**Top Rust Libraries**

[[](https://serokell.medium.com/?source=post_page-----eaca032a7b72--------------------------------)](https://serokell.medium.com/?source=post_page-----eaca032a7b72--------------------------------)

[Serokell](https://serokell.medium.com/?source=post_page-----eaca032a7b72--------------------------------)

·

Follow

13 min read

·

Nov 25

167

1



The Rust programming language has been gaining more supporters in the developer community thanks to its focus on performance, safety, concurrency, and a rich ecosystem of libraries.

In this blog post, we offer a concise overview of some noteworthy Rust libraries, selected based on their functionality and popularity among developers.

**What is a Rust library?**

In Rust, a library, commonly termed as a “crate,” provides precompiled routines (functions), scripts, and modules, allowing programmers to avoid redundant implementations. There are two primary types of crates: Binary and Library. The default prefix for a library is “lib.” You can change it by writing specific attributes or compiler options.

For a Binary crate, Cargo automatically generates a Cargo.toml file when creating a new project. The main entry point for this Binary crate is located at src/main.rs. In the case of a Library crate, the root is found in src/lib.rs within its corresponding package.

**How to use a Rust library?**

Using a Rust library involves the following steps:

1. **Choose a library**

First, identify the library that you want to use. You can search for Rust libraries on [crates.io](https://crates.io/), which is the official crate registry for Rust.

**2. Add dependency to Cargo.toml**

Once you’ve identified the crate, add it to your project by including it in the [dependencies] section of your Cargo.toml file, for instance:

You can use semver notation to specify a version range. See [Cargo documentation](https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html) for more information.

**3. Fetch and compile dependency**

When you run cargo build or cargo run, Cargo, Rust's package manager, will fetch the crate and its dependencies from [crates.io](http://crates.io/) and compile them as part of your project.

**4. Use the library in your code**

After adding the dependency, you can use the library in your Rust code by bringing it into scope with the use keyword.

**5. Consult the documentation**

Each crate typically has its own documentation that provides information about how to use its functionality. You can often find this documentation linked from the crate’s page on [crates.io](http://crates.io/) or by searching for the crate’s name followed by “Rust docs” in a search engine.

**6. Update the library**

Over time, libraries may receive updates that offer new features, bug fixes, or security patches. You can update a library by changing its version in Cargo.toml and running cargo update. Cargo will then resolve the latest compatible version and update your Cargo.lock file accordingly.

**Are Rust libraries cross-platform?**

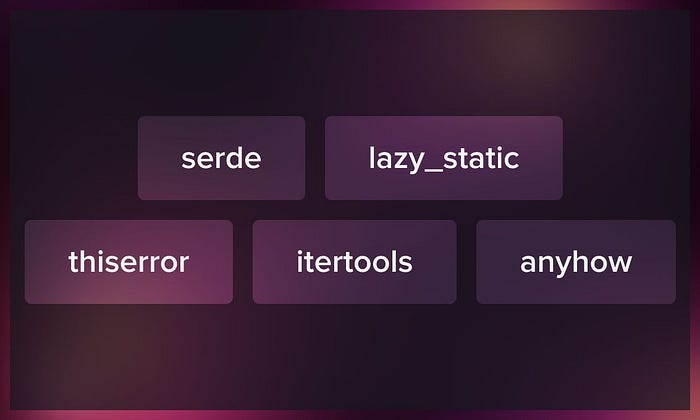
Rust libraries often aim to be cross-platform, with the language emphasizing cross-platform compatibility. However, a library’s cross-platform nature depends on its purpose, dependencies, any platform-specific features it might have, and how regularly it’s tested across platforms. While Rust’s standard library supports a consistent API across systems, it also contains platform-specific modules. In practice, many Rust libraries are cross-platform, but it’s essential to consult a library’s documentation and repository for specific platform support details.

**Is Rust memory-safe?**

Rust is designed for memory safety. Yet, it also includes an ‘unsafe’ programming feature, highlighted by the unsafe keyword. When you see unsafe, it indicates that developers and reviewers need to manually check that the code follows certain contracts. Not doing so can result in memory vulnerabilities and crashes in those specific sections.

Now that we’re done with the basics, let’s explore some of the most popular Rust libraries that can help you in your work.

**5 most popular Rust libraries**



In this section, we introduce libraries with the highest number of downloads and the most helpful features, in our opinion. Further sections also contain popular crates, but we have included some that may be less popular but are worth paying attention to due to their functionality.

**Serde**

[Serde](https://serde.rs/) is *the* crate for data serialization and deserialization, and the name is actually composed of the initial letters of these two words. It’s very popular, with more than 216 million downloads and more than 270 versions, so you’re likely to find assistance with almost any issue. Data formats supported by Serde include JSON, YAML, TOML, Pickle, BSON, and much more. Additionally, Serde offers a derive macro that allows you to automatically generate serialization methods for structs within your application. It can produce implementations for a majority of structs and enums, even those with complex generic types or trait limitations.

**Lazy\_static**

[Lazy\_static](https://docs.rs/lazy_static/latest/lazy_static/) is the crate for lazy static data initialization. Static variables in Rust are normally initialized at compile time, but there are cases where you might want to initialize them at runtime (e.g. if you want to do runtime computations during initialization). They also ensure that the initialization only happens once. The lazy\_static macro provides a way to handle this.

Lazy\_static offers several advantages when working with static variables in Rust. Firstly, it employs “Lazy Initialization.” This means that the static variable is only initialized when accessed for the first time. Such an approach can boost performance and sidestep potential issues related to initialization order. Secondly, it guarantees “Thread Safety.” If multiple threads attempt to access the variable simultaneously during the initialization phase, lazy\_static ensures that the variable is initialized just once, preventing any race conditions or redundant operations.

**Thiserror**

[Thiserror](https://crates.io/crates/thiserror) is a utility crate in Rust designed to streamline the creation of custom error types. It complements the std::error::Error trait from Rust's standard library by eliminating the need for repetitive boilerplate code. With thiserror, developers can effortlessly derive the Error trait for their custom error types. For instance, consider the a enum representing various error types. Using the #[error(...)] attribute, you can define how each error variant is displayed, and thiserror will derive std::fmt::Display and std::error::Error trait implementations automatically. It is also possible to use the #[from] attribute to automatically derive std::convert::From implementation.

In essence, thiserror enhances error handling and reporting, making it more concise and user-friendly in Rust applications.

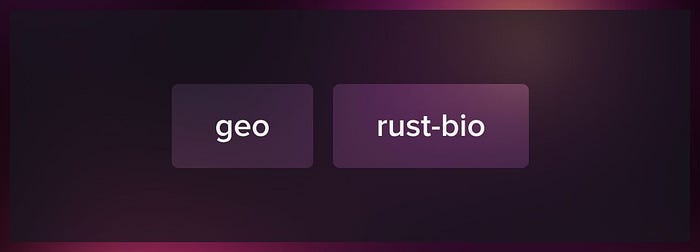
**Itertools**

[Itertools](https://docs.rs/itertools/latest/itertools/) is a Rust crate that provides additional iterator traits, functions, and adaptors extending the capabilities of Rust’s standard iterators. It offers utility functions that allow for more complex and specialized iterator operations that aren’t available in the standard library, for example, interleave, unique, batching, and group\_by. Beyond these, itertools also offers special iterator adaptors such as multi\_peel, tuple\_combinations, and kmerge. In a sense, itertools is to Rust what itertools (the module with the same name) is to Python, in that both provide a set of utilities that make it easier to work with iterators in their respective languages.

