The Rust Programming Language and its Benefits

By

Viacheslav Knyazev

Systems IV

Christine Donahue

12/03/2020

Contents

[Executive Summary 3](#_Toc37004407)

[Introduction 4](#_Toc37004408)

[Who is behind Rust? 4](#_Toc37004409)

[Immutability 4](#_Toc37004410)

[Memory Models, Ownership and Borrowing 4](#_Toc37004411)

[Type System 8](#_Toc37004412)

[No unexpected behavior 9](#_Toc37004413)

[Fearless Concurrency 9](#_Toc37004414)

[Documentation and Testing 10](#_Toc37004415)

[Conclusion 11](#_Toc37004416)

[Sources 11](#_Toc37004417)

[Figure 1: Lexical scoping 5](#_Toc37004396)

[Figure 2: Ownership with stack values 5](#_Toc37004397)

[Figure 3: Ownership with heap values 5](#_Toc37004398)

[Figure 4: Ownership with methods 6](#_Toc37004399)

[Figure 5: Returning ownership 7](#_Toc37004400)

[Figure 6: Borrowing ownership 7](#_Toc37004401)

[Figure 7: Dangling reference 8](#_Toc37004402)

[Figure 8: IpAddr Enum, from the Rust standard library 8](#_Toc37004403)

[Figure 9: Extending and matching against a new trait 9](#_Toc37004404)

[Figure 10: Synchronizing a counter using a mutex 10](#_Toc37004405)

[Figure 11: Inline documentation tests 11](#_Toc37004406)

# Executive Summary

Rust is a new, modern, low-level language which is rapidly gaining appreciation in the programming world. Mozilla has set a high bar for itself with respect to the design of the language and is succeeding in delivering an ergonomic language with novel memory-management concepts. Learning about these concepts through the exercise of discovering Rust will yields a greater understanding of how memory works and patterns which produce cleaner code and more robust software.

The feature-set offered by Rust is unique in the Computer Science world with its novel concept of ownership and innovations in the usage of type algebra. Interacting with these new features can give a new perspective on other languages and teach about programming constructs often lacking in more mainstream programming languages.

# Introduction

As part of the research project of the Computer Science program, this report aims to cover Rust from a very high-level in terms of its safety guarantees and unusual features, without diving into the details of syntax or underlying implementations.

Featuring marketing pitches such “Fast, reliable, productive – pick three” [1] and “Fearless concurrency”, Rust is modern, low-level language heavily inspired by a multitude of languages such as OCaml, C++, Haskell, Erlang, C#, Swift and many others[17], while including multiple novel innovations such as the borrow-checker at the heart of the language.

It is winning the hearts of many developers as reflected in Rust topping the results in the yearly StackOverflow developer surveys [4].

In brief, Rust is a language featuring no garbage-collector or runtime backed by an elaborate yet strict type system which compiles directly to machine-code [14].

# Who is behind Rust?

While the project has been initiated and shepherded by Mozilla, the development of every part of the language, ranging from the specifications, to the compiler details is done in the open by a very large group of volunteers. To date, the project has been contributed by over 2,500 developers.

# Memory Models, Ownership and Borrowing

Traditionally, there have been two approaches to manage memory. The first is as is done by languages such as C, featuring manual memory allocation and release [2]. This strategy is very efficient; however, mistakes can lead to fatal crashes, memory leaks and a wide variety of security vulnerabilities such a Use-After-Free. This model has been largely superseded by the second strategy of using a garbage collector in non-performance-critical contexts, as the garbage collector must periodically analyze the program in order to scan the free-able memory which carries a performance penalty [3].

Rust introduces a novel paradigm, featuring deterministic garbage collection via the concept of ownership. By the adherence to three rules, Rust can guarantee cheap and corruption-free memory management [5]:

* Each value has a variable which is its owner
* There can only be one owner at a time
* When the owner goes out of scope, the value will be disposed of

The scope is defined lexically as demonstrated in Figure 1.



