**Anatomy Rust: Security Concepts (Ownership, Borrowing and Lifetimes)**

[[ahmetw.eth](https://ahmetwkulekci.medium.com/?source=post_page-----b05d35e5e185--------------------------------)](https://ahmetwkulekci.medium.com/?source=post_page-----b05d35e5e185--------------------------------)

[ahmetw.eth](https://ahmetwkulekci.medium.com/?source=post_page-----b05d35e5e185--------------------------------)

·

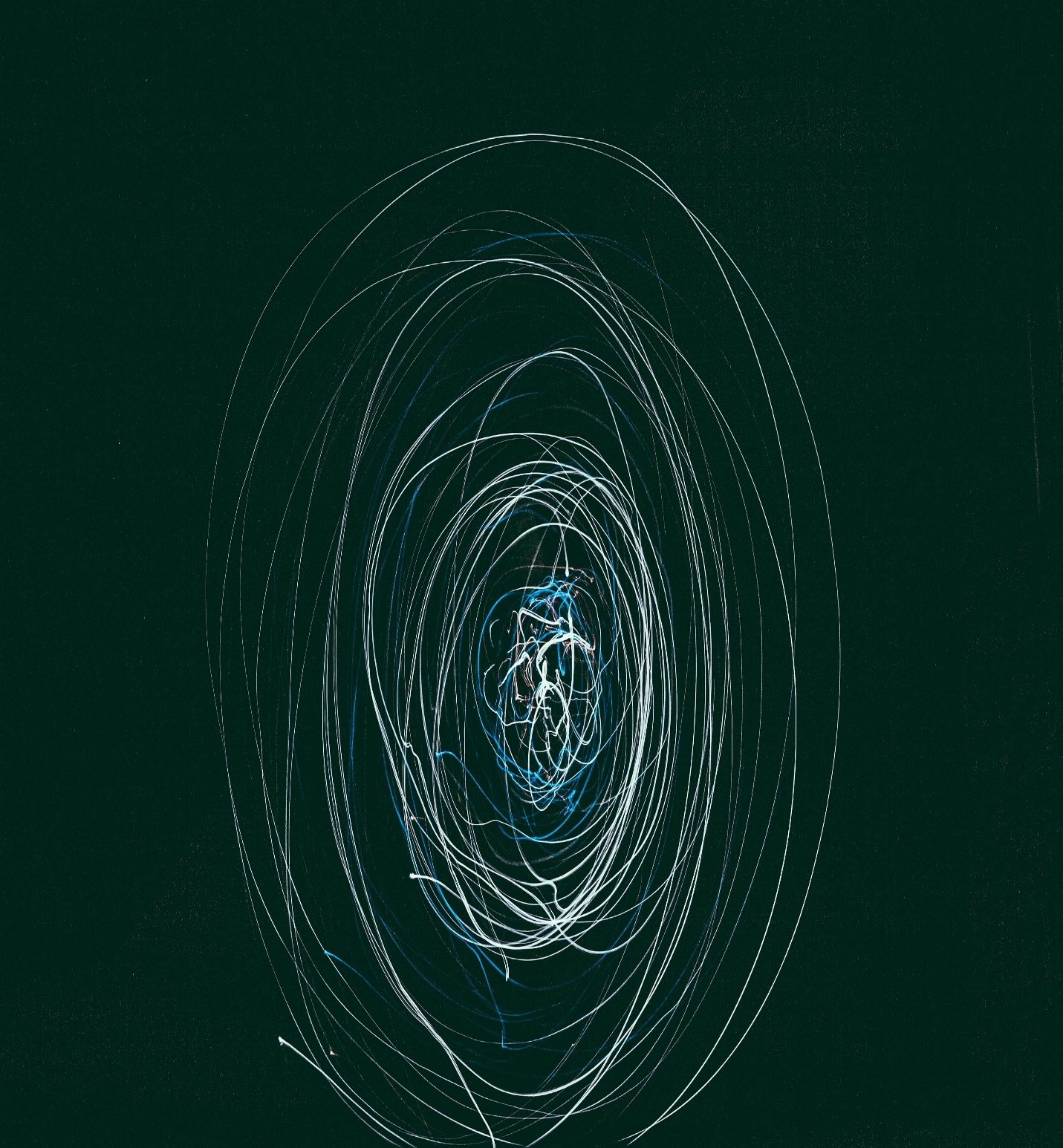
Follow

6 min read

·

Oct 7

138

Photo by [Gertrūda Valasevičiūtė](https://unsplash.com/@skraidantisdrambliukas?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash) on [Unsplash](https://unsplash.com/photos/xMObPS6V_gY?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash" \t "_blank)

Hello!