**Anyhow**

The [Anyhow](https://docs.rs/anyhow/latest/anyhow/) crate in Rust streamlines error handling. The best way to describe what anyhow does is by comparing it with dyn Error. dyn Error is a vanilla "unified error type," but it has to be boxed as a dyn type. This can make it somewhat cumbersome to work with, and it's represented using a "fat pointer," which occupies twice the memory size of a regular pointer. Anyhow errors don't need to be boxed and sidestep the "fat pointer" issue. Additionally, anyhow has utilities for enriching errors with context information. So whenever you want to use dyn Error, Anyhow is a better alternative.

**Scientific Rust libraries**



From geospatial calculations to bioinformatics solutions, these Rust libraries can help researchers in a variety of tasks.

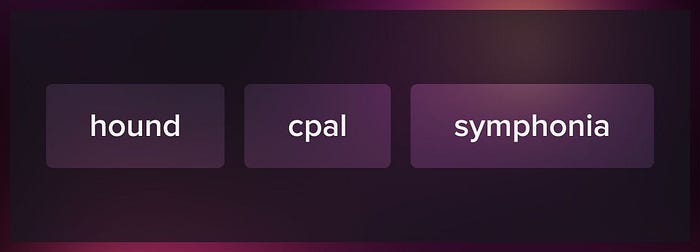
**Geo**

With more than 120k downloads per month, the [geo](https://lib.rs/crates/geo) crate offers geospatial types like Point, LineString, and Polygon, and includes various operations: area calculation, distance measurement, and affine transformations. Additional functionalities encompass coordinate transformations, serialization to GeoJSON and WKT, geocoding, and GPS data handling. Contributions to the library can be licensed under either the Apache License, Version 2.0 or the MIT license.

**Rust-bio**

The [Rust-bio](https://github.com/rust-bio) library offers algorithmic and data structural solutions for bioinformatics. All features undergo thorough testing via continuous integration. For those contributing, it’s recommended to use the pre-commit tool to ensure code quality. There are two ways to submit pull requests: one-time contributors can fork and submit changes, while frequent contributors can join the rust-bio team by introducing themselves on [Discord](https://discord.com/invite/rust) and then push changes directly. You are required to provide documentation for every public function/module, specifying content and format. The library currently supports Rust version 1.64.0 and is licensed under the MIT license.

**Audio and sound**



The libraries in this section will streamline audio and sound software development.

**Hound**

[Hound](https://crates.io/crates/hound)is a Rust-based library for encoding and decoding in the WAV audio format, commonly used for raw, uncompressed audio. It was developed primarily to test Claxon, another Rust library for FLAC decoding. Hound operates under the Apache 2.0 license and can be used in both open-source and closed-source projects, whether commercial or not. For integration into GPLv2-licensed software, you can add an exception to their copyright notice.

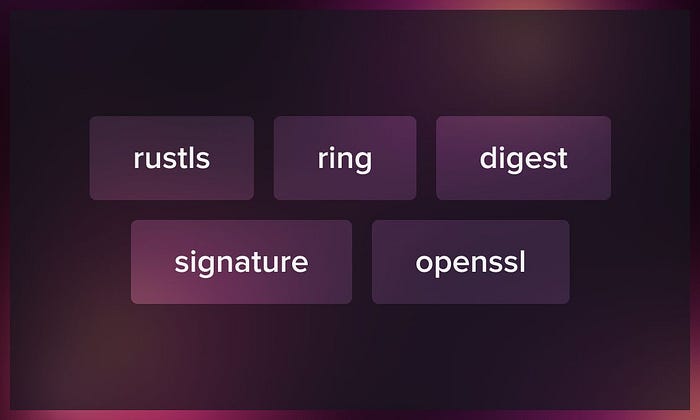
**CPAL**

[CPAL](https://github.com/RustAudio/cpal#cpal---cross-platform-audio-library) (Cross-Platform Audio Library) is a low-level audio input and output library written in pure Rust. Key features include enumerating audio hosts, identifying available audio devices, determining default input/output devices, and managing PCM streams. Supported hosts currently encompass Linux (through ALSA or JACK), Windows (primarily WASAPI), macOS and iOS (both via CoreAudio), Android (via Oboe), and Emscripten. Linux users need the ALSA development files from libasound2-dev or alsa-lib-devel packages.

**Symphonia**

[Symphonia](https://github.com/pdeljanov/Symphonia) is a Rust-based multimedia library specializing in demuxing, tag reading, and audio decoding. It supports decoding popular audio codecs with gapless playback, demuxing prevalent media containers, reading most tagging formats, and has an efficient system for manipulating audio data. Symphonia prioritizes 100% Rust safety and speed without sacrificing performance, and has minimal dependencies. Future enhancements include creating C and WASM APIs for diverse integrations. Symphonia uses separate crates for different audio codecs and media formats. By default, it supports only royalty-free open standard codecs, but others can be enabled using feature flags. To describe the reliability and completeness of the library it is categorized as “Good,” “Great,” or “Excellent”. A “Great” status denotes that the feature is largely developed, while “Excellent” confirms it meets or exceeds reference standards.

**Cryptography**



The following crates provide functionalities related to cryptographic operations and concepts in the Rust programming language. These operations are fundamental for various security-related tasks, such as hashing, digital signatures, key generation and management, etc.

**Rustls**

[Rustls](https://lib.rs/crates/rustls) is a TLS library developed in Rust. It’s used in various production environments, and although its API is mostly stable, it’s subject to change. Designed for optimal cryptographic security, Rustls requires no configuration and omits unsafe or outdated cryptographic practices by default.

Its key features include:

* Support for TLS1.2 and TLS1.3 protocols.
* Various server authentication methods, including ECDSA, Ed25519, and RSA.
* Forward secrecy facilitated by curve25519, nistp256, and nistp384.
* Encryption techniques, such as AES128-GCM, AES256-GCM, and ChaCha20-Poly1305.
* ALPN, SNI, session resumption, 0-RTT data, client authentication, extended master secret, exporters, and OCSP stapling.

**Ring**

Ring is a cryptographic library in Rust, C, and assembly language, designed to facilitate user-friendly and secure operations. It easily integrates with applications, with a special focus on optimization for small IoT devices. Derived from BoringSSL, Ring regularly receives updates and actively contributes to the development of that library. Given its platform-specific nature, it is recommended to run benchmarks on your target devices from the repository’s benches folder. The library is compatible with recent Rust releases. Comprehensive documentation is available [here](https://docs.rs/ring/latest/ring), and for building, you can refer to [BUILDING.md.](http://building.md./)

**Digest**

[Digest](https://lib.rs/crates/digest)is a base library for implementing specific cryptographic hash functions on top of. Since it doesn’t provide any implementations, you need crates providing specific hash implementations, such as sha1, sha2, sha3, hmac, pbkdf2, blake2, blake3. It requires a minimum Rust version of 1.57. While this may change in the future, you can always tell when this change has occurred by looking at the library's version number, specifically the minor version component. According to its Semantic Versioning policy, the library includes default features but does not impose version requirements on the minimum Rust version. You can license under either the Apache License, Version 2.0 or the MIT license. Contributions are assumed to be dual-licensed under these terms unless stated otherwise by the contributor.

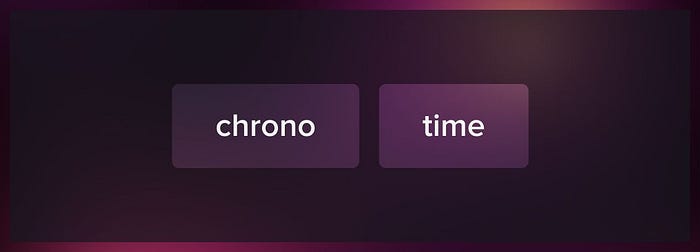
**Signature**

[Signature](https://crates.io/crates/signature) depends on Digest and contains traits that provide generic, object-safe interfaces for both creating and authenticating digital signatures. Libraries using this crate provide specific signature implementations, which include ed25519, ecdsa, rsa, pgp and others. This crate was developed over years to offer a user-friendly approach to digital signatures. It emphasizes ease-of-use and flexibility but has made some tradeoffs, like limiting signature representations. While there are suggestions to improve it for those implementing the traits, these changes could complicate things for users. Future updates might consider balancing both user and developer needs, but current challenges include technical complexities and design limitations.

