Data allocated on the heap persists until it is explicitly deallocated. Heap allocation can be slower and less efficient than stack allocation, because it requires additional runtime overhead to allocate and deallocate memory.

In Rust, heap allocation is typically done using the Box type, which creates a pointer to a value that is stored on the heap. For example:

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

Here, `x` is a pointer to a value of 5 that is stored on the heap. The `Box::new()` function allocates memory on the heap and returns a pointer to the allocated memory. The `println!()` macro prints the value of `x`.

Memory ownership - example

Example of memory ownership and borrowing in Rust:

```
fn main() {
    let mut vec = vec![1, 2, 3];
    print_vec(&vec); // Pass a reference
    vec.push(4); // Modify the vector
    take_vec(vec); // Pass ownership
}

fn print_vec(vec: &Vec<i32>) {
    for num in vec {
        println!("{}", num);
    }
}

fn take_vec(vec: Vec<i32>) {
    println!("Took ownership of {:?}", vec);
}
```

In this example, we define a vector of integers and then pass a reference to the vector to a function called `print_vec`. The `print_vec` function borrows the reference to the vector and iterates over it, printing each element.

Next, we modify the vector by pushing another element onto it, and then pass ownership of the vector to a function called `take_vec`. The `take_vec` function takes ownership of the vector and prints a message to indicate that it has ownership of the vector.

Notice that we use the `&` operator to pass a reference to the vector to `print_vec`. This is an example of borrowing in Rust - we borrow a reference to the vector without taking ownership of it.

In contrast, when we pass the vector to `take_vec`, we don't use the `&` operator. This is an example of taking ownership in Rust - we transfer ownership of the vector to the `take_vec` function.

Mutability and immutability

Rust provides strict control over mutable and immutable references to data. Rust's approach to mutability and immutability helps to prevent many common programming errors, such as null pointer references, race conditions, and other types of undefined behavior.

In Rust, a variable's mutability is determined by whether or not it was declared with the `mut` keyword. If a variable is declared with `mut`, it is mutable, meaning it can be changed. If it is not declared with `mut`, it is immutable, meaning it cannot be changed.

Here is an example of a mutable variable in Rust:

```
let mut x = 5;
x = 6; // This is allowed because x is mutable.
```

And here is an example of an immutable variable in Rust:

```
let x = 5;
x = 6; // This is not allowed because x is immutable.
```

Immutable variables are useful for ensuring that data remains constant and unchanging. They can help to prevent accidental modification of data and make programs easier to reason about. On the other hand, mutable variables can be useful for cases where data needs to be updated or changed.

In Rust, mutability is also closely tied to references to data. Rust uses a concept called borrowing to ensure that mutable and immutable references to data do not overlap in ways that could cause undefined behavior.

When a variable is borrowed as mutable, the borrowing function gains exclusive access to the data, meaning that no other function can access it until the mutable reference goes out of scope. Conversely, when a variable is borrowed as immutable, multiple functions can access the data at the same time, as long as they are not trying to modify it.

No garbage collection

When a program creates objects or data in memory, the program must manage the memory. Some languages such as C rely on the developer to allocate memory and free it. Some languages such as Java use garbage collection. Rust has a unique approach that uses no garbage collection.

What is garbage collection?

Garbage collection (GC) is a mechanism that automatically frees up memory that is no longer being used.

Garbage collection works by periodically scanning the memory used by a program to identify objects that are no longer being used. Once identified, the garbage collector frees up the memory used by these objects, making it available for future use by the program.

There are different types of garbage collection algorithms, such as reference counting, mark and sweep, and copying. Each algorithm has its strengths and weaknesses, depending on context.

Rust doesn't use garbage collection

One of Rust's key innovations is guaranteeing memory safety (meaning no segfaults) without requiring garbage collection. Rust avoids GC by tracking memory ownership and enforcing safety via the borrow checker.

By avoiding GC, Rust can offer numerous benefits: predictable cleanup of resources, lower overhead for memory management, and essentially no runtime system. These benefits make it easier to embed Rust into arbitrary contexts, and also easier to integrate Rust with languages that do have a GC.

For when single ownership does not suffice, Rust programs can use the standard library reference-counting smart pointer types: `Rc` for single-thread reference counting, and Arc for multi-thread reference counting.

Borrow splitting

Borrow splitting, a.k.a. partial borrowing, is when you try to borrow in multiple ways that can interfere with each other.

This example fails to compile because of borrow splitting:

```
// Create a typical struct
struct Foo {
    a: i32,
    b: i32,
}

// Create mutabale accessors
impl Foo {
    pub fn a_mut(&mut self) -> &mut i32 {
        &mut self.a
    }
    pub fn b_mut(&mut self) -> &mut i32 {
        &mut self.b
    }
}

// Compile succeeds because `a` and `b` are independent
pub fn increment(a: &mut i32, b: &mut i32) {
    *a = *a + 1;
    *b = *b + 1;
}

// Compile error because `a` and `b` are borrow splitting:
// cannot borrow `*self` as mutable more than once at a time
impl Foo {
    pub fn increment(&mut self) {
        let a = self.a_mut();
        let b = self.b_mut();
        *a = *a + 1;
        *b = *b + 1;
    }
}
```

Test framework

Rust has a built-in testing framework that allows developers to write and run automated tests for their Rust code. The testing framework is designed to be easy to use, and it supports a wide range of testing scenarios, including unit tests, integration tests, and benchmark tests.

To write tests in Rust, developers create test functions that are annotated with the #[test] attribute. These functions can contain one or more test assertions that check whether a particular condition is true or false. If all assertions in a test function pass, the test is considered to have passed. If any assertion fails, the test is considered to have failed.

