Hot-or-Not RUST
with Niko Matsakis

Author: Bart Sanders, Manager Embedded Control Software Discipline

January 8th, 2018

Welcome

Timetable

18:00h	Introduction
18:03h	Rust, part 1
19:30h	Break
20:00h	Rust, part 2
20:45h	Q & A
21:00h	Drinks

```
process::exit(0xBB);
```

HOT
OR
NOT

Niko Matsakis will explain how Rust helps us become more productive.

Hack without fear!

Rust: Reach Further!
Nicholas Matsakis

Rust:

Safe, fast code that **works**.

Safe

Double free?

Dangling pointers?

Buffer overflow?

Data races?

Safety =

Eat your spinach!

Photo credit: Sanjoy Ghosh
https://www.flickr.com/photos/sanjoy/4016632253/

Safety = Eat your spinach!

Photo credit: Salim Virji
https://www.flickr.com/photos/salim/8594532469/

Saved by the compiler: Parallelizing a loop with Rust and rayon

Eric Kidd on Thursday 20 Oct 2016

The Rust compiler just saved me from a nasty threading bug. I was working on cage (our open source development tool for Docker apps with lots of microservices), and I decided to parallelize the routine that transformed docker-compose.yml files.

Rust:

Safe, fast code

that **works**.

Fast

Zero-cost abstractions:

- High-level code, low-level efficiency
- No compiler heroics needed

No garbage collector:

- Predictable, memory usage; no pauses
- Apply techniques to other resources (sockets, etc)

Indeed, no mandatory runtime at all:

- Embedded, WASM, or standalone libraries

```ruby
class ::String
  def blank?
    /\A[[:space:]]*\z/ == self
  end
end
```

Performance

Ruby:
964K iter/sec

```c
static VALUE
rb_str_blank_as(VALUE str)
{
  rb_encoding *enc;
  char *s, *e;

  enc = STR_ENC_GET(str);
  s = RSTRING_PTR(str);
  if (!s || RSTRING_LEN(str) == 0) return Qtrue;

  e = RSTRING_END(str);
  while (s < e) {
    int n;
    unsigned int cc = rb_enc_codepoint_len(s, e, &n, enc);

    switch (cc) {
      case 9:
      case 0xa:
      case 0xb:
      case 0xc:
      case 0xd:
      case 0x20:
      case 0x85:
      case 0xa0:
      case 0x1680:
      case 0x2000:
      case 0x2001:
      case 0x2002:
      case 0x2003:
      case 0x2004:
      case 0x2005:
      case 0x2006:
      case 0x2007:
      case 0x2008:
      case 0x2009:
      case 0x200a:
      case 0x2028:
      case 0x2029:
      case 0x202f:
      case 0x205f:
      case 0x3000:
#if ruby_version_before_2_2()
      case 0x180e:
#endif
          /* found */
          break;
      default:
          return Qfalse;
    }
    s += n;
  }
  return Qtrue;
}
```

https://github.com/SamSaffron/fast_blank

Performance

Ruby:
964K iter/sec

10x!

C:
10.5M iter/sec

```ruby
class ::String
  def blank?
    /\A[[:space:]]*\z/ == self
  end
end
```

```rust
extern "C" fn fast_blank(buf: Buf) -> bool {
  buf.as_slice().chars().all(|c| c.is_whitespace())
}
```

Get Rust string slice

Get iterator over each character

Are all characters whitespace?

Performance

Ruby:
964K iter/sec

C:
10.5M iter/sec

Rust:
11M iter/sec

High-level, zero-cost abstractions

```
fn is_whitespace(text: &str) -> bool {
  text.chars()
    .all(|c| c.is_whitespace())
}
```

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.par_iter()
    .map(|path| Image::load(path))
    .collect()
}
```

CARGO
packages for Rust
Click or press 'S' to search...
Browse All Crates | Docs ▾ | Log in with GitHub

The Rust community's crate host

Install Cargo
Getting Started

Rust:

Safe, fast code that **works**.

Documentation Install Community Contribute

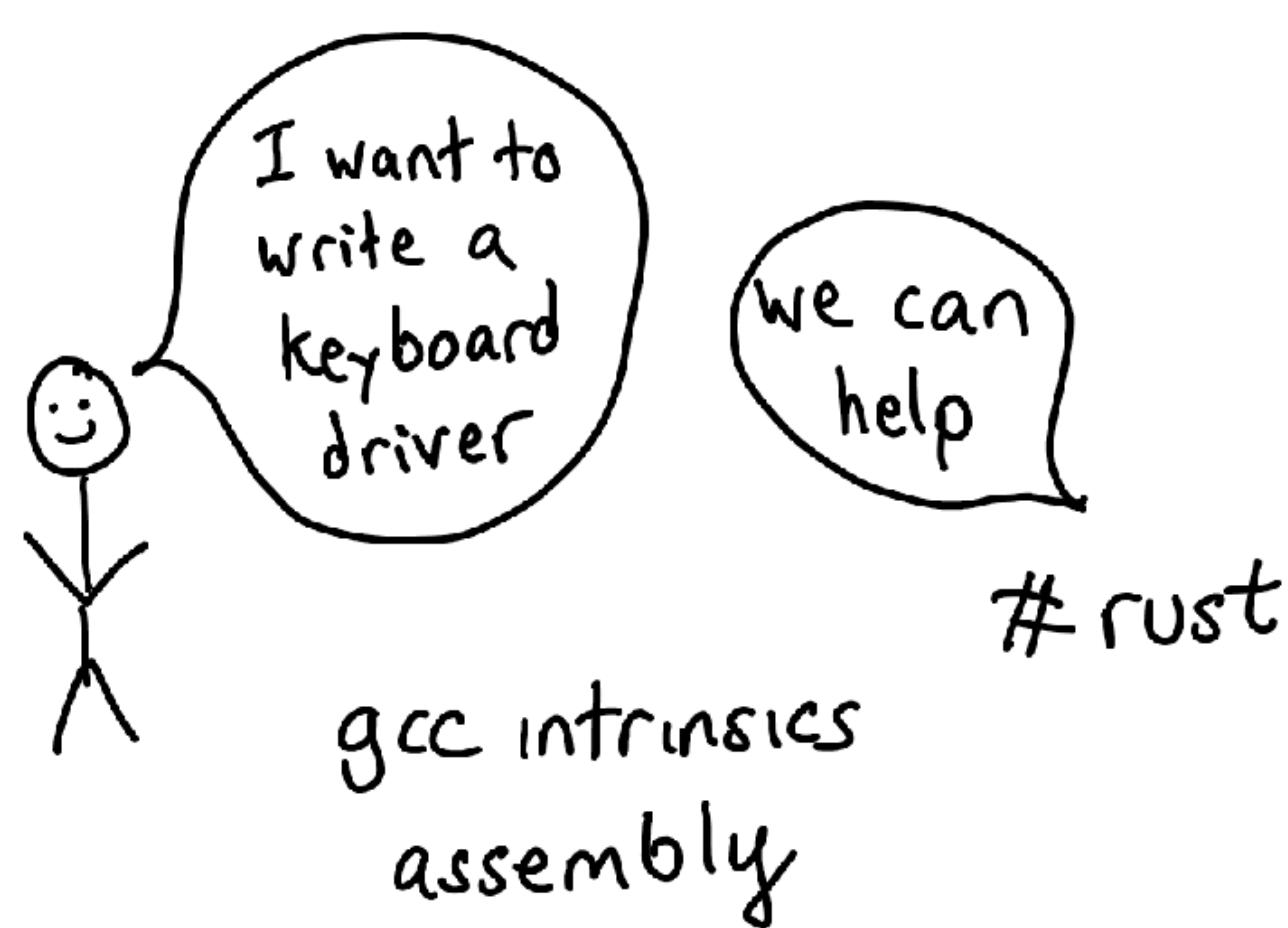
Friends of Rust

(Organizations running Rust in production)

npm

I like Rust because it is **boring**.
— CJ Silverio, npm CTO

Open and welcoming



From http://jvns.ca/blog/2016/09/11/rustconf-keynote/

Ownership and Borrowing

Parallelism in Rust

Traits

Unsafe Rust

Rust in Production

Ownership and Borrowing
Photo Credit: Nathan Kam
https://www.youtube.com/watch?v=Tnssn9KcWLg

Rust

Zero-cost abstractions

Memory safety & data-race freedom

Zero-cost abstractions