**Openssl**

The [openssl](https://lib.rs/crates/openssl)crate provides Rust bindings that encompass a wide range of OpenSSL versions, spanning from 1.0.1 to 3.x.x, as well as compatibility with LibreSSL versions from 2.5 to 3.7.x. Currently, the recommended release versions are 0.10 for openssl and 0.9 for openssl-sys. Major updates are typically released annually, and the previous version continues to receive bug fixes for a period of three months following the new release. For building, users can choose the “vendored” feature for a static link to OpenSSL or use the “automatic” method which detects existing OpenSSL installations. Contributions to the crate are typically dual-licensed under the Apache and MIT licenses unless stated otherwise by the contributor.

**Date and time**



Time and date Rust crates are used for handling, manipulating, and formatting dates, times, and timestamps in Rust programs.

**Chrono**

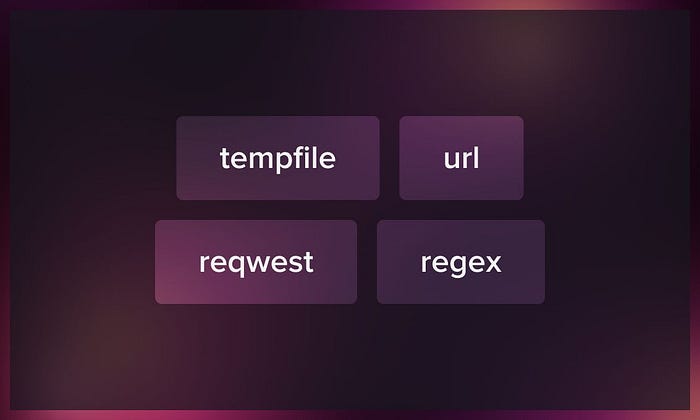
[Chrono](https://lib.rs/crates/chrono), a date and time library for Rust, provides a solution for managing dates and times within the proleptic Gregorian calendar. It offers strftime-inspired flexible parsing and formatting. Its features include alloc, std, clock, and wasmbind, with optional extensions such as serde and rkyv. To keep the core library lightweight, timezone data isn't included by default. However, you can easily incorporate Chrono-TZ or tzfile for this pupose. Comprehensive documentation is accessible on [docs.rs](http://docs.rs/).

Among its few limitations are exclusive support for the proleptic Gregorian calendar, date range constraints of +/- 262,000 years from the standard epoch, and nanosecond precision for time. It does handle lead seconds either. The Minimum Supported Rust Version (MSRV) is Rust 1.57.0.

**Time**

[Time](https://crates.io/crates/time)is a date and time library for Rust. It is written to be easy, safe, space and time efficient. It supports integration with serde, has a rich support for const (i.e. compile-time) computations, no-std support and includes numeric traits to simplify specifying durations (e.g. 2.seconds()).

**Miscellaneous**



Below, we present libraries for managing temporary files, handling URLs, making HTTP requests, and using regular expressions.

**Tempfile**

The “ [tempfile](https://lib.rs/crates/tempfile)” library aids in managing temporary files and directories. To create temporary files, it is recommended to use the tempfile() function, and for temporary directories, you should opt for the tempdir() function. The library provides multiple methods for creating temporary entities. The tempfile() function delegates the task of removing the temporary file to the operating system when it's no longer needed. On the other hand, the TempDir and NamedTempFile methods rely on Rust's destructors to handle cleanup.

In most scenarios, using tempfile() suffices unless there is a specific requirement to access the file's path or retain it for extended periods. It's important to note a security consideration: in environments with aggressive temporary file cleaners, relying solely on file paths can be risky. This is because such cleaners might unintentionally remove a temporary file, potentially creating an opportunity for attackers to exploit and replace the file. In contrast, tempfile() doesn't depend on file paths, making it a safer option in this context. However, NamedTempFile does rely on file paths for certain operations.

**Url**

The [url](https://crates.io/crates/url) crate for Rust is a library that provides tools for parsing and manipulating URLs, aligning with the [WHATWG URL Living Standard](https://www.google.com/url?q=https%3A%2F%2Furl.spec.whatwg.org%2F&sa=D&ust=1697545307228049&usg=AOvVaw09KgGlFjneVBebeO2oi7Ui). It offers a comprehensive URL type, Url, that makes it easy to read, write, and manipulate components of URLs, such as the scheme, domain, path, and query string. The crate ensures URL validity, performs automatic percent-encoding of specific URL components, and supports relative URL resolution and normalization.

**Reqwest**

[Reqwest](https://crates.io/crates/reqwest)is a widely-used Rust crate that offers a high-level HTTP client, simplifying interactions with web services. Built atop the hyper library, it provides a user-friendly API for dispatching and receiving HTTP requests. Notably, it supports Rust's async/await for non-blocking tasks, seamlessly integrates JSON serialization with the serde crate, ensures secure connections via rustls or native-tls, and manages cookies efficiently. Its comprehensive features and ease of use, make reqwest a preferred choice for Rust developers interfacing with web APIs.

**Regex**

The [regex](https://crates.io/crates/regex) crate provides tools for working with regular expressions, allowing for efficient text processing. To use it, you need to include it in your Cargo.toml file. With the regex crate, you can match strings against patterns, search within strings, split them based on patterns, or replace portions of them. For instance, you can create patterns to match date formats or extract parts of a string using capturing groups. Moreover, it’s possible to iterate over all matches in a string or replace matched patterns with specified text. This library offers an extensive API, making it an essential tool for text manipulation tasks in Rust.

**Conclusion**

You can leverage Rust’s performance benefits for tasks in various fields, from fintech and biomedicine to machine learning and business intelligence. Rust is widely used in fault-tolerant software, especially for production-ready systems and performance-critical components. Check out this crash course to become a Rust programmer:

***We also invite you to discover Serokell’s expert***[***Rust development solutions***](https://serokell.io/rust-development)***.***

*Originally published at*[*https://serokell.io*](https://serokell.io/blog/most-popular-rust-libraries)*.*

[Rust](https://medium.com/tag/rust?source=post_page-----eaca032a7b72---------------rust-----------------)

[Programming](https://medium.com/tag/programming?source=post_page-----eaca032a7b72---------------programming-----------------)

[Rust Libraries](https://medium.com/tag/rust-libraries?source=post_page-----eaca032a7b72---------------rust_libraries-----------------)

[Rust Crates](https://medium.com/tag/rust-crates?source=post_page-----eaca032a7b72---------------rust_crates-----------------)

**Rust Development Roadmap**

[[](https://serokell.medium.com/?source=post_page-----805419c487f--------------------------------)](https://serokell.medium.com/?source=post_page-----805419c487f--------------------------------)

[Serokell](https://serokell.medium.com/?source=post_page-----805419c487f--------------------------------)

·

Follow

9 min read

·

Sep 28

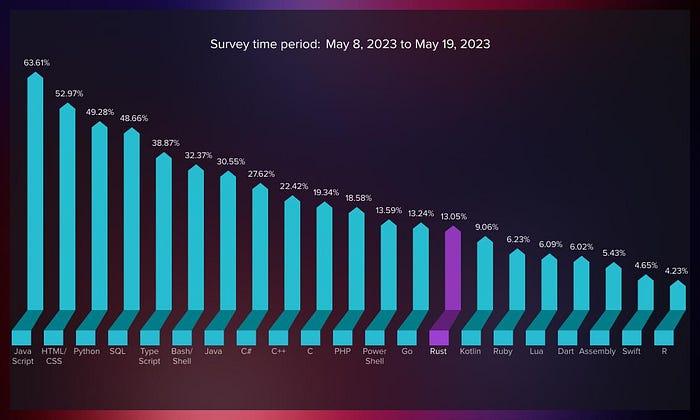
217

2



Rust has been one of the top 15 most popular and loved programming languages over the past five years due to the incredible developer experience it offers and its flexibility. It boasts powerful features, transparent documentation, and an engaged and supportive community of contributors.