Figure : Lexical scoping

Like most languages, Rust has two types of values: primitives and pointers. Primitives include types such as Integers, Floats, Booleans, and other types which can fit on the stack via the attribute of having a known size at compile-time, while more complex types such as Strings will be stored as pointers on the stack pointing to somewhere on the heap.

Generally, when re-assigned to a variable, stack-allocated values will be copied such that both variables will be the owners of different copies of the same value as shown in Figure 2.

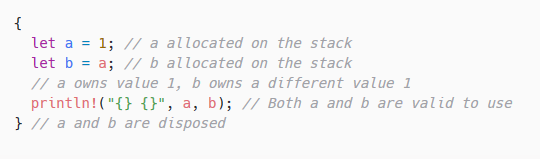


Figure : Ownership with stack values

Heap-allocated values on the other hand, will transfer ownership and multiple pointers to the same location would result in two owners of that data as shown in Figure 3.

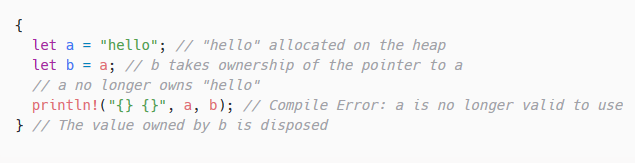


Figure : Ownership with heap values

This behavior extends in a natural way to methods, as shown in Figure 4.

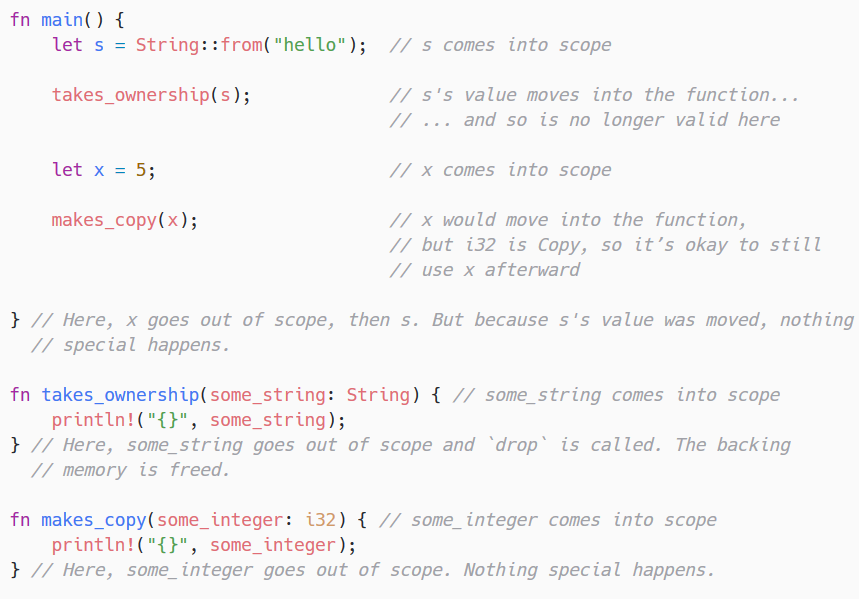


Figure : Ownership with methods

If it is not desirable for Rust to dispose of a value, it can either be returned from the called method or passed as a reference in which case the compiler will enforce that the owner is not disposed of prior to the completion of the invoked method. This behavior is referred to as “borrowing”. The two approaches are displayed in Figure 5 and Figure 6 respectively.