Here's an example of a simple test function in Rust:

```
#[test]
fn test_addition() {
    let result = 2 + 2;
    assert_eq!(result, 4);
}
```

In this example, the `test_addition` function tests whether the addition of two numbers results in the expected value. The `assert_eq!` macro compares the result of the addition with the expected value of 4. If the addition results in anything other than 4, the assertion will fail, and the test will fail.

To run tests in Rust, developers use the `cargo test` command, which runs all tests in a Rust project and reports the results. The cargo test command can also be used to run specific tests or groups of tests, and it provides a range of options for controlling the behavior of the testing framework.

In addition to unit tests, Rust's testing framework also supports integration tests, which test the interaction between different modules or components of a Rust application, and benchmark tests, which measure the performance of Rust code under different conditions.

Test assertions

The Rust testing framework provides macros for test assertions, such as:

- `assert!(condition)`: assert `condition` is true.
- `assert_eq!(a, b)`: assert `a` is equal to `b`.
- `assert_ne!(a, b)`: assert `a` is not equal to `b`.

Example:

```
let x = 1;
let y = 2;
assert!(x < y);
```

Example with an optional message:

```
let x = 1;
let y = 2;
assert!(x < y, "We want x to be less than y");
```

Assertables crate

The Assertables crate provides more assert macros, such as:

- `assert_starts_with!(x, y)`: Does `x` start with `y`?
- `assert_contains!(array, element)`: Does `array` contains `element`?
- `assert_is_match!(regex, string)`: Does `regex` match `string`?

Example:

```
use assertables;
let a = "hello world";
let b = "hello";
assert_starts_with!(&a, &b);
```

Unit testing

Unit testing is a software testing technique where individual software components or units are tested in isolation to ensure that they behave as expected. In Rust, unit testing involves writing tests that validate the expected behavior of functions, methods, and other individual units of code.

Rust provides a built-in testing framework for unit testing called `cargo test`.

- Unit tests are typically placed in the same file as the code they are testing. These tests should be written to validate the expected behavior of each function and method.
- Use the `#[cfg(test)]` attribute indicates that a Rust module contains tests.
- Use assertions, such as the Rust standard library `assert_eq!` assertion, or Assertables crate `assert_starts_with!` assertion.
- Unit tests in Rust can be run using the `cargo test` command. This command compiles and runs all the tests in the project, including the unit tests.
- After the tests have run, the output of the tests can be analyzed to determine whether the unit tests have passed or failed. Rust's testing framework provides detailed information about the tests that have been run, including the number of tests that have passed or failed and the reason for the failures.

By following these steps, developers can use Rust's unit testing framework to validate the behavior of individual components of the software, ensuring that each unit behaves as expected and functions correctly as part of the larger system.

Integration testing

Integration testing is a software testing technique where individual software modules are tested as a group to validate their combined functionality. In Rust, integration testing involves testing the interactions between different modules or components of the software.

Rust provides a built-in testing framework for integration testing called `cargo test`. Here are the steps involved in Rust integration testing:

- Create a separate directory for integration tests: Integration tests in Rust are typically placed in a separate directory called `tests` at the top level of the project. This directory contains Rust files that end with `_test.rs`.
- Write the integration tests: Integration tests in Rust are similar to unit tests but test the interaction between different modules or components. These tests should be written to validate the expected behavior of the system as a whole.
- Use Rust's testing framework: Rust's testing framework provides a set of macros and functions for writing and running tests. The `#[cfg(test)]` attribute indicates that a Rust module contains tests.
- Run the tests: Integration tests in Rust can be run using the `cargo test` command. This command compiles and runs all the tests in the project, including the integration tests.
- Analyze the test results: After the tests have run, the output of the tests can be analyzed to determine whether the integration tests have passed or failed. Rust's testing framework provides detailed information about the tests that have been run, including the number of tests that have passed or failed and the reason for the failures.

By following these steps, developers can use Rust's integration testing framework to validate the interactions between different modules or components of the software, ensuring that the software functions correctly as a whole.

Documentation testing

Rust doc tests are a form of Rust's testing framework that allows developers to include tests in the documentation of their code. This enables developers to write code examples and tests in the documentation itself, ensuring that the documentation remains up-to-date and accurate.

Example:

```
/// This is a document comment with a doc test.
///
/// This doc test must succeed.
///
/// ```
/// assert!(true);
/// ```
```

To run all the doc tests:

```
cargo test --doc
```

To also show warnings:

```
cargo test --doc -- --show-output
```

Rust doc tests have a variety of options to make them more powerful and more flexible.

- Annotations enable you to specify code blocks that should be ignored, or should panic, or should be compiled but not run.
- Embedded comments enable you to write code that is hidden, so your documentation is shorter and more readable.
- Trailing returns enable you to skip lengthy error handling, and instead use `?` error returns.

Documentation testing annotations

Documentation comment code blocks can use annotations with attributes that help `rustdoc` do the right thing when testing your code. Here are the annotations.

This test must panic:

```
/// ```should_panic
/// assert!(false);
/// ```
```

This test must compile, but is not run:

```
/// ```no_run
/// assert!(true);
/// ```
```

This test must fail to compile:

```
///  ```compile_fail
/// snafu
/// ```
```

This test is only for Rust 2018 edition:

```
/// ```edition2018
/// assert!(true);
/// ```
```

This code block is ignored, and not a test:

```
///  ```ignore
/// This is something else besides a test.
/// ```
```

This code block is text, and not a test:

```
///  ```text
/// Hello, World!
/// ```
```

Source-based code coverage

https://doc.rust-lang.org/rustc/instrument-coverage.html

In Rust, source-based code coverage is a way of measuring how much of a Rust codebase is executed during a test suite. This type of coverage analysis works by instrumenting the Rust code and tracking which lines of code are executed during a test run.

The process of generating source-based coverage typically involves the following steps:

- Instrumentation: The Rust code is modified to include extra code that tracks which lines of code are executed.
- Test Execution: The test suite is run against the instrumented code.
- Coverage Report Generation: The data collected during the test run generates a report that shows which lines of code were executed and which were not.