```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
  …
}
```

Stack and inline layout.

Lightweight references

Deterministic destruction

vector

Stack
…
data
length
capacity
elem

Heap
[0]
[…]
[n]

string

‘H’
‘e’
…

Stack

Heap

C++

Memory safety

```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
  vector.push_back(some_string);
  cout << elem;
}
```

Mutating the vector freed old contents.

vector

…

data

length

capacity

elem

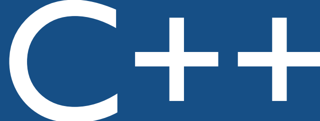
[0]

[1]

[0]

Dangling pointer: *pointer to freed memory.*

Aliasing: *more than one pointer to same memory.*

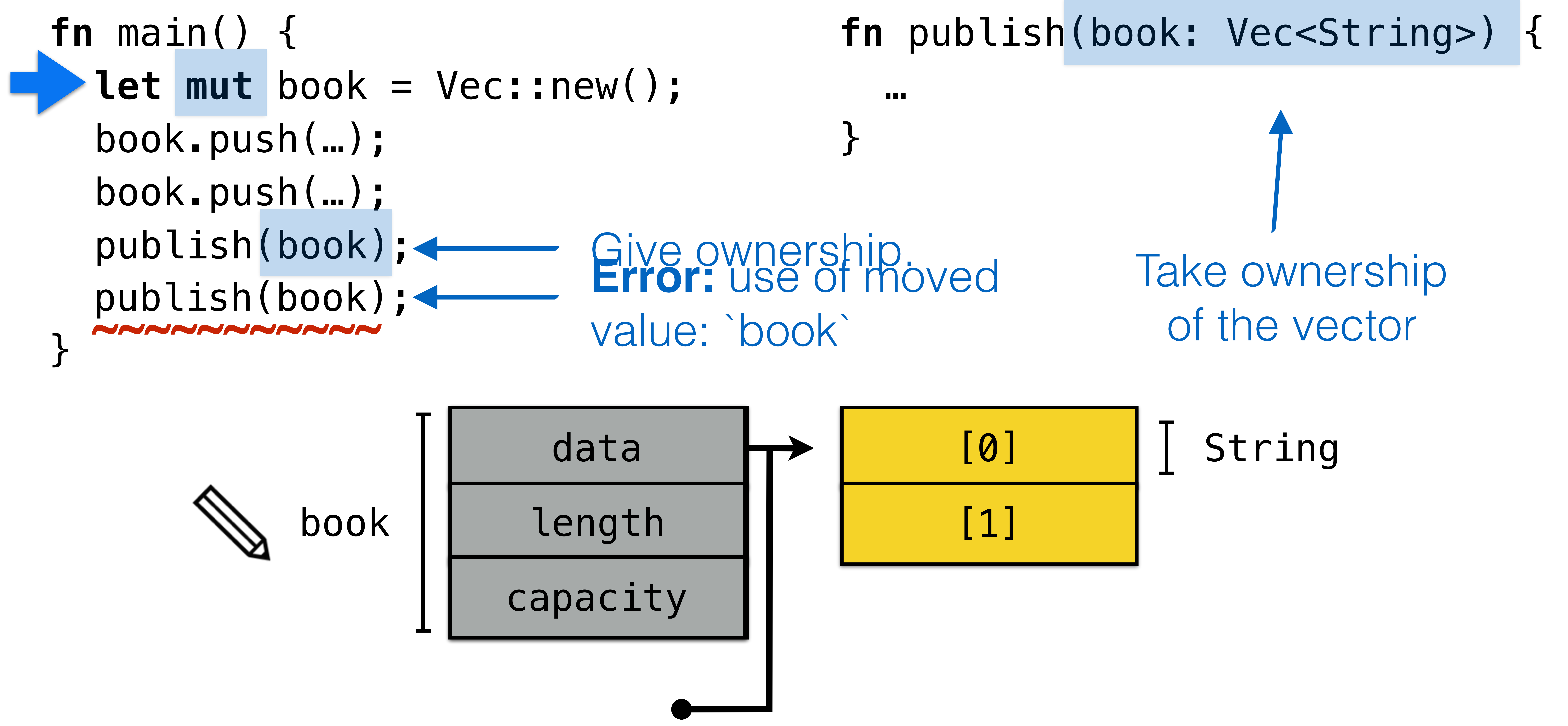


Not just about memory allocation

```
union {
  void *ptr;
  uintptr_t integer;
} u;