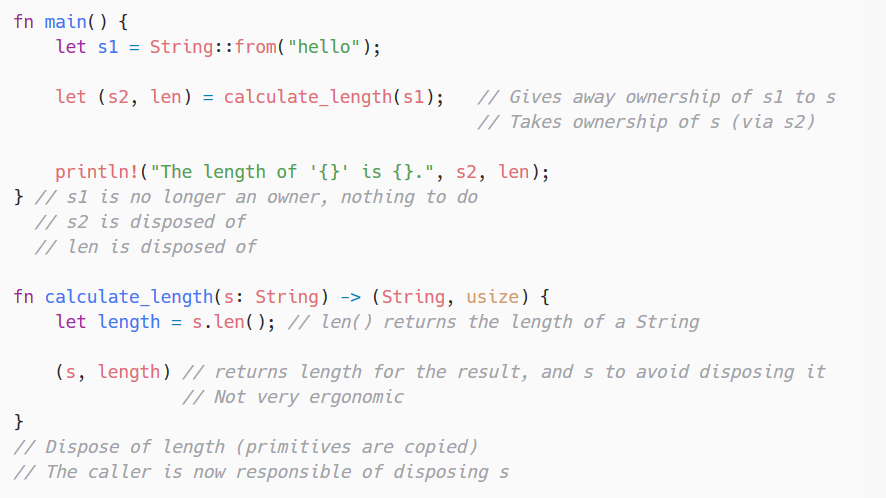


Figure : Returning ownership

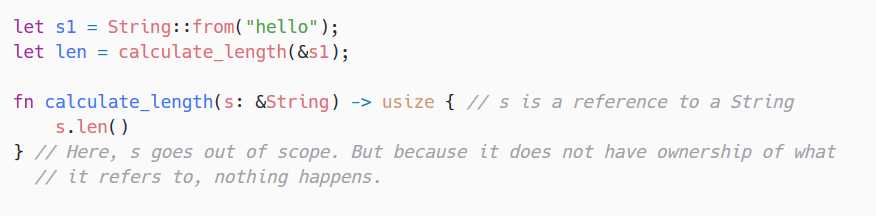


Figure : Borrowing ownership

The concept of borrowing introduces two new rules:

* Borrowing values whose lexical lifetime is shorter than the borrowers is not allowed
* Each value can be borrowed in one of two ways, but not both at the same time.
  + One of more immutable references
  + One mutable reference

The first rule is there simply to ensure that dangling references never happen, which guarantees the absence of use-after-free errors. An example of a dangling reference is shown in Figure 7.

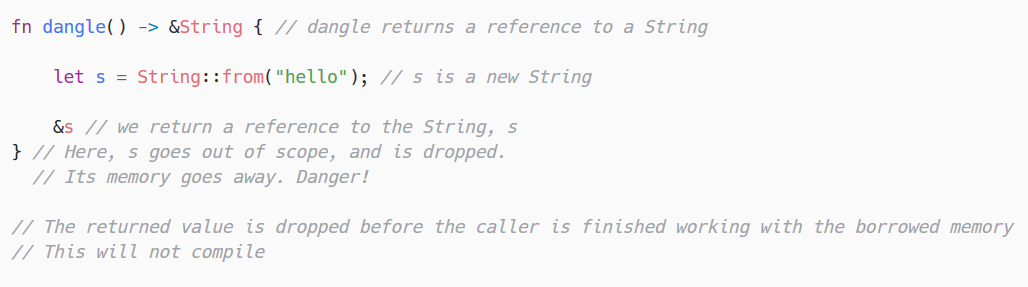


Figure : Dangling reference

The second rule is to ensure that the state of data does not change when not expected to. Simply put, when a value is borrowed as immutable, the borrower expects that the value will not suddenly change between calls. If the value is mutable elsewhere, this guarantee is impossible uphold in a concurrent environment.

# Type System

Rust has a powerful type system largely inspired by the OCaml’s type algebra, which has allowed to incorporate ergonomic features such as type inference and pattern matching. Beyond the type theory and ergonomics, Rust presents a new programming paradigm similar to C#’s interfaces pushed to an extreme. Rust offers 3 fundamental constructs which you can build your objects with: Enums, Structs and Traits [13][9].