The resulting coverage report provides developers with insights into the effectiveness of their tests and helps identify areas of the code that are not being sufficiently exercised by the test suite.

One key advantage of Rust source-based coverage is that it can provide more accurate coverage measurements than alternative methods, such as binary-based coverage. This is because source-based coverage is able to account for control structures, such as branches and loops, which can lead to different paths through the code being executed.

To run unit tests with coverage:

```
RUSTFLAGS="-C instrument-coverage" cargo test --tests
```

After the tests run, there are a variety of ways to use the output files and view the coverage reports. The steps are detailed, so please see the link above for specifics.

Liskov substitution principle - example

Example:

```
trait Drawable {
    fn draw(&self);
}

fn draw_anything(drawable: &dyn Drawable) {
    drawable.draw();
}

struct Circle {
    radius: i32,
}

impl Drawable for Circle {
    fn draw(&self) {
        println!("Circle with radius {}", self.radius);
    }
}

fn main() {
    let circle = Circle { radius: 1 };
    draw_anything(&circle);
}
```

This defines a struct `Circle` and implements the `Drawable` trait. The `draw_anything` function takes any object that implements the `Drawable` interface, which means that it can accept circles or anything else that implements `Drawable`. interface. The function is an example of Liskov substitution principle in action, because any `Drawable` can be given .

rustup command-line tool

In Rust, `rustup` is a command-line tool that manages the installation and configuration of Rust toolchains. A Rust toolchain is a set of tools and libraries that are used to compile and run Rust programs.

`rustup` installs and updates Rust toolchains, including the Rust compiler and associated tools such as `cargo`. It also allows for the management of multiple toolchains and makes it easy to switch between them.

Some of the commonly used rustup commands include:

- `rustup install`: Installs a specific version of the Rust toolchain.
- `rustup default`: Sets the default Rust toolchain to use.
- `rustup update`: Updates the Rust toolchain to the latest stable release.
- `rustup self update`: Updates rustup itself to the latest version.
- `rustup component add`: Adds a component to the Rust toolchain, such as a specific target or a specific version of rustfmt.
- `rustup target add`: Adds a new target to the Rust toolchain, such as armv7-unknown-linux-gnueabihf for cross-compiling to an ARM-based Linux system.
- `rustup toolchain list`: Lists all installed Rust toolchains.
- `rustup override`: Sets a toolchain override for a specific directory or project.

`rustup` also allows for the installation of Rust-related components such as the `rust-src` component, which includes the source code for the Rust standard library, or the `rls` component, which provides support for Rust language server integration.

Overall, rustup is a powerful tool that makes it easy to manage Rust toolchains, enabling Rust developers to work with multiple versions of Rust and target different platforms.

Cargo package manager and crates

In Rust, Cargo is the package manager and build tool that creates and manages projects and their dependencies. Cargo provides ways to easily build, test, document, and publish code.

Cargo uses a file called `Cargo.toml` to manage the configuration and dependencies of a Rust project. The `Cargo.toml` file specifies the name of the package, version information, and the dependencies of the project. Cargo also provides a command-line interface that allows developers to manage their Rust projects and dependencies easily.

A cargo package is called a "crate". A crate can be a binary or a library. A binary crate is an executable program. A library crate is code that can be used by other programs.

Cargo provides a standardized directory structure for Rust projects. By convention, the main source code of a project is placed in a directory called src, and the project configuration and dependencies are specified in a file called `Cargo.toml`. Cargo uses the `Cargo.lock` file to keep track of exact dependency versions used in the project.

Cargo also provides a number of commands to manage a Rust project. Some of the commonly used commands include:

- `cargo new`: Create a new Rust project.
- `cargo build`: Build the project and its dependencies.
- `cargo run`: Build and run the project.
- `cargo test`: Run the project tests.
- `cargo doc`: Generates documentation for the project.
- `cargo publish`: Publishes a crate to the official registry.

cargo-install-favorites

https://github.com/sixarm/cargo-install-favorites

The `cargo-install-favorites` shell script is a list of our favorite Rust projects for use on our daily machines, such as enhanced command line utilties. Here are highlights.

bat: Show terminal text with highlights, git integration, fzf. Like `cat`.

bottom: A graphical process/system monitor for the terminal. Like `top`.

broot: A file manager with better ways to navigate directories. Like `tree`.

difftastic: Compare files via syntax, alignments, etc. Like `diff`.

du-dust: Show disk usage, with trees, colors, rollups, and more. Like `du`.

exa: Examine file lists, with colors, attributes, git awareness. Like `ls`.

fd-find: A simple, fast and user-friendly alternative to Unix `find`.

gitui: Blazing fast terminal user interface for git. Like `git`, `gitk`.

gping: Graphical ping network tracer, plus multiple hosts. Like `ping`.

helix: Terminal text editor, with modern capabilties built-in. Like `vim`.

just: Command runner for project-specific tasks. Like `make`.

procs: Monitor system processes, with colors, search, extras. Like `ps`.

ripgrep: Fast flexible regular expression text search tool. Like `grep`.

starship: Fast, minimal, infinitely customizable shell prompt.

watchexec-cli: Watch files for modifications then execute commands.

zellij: Terminal workspace with batteries included. Like `screen`, `tmux`.

zoxide: A faster way to navigate your directories. Like `cd`, `jump`.

Blessed recommendations

https://blessed.rs

Blessed is an unofficial guide to the Rust ecosystem. New Rust developers frequently ask which tools and crates to use and trust.

Blessed aims to answer these questions with listings such as:

- General purpose: regex, Serde, tempfile, etc.
- Developer tooling: Clippy, Criterion, cargo-release, etc.
- Language extensions: itertools, once_cell, syn, etc.
- HTTP services: axum, hyper, reqwest, etc.
- Databases: diesel, rusqlite, sqlx, etc.
- CLIs: clap, walkdir, ratatui, etc.
- Concurrency: flume, parking_lot, rayon, etc.
- Graphics: egui, gtk4, etc.