Rust prioritizes safety, performance, and concurrency. It allows programmers to create applications with control of a low-level language, but with the powerful abstractions of a high-level language. Overall, there’s so much you can do with Rust that every developer should at least give it a try.



Source: [Statista](https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/)

**What are the core Rust features?**

Now let’s look at what makes Rust so effective and applicable to numerous use cases.

**Memory safety**

* The primary design goal of Rust is to guarantee memory safety without relying on a garbage collector. This is achieved through transparent data ownership, a robust type system, and rigorous compile-time checks. Data ownership paradigm dictates that every piece of data in Rust has a unique owner. When ownership is transferred (known as a “move”), the previous owner can no longer access the data. This prevents duplicate free operations and eliminates double-deletion bugs.This approach eliminates a class of bugs right at the compile stage.
* Bugs like null pointer dereferencing, which is a prevalent issue in many languages, are eliminated by using the Option<T> type in Rust. It explicitly handles the possibility of absence of value.
* Data races and buffer overflows are also prevented due to Rust’s borrowing and ownership system. If a piece of code is potentially unsafe, Rust won’t compile it unless wrapped in an unsafe block, signaling the developer to proceed carefully.

**Concurrency model**

* Concurrency in Rust is built upon the same principles that offer memory safety: ownership and borrowing. By leveraging these, Rust ensures thread safety.
* The language offers several concurrency primitives, like Arc (atomic reference counting) for shared state and Mutex or RwLock for mutable access. Channels in Rust offer a way to communicate between threads.
* The async/await syntax, introduced in Rust 1.39.0, allows for writing asynchronous code that looks like synchronous code, simplifying complex concurrency scenarios.

**Performance characteristics**

* Rust’s “zero-cost abstractions” mean that using high-level constructs doesn’t impose a runtime overhead. You get the benefit of abstraction without sacrificing performance.
* Rust, being a systems language, provides extensive control over hardware resources. This makes it as performant as C and C++ in many benchmarks, but with added safety guarantees.
* Inlining, loop unrolling, and other aggressive optimizations can be performed due to Rust’s explicitness and lack of a runtime.

Watch this comprehensive video to learn more about Rust features:

The above features aren’t unique to Rust; its strength lies in their effective combination. Many Rust properties are inspired by or borrowed from other languages and various models:

**Core foundations**:

* *Abstract machine model:* Inspired by C.
* *Memory model and management*: Borrowed from languages and tools like C++, ML Kit, and Cyclone.

**Data structures & types**:

* *Data types:* Rust’s data types have roots in C, SML, OCaml, Lisp, and Limbo.
* ***Optional bindings:*** This feature is influenced by Swift.

**Programming paradigms**:

* *Functional programming:* Elements taken from Haskell, OCaml, and F#.
* *Hygienic Macros*: Adapted from Scheme.

**Advanced features**:

* *Attributes*: Deriving from the ECMA-335 standard.

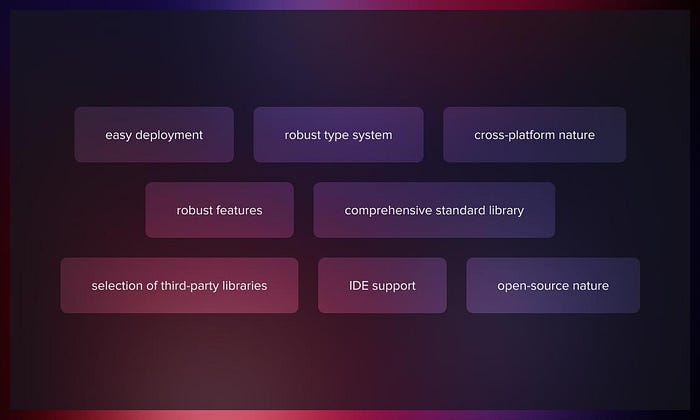
**Packaging & modularity**:

* *Crate system*: Analogous to the Assembly in the ECMA-335 CLI model.

**Concurrency & communication**:

* *Channels and concurrency:* Influences from Newsqueak, Alef, and Limbo.
* *Message passing and thread failure*: Principles borrowed from Erlang.

**Why adopt Rust development?**



Rust combines various programming styles: procedural, actor-based concurrency, object-oriented, and purely functional. It also supports both static and dynamic forms of generic and metaprogramming. Below we provide an overview of the main Rust benefits.

**Easy deployment**

Rust’s robust safety and integrity tools wouldn’t mean much without active utilization. Rust’s creators and community make the language accessible and friendly for beginners. The package required to generate Rust binaries is unified. You need external compilers like GCCO only when working with components outside of the Rust system, like compiling a C library from its source.

**Robust type system**

With Rust, you aren’t allowed to compile code that contains detectable bugs. This feature enables developers to concentrate on the program’s core logic rather than on rectifying issues.

**Cross-platform nature**

While Rust is compatible with most modern platforms, its developers don’t aim to make it universal, focusing on popular platforms. Rust functions on three major operating systems: Linux, Windows, and MacOS. If there’s a need to cross-compile or generate binaries for another platform or architecture, Rust makes it relatively easy as well.

**Robust features**

Rust boasts a set of native features that can rival those in C++: Macros, generics, pattern matching, and composition via “traits.”

**Comprehensive standard library**

Part of Rust’s broader objective is to present a compelling alternative to C and C++. As such, it offers an extensive standard library that includes containers, collections, iterators, string operations, process and thread management, and more. Since Rust aims to be cross-platform, its standard library only includes universally portable features. Platform-specific functions are available through third-party libraries. Moreover, Rust can function without its standard library, which is especially useful when developing platform-independent binaries like those for embedded systems.

**Selection of third-party libraries**

A language’s versatility is often determined by the third-party support it receives. [Cargo](https://crates.io/), Rust’s official library repository, hosts over 60,000 “crates.” Many of these are API bindings for popular libraries or frameworks. Yet, there isn’t a comprehensive ranking system based on quality for these crates, so you have to rely on your personal experience or community advice.

Find out more about popular [open-source Rust libraries](https://serokell.io/blog/open-source-rust) in our earlier post.

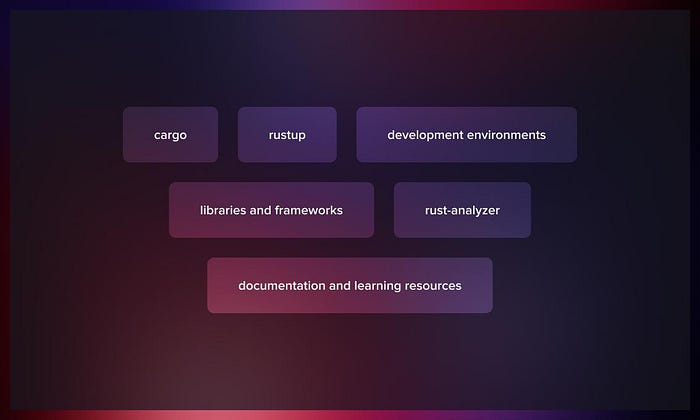
**IDE support**

The Rust team developed [rust-analyzer](https://rust-analyzer.github.io/) to offer real-time feedback from the Rust compiler to IDEs, for example, Microsoft Visual Studio Code.