Enums are enumeration types based upon OCaml’s notion of an enum and similar to those found in languages such as Java, with the notable exception that each enumerated type can hold data. An example of their usage is shown in Figure 8.

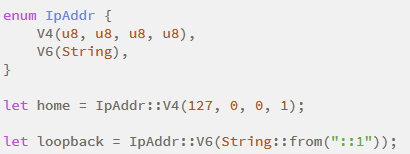


Figure : IpAddr Enum (differs from the Rust standard library)

The Rust concept of a struct analogous to the concept of Classes in traditional Object-Oriented languages. It is the description of the shape of objects including properties and methods. It does not, however adhere to ideas of inheritance. Code-sharing and polymorphism is accomplished via the mechanism of implementing traits, which are reminiscent of interfaces [18].

Any struct can selectively implement traits for generic behavior, which can then be matched against in method signatures. Unlike C#, these implementations are written separately from the struct’s definition. This enables third party libraries to extend native types and later match against those extended implementations. An example of this capability is displayed in Figure 9.



Figure : Extending and matching against a new trait

## No unexpected behavior

The design of the type system eliminates frustratingly ambiguous experiences found in other programming languages. In Java for example, when invoking a method perform a computation, there are multiple ways an error case can be handled:

* Return a null
* Throw an exception
* Return result object with an error field/flag

Furthermore, there is no distinction between critical errors and non-critical errors as both are the same class of exceptions.

Rust solves both issues by replacing the ambiguous errors with a standard variant system, which must be explicitly checked. Each potentially failing method call will return a Result<T, E> type containing either a Ok<T> or an Error<E> for non-critical cases where the program can recover, such that the type system forces the logic to explicitly handle both cases. Additionally, Rust features a second type of errors called panics, which are invoked when there was a fatal failure, which indicates a non-recoverable state such as running out working memory.

# Fearless Concurrency

The ownership rules and the type system culminate in one of Rust’s marketing pitches: “Fearless concurrency”. The concept of concurrent programming typically groups two types of processes together: Concurrency where the program does multiple different things at the same time and parallelism where the program distributes computation across multiple threads. [11]

Any sort of concurrent programming has been historically very error-prone and difficult to debug due its non-deterministic nature resulting in issues such as race conditions, poor state isolate and dead-locks. While Rust is not a panacea for these issues, it does have many preventative measures to avoid them proactively. For instance, unlike the manual mutex-management patterns offered by most languages, Rust can take advantage of its ties between lexical scoping and lifetimes in order to automatically manage data locking as shown in Figure 11. Another example is how types are individually profiled to be thread-safe based on their characteristics, which avoids a plethora of issues stemming from the lack of underlying atomicity in data structures in languages such as C++.

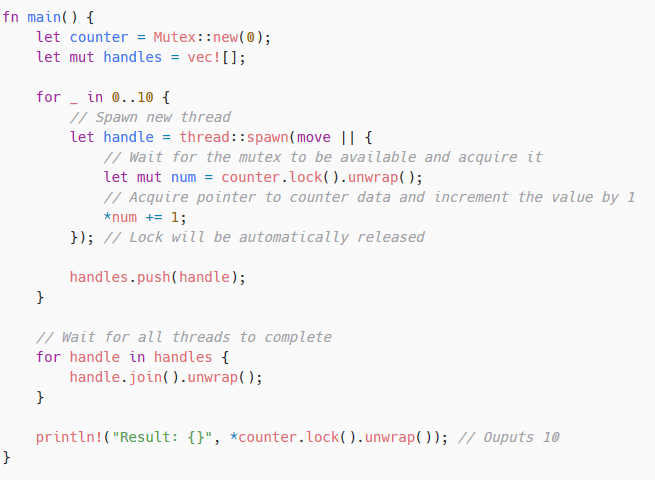


Figure : Synchronizing a counter using a mutex

# Documentation and Testing

The Rust toolchain features extensive support for best practices surrounding testing and documentation. This is to say that the Rust compiler has built-in support for producing publishable, quality documentation based solely on source-code, tests and comments and first-class support for tests.