auto p = &u.ptr;
u.integer = 0x12345678;
use(*p); // uh-oh
```

~ Ownership and borrowing ~

Type	Ownership	Alias?	Mutate?
T	Owned		✓



Ownership

“Manual” memory management in Rust:

Values owned by **creator**.

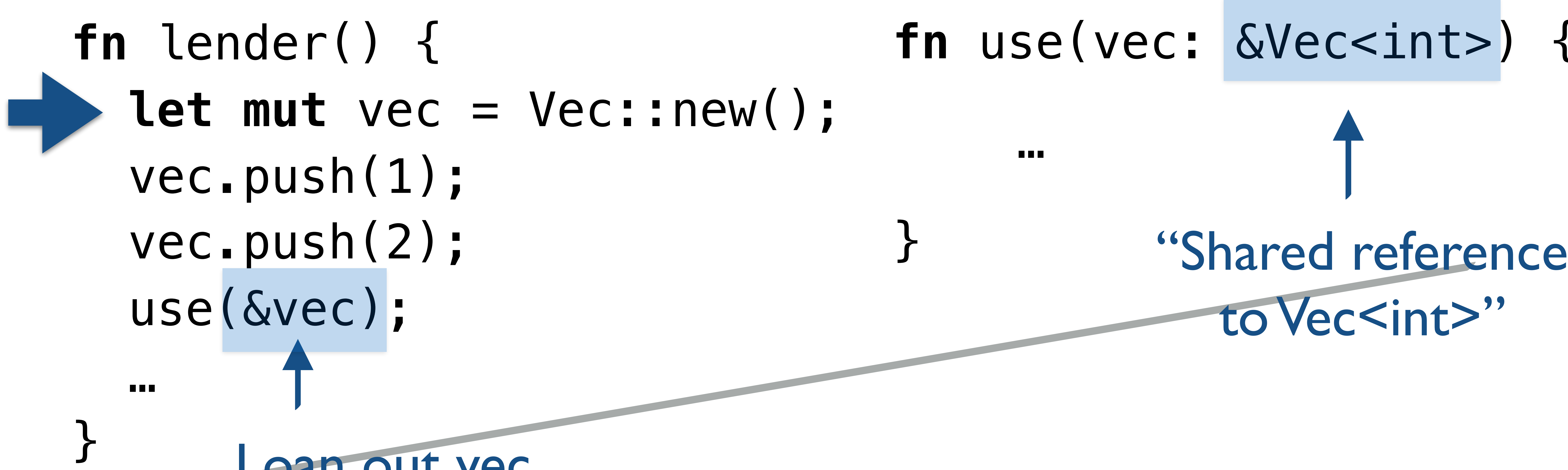
Values **moved** via assignment.

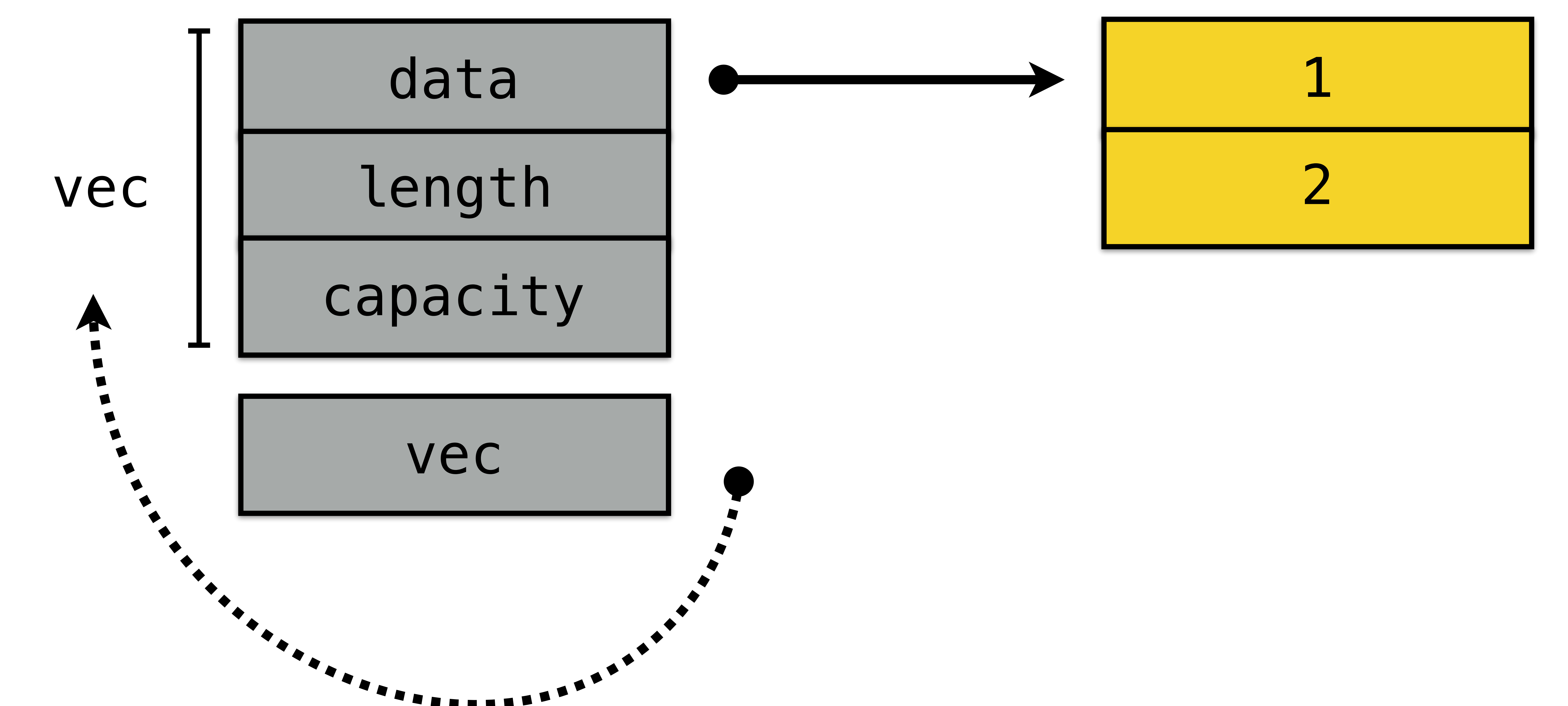
When final owner returns, **value is freed**.

Feels invisible.

~ Ownership and borrowing ~

Type	Ownership	Alias?	Mutate?
T	Owned		✓
&T	Shared reference	✓	


fn lender() {
let mut vec = Vec::new();
vec.push(1);
vec.push(2);
use(&vec);
…
}
Loan out vec
fn use(vec: &Vec<int>) {
…
}
“Shared reference to Vec<int>”


vec
data
length
capacity
1
2
vec

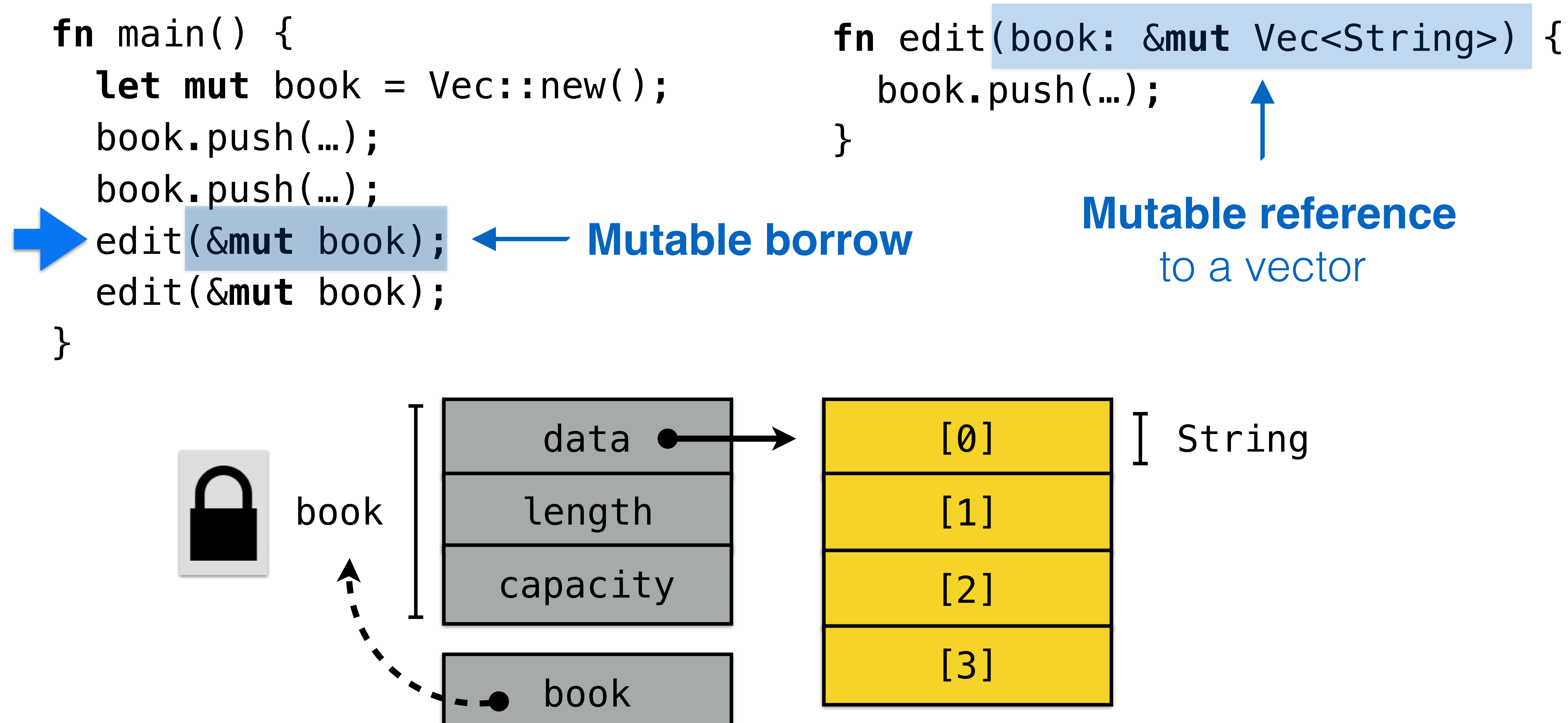
Sharing “freezes” data (temporarily)

