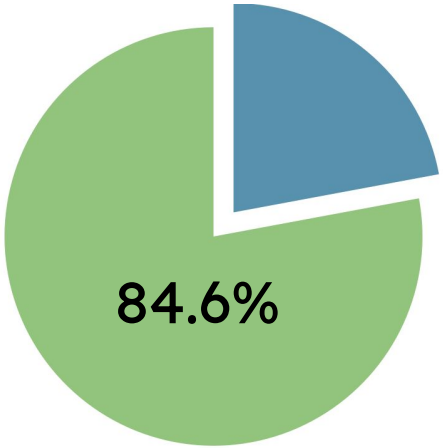
Examining Rust Developers' Motivations for Using `unsafe` Code

Ian McCormack

Software & Societal Systems
icmccorm.me

Rust has been voted
most loved or **admired language**
on Stack Overflow's annual developer survey
for the past 8 years.

Developers who used Rust in 2023 and want to continue in 2024.



Energy efficiency across Programming Languages:
how do energy, time, and memory relate?

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57

Pereira et al., 2017

Rust restricts *aliasing* and *mutability* to provide **static guarantees of memory and thread safety.**

The Borrow Checker

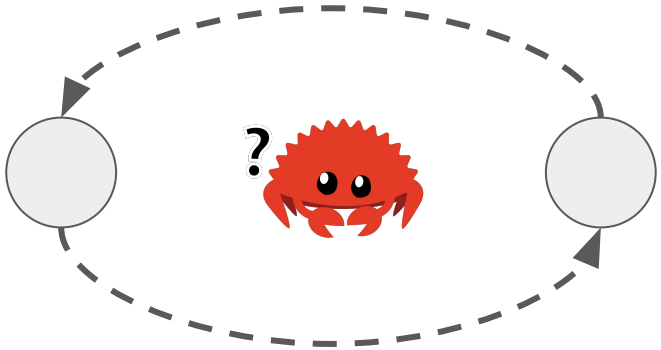
Crichton, 2021

All values have **exactly one owner**.

A reference to a value **cannot outlive** the owner.

A value can have **one mutable** reference **or many immutable** references, but not both.

Rust's borrow checker is necessarily **conservative**.
There are safe patterns it **cannot prove**.



There are memory effects **beyond its scope.**

Rust's **unsafe** keyword enables features that escape these restrictions.

- *Dereference raw pointers*
- *Call* **unsafe** *functions*
- *Mutable static variables*
- *Implement* **unsafe** *traits*
- *Access fields of union types*

These features are **essential but dangerous!**

We interviewed **19 Rust developers** who “regularly write or edit” `unsafe` code to understand their motivations.

We recruited in April, 2023 from **Reddit**, **Twitter**, the **Rust Forums**, **Discord**, and direct contacts.

Hanan Hibshi
CMU - INI

Joshua Sunshine
CMU - S3D

Jonathan Aldrich
CMU - S3D

Tomas Dougan
Brown

Sam Estep
CMU - S3D

Rust was intuitive for C++ developers.

> This is how we think anyway.
>
> We're working on a large
>
> C++ code base; this is what we're doing in
>
> our head, so now it's doing it for us. (P2)

Developers used **unsafe** code
for three distinct reasons.

1. Performance
2. Ease-of-use
3. No safe alternative

`unsafe` performs better.

> “ …you’re writing `unsafe` code anyway.
> You understand how to like use this very
> like sharp-edged abstraction properly.
>
> And so when I switched over to using
> [a bump allocator], it cut like, I think it
> was like 20 to 40% off some benchmarks. (P3)

`unsafe` is easier to use.

> So I just made that one `'static`. Transmuted that one to `'static`, which is another use case for `unsafe` is working around a bad API when you know that you can use it correctly. (P14)

There is no safe alternative.

> That's just an address. I just have to trust it. I have to jump to that address and I really can't do anything to verify it...that information is lost during compilation. (P12)

These motivations are **not** mutually exclusive.

1. ## Performance

   Zero-copy deserialization achieves better performance,
   which is necessary for our domain.

2. ## Ease-of-use

   An existing library, `rkvy`, is not ergonomic for this use case.

3. ## No safe alternative

   Zero-copy deserialization fundamentally requires some `unsafe` code.

Developers prefer to expose safe APIs, but they can't always guarantee correctness.

> No library author knows when they're going to trigger [undefined behavior]... we do our best to give you like a sound interface, but god knows if there's a hole in it...
>
> You never know when a user is going to stumble like into something you just didn't cover. (P9)

Developers prefer to minimize interactions with foreign functions.

> If you’re getting to that point where it’s a bit concerning...you probably need a really good reason to continue...
>
> You should be stepping back and being like, “okay, why do I need to get down? Why do I have to worry about this?” (P7)

Developers are uncertain about Rust's semantics.

> I think the biggest issue with Rust's **`unsafe`** isn't necessarily the tools, but just the spec—or rather, the absence of the spec.
>
> If you write **`unsafe`** code today, you write it against the void. (P9)

Are you a Rust developer?
Have you used `unsafe` code?

Please consider taking
our 20-minute survey!