Writing tests is as simple as writing methods labeled with a #[test] attribute within a project to create unit tests, or outside of it for integration tests. [16]

The deeply interwoven integration of these features allows for functionality rarely found in other platforms. For example, any code written in inline documentation for demonstrative purposes will be compiled and executed to ensure that it is not out-of-date. This both ensures quality documentation and actively rewards writing tests and documentation instead of seeing it as a time sink. [8]

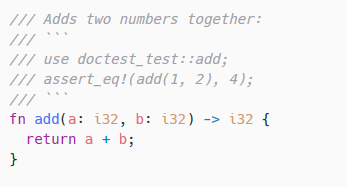


Figure 11: Inline documentation tests

# Conclusion

Programming courses should include Rust in some capacity as it helps ingrain in students both good programming practices and a deeper understanding of memory management as it relates to all programming languages. This would be helpful as patterns which Rust strictly enforces can be transferred to any language and be applied in order to produce higher quality code with cleaner abstractions and less fear of multithreading.

Additionally, it could help students discover a new appreciation for both testing and documentation which is often done as an after-thought instead of a primary concern.

# Sources

1: “Announcing Rust 1.12: Rust Blog.” The Rust Programming Language Blog, 29 Sept. 2016, blog.rust-lang.org/2016/09/29/Rust-1.12.html.

2: “C - Memory Management.” Tutorialspoint, www.tutorialspoint.com/cprogramming/c\_memory\_management.htm.

3: “Garbage Collection and Application Performance.” Dynatrace, www.dynatrace.com/resources/ebooks/javabook/impact-of-garbage-collection-on-performance/.

4: Goulding, Jake. “What Is Rust and Why Is It so Popular?” Stack Overflow Blog, 6 Feb. 2020, stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular.

5: Klabnik, Steve. “The Rust Programming Language.” Understanding Ownership - The Rust Programming Language, doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

6: Klabnik, Steve. “The Rust Programming Language.” Using Structs to Structure Related Data - The Rust Programming Language, doc.rust-lang.org/book/ch05-00-structs.html.

7: Klabnik, Steve. “The Rust Programming Language.” Understanding Ownership - The Rust Programming Language, doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

8: Klabnik, Steve. Documentation, doc.rust-lang.org/1.4.0/book/documentation.html.

9: Klabnik, Steve. “The Rust Programming Language.” Using Structs to Structure Related Data - The Rust Programming Language, doc.rust-lang.org/book/ch05-00-structs.html.

10: Klabnik, Steve. “The Rust Programming Language.” Object Oriented Programming Features of Rust - The Rust Programming Language, doc.rust-lang.org/book/ch17-00-oop.html.

11: Klabnik, Steve. “The Rust Programming Language.” Fearless Concurrency - The Rust Programming Language, doc.rust-lang.org/book/ch16-00-concurrency.html.

12: “Production.” Rust Programming Language, www.rust-lang.org/production.

13: “Rust By Example.” Traits - Rust By Example, doc.rust-lang.org/rust-by-example/trait.html.

14: “Rust Programming Language.” The Rust Design FAQ, doc.rust-lang.org/1.6.0/complement-design-faq.html.

15: “Rust Standard Library.” Documentation, doc.rust-lang.org/1.4.0/book/documentation.html.

16: “Testing.” Rust Standard Library, doc.rust-lang.org/1.4.0/book/testing.html.

17: “The Rust Reference.” Influences - The Rust Reference, doc.rust-lang.org/reference/influences.html.

18: Turon, Aaron. “Abstraction without Overhead: Traits in Rust: Rust Blog.” The Rust Programming Language Blog, 11 May 2015, blog.rust-lang.org/2015/05/11/traits.html.