Clippy linting

Rust Clippy is a popular linting tool for Rust that provides additional static analysis to help catch bugs and improve code quality. It is an external tool that runs alongside the Rust compiler and analyzes Rust code to check for common programming errors, style issues, and other potential problems.

Clippy is built on top of Rust's existing linting infrastructure and provides additional lints that are not included in the standard library. These lints are organized into several categories, including:

- Correctness: These lints check for potential errors that can cause undefined behavior, such as null pointer dereferences, out-of-bounds array access, and other common issues.
- Style: These lints check for coding style issues, such as using inconsistent indentation, unnecessary parentheses, and redundant code.
- Performance: These lints check for potential performance issues, such as using slow algorithms or redundant calculations.
- Complexity: These lints check for overly complex code, such as deeply nested functions or overly complicated expressions.
- Security: These lints check for potential security vulnerabilities, such as buffer overflows, unsafe code, and other issues.

Clippy is highly customizable, allowing developers to enable or disable specific lints, customize the severity level of lints, and even create custom lints tailored to their specific needs. It is also regularly updated with new lints and improvements, making it a valuable tool for improving Rust code quality and preventing bugs.

Helix text editor

https://helix-editor.com/

Helix is terminal-based text editor written in Rust, with excellent capabilities for programming in Rust. Helix is inpsired by Neovim and Kakoune, and is similar in ways to vim, emacs, and nano.

Key benefits:

- Multiple selections as a core editing primitive, so commands can manipulate selections, which allows concurrent code editing.
- Tree-sitter integration, which enables better syntax highlighting, indent calculation, and code navigation.
- Powerful code manipulation to navigate and select functions, classes, comments, etc and select syntax tree nodes instead of plain text.
- Language server support provides language-specific auto completion, goto definition, documentation, diagnostics and other IDE features with no additional configuration.
- Built in Rust, for the terminal. No Electron. No VimScript. No JavaScript. Use it over ssh, tmux, or a plain terminal. Your laptop battery life will thank you.
- Modern features such as fuzzy finder to jump to files and symbols, project wide search, beautiful themes, auto closing bracket pairs, surround integration and more.

Rustfmt - examples

Rustfmt as a standalone tool

You can use Rustfmt directly from the command line:

```
rustfmt <filename.rs>
```

This command will format the Rust code in the specified file and print the formatted output to the terminal. If you want to save the formatted output to a file, you can use the -w option followed by the filename, like this:

```
rustfmt -w <filename.rs>
```

Rusfmt within a code editor

You can use Rustfmt within a code editor such as vim, emacs, Helix, and VSCode. To do this, you install a Rustfmt extension for your editor, then configure it to format your code on save or on demand.

For example, in VSCode, you can install the “Rustfmt” extension and configure it to format your code on save by adding the following line to your `settings.json` file:

```
"editor.formatOnSave": true
```

Rustfmt via a build script

You can use Rustfmt as a step of your build process, before compiling it. One way to do this is to create a build script by adding the following line to your `Cargo.toml` file:

```
[package]
build = "rustfmt <filename.rs>"
```

Rust mdBook for documentation

Rust mdBook is a tool for creating and publishing documentation in the form of books or websites. Rust mdBook is designed for documenting Rust projects, but it can be used for any kind of documentation. Rust mdBook supports various Markdown features and more, including syntax highlighting for code blocks, table of contents generation, cross-referencing between pages, customizable themes, and documentation by using Rustdoc comments.

To install the `mdbook` tool and the `mdbook-pdf` tool:

```
cargo install mdbook
cargo install mdbook-pdf
cargo install mdbook-toc
```

To use Rust mdBook, you create a book directory that contains the Markdown files and any associated assets, such as images or code samples. You can then use the mdBook command-line tool to compile the book into the desired format. The resulting output can be published as a website or distributed as an eBook or PDF.

To use Rust mdBook PDF, you may need to install additional software, such as a web browser that can render PDF. Rust mdBook PDF has installation options to automatically download and install the Chromium web browser, which can render PDF. See the Rust mdBook PDF documentation for more information.

To use Rust mdBook TOC (table of contents), you can use the default markup `<!-- tod -->`, then the build will automatically generate a table of contents. See the rust mdBook TOC documentation for more information.

Overall, Rust mdBook makes it easy to create high-quality documentation that is easy to read and understand.

Cross-compiling for multiple platforms

Cross-compiling is the process of compiling code for a platform different from the one on which the code is compiled.

Rust supports cross-compiling, which means that you can compile Rust code on one platform and generate executable code for another platform, such as Windows, Linux, or macOS.

To cross-compile Rust code, you need to install a cross-compiler toolchain for the target platform. This toolchain includes the Rust compiler, standard library, and any other dependencies required to build the code. You can install cross-compilers for different architectures using Rust's built-in tool, rustup.

Once the cross-compiler toolchain is installed, you can use the cargo command to build your Rust project for the target platform. You can specify the target platform by setting the –target option when running the cargo build or cargo run command.

For example, to build a Rust project for the ARM architecture, you would use the following command:

```
cargo build --target=arm-unknown-linux-gnueabihf
```

This command tells cargo to build the project for the ARM architecture using the GNU toolchain and the Hard Float ABI.

Cross-compiling Rust code can be useful for a variety of scenarios, such as building applications for embedded systems or developing software that needs to run on multiple platforms. Rust's strong type system and memory safety guarantees make it a good choice for writing cross-platform applications that require high performance and reliability.

Rhai script

Rhai is an embedded scripting language for Rust. Rhai is a dynamically typed language with support for high-level data types such as arrays, maps, and functions. Rhai supports Rust-style ownership and borrowing, making it easy to integrate with Rust's memory management.