**Open-source nature**

As an open-source language, Rust benefits from the contributions of an extensive global developer community that continuously improves its performance. Friendly, especially for those new to programming. It also regularly updates its detailed and transparent language documentation.

**The Rust development ecosystem**



One of Rust’s significant advantages is its wide ecosystem of tools, frameworks, and libraries.

**Cargo**

[Cargo](https://doc.rust-lang.org/cargo/) is the official package manager and compiler for Rust. It handles project building, dependency management, and test execution. For the official registry of Rust packages, developers use [crates.io](https://crates.io/), which is governed by the Rust community and is integrated with Cargo.

**Rustup**

A utility to streamline the management of Rust versions, [Rustup](https://rustup.rs/) allows for switching between different Rust toolchains and updating them.

**Development environments**

Many IDEs and text editors now support [Rust development](https://serokell.io/rust-development), including Visual Studio Code, Atom, and IntelliJ IDEA.

**Libraries and frameworks**

Rust’s [library and framework ecosystem](https://lib.rs/std) is constantly expanding and encompasses applications such as web development, gaming, and machine learning, among others. Among its popular libraries and frameworks are Rocket, Tokio, Serde, and Actix.

**Rust-analyzer**

An efficient and dependable language server, [rust-analyzer](https://rust-analyzer.github.io/) offers functionalities like code analysis and auto-completion for Rust code. It is easy to use and compatible with multiple text editors and IDEs.

[Rustfmt](https://doc.rust-lang.org/book/appendix-04-useful-development-tools.html#automatic-formatting-with-rustfmt) is a dedicated formatting tool to ensure the uniformity of Rust code.

**Documentation and learning resources**

Rust provides a lot of useful documentation. Check out some useful resources that we’ve collected here for your reference.

Whether you are starting out with Rust or looking for answers to specific questions related to your projects, Rust’s [official website](https://www.rust-lang.org/) is always an excellent source of perfectly structured information.

**Rust development challenges**

The Rust programming language offers numerous advantages, such as enhanced security, superior performance, and streamlined concurrent programming, along with ensured memory safety. Nevertheless, Rust isn’t without its flaws. Some of the commonly cited challenges when working with Rust include:

**Steep learning curve**: Rust’s ownership, borrowing, and lifetimes can be confusing for newcomers, especially those coming from languages without these concepts.

**Verbose syntax**: The language is sometimes critiqued for its verbose syntax, which can be seen as both a pro (explicit over implicit) and a con (lengthy code for simple tasks).

**Limited libraries**: While the Rust ecosystem is rapidly growing, it still lags behind more mature languages like Python, Java, or JavaScript in terms of the sheer number and diversity of third-party libraries available.

**Compilation times:** One of the often-cited drawbacks is longer compilation times. This can slow down development, especially in larger projects.

**Maturity:** While Rust is a stable language, it’s still younger than many of its competitors. As a result, certain features and best practices are still evolving.

**Minimal runtime**: This is both a strength and a challenge. While it allows Rust to be used in systems programming and places with constrained resources, it also means that the language doesn’t come with as many built-in features as languages with more extensive runtimes.

**Rust vs. C++**

Below we introduce a comparison between Rust and C++ provided by Serokell’s site reliability engineer Richard Brežák.

What makes Rust stand out from other languages is its type system, library functions that lean towards the functional paradigm, and its ownership system.

Modern C++ has many features and, in fact, implements much of Rust’s feature set. C++20 forgoes the traditional header/source file split and adopts a modern module system. It also introduces functional aspects like lambdas, maps, folds and even std::optional, which behaves like Option.

Some of these modern C++ features can be a bit clunky to use, but they do exist and work. There is even talk within the C++ developer community of introducing features like proper pattern matching and lifetimes, similar to those in the Rust language, into future C++ standards.

Commonly it’s said that the memory management models of C++ and Rust differ. While it’s partially true that both rely on destructors to deallocate resources, the key distinction lies in the fact that Rust is smarter and stricter in tracking the lifetime of values, enabling it to maintain a reference’s relationship with a value. In contrast, C++ doesn’t preserve this relationship after creating a reference; it essentially only creates a faster copy.

The problems with C++ arise from the necessity of ensuring that only the most recent programming paradigms and features are used. This complicates development and adds mental overhead. In contrast, Rust doesn’t allow you to write subpar code. In some cases, C++’s lenience can be a benefit, especially when interfacing with legacy codebases or quickly prototyping a new program. In the end, C++ offers numerous advantages: It is a very versatile language that can compile to virtually any platform and runs with minimal overhead. Its issues stem from too many options, which sometimes leads programmers to make incorrect choices.

**Where is Rust already used?**



Rust has found [multiple applications](https://en.wikipedia.org/wiki/Rust_%28programming_language%29#Adoption)in various tech domains, which include:

**Web browsers**

Components from Servo, an experimental parallel browser, were incorporated into Mozilla’s Firefox.

**Web services and platforms**

Companies like OpenDNS (owned by Cisco) and Cloudflare employ Rust for services like DNS resolution and firewall pattern matching.

* Discord ( an instant messaging social platform) uses Rust for parts of its backend and video encoding.
* Dropbox turned to Rust in 2021 for media capturing.
* Facebook’s (Meta) Mononoke and Google’s Android OS have integrated Rust components.

**Cloud services**

* Amazon Web Services (AWS) has been using Rust since 2017 for virtualization software (Firecracker), containerization (Bottlerocket), and the asynchronous networking stack (Tokio).
* Microsoft Azure IoT Edge has Rust components. Microsoft also employs them to support containerized modules with WebAssembly and Kubernetes.

**Operating systems**

* In 2021, the Rust for Linux project aimed to add Rust to the Linux kernel. By 2023, Rust (along with C and Assembly) was incorporated into Linux’s version 6.1.
* Redox OS has a microkernel in Rust.
* Parts of Microsoft Windows were rewritten in Rust for enhanced performance.

**Web development**

* Deno provides a secure runtime for JavaScript and TypeScript using Rust.
* Ruffle, an open-source SWF emulator, is built in Rust.

To learn more about Rust applications, read our interview series “ [Rust in Production](https://serokell.io/blog/rust-in-production).”

**Conclusion**

Surveys show a growing popularity of Rust amongst developers. [The 2023 Stack Overflow Developer Survey](https://survey.stackoverflow.co/2023/) indicated 13% had recently worked extensively in Rust, with the language being the “most loved” every year from 2016 to 2023. Rust was also ranked 6th in “most wanted technology” in 2023.

The combination of high-level abstractions and granular control make Rust a preferred choice for developing safe and efficient system software. With the increasing recognition of Rust within the developer community it has the potential to become the predominant language in more domains.

*Originally published at*[*https://serokell.io*](https://serokell.io/blog/rust-development)*.*

**Was Rust Worth It?**

From JavaScript to Rust, three years in.

[[](https://jsoverson.medium.com/?source=post_page-----f43d171fb1b3--------------------------------)](https://jsoverson.medium.com/?source=post_page-----f43d171fb1b3--------------------------------)

[Jarrod Overson](https://jsoverson.medium.com/?source=post_page-----f43d171fb1b3--------------------------------)

·

Follow

8 min read

·

Oct 26

2.9K

30

A few years ago, I dropped everything to focus 100% on WebAssembly. At the time, Rust had the best support for compiling into WebAssembly, and the most full-featured WebAssembly runtimes were Rust-based. Rust was the best option on the menu. I jumped in, eager to see what all the hype was about.

Since then, I (along with some other awesome people) built [Wick](https://github.com/candlecorp/wick), an application framework and runtime that uses WebAssembly as its core module system.