```
let mut book = Vec::new();

book.push(…);

let r = &book;

book.len();

book.push(…);

r.push(…);

book.push(…);
```

- `book` **mutable** here
- `book` **borrowed** here
- reading `book` ok while shared
- cannot mutate while shared
- cannot mutate through shared ref
- after last use of `r`, `book` is **mutable again**

~ Ownership and borrowing ~

Type	Ownership	Alias?	Mutate?
T	Owned		✓
&T	Shared reference	✓	
&mut T	Mutable reference		✓



Mutable borrow

Mutable references: no other access

```
let mut book = Vec::new();

book.push(…);

let r = &mut book;

book.len();

r.push(…);

book.push(…);
```

book **mutable** here

book **borrowed** here

cannot **access** while borrowed

but can mutate through `r`

after last use of `r`,
book is accessible again

~ Closures ~

Definition: a closure is a callback that Just Works.
— The Reinvigorated Programmer

https://reprog.wordpress.com/2010/02/27/closures-finally-explained/

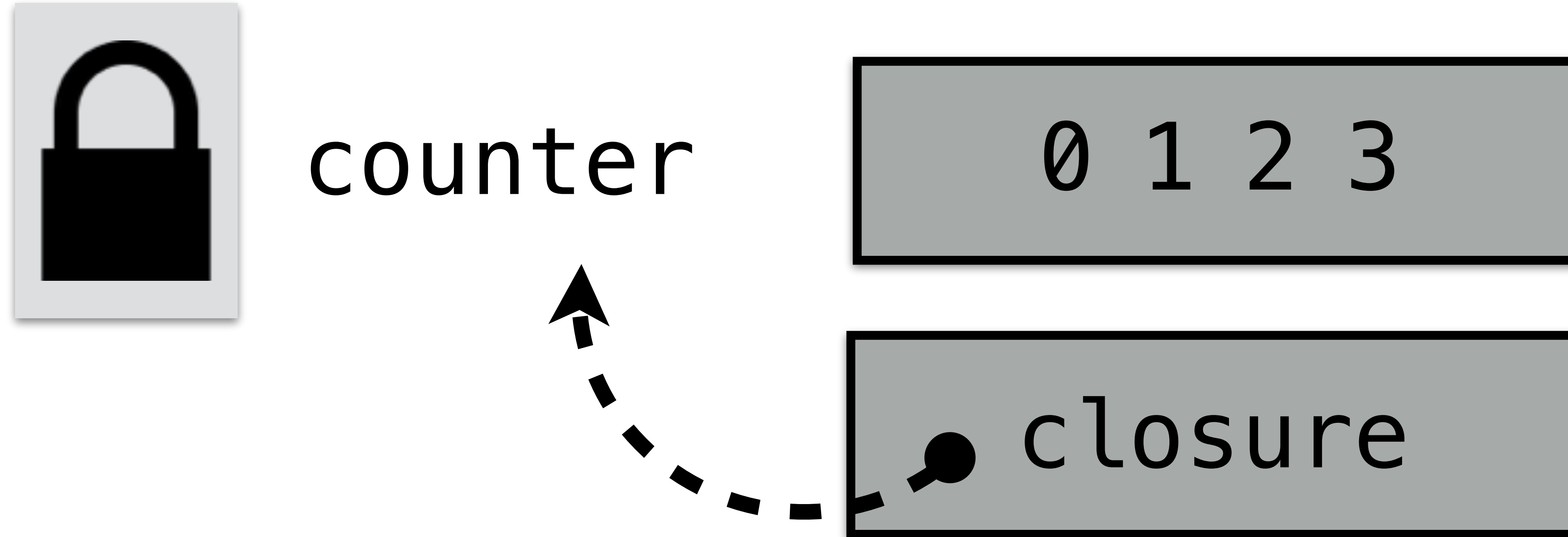
```rust
fn main() {
    let mut counter = 0;
    let mut closure = || {
        counter += 1;
    };
    closure();
    counter += 1;
    closure();
    counter += 1;
}
```

creates a **closure**

closure **borrows** `counter` from enclosing stack frame

cannot access while borrowed

done using closure; ok



~ Named lifetimes ~

There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-one errors.
— Leon Bambrick

```
impl<T> [T] {
  fn split_at_mut(
    &'a mut self,
    mid: usize,
  ) -> (&'a mut [T], &'a mut [T]) {
    …
  }
}
```

given a slice of T elements…

and a midpoint

divide slice into two

self: &mut [i32]

[0]	[1]	[2]	[3]	[…]	[n]

less: &mut [i32]   greater: &mut [i32]

```
fn foo(vec: &mut [T]) {
  let (less, greater) =
    vec.split_at_mut(3);
  …
}
```

`vec` **borrowed** here while `less` and `greater` still in use

```c
const git_tree_entry *
git_tree_entry_byname(const git_tree *tree,
                      const char *filename);
```

This returns a git_tree_entry that is owned by the git_tree. You don't have to free it, but you must not use it after the git_tree is released.

```rust
impl Tree {
  fn by_name<‘a>(&’a self, filename: &str) -> &’a TreeEntry {
    ..
  }
}
```

“Returns a reference derived from `self`”