One of the key features of Rhai is its safety and security. Rhai enforces sandboxing by default, which means that scripts executed within a Rhai interpreter cannot access or modify the host environment. Rhai also supports a variety of security features such as timeouts, memory limits, and access controls to ensure that scripts are safe to use.

Rhai's syntax is similar to Rust's syntax, making it easy for Rust developers to learn and use. Rhai also provides a number of built-in functions and operators that simplify common scripting tasks such as string manipulation, math operations, and control flow.

Example of using Rust as an embedded language in Rhai script:

```
use rhai::{Engine, EvalAltResult};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut engine = Engine::new();

    let result = engine.eval::<i32>("2 + 2")?;
    println!("2 + 2 = {}", result); // should print 4

    let result = engine.eval::<f64>("3.14 * 2.0")?;
    println!("3.14 * 2.0 = {}", result); // should print 6.28

    let result = engine.eval::<i32>("10 / 3")?;
    println!("10 / 3 = {}", result); // should print 3

    Ok(())
}
```

In this example, the Rhai script evaluates arithmetic expressions, and Rust performs the actual calculations. This combines Rhai's dynamic code and Rust's strong typing and optimized performance.

Abstract syntax tree (AST)

An abstract syntax tree (AST) is a data structure used in computer science to represent the structure of a program in a way that can be easily analyzed and manipulated by algorithms.

An AST is created by analyzing the source code of a program and breaking it down into a tree-like structure that represents its syntax.

Each node in the tree represents a syntactic construct in the program, such as a function call, a variable declaration, or an if statement. The nodes in the tree are connected by edges that represent the relationships between the constructs.

The main advantage of using an AST is that it provides a way to analyze the program's structure and behavior without having to execute the code. This makes it possible to perform tasks such as code optimization, program transformation, and error detection without having to actually run the program.

ASTs are commonly used in compilers, interpreters, and other tools that analyze or manipulate source code. For example, a compiler may use an AST to perform optimizations such as dead code elimination or loop unrolling, while a static analyzer may use an AST to detect potential security vulnerabilities or other code quality issues.

Overall, abstract syntax trees are a powerful tool for working with programs, allowing developers to reason about their structure and behavior in a way that is both precise and efficient.

Tree-sitter parsing library

Tree-sitter is a parsing library that allows developers to create robust and efficient parsers for programming languages, configuration files, and other structured documents. It was created by Rasmus Andersson and is written in Rust.

The library uses the tree-sitter parsing algorithm, which is a powerful parsing technique that builds an abstract syntax tree (AST) for the parsed code. The AST is a tree structure that represents the structure of the code, making it easier to analyze and manipulate.

One of the key advantages of Rust tree-sitter is its speed and efficiency. It is designed to be extremely fast, allowing it to handle large codebases and parse files in real-time. It also uses incremental parsing, which means that it can efficiently update the AST as changes are made to the code.

Rust tree-sitter is also highly modular, with a simple and flexible API that allows developers to easily create custom parsers for new languages or modify existing parsers. It supports a wide range of programming languages, including C, C++, Java, Python, Ruby, and many others.

Overall, Rust tree-sitter is a powerful and flexible parsing library that can be used to create high-performance parsers for a wide range of programming languages and structured documents.

Language Server Protocol (LSP)

Language Server Protocol (LSP) is a communication protocol between an editor or an IDE and a language server that provides language-specific features such as code completion, error checking, and symbol search.

The Language Server Protocol is used by many popular editors and IDEs, and is supported by many programming languages.

Using the Language Server Protocol, editors and IDEs can provide consistent language features across multiple programming languages and language servers, without having to implement language-specific functionality themselves. This allows for faster and more efficient development, as developers can use their preferred editor or IDE and still have access to advanced language features.

The LSP defines a set of standard JSON-RPC methods that the client and server can use to communicate. These methods include:

- `initialize`: Iinitialize the language server and configure it.
- `shutdown`: Shut down the language server.
- `textDocument/didOpen`: Notify when a document is opened.
- `textDocument/didChange`: Notify when a document is modified.
- `textDocument/completion`: Request code completion suggestions.
- `textDocument/hover`: Request information about a symbol.
- `textDocument/references`: Request references to a symbol.

The Language Server Protocol is an open standard. The protocol is implemented in a client-server architecture, where the client is an editor or IDE that supports the LSP, and the server is a language server that provides language-specific functionality.

Static analysis for error detection

Static analysis is the process of analyzing code without executing it, to detect potential errors or issues before the code is actually run. Rust has a strong focus on static analysis, with the goal of catching as many errors as possible at compile time, before the code is even executed.

Rust's static analysis features include:

- Static typing: Rust is a statically typed language, meaning that the type of a variable is known at compile time. This helps catch many common errors, such as trying to add a string and a number, before the code is even run.
- Ownership and borrowing: Rust's ownership and borrowing system helps prevent memory errors such as null pointer dereferences or use-after-free bugs. The compiler enforces rules around how references to data are created, modified, and used, to ensure that they are safe and sound.
- Lifetimes: Rust's lifetime system helps ensure that references to data are valid for as long as they are needed. This prevents common errors such as dangling pointers or double frees.
- Macros: Rust's macro system allows developers to write code that generates other code at compile time. This can be used to perform custom static analysis or generate repetitive code automatically.
- Clippy: Clippy is a community-maintained linter for Rust that provides additional static analysis checks beyond what the compiler itself does. Clippy checks for common coding mistakes, such as unused variables, and provides suggestions for how to fix them.

Overall, Rust's strong focus on static analysis helps catch many errors before they occur, reducing the likelihood of bugs and making it easier to write safe and reliable code.

Design patterns: introduction

Design patterns in programming refer to reusable solutions to common problems that arise during software development. These patterns provide a standard set of practices, templates and a recommended course of action for solving recurring problems. They are proven solutions that help developers to build software that is more modular, maintainable, and scalable.