Wick was the primary target of our Rust experimentation

After three years, multiple production deployments, an [ebook](https://github.com/jsoverson/node-to-rust), and ~100 packages deployed on [crates.io](https://crates.io/me/crates), I feel it’s time to share some thoughts on Rust.

**The Good**

**You can maintain more with less**

I am a massive proponent of test-driven development. I got used to testing in languages like Java and JavaScript. I started writing tests in Rust as I would in any other language but found that I was writing tests that *couldn’t fail*. Once you get to the point where your tests can run – that is, where your Rust code compiles – Rust has accounted for so many errors that many common test cases become irrelevant. If you avoid unsafe {} blocks and panic-prone methods like .unwrap(), you start with a foundation that sidesteps many problems by default.

The aggressiveness of Rust’s borrow checker, the richness of Rust’s type system, the functional patterns and libraries, and the lack of “null” values all lead to maintaining more with less effort spent in places like testing. I’ve maintained the 70,000+ lines of code in the Wick project with far fewer tests than I would need in other languages.

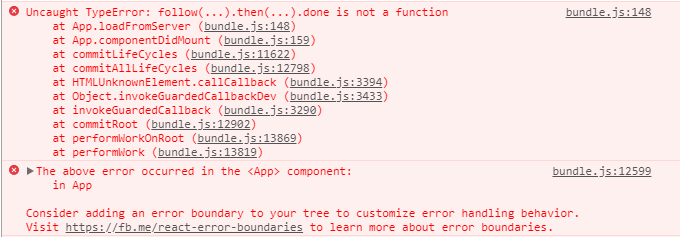
When you need to write tests, adding them on the fly is easy without thinking about it. Rust’s integrated test harness lets you add tests right next to code with barely a second thought.

**I code better in other languages now**

Programming in Rust is like being in an emotionally abusive relationship. Rust screams at you all day, every day, often about things that you would have considered perfectly normal in another life. Eventually, you get used to the tantrums. They become routine. You learn to walk the tightrope to avoid triggering the compiler’s temper. And just like in real life, those behavior changes stick with you forever.

Emotional abuse is not generally considered a *healthy* way to encourage change, but it does effect change nonetheless.

I can’t write code in other languages without feeling uncomfortable when lines are out of order or when return values are unchecked. I also now get irrationally upset when I experience a runtime error.



*What do you mean “done" is not a function? Why didn’t you let me know "done” might not be a function??*

**Clippy is great!**

[Clippy](https://github.com/rust-lang/rust-clippy) is Rust’s linter, but calling it a linter is a disservice. In a language where the compiler can make you cry, Clippy is more of a gentle friend than a linter.

The Rust standard library is *enormous.* It’s hard to find functions you know probably exist when so much functionality is spread across myriad granular types, traits, macros, and functions. Many Clippy rules (e.g., [manual\_is\_ascii\_check](https://rust-lang.github.io/rust-clippy/master/index.html#/manual_is_ascii_check)) look for common patterns that stdlib methods or types would better replace.

Clippy has [hundreds of rules](https://rust-lang.github.io/rust-clippy/master/index.html) that tackle performance, readability, and unnecessary indirection. It will frequently give you the replacement code when possible.

It also looks like (soon) you’ll [finally](https://github.com/rust-lang/cargo/issues/12115) be able to configure global lints for a project. Until now, you had to hack your solution to keep lints consistent for projects. In Wick, we use a script to automatically update [inline lint configurations](https://github.com/candlecorp/wick/blob/main/src/main.rs#L8-L84) for a few dozen crates. It took [*years*](https://github.com/rust-lang/cargo/issues/5034) for the Rust community to land on a solution for this, which brings us to…

**The Bad**

**There are gaps that you’ll have to live with**

I questioned my sanity every time I circled back around to the Clippy issue above. Surely, I was wrong. There must be a configuration I missed. I couldn’t believe it. I still can’t. There ***must*** be a way to configure lints globally. I [quadruple](https://github.com/rust-lang/cargo/issues/5034)-[checked](https://github.com/rust-lang/rust/issues/45832) [when](https://github.com/rust-lang/rust-clippy/issues/1313) I [wrote](https://github.com/rust-lang/rust-clippy/issues/6625) [this](https://www.appsloveworld.com/rust/4/how-can-i-have-a-shared-clippy-configuration-for-all-the-crates-in-a-workspace) [to](https://github.com/EmbarkStudios/rust-ecosystem/issues/22) make sure I wasn’t delusional. Those issues are closed now, but they had been open for years.

Clippy’s awesome, but this use case is one example of many around the Rust world. I frequently come across libraries or tools where my use cases aren’t covered. That’s not uncommon in newer languages or projects. Software takes time (usage) to mature. But Rust isn’t *that* new. There’s something about Rust that feels different.

In open source, edge cases are frequently addressed by early adopters and new users. They’re the ones with the edge cases. Their PRs refine projects so they’re better for the next user. [Rust has been awarded the “most loved language” for the better part of a decade](https://github.blog/2023-08-30-why-rust-is-the-most-admired-language-among-developers/). It’s got no problem attracting new users, but it’s not resulting in dramatically improved libraries or tools. It’s resulting in one-off forks that handle specific use cases. I’m guilty of that, too, but not for lack of trying to land PRs.

I don’t know why. Maybe the pressure to maintain stable APIs, along with Rust’s granular type system, makes it difficult for library owners to iterate. It’s hard to accept a minor change if it would result in a major version bump.

Or maybe it’s because writing Rust code that does everything for everyone is exceedingly difficult, and people don’t want to deal with it.

**Cargo, crates.io, and how to structure projects**

I modeled the Wick repository structure around some other popular projects I saw. It looked reasonable and worked fine until it didn’t.

You can build, test, and use what feels like a module-sized crate easily with Cargo. Deploying it to crates.io, though? That’s a whole different story.

You can’t publish packages to crates.io unless **every** referenced crate is also published individually. That makes some sense. You don’t want to depend on a crate that depends on packages that only exist on the author’s local filesystem.

However, many developers break large projects down into smaller modules naturally, and you can’t publish a parent crate that has sub-crates that *only exist within itself.*You can’t even publish a crate that has local dev dependencies. You must choose between publishing random utility crates or restructuring your project to avoid this problem. This limitation feels arbitrary and unnecessary. You can clearly build projects structured like this, you just can’t publish them.

*Edit:*[*Ed Page reached out*](https://www.reddit.com/r/rust/comments/17gnkh2/was_rust_worth_it/k6jf2c0/)*to note that you*can*publish with local dev dependencies, as long as you don’t include a version in Cargo.toml*

Cargo does have excellent workspace support, though! Cargo’s workspaces offer a better experience managing large projects than most languages. But they don’t solve the deployment problem. Turns out, you can set workspaces up in any of a dozen ways, *none* of which make it easy to deploy.

You can see the problem manifest in the sheer number of [utility crates](https://crates.io/search?q=cargo+workspace+publish) designed to simplify publishing workspaces. Each works with a subset of configurations, and the “one true way” of setting workspaces up still eludes me. When I publish Wick, it’s frequently an hour+ of effort combining manual, repetitive tasks with tools that only partially work.

**Async**

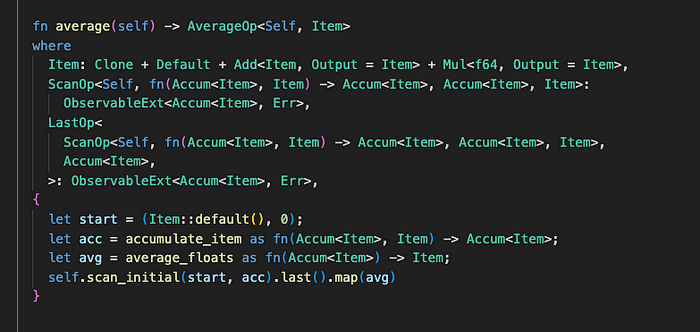
Rust added async-iness to the language after its inception. It feels like an afterthought, acts like an afterthought, and frequently gets in your way with errors that are hard to understand and resolve. When you search for solutions, you have to filter based on the various runtimes and their async flavors. Want to use an async library? There’s a chance you can’t use it outside of a specific async runtime.