```
const git_tree_entry *
git_tree_entry_byname(const git_tree *tree,
                      const char *filename);
```

Read-only, yes, but mutable through an alias?

Will `git_tree_entry_byname` keep this pointer?
Start a thread using it?

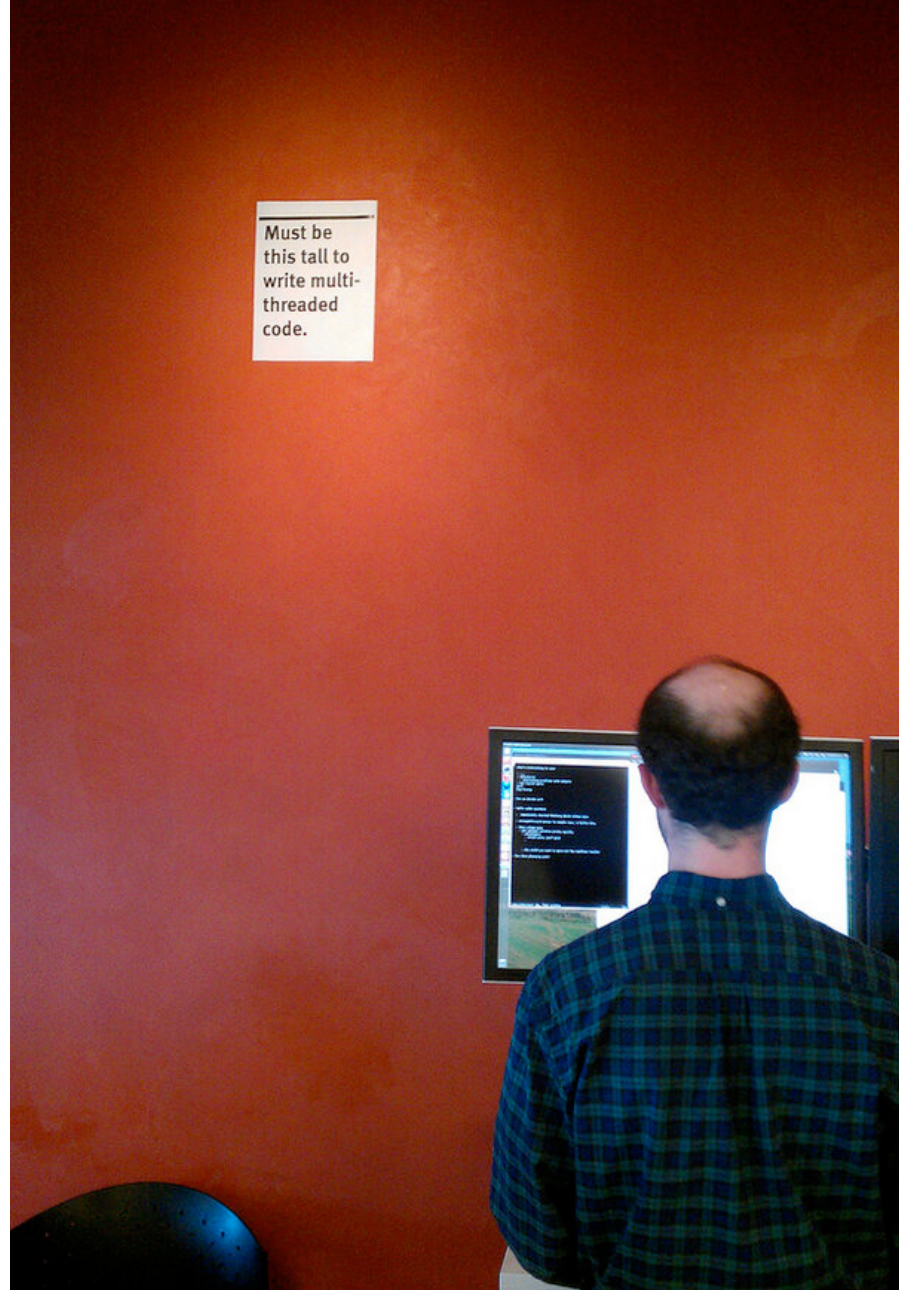
This returns a git_tree_entry that is owned by the git_tree. You don't have to free it, but you must not use it after the git_tree is released.

```
impl Tree {
  fn by_name<'a>(&'a self, filename: &str) -> &'a TreeEntry {
    ..
  }
}
```

Borrowed string
Does not escape `by_name`
Immutable while `by_name` executes

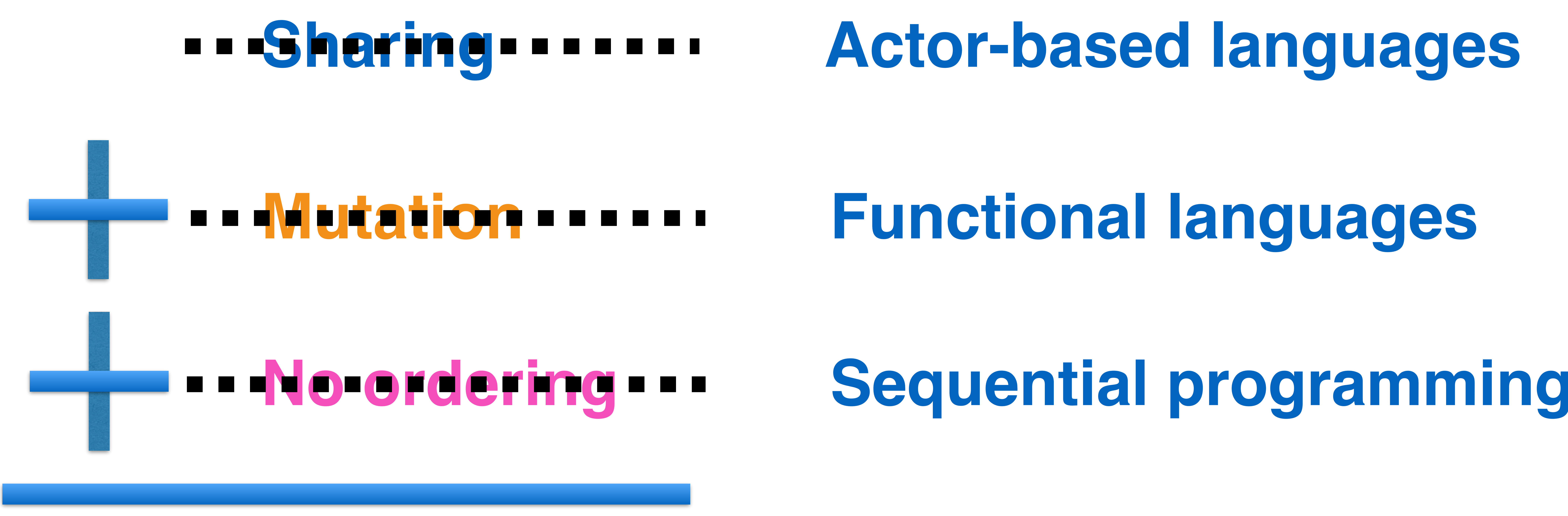
Parallelism
Photo credit: Dave Gingrich
https://www.flickr.com/photos/ndanger/2744507570/

“Must be this tall to write multi-threaded code”

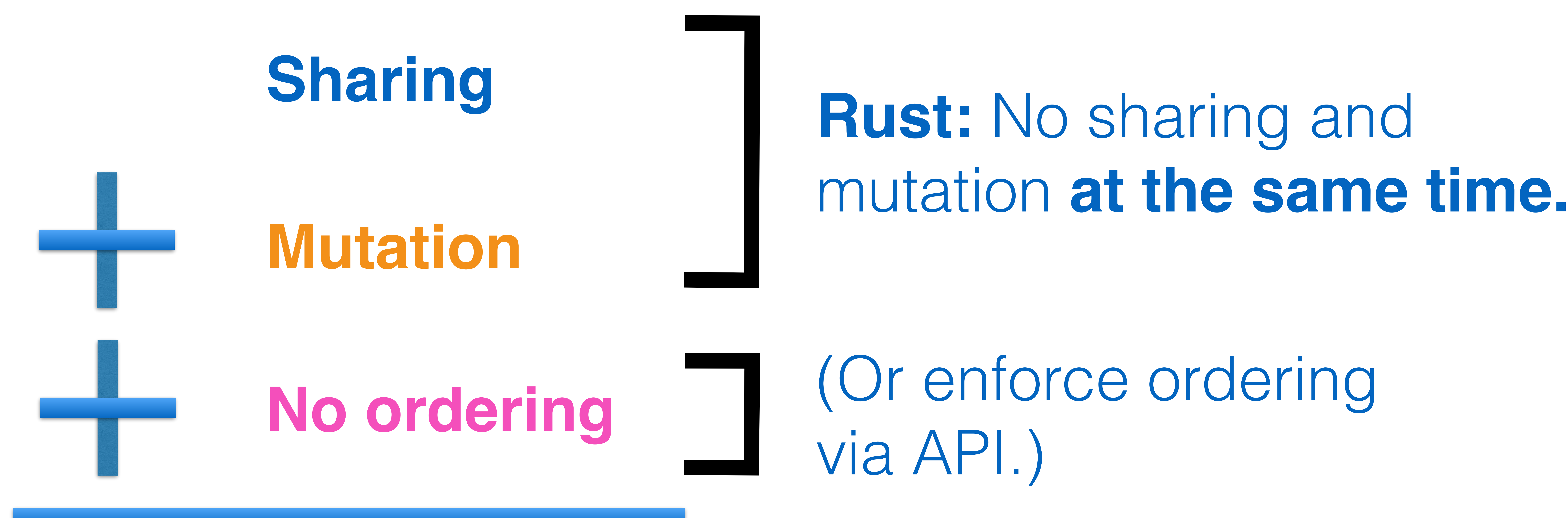


David Baron
Mozilla Distinguished Engineer

Data races



Data races



Observation:

Building parallel abstractions is easy.

Misusing those abstractions is also easy.

Go Code