Design patterns are used by software developers to ensure that the code follows best practices while tackling common problems. They are grouped into three categories: creational, structural, and behavioral.

Creational patterns are used to create objects and instances of classes during runtime. Structural patterns are aimed at developing the overall structure of the code, while behavioral patterns are used to manage communication between object instances.

Using a well-defined design pattern allows developers to focus on the software's functionalities rather than the design aspects of the code. Some of the commonly used design patterns in programming include Singleton, Observer, Decorator, Facade, Adapter, Iterator, Builder, and many more.

As one example, the Iterator design pattern provides a way to iterate over a collection of objects. In Rust, this is built into the language with the Iterator trait.

Example:

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

Design patterns: adapter

The “adapter” structural design pattern enables incompatible interfaces to collaborate. This can be implemented using a adapter struct that wraps an adaptee struct.

```
// Suppose we have a typical struct that we want to adapt.
// This struct is typically know as the "adaptee".
struct CircleWithRadius {
    radius: f32;
}

// The adapter structural design pattern typically means
// we define an outer struct that wraps an inner struct.
// The outer struct is typically known as the "adapter".
// The inner struct is typically known as the "adaptee".
struct CircleWithDiameter {
    adaptee: CircleWithRadius;
}

// We implement the adapter methods, such as these accessors,
// so the methods actually get and set the adaptee's data.
// This is similar to a proxy object, or to a facade object.
impl CircleWithDiameter {
    fn diameter(&self) -> f32 {
        adaptee.radius * 2;
    }

    fn set_diameter(&self, diameter: f32) {
        adaptee.radius = diameter / 2;
    }
}
```

Design patterns: builder

The “builder” design pattern creates complex objects via simpler steps. This can be implemented using a struct with setter methods.

```
struct Foo {
    a: i32,
    b: i32,
}

struct FooBuilder {
    a: Option<i32>,
    b: Option<i32>,
}

impl FooBuilder {
    fn new() -> Self {
        FooBuilder {
            a: None,
            b: None,
        }
    }

    fn a(mut self, a: i32) -> Self {
        self.a = Some(a); self
    }

    fn b(mut self, b: u32) -> Self {
        self.b = Some(b); self
    }

    fn build(self) -> Foo {
        Foo {
            a: self.a.expect("missing field a"),
            b: self.b.expect("missing field b"),
        }
    }
}

let foo = FooBuilder::new().a(1).b(2).build();
```

Design patterns: observer

The “observer” design pattern enables one object to notify others of its state changes. This can be implemented using Rust’s channels or event emitters.

Example:

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
    let (tx, rx) = channel();

    thread::spawn(move || {
        tx.send("Hello, World!").unwrap();
    });

    let message = rx.recv().unwrap();
    println!("{}", message);
}
```

Design patterns: singleton

The “singleton” design pattern ensures that only one instance of a particular object is ever created. This can be implemented using a static variable or a lazy static variable.

Example:

```
struct Singleton;

impl Singleton {
    fn instance() -> &'static Self {
        static mut INSTANCE:
            *const Singleton = 0 as *const Singleton;
        static ONCE: Once = Once::new();
        unsafe {
            ONCE.call_once(|| {
                let singleton = Singleton {};
                INSTANCE = mem::transmute(Box::new(singleton));
            });

            &*INSTANCE
        }
    }
}
```

cargo-cache crate for caching builds

https://crates.io/crates/cargo-cache

The Rust cargo-cache crate provides a command-line interface (CLI) for managing the cache directory used by the Cargo package manager.

When you use Cargo to build a Rust project, it downloads and caches dependencies, build artifacts, and other files related to the build process in a directory called "cargo-cache". Over time, this directory can become quite large, taking up valuable disk space on your system.

The `cargo-cache` crate provides several commands that allow you to manage the cache directory. Some of the key features:

- Listing the contents of the cache directory
- Clearing the cache directory
- Showing the size of the cache directory
- Displaying information about individual cached packages

Example listing:

```
Cargo cache '~/.cargo':
Total:                                4.41 GB
  75 installed binaries:            481.75 MB
  Registry:                           3.92 GB
    Registry index:                 503.26 MB
    2563 crate archives:            403.60 MB
    2563 crate source checkouts:      3.02 GB
  Git db:                             2.67 MB
    1 bare git repos:               905.51 KB
    1 git repo checkouts:             1.77 MB
```

Using `cargo-cache`, you can easily clear out old or unnecessary cached files, reclaiming valuable disk space on your system. You can also use the `cargo-cache` CLI to better understand the contents of the cache directory and diagnose any issues related to the build process.

cargo-crev for community-driven trust

https://crates.io/crates/cargo-crev

The Rust cargo-crev crate helps developers build a community-driven trust system for their packages. The crate provides a way for developers to create signed reviews of their dependencies and share them with other developers. These reviews can include information about the quality of the code, how well the documentation is written, and any security concerns.

The idea behind cargo-crev is to create a trusted network of developers who can vouch for the quality and safety of each other's code. This can help prevent malicious packages from being added to the Rust ecosystem and can provide a sense of security for developers who rely on Rust packages in their projects.

Using cargo-crev, developers can create public or private reviews of their dependencies, and other developers can use these reviews to make informed decisions about which packages to use in their projects. The crate also provides a command-line interface that makes it easy to manage reviews and share them with the community.

Key features:

- Build a web of trust of users to help verify the code you use
- Warn you about untrustworthy crates and security vulnerabilities
- Increase trustworthiness of your own code

cargo-dist crate for distribution archives

https://crates.io/crates/cargo-dist

The Rust cargo-dist crate is a Rust crate that provides a simple and convenient way to package a Rust project as a distributable archive. The crate is designed to work with the Rust cargo build system, and provides a number of features that make it easy to create archives for various platforms.

One of the main features of `cargo-dist` is its support for cross-compiling. The crate can automatically build and package your Rust project for a number of different platforms, including Windows, macOS, Linux, and Android, all from a single command. This can save a lot of time and effort when distributing your project to users on multiple platforms.

Another useful feature of `cargo-dist` is its support for packaging dependencies. When you create a distributable archive with `cargo-dist`, it will automatically include all of the dependencies for your Rust project, so users don't have to manually install them. This can help simplify the installation process for your project and reduce the risk of dependency conflicts.

Finally, `cargo-dist` provides a number of options for customizing the packaging process. You can specify the format of the archive (e.g. `.tar.gz`, `.zip`, etc.), include or exclude specific files or directories, and more. This can help ensure that the distributable archive contains exactly what you want, and nothing more.

cargo-release crate - examples

The cargo-release crate provides many features and functions, including these examples.

Release Management: The `cargo-release` crate provides a range of tools for managing the release process, including the ability to automatically generate a new version number based on a specified release type (e.g. major, minor, or patch), update the changelog and version number in your crate's Cargo.toml file, tag the release in Git, and publish the crate to crates.io:

```
cargo release --dry-run  # preview the release process
cargo release            # perform the release
```

Pre-Release Management: The `cargo-release` crate also provides tools for managing pre-releases, including the ability to create and publish pre-release versions of your crate (e.g. 0.2.0-alpha.1), and to promote pre-release versions to stable releases:

```
cargo release --pre-release  # create a pre-release version
cargo release --continue     # promote a pre-release to stable
```

Customization: The `cargo-release` crate is highly configurable, allowing you to customize the release process to suit your needs. For example, you can specify which branches to release from, configure the changelog format and location, and specify additional steps to perform during the release process:

```
[release]
branches = ["main"]
changelog = "docs/CHANGELOG.md"
pre-release = false

[release.steps.post]
## ... additional steps to perform after the release ...
```

cargo-make crate - example

Here's an example `cargo-make` configuration file `Makefile.toml`:

```
[tasks.build]
command = "cargo build --release"

[tasks.test]
command = "cargo test"

[tasks.lint]
command = "cargo clippy"

[tasks.default]
dependencies = ["build", "test", "lint"]
```

In this example, we've defined three tasks: `build`, `test`, and `lint`. Each task has a command that specifies what action to perform when the task is executed. The `default` task depends on the `build`, `test`, and `lint` tasks, and is executed when no task is specified.

You can then run your build process using the following command:

```
cargo make
```

This will execute the default task and all its dependencies in the correct order.

If you want to execute a specific task, you can use the following command:

```
cargo make <task-name>
```

Criterion crate - example

Add to your file `Cargo.toml`:

```
[dev-dependencies]
criterion = { version = "0.4", features = ["html_reports"] }

[bench]
name = "my_benchmark"
harness = false
```

Create a file `$PROJECT/benches/demo.rs` with this code:

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

fn fibonacci(n: u64) -> u64 {
    match n {
        0 | 1 => 1,
        n => fibonacci(n-1) + fibonacci(n-2),
    }
}

fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function(
        "fib 20",
        |b| b.iter(|| fibonacci(black_box(20)))
    );
}

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

Run this benchmark with `cargo bench`. You should see output like below.

```
fib 20   time:   [26.029 us 26.251 us 26.505 us]
Found 11 outliers among 99 measurements (11.11%)
  6 (6.06%) high mild
  5 (5.05%) high severe
```

Rust governance

https://www.rust-lang.org/governance

Rust governance refers to the system in place for managing and directing the development and maintenance of the Rust programming language. Rust governance is characterized by the following key components:

- RFC process: The Rust RFC (Request for Comments) process is how changes are proposed and accepted into Rust. An RFC is a document proposing a significant change or new feature to Rust, which is rigorously reviewed and discussed by the community before it is accepted or rejected.
- Core team: The Rust core team is responsible for the overall strategy and direction of Rust development, managing the Rust project's infrastructure, and overseeing various Rust teams.
- Teams: There are many different teams within the Rust community that work on specific areas of Rust development, such as the compiler, Cargo (the Rust package manager), documentation, and community outreach. Each team has a leadership structure and is responsible for driving the development of their respective area.
- Working groups: There are a working groups of contributors focused on specific aspects of Rust development, such as asynchronoous programming, command-line interfaces, embedded devices, security responses, and WebAssembly.
- Stewardship: The Rust community has a concept of "stewardship," in which individuals or groups take responsibility for specific areas of Rust development. Stewards are responsible for ensuring that their area of Rust is maintained and developed in a way that aligns with the overall goals and values of Rust.

The Rust Foundation

The Rust Foundation is a nonprofit organization that was formed in 2020 to manage the development and direction of the Rust programming language.

The Rust Foundation aims to provide an official structure for the Rust community, protect the intellectual property of the language, and oversee the development of Rust. The foundation will allow Rust to be more widely adopted in industry settings, and provide financial support for key initiatives and infrastructure projects.

The Rust Foundation is governed by a board of directors, made up of individuals and organizations that have an interest in the success of Rust. These directors are responsible for setting the direction of the foundation, managing financial resources, and ensuring the continued success of the Rust programming language.

Link: https://foundation.rust-lang.org/

Community Grants Program

The Rust Foundation runs a grants program for the Rust community. These grants are complementing the existing ecosystem of funding for the Rust community, not replacing it. The objectives are to support the maintainers of Rust, and to support and grow the community of Rust users.

Grants include event support grants, Rust Foundation Fellowships, project grants for discrete pieces of work, and more.

Link: https://foundation.rust-lang.org/grants/

The Rust RFC process

https://rust-lang.github.io/rfcs/

The Rust RFC (Request For Comments) process is the formal mechanism for proposing, discussing, and deciding on changes to Rust. The process is intended to be transparent, collaborative, and community-driven, and it involves several stages.

1. RFC submission: Anyone can submit an RFC by creating a new issue on the Rust GitHub repository, describing the proposed change, its motivation, potential impact, and alternative solutions.
2. Initial triage: The Rust team evaluates its feasibility, alignment with the language's vision and goals, and potential impact. If the RFC is deemed appropriate for further consideration, it is assigned a tracking issue number and enters the "active" state.
3. Community feedback: The RFC is then open for community feedback and discussion on GitHub. The community can provide comments, suggestions, concerns, and questions, which the RFC author should address and update the RFC accordingly.
4. FCP (Final Comment Period): If there's consensus on the proposed change, then the RFC enters the Final Comment Period (FCP). Stakeholders review and incorporate the feedback, evaluate the consensus, and decide whether to accept, reject, or postpone the RFC.
5. Implementation: If an RFC is accepted, it is assigned to a team or an individual to implement it. The implementation is done on a separate branch or fork of the Rust repository, and it is subject to code review, testing, and community feedback.
6. Merge: Once the implementation is deemed stable, it is merged into the Rust main branch, and becomes part of the next Rust release.

The Rust roadmap

The Rust roadmap is a high-level plan that outlines the goals, priorities, and direction of the Rust programming language for the next several years. The roadmap is developed by the Rust core team and is informed by community feedback, user needs, and technical challenges.

The Rust roadmap is divided into three main categories:

- Language: This category includes improvements to the Rust language itself, such as adding new syntax, improving performance, and simplifying usage. Some of the specific goals in this category include stabilizing async/await, improving const generics, and adding better support for embedded systems.
- Ecosystem: This category includes improvements to the Rust tooling, libraries, and community, such as making it easier to use Rust for web development, improving the Rust packaging and distribution system, and enhancing the Rust documentation and learning resources.
- Community: This category includes initiatives to make the Rust community more diverse, inclusive, and welcoming, such as improving the Rust governance and decision-making process, promoting Rust education and outreach, and supporting the Rust community events and conferences.

The Rust roadmap is not a fixed plan, and it is subject to change based on feedback and new developments. The Rust core team periodically updates the roadmap to reflect new priorities, challenges, and opportunities. Developers who want to contribute to Rust or use Rust for their projects can consult the roadmap to understand the direction and focus of the language and the community.

About the author

I'm Joel Parker Henderson. I'm a software developer and writer.

https://linkedin.com/in/joelparkerhenderson

https://github.com/joelparkerhenderson

Professional

For work, I consult for companies that seek to leverage technology capabilities and business capabilities, such as hands-on coding and growth leadership. Clients range from venture capital startups to Fortune 500 enterprises to nonprofit organziations.

For technology capabilities, I host repositories for developers who work with architecture decision records, functional specifications, system quality attributes, git workflow recommendations, monorepo versus polyrepo guidance, and hands-on code demonstrations.

For business capabilities, I host repostitories for managers who work with objectives and key results (OKRs), key performance indicators (KPIs), strategic balanced scorecards (SBS), value stream mappings (VSMs), statements of work (SOWs), and similar practices.

Personal

I'm a strong believer in free libre open source software (FLOSS). I'm an avid traveler and enjoy getting to know new people, new places, and new cultures. I love music and play guitar.

I advocate for chariable donations to help improve our world. Some of my favorite charities are Apache Software Foundation (ASF), Electronic Frontier Foundation (EFF), Free Software Foundation (FSF), Amnesty International (AI), Center for Environmental Health (CEH), Médecins Sans Frontières (MSF), and Human Rights Watch (HRW).

About the ebook PDF

The Rust Quick Guide ebook PDF is generated from the repository markdown files. The process uses custom book build tools, fonts thanks to Adobe, our open source tools, and the program `pandoc`.

Book build tools

The book build tools are in the repository, in the directory `book/build`. The tools select all the documentation links, merge all the markdown files, then process everything into a PDF file.

Book fonts

The book fonts are Source Serif Pro, Source Sans Pro, and Source Code Pro. The fonts are by Adobe and are free open source.

markdown-text-to-link-urls

https://github.com/sixarm/markdown-text-to-link-urls

This is a command-line parsing tool that we maintain. The tool reads markdown text, and outputs all markdown link URLs. We use this to parse the top-level file `README.md`, to get all the links. We filter these results to get the links to individual guidepost markdown files, then we merge all these files into one markdown file.

pandoc-from-markdown-to-pdf

https://github.com/sixarm/pandoc-from-markdown-to-pdf

This is a command-line pandoc tool that we maintain. The tool providers our preferred pandoc settings in order to convert from an input markdown text file to an output PDF file. The tool adds a table of contents, loads our preferred fonts, configures source code syntax highlighting, sets the page size and margins, and more.

About related projects

Several Rust projects by the same author may be helpful.

Demo Rust Axum

https://github.com/joelparkerhenderson/demo-rust-axum

This project demonstrates Rust and the axum web framework. The project also shows tower for clients and servers, hyper for HTTP, tokio for asynchronous I/O, and Serde for serialization/deserialization.

Assertables Rust crate

https://github.com/sixarm/assertables-rust-crate

The Assertables Rust crate provides many assert macros, such as `assert_starts_with`, `asserts_contains`, and `asserts_is_match`. There are also macros for testing arrays and vectors, function results, readers and streams, and more.

Collectables Rust crate

https://github.com/sixarm/collectibles-rust-crate

The Collectables Rust crate provides helpers for standard library collections. The crate provides two general-purpose collections helpers: `BTreeMapToSet` based on `BTreeMap` and `BTreeSet`, and `HashMapToSet` based `HashMap` and `HashSet`.

checkline Rust crate

https://github.com/sixarm/checkline-rust-crate

The `checkline` crate is a command-line checkbox line picker: it reads lines from stdin, prompts the user with a checkbox per line, then outputs lines to stdout. This crate is a good introduction to the Cursive TUI crate.