After two decades of JavaScript and decent experience with Go, this is the *most significant*source of frustration and friction with Rust. It’s not an insurmountable problem, but you must always be ready to deal with the async monster when it rears its head. In other languages, async is almost invisible.

**The Ugly**

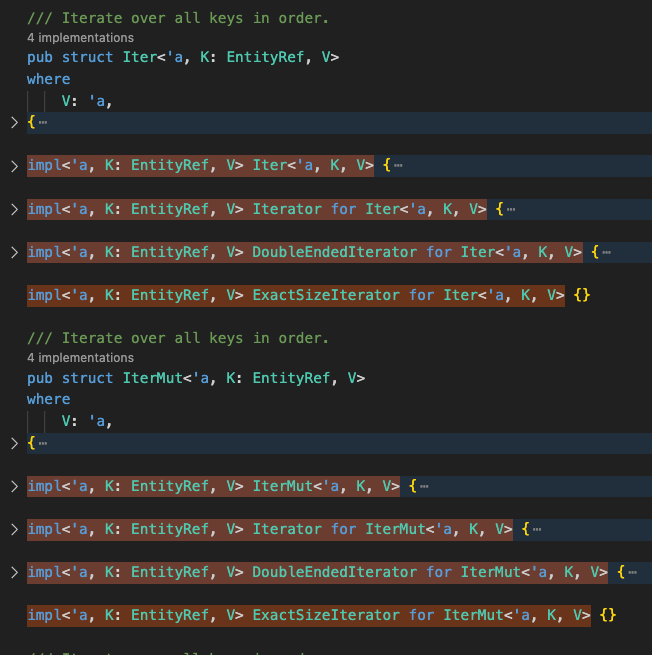
**Refactoring can be a slog**

Rust’s rich type system is a blessing and a curse. Thinking in Rust types is a dream. Managing Rust’s types can be a nightmare. Your data and function signatures can have generic types, generic lifetimes, and trait constraints. Those constraints can have their own generic types and lifetimes. [Sometimes, you’ll have more type constraints than actual code](https://github.com/rxRust/rxRust/blob/master/src/observable.rs#L1134-L1142).



Constraints that outweigh logic

You also need to define all your generics on [every impl](https://github.com/bytecodealliance/wasmtime/blob/038ddfeb6699591b5d82546c9b2d5076097bc9ce/cranelift/entity/src/iter.rs#L29-L58). It’s tedious when writing it the first time. When refactoring though, it can turn a minor change into a cascading mess.



Simple generic IDs are duplicated over and over again.

It’s hard to make rapid progress when you need to tweak 14 different definitions before you can take a single step forward.

*Edit to address external comments: The problem isn’t the expressibility, the problem is no language or tooling solution to reduce the duplication. There are frequent reasons to have the same constraints or refer to the same generic lists, but there’s no way to alias or otherwise refer to a central definition. I’m not sure there should be, but it doesn’t change the burden of duplication.*

**The Verdict**

I love Rust. I love what it can do and how versatile it is. I can write system-level code in the same language as CLI apps, web servers, *and* web clients. With WebAssembly, I can use the same exact binary to run an LLM [in the browser](https://wasm.candle.dev/llama2) as on the command line. That still blows my mind.

I love how rock-solid Rust programs can be. It’s hard to return to other languages after you learn to appreciate what Rust protects you from. I went back to Go for a brief period. I quickly became intoxicated with the speed of development again. Then I hit the runtime panics, and the glass shattered.

But Rust has its warts. It’s hard to hire for, slow to learn, and too rigid to iterate quickly. It’s hard to troubleshoot memory and performance issues, especially with async code. Not all libraries are as good about safe code as others, and dev tooling leaves much to be desired. You start behind and have a lot working against you. If you can get past the hurdles, you’ll leave everyone in the dust. That’s a big if.

Was Rust worth it for us? It’s too early to tell. We’ve done amazing things with a small team but also had immense roadblocks. We also had technical reasons that made Rust more viable.

Will it be worth it for you? If you need to iterate rapidly, probably not. If you have a known scope, or can absorb more upfront cost? Definitely consider it. You’ll end up with bulletproof software. With the WebAssembly angle becoming stronger every month, the prospect of writing perfect software *once* and reusing it *everywhere* is becoming a reality sooner rather than later.

# Migrating from JavaScript to Rust: A Developer’s Experience

[[Alexander Obregon](https://medium.com/@AlexanderObregon?source=post_page-----8adcddd33e71--------------------------------)](https://medium.com/@AlexanderObregon?source=post_page-----8adcddd33e71--------------------------------)

[[Stackademic](https://blog.stackademic.com/?source=post_page-----8adcddd33e71--------------------------------)](https://blog.stackademic.com/?source=post_page-----8adcddd33e71--------------------------------)

[Alexander Obregon](https://medium.com/@AlexanderObregon?source=post_page-----8adcddd33e71--------------------------------)

·

Follow

Published in

[Stackademic](https://blog.stackademic.com/?source=post_page-----8adcddd33e71--------------------------------)

·

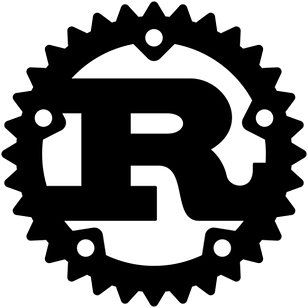
4 min read

·

Aug 21

137

1



[Image Source](https://commons.wikimedia.org/wiki/File:Rust_programming_language_black_logo.svg)

# Introduction

For any software developer, change is a constant. New languages, tools, and paradigms continuously evolve, urging us to reassess our choices. Recently, the rise of WebAssembly and the systems programming language, Rust, has piqued the interest of many web developers previously steeped in JavaScript. Here, I’ll detail my personal journey from the dynamically-typed world of JavaScript to the statically-typed realm of Rust.

# Why the Shift?

To kick off, it’s essential to understand why someone would consider such a migration.

* **Performance:** Rust offers near C-level performance, making it suitable for tasks that need raw speed. In contrast, JavaScript, being an interpreted language, can’t match this.
* **Memory Safety:** One of Rust’s selling points is its guarantee of memory safety without sacrificing performance.
* **Concurrency:** Rust has first-class support for concurrent programming, which is becoming crucial in today’s multi-core CPU world.
* **Interoperability:** With the advent of WebAssembly, Rust can be used in web browsers, providing a bridge between systems programming and web development.

# Initial Challenges

However, it wasn’t all roses. My journey was fraught with challenges:

## Syntax Differences

While JavaScript uses a C-style syntax that is friendly and familiar to many, Rust introduces several unique constructs.

fn main() {  
 let mut x = 5; // `mut` indicates mutability  
 println!("The value of x is: {}", x);  
 x = 6;  
 println!("The value of x is: {}", x);  
}

In this Rust example, variable mutability is explicit, unlike JavaScript.

## Static Typing

Coming from JavaScript, Rust’s strict type system felt intimidating. In JavaScript, I was used to:

let x = 5;  
x = "Hello";

But in Rust, type inference is powerful, but once a type is set, it’s set:

let x = 5; // x is i32 by default  
x = "Hello"; // This will throw an error

## Ownership & Borrowing

Rust introduces the concepts of ownership and borrowing to guarantee memory safety. This was foreign to me as a JavaScript developer. In Rust:

fn main() {  
 let s1 = String::from("hello");  
 let s2 = s1;   
 println!("{}, world!", s1); // This will throw an error  
}

Once you pass the value of s1 to s2, you can't use s1 anymore. This ownership model ensures no two parts of the code can change the same piece of memory at the same time.

# Rust’s Strengths

As I delved deeper into Rust, I began to appreciate its strengths:

## Fearless Concurrency

Concurrency in JavaScript, with its single-threaded event-loop, often led me to callback hell. Rust, however, offers a cleaner way with threads, channels, and the async/await syntax.

## No Garbage Collector

Rust doesn’t have a runtime or garbage collector. Its memory safety guarantees are compile-time, leading to faster execution.

## Rich Standard Library

Rust’s standard library, std, is extensive. It offers everything from data structures to file systems operations, often negating the need for external libraries.

# WebAssembly and Rust

With WebAssembly (Wasm), you can run code in the browser at near-native speed. Rust has become one of the favored languages to compile to Wasm. This allowed me to use Rust for heavy computations in the web environment, making applications snappier.

# The Learning Curve

Switching from JavaScript to Rust was, in many ways, like learning to program all over again:

* **Reading Extensively:** The “Rust Book” became my bible. It’s a comprehensive guide to all things Rust.
* **Community Engagement:** Rust’s community is vibrant, helpful, and growing. Engaging on platforms like the Rust forum or Reddit was beneficial.
* **Practice:** Like any language, mastering Rust required practice. I started with small projects, gradually increasing complexity.

# Integration into the Workflow

Despite its strengths, Rust isn’t always the best tool for every job. I found that integrating Rust into my workflow, rather than replacing JavaScript entirely, was more effective. For instance, frontend user interactions remained in JS, while heavy computations shifted to Rust.

# Conclusion

Migrating from JavaScript to Rust was a transformative experience. It reshaped my understanding of memory management, concurrency, and type systems. While Rust might not entirely replace JavaScript in the web ecosystem, it certainly carves a significant niche for itself.

For developers considering a similar shift, my advice is: be patient. The learning curve is steep but rewarding. Embrace the Rustacean way, and you’ll undoubtedly see your software skills reach new horizons.

1. [The Rust Programming Language — Website](https://www.rust-lang.org/)
2. [The Rust Programming Language — Official Book](https://doc.rust-lang.org/book/)
3. [WebAssembly Official Website](https://webassembly.org/)

Thank you for reading until the end. Please consider following the writer and this publication. Visit [Stackademic](https://stackademic.com/) to find out more about how we are democratizing free programming education around the world.

# Why Rust is Making You Fail.

[[Chad Johnson](https://medium.com/@ChadJohnsonOfficial?source=post_page-----586dc7e2edbc--------------------------------)](https://medium.com/@ChadJohnsonOfficial?source=post_page-----586dc7e2edbc--------------------------------)

[Chad Johnson](https://medium.com/@ChadJohnsonOfficial?source=post_page-----586dc7e2edbc--------------------------------)

·

Follow

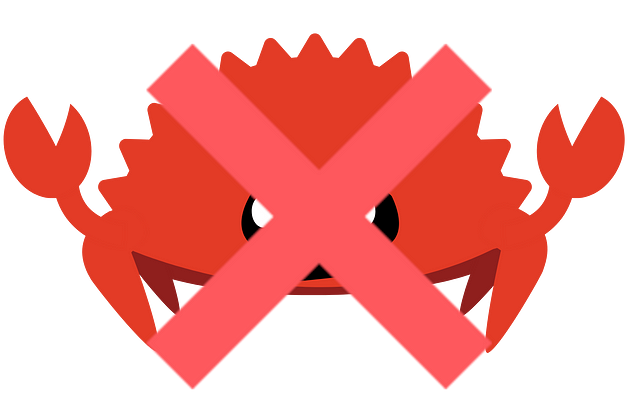
3 min read

·

Oct 4

928

82



A Ferris with a cross overlaid on it.

Programming languages are tools, not religions, but you wouldn’t know that from talking to some Rust propagandists. Sure, Rust has its merits — memory safety without garbage collection, yada yada — but let’s talk about getting actual, market-ready products out the door. That’s where garbage-collected languages like C#, JavaScript, and Python shine. Here’s why.

# The Borrow Checker: A Double-Edged Sword

Ah, the borrow checker, Rust’s claim to fame. It’s like that overly cautious friend who won’t let you climb a tree because you might fall and break a leg. It’s supposed to enforce strict rules to avoid pitfalls.

* **In Practice:**It makes you wrestle with it just to get your code to compile. Every minute spent doing that is a minute not spent developing features, fixing other kinds of bugs, or doing literally anything else productive.

# The Learning Curve Fallacy

Rust proponents claim you’ll get the hang of the borrow checker, eventually.

* **Hard Truth**: The upfront time isn’t worth the so-called long-term gains, especially if you’re in a rapid development cycle.

# Striving for Perfection at the Cost of Progress

Here’s the kicker with Rust — you’re expected to write perfect code right from the start. Sounds ideal, doesn’t it? Except, what if you’re in startup mode or another fast-paced development environment?

* **Reality Check**: You need to ship products fast to even know if they’re worth refining. Wrestling with the borrow checker to produce “perfect” code is just burning time that could’ve been better spent actually testing your product in the market.

# The Myth of Future Time-Saving

Ah, the old “it’ll save you debugging time in the long run” argument. That’s not completely bogus, but if you’re racing against the clock, that future time saved is purely theoretical.

* **Hard Truth**: What’s the point of crafting “perfect,” bug-free code if you haven’t even validated the product’s market fit? You could spend all that time fine-tuning, only to find out nobody wants what you’re selling.

# Productivity over Perfection

Garbage collection? Yeah, it might be sneered at by hardcore systems programmers. But let’s get real.

* **The Deal**: Speed trumps perfection in the business world. If the product freezes for a few milliseconds to collect garbage but ships six months faster, most businesses will pay that price gladly.

# Time is Money, and Compilers Don’t Pay Bills

Your boss couldn’t care less about the theoretical benefits of zero-cost abstractions if the product’s late.

* **The Reality**: With languages like C#, JavaScript, and Python, you’re writing code that actually brings in revenue. These languages come packed with rich libraries that let you do more with less. Rust? Not so much.

# Ecosystem Support

Sure, Rust’s ecosystem is growing, but let’s not kid ourselves. It’s nowhere near as extensive as that of languages like C#, JavaScript, and Python.

* **The Fact**: Massive user bases, extensive libraries, and heaps of resources. Got a problem? A quick Google search usually has you covered. Rust? Not quite there yet.

# The “It Works” Factor

At the end of the day, the market wants something that works. Languages like C#, JavaScript and Python have a long track record of working well in a variety of domains, from game development to backends and to user interfaces.

* **The Truth**: Rust might catch up one day, but for now, it’s more of a gamble.

# Conclusion

Look, if you’re not building low-level stuff like drivers, embedded systems, or operating systems, and you don’t have a laser-focused goal, do not use Rust.

In scenarios where time-to-market is king, garbage-collected languages like C#, JavaScript, and Python are the real MVPs. Why? Because they get you across the finish line when it really counts.

* **The Verdict**: In the real world, a so-called “perfect” product that’s collecting dust on the shelf is worthless. On the other hand, a good-enough product that actually makes it into users’ hands? That’s where the money is.

So, the next time you’re in a rapid development cycle, think twice before hopping on the Rust train.

# Special Thanks

Thank you to all contributors and those who engaged in the comments, leading to revisions in this article in a more balanced perspective.

# We are done here.

You’ve read the rest, now **follow**the best. You’ll get more no-BS deep-dives here on Medium. Don’t miss out.