```
func foo(…) {
  m := make(map[string]string)
  m[“Hello”] = “World”
  channel <- m
  m[“Hello”] = “Data Race”
}
```

send data over channel

but how to stop sender from using it afterwards?

```
fn foo(…) {
  let m = HashMap::new();
  m.insert("Hello", "World");
  channel.send(m);
  m.insert("Hello", "Data Race");
}
```

Error: use of moved value: `book`

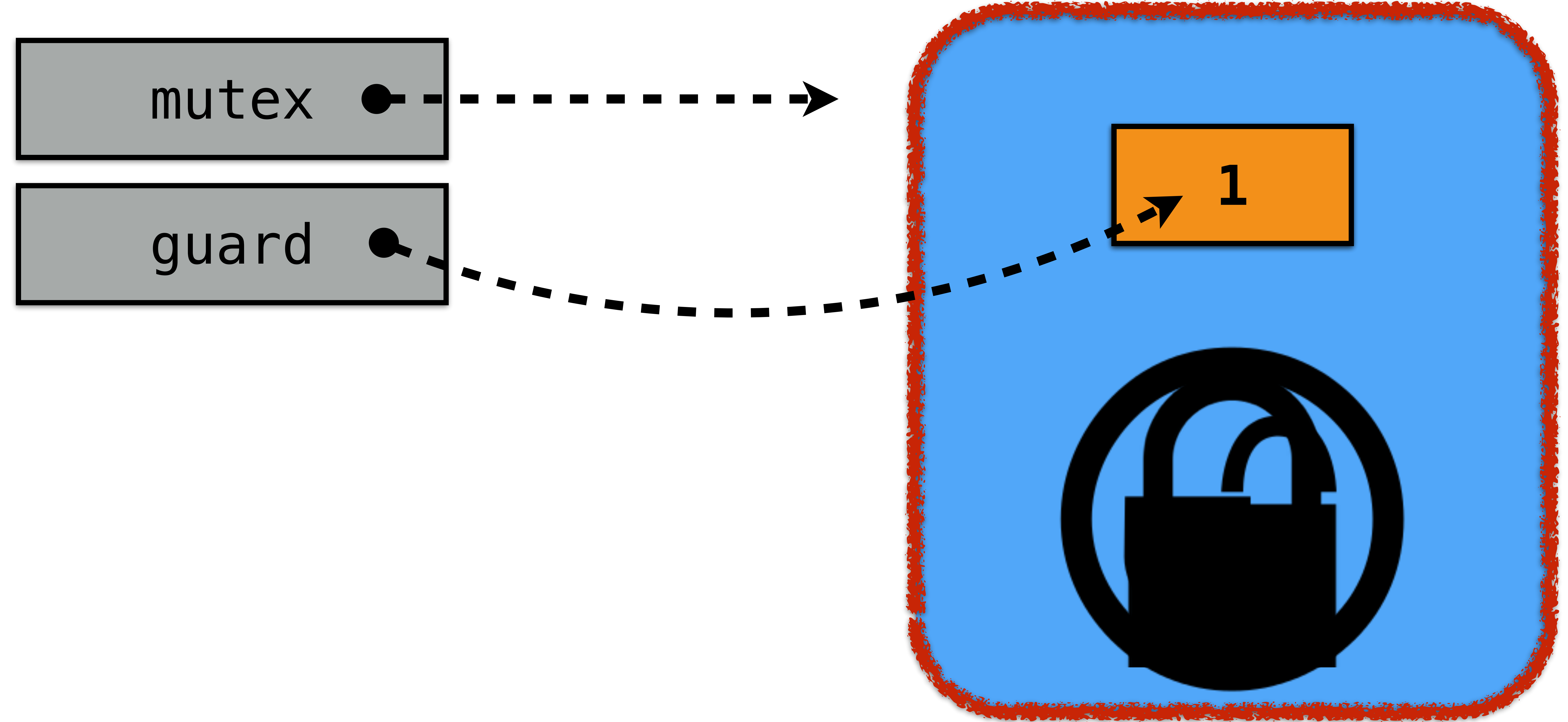
```
impl<T> Channel<T> {
  fn send(&mut self, data: T) {
    …
  }
}
```

Take ownership of the data

~ *Concurrency paradigms* ~

Paradigm	Ownership?	Borrowing?
Message passing	✓	
Locking	✓	✓

```
fn sync_inc(mutex: &Mutex<i32>) {
  let mut guard: Guard<i32> = counter.lock();
  *guard += 1;
}
```



https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

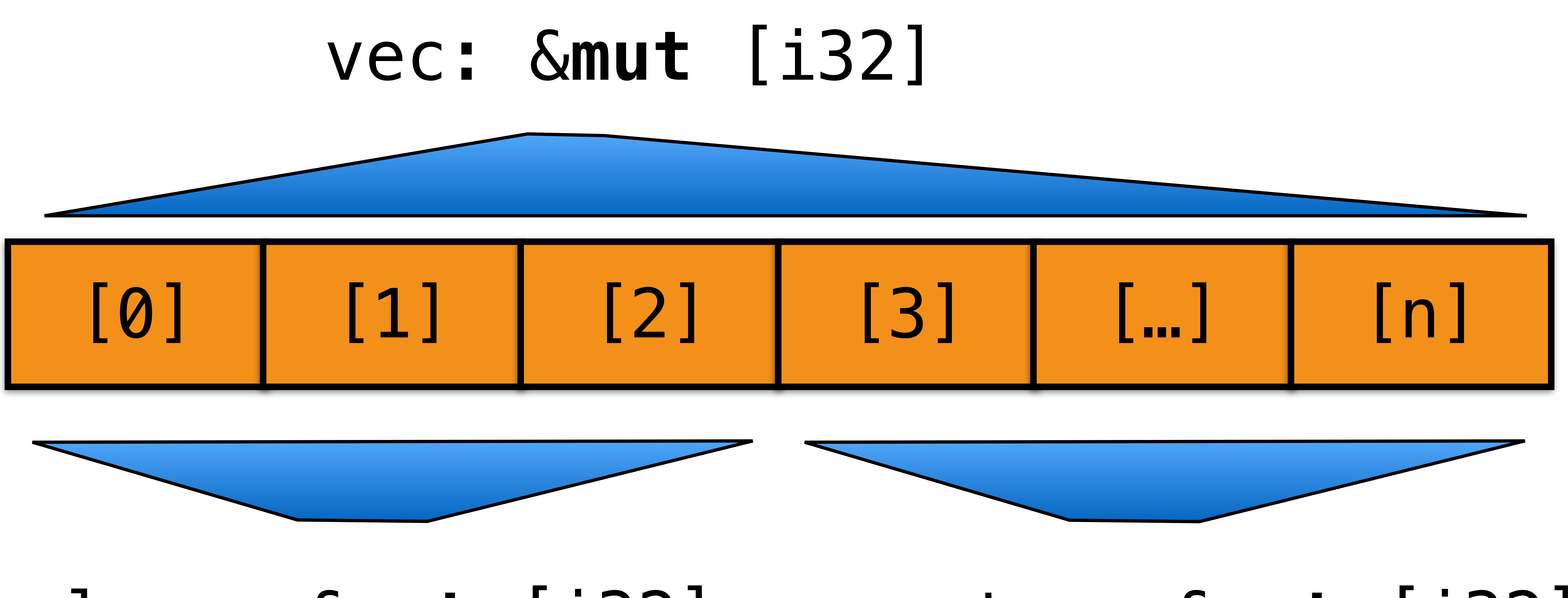
~ *Concurrency paradigms* ~

Paradigm	Ownership?	Borrowing?
Message passing	✓	
Locking	✓	✓
Fork join		✓

```
fn divide_and_conquer(…) {
  rayon::join(
    || do_something(),
    || do_something_else(),
  );
}
```

Conceptually:
- Start two threads
- Wait for them to finish

```
fn qsort(vec: &mut [i32]) {
  if vec.len() <= 1 { return; }
  let pivot = vec[random(vec.len())];
  let mid = vec.partition(vec, pivot);
  let (less, greater) = vec.split_at_mut(mid);
  qsort(less);
  qsort(greater);
}
```



```
fn qsort(vec: &mut [i32]) {
  if vec.len() <= 1 { return; }
  let pivot = vec[random(vec.len())];
  let mid = vec.partition(vec, pivot);
  let (less, greater) = vec.split_at_mut(mid);
  rayon::join(
    || qsort(less),
    || qsort(greater)
  );
}
```

```
rayon::join(
  || qsort(less),
  || qsort(less),
);
```

?

[0] [1] [2] [3] […] [n]

less: &mut [i32]   greater: &mut [i32]

~ *Concurrency paradigms* ~

Paradigm	Ownership?	Borrowing?
Message passing	✓	
Locking	✓	✓
Fork join		✓
Lock-free	✓	✓
Futures	✓	✓
…		

Traits
Photo credit: Andy Leppard
https://www.flickr.com/photos/creativecomputer/261445720/

“Zero cost” abstraction

```
vec1.iter()                      // vec1’s elements
    .zip(vec2.iter())            // paired with vec2’s
    .map(|(i, j)| i * j)         // multiplied
    .sum()                       // and summed
```

```
.LBB0_8:
  movdqu (%rdi,%rbx,4), %xmm1
  movdqu (%rdx,%rbx,4), %xmm2
  pshufd $245, %xmm2, %xmm3
  pmuludq  %xmm1, %xmm2
  pshufd $232, %xmm2, %xmm2
  pshufd $245, %xmm1, %xmm1
  pmuludq  %xmm3, %xmm1
  pshufd $232, %xmm1, %xmm1
  punpckldq %xmm1, %xmm2
  paddd%xmm2, %xmm0
  addq $4, %rbx
  incq %rax
  jne .LBB0_8
```

Parallel execution

```
vec1.par_iter()
    .zip(vec2.par_iter())
    .map(|(i, j)| i * j)
    .sum()
```

Multicore (work stealing)
+ SIMD
+ Guaranteed thread safety

Implemented for a given type (`Self`) ✳

✳ Yes, we support "multiparameter" traits too.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

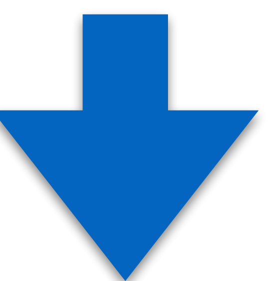
Associated type

Method that takes `&mut` reference

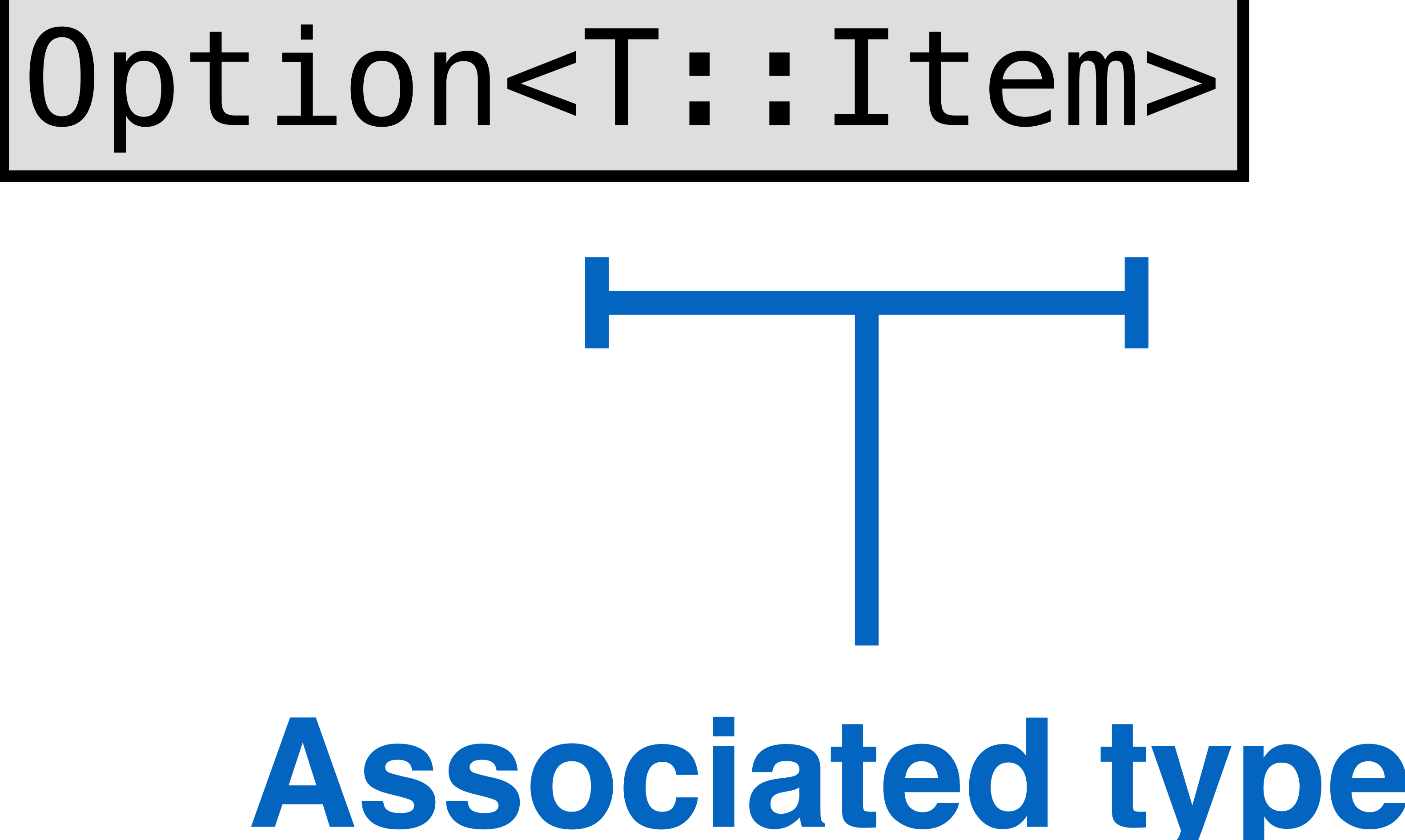
Reference to the associated type

`iter` is of some type `T` that implements `Iterator`

```
iter.next()
```



```
Iterator::next(&mut iter)
```



“Auto-ref”

Use method as a function

```
struct Zip<A: Iterator, B: Iterator> {
  a: A,
  b: B,
}
```

```rust
struct Zip<A: Iterator, B: Iterator> {
  a: A,
  b: B,
}
```

```rust
impl<A: Iterator, B: Iterator> Iterator for Zip<A, B> {
  type Item = (A::Item, B::Item);
  fn next(&mut self) -> Option<(A::Item, B::Item)> {
    match (self.a.next(), self.b.next()) {
      (Some(a), Some(b)) => Some((a, b)),
      _ => None,
    }
  }
}
```

“Zero cost” abstraction

```
vec1.iter()                         // vec1’s elements
    .zip(vec2.iter())               // paired with vec2’s
    .map(|(i, j)| i * j)            // multiplied
    .sum()                          // and summed
```

Default methods

```
trait Iterator {
  // Required items
  type Item;
  fn next(&mut self) -> Option<Self::Item>;

  // Provided items
  fn zip<I>(self, other: I) -> Zip<Self, I>
  where I: Iterator {
    Zip { a: self, b: other }
  }

  …
}
```

```
impl<A: Iterator, B: Iterator> Iterator for Zip<A, B> {
    …
}
```

At compilation time, will generate fully specialized variants for each value of `A`, `B`.

Can also use traits as “types”:

```
Vec<&Iterator<Item=i32>>
```

=> Dynamic dispatch, heterogeneity.

Unsafe

Vision: An Extensible Language

Core language:

Ownership and borrowing

Libraries:

Reference-counting

Files

Parallel execution

...

Use ownership/ borrowing to enforce correct usage.

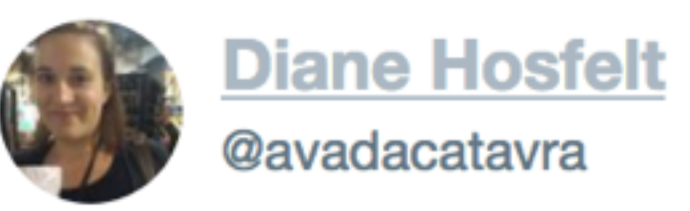
Safe abstractions

```
fn split_at_mut(…) {
    unsafe {
      …
    }
}
```

Validates input, etc.

Trust me.

Ownership/borrowing/traits give tools to enforce **safe abstraction boundaries.**



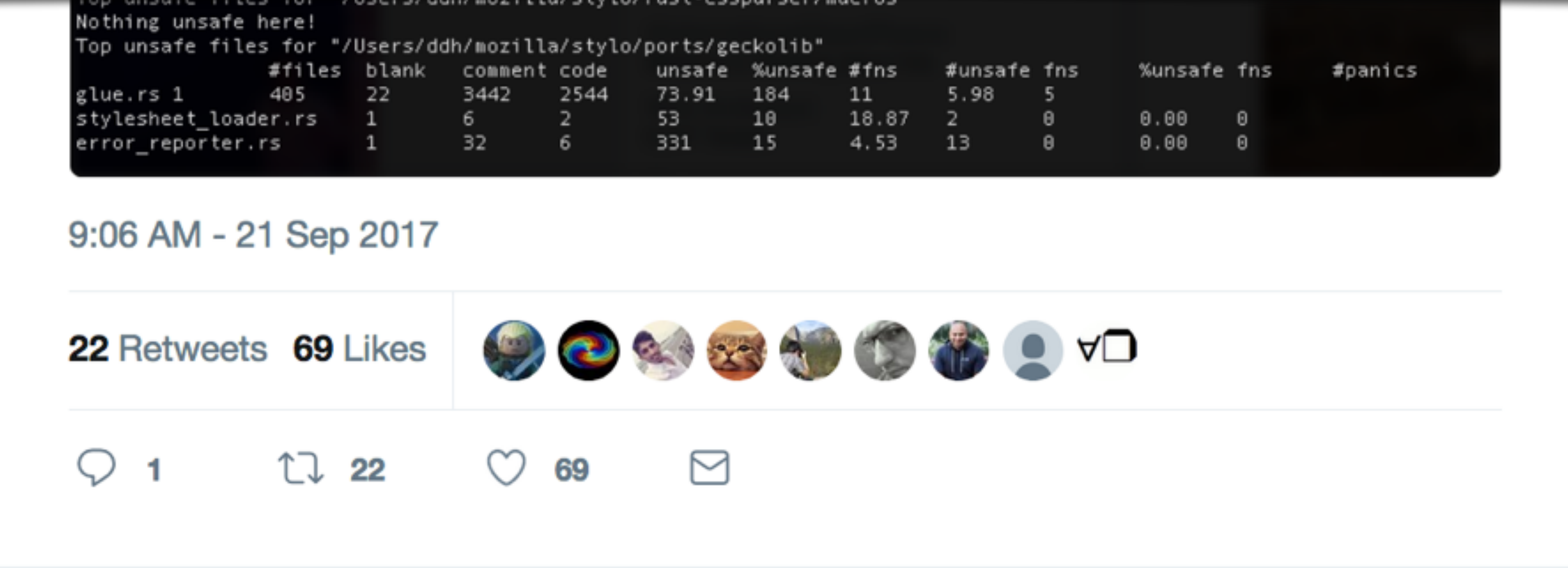
Diane Hosfelt
@avadacatavra

Following

Published github.com/avadacatavra/u … to analyze unsafe code usage in @rustlang

Stylo (Parallel CSS Rendering — coming in FF57)

	Total KLOC	Unsafe KLOC	Unsafe %
Total	146.2	51.7	35%
Interesting stuff	71.6	1.4	1.9%
FFI Bindings	74.5	50.3	67.4%



Mission accomplished
Rust in Firefox 48

STYLE

```
body {
    color: grey;
}
```

```
h1 {
    color: blue;
    font-size: 2em;
    transform: skew(45deg);
    will-change: transform;
}
```

```
p {
    margin-top: 2em;
}
```

COMPUTED STYLES

html

head

body

COMPUTED STYLES

link

div

COMPUTED STYLES

h1

COMPUTED STYLES

p

Code Cartoons