In the first post of the Anatomy series, I will explain the 3 most important concepts that bring Rust from other programming languages with code examples. So, let’s dive in.

Rust is a low-level programming language designed for system programming that provides high performance and memory safety while also supporting modern software development practices. Ownership, borrowing, and lifetimes, what I want to explain in this post, make it possible to avoid common programming angling references.

Rust gives us a safe and clear way to manage our resources. And it doesn’t allow us to manage them any other way. This is very restrictive, but that’s what we want.

Rust restrictions are indeed awesome for several reasons. Here are a few explanations:

* **Memory Safety:** Rust’s ownership and borrowing system ensures that developers write code that is free from common memory-related bugs such as null pointer dereferences, buffer overflows, and dangling pointers. The strict compiler-enforced restrictions on memory access prevent these issues at compile-time, reducing the risk of runtime errors.
* **Data Race Prevention**: Rust’s ownership system also guarantees data race prevention. By enforcing strict rules on mutable references and concurrency, ***Rust ensures that multiple threads cannot modify the same data concurrently, eliminating a class of bugs common in other programming languages.***
* **Cross-Platform Compatibility:**Rust’s restrictions make the language suitable for systems programming across various platforms. It provides low-level control and efficiency while still being portable and safe.
* **Trustworthy Libraries:** Rust's restrictions make building high-quality libraries easier. The emphasis on strong typing, immutability by default, and strict interface specifications make it easier for developers to reason about the correctness and behavior of libraries.
* **Performance Optimization:** The restrictions in Rust allow for powerful optimizations by the compiler. The ownership and borrowing system enables the compiler to generate highly optimized machine code without sacrificing safety, resulting in efficient and performant programs.

**Understanding Ownership**

Ownership is a fundamental concept in Rust that determines how resources are managed. At the core of Rust’s memory management model is the concept of ownership. ***In Rust, every value has a unique owner, which is responsible for allocating and deallocating memory.*** When a value goes out of scope, its owner is automatically responsible for cleaning up the memory associated with it. ***This ownership model ensures that there are no dangling pointers or memory leaks in Rust programs.***

Here’s an example of transferring ownership from one variable to another:

fn main() {  
 let s1 = String::from("Hello");  
 let s2 = s1;  
 println!("{}, world!", s2); //prints "Hello, world!"  
}

In the code above, *s1* owns the string “Hello” in line 2. When *s2* is assigned to s1, ownership of the string is moved to *s2* in line 3. When *s1* goes out of scope, nothing happens because its value was transferred to *s2*.

Here’s another example of ownership transfer:

fn main() {  
 let s1 = String::from("Hello");  
 let s2 = s1;  
 //println!("{}, world!", s1); <- error  
 println!("{}, world!", s2);  
}

In the code above, the same thing happens: ownership of the string “Hello” is transferred from *s1* to *s2*. However, if we try to print *s1* after it has been assigned to *s2*, the code will not compile. This is because Rust prevents us from reading a value that we don’t own anymore.

One of the key advantages of the ownership model in Rust is that it eliminates the need for garbage collection. Unlike languages like Java or Python, Rust doesn’t rely on a garbage collector to reclaim memory. Instead, ownership allows for deterministic memory management, making Rust programs more predictable and efficient.

**Comparing Rust’s Ownership Model to Garbage Collection**

Garbage collection is a commonly used memory management technique in languages like Java and Python. It automatically reclaims memory that is no longer in use by identifying and freeing objects that are no longer reachable. While garbage collection provides convenience and ease of use, it comes with some overhead in terms of runtime performance and memory usage.

In return, Rust’s ownership model eliminates the need for garbage collection by enforcing strict compile-time checks on memory usage. This enables Rust programs to have predictable and efficient memory management without sacrificing safety. Also, the ownership model allows for fine-grained control over memory allocation and deallocation, which can be beneficial in resource-constrained environments.

Now it’s time for our other important concept: Borrowing!

**Borrowing**

Borrowing is the act of borrowing a value without ownership of it. This allows us to reuse the same allocation for multiple purposes, which reduces the amount of memory our application uses. There are two types of borrowing in Rust: immutable and mutable.

An immutable borrow is used when we want to read a value without changing it:

fn main() {  
 let s1 = String::from("Hello");  
 let len = calculate\_length(&s1);  
 println!("The length of '{}' is {}.", s1, len);  
}  
  
fn calculate\_length(s: &String) -> usize {  
 s.len()  
}

The code above shows how to calculate the length of a string slice without actually changing the *String* value. The *&* character is used to indicate that there’s a reference to the *s1* string that *calculate\_length* function can calculate the length off.

Mutable borrowing, on the other hand, allows for both reading and writing values:

fn main() {  
 let mut s1 = String::from("Hello");  
 change(&mut s1);  
 println!("{}", s1);  
}  
  
fn change(s: &mut String) {  
 s.push\_str(", world!");  
}

In the example above, *s1* is declared as mutable using the *mut* keyword, and then it is passed to the *change* function as a mutable reference. The change function then appends the string “, world!” to the end of the string. Since *s1* is mutable, the function easily modifies it in place.

**Rust Borrowing vs. Ownership**

Rust’s borrowing and ownership are closely related concepts, but they have distinct roles in the language. Ownership refers to the exclusive control a variable has over a piece of data, allowing it to be modified or deallocated. On the other hand, borrowing allows temporary access to data without taking ownership.

Borrowing and ownership systems work together to provide memory security guarantees. Ownership ensures that there is always a single, definitive owner of the data, preventing data races and memory leaks. Checkout provides controlled and secure access to data, ensuring that multiple references do not interfere with each other.

**Lifetimes**

Lifetimes are Rust’s way of keeping track of the lifetime of a reference to ensure that the value borrowed outlives its reference. A lifetime is a time span during which a reference remains valid.

The Rust compiler understands lifetimes and uses them to verify that code stays within the memory safety rules.

Lifetimes are usually not specified explicitly in Rust programs since Rust’s rules for how lifetimes work are designed to reduce the amount of [boilerplate code](https://en.wikipedia.org/wiki/Boilerplate_code#:~:text=In%20computer%20programming%2C%20boilerplate%20code,to%20accomplish%20only%20minor%20functionality.) that is needed to describe the lifetimes accurately. However, there are some cases where explicit annotation is necessary.

Here’s an example with explicit lifetime annotation:

fn main() {  
 let string1 = String::from("Hi");  
 let string2 = String::from("Bye");  
 let result = longest(string1.as\_str(), string2.as\_str());  
 println!("The longest string is '{}'", result);  
}  
  
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
 if x.len() > y.len() {  
 x  
 } else {  
 y  
 }  
}

In this example, the *longest* function takes two string slices as parameters and returns the slice with the longest string. The *<’a>* syntax indicates that the lifetime of the return value and the two parameters have to be the same. The *&’a str* syntax indicates that the two parameters are string slices with the same lifetime.

**Conclusion**

Ownership, borrowing, and lifetimes are key concepts in Rust. Ownership implies that the owner of an object is responsible for its memory allocation and deallocation when the owner is dropped. Borrowing allows multiple immutable and one mutable reference to be borrowed from a value without transferring ownership. Lifetimes ensure that the lifetime of a reference stays within its current scope.

The combination of ownership, borrowing, and lifetimes helps make Rust code safe without compromising on performance. Memory issues like null references and data races are mostly eliminated because of Rust’s static checks on the code. Understanding ownership, borrowing, and lifetimes is essential for writing correct and efficient Rust code.

See you soon, take care. 🚀

**References**

* <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>
* <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html>
* <https://squidarth.com/rc/rust/2018/05/31/rust-borrowing-and-ownership.html>
* <https://medium.com/@bugaevc/understanding-rust-ownership-borrowing-lifetimes-ff9ee9f79a9c>
* <https://medium.com/tadaweb/security-by-design-a-brief-introduction-to-rust-378060e45038>
* <https://gtsslr19.sciencesconf.org/data/program/PChifflier_rust.pdf>
* <https://blog.skylight.io/rust-means-never-having-to-close-a-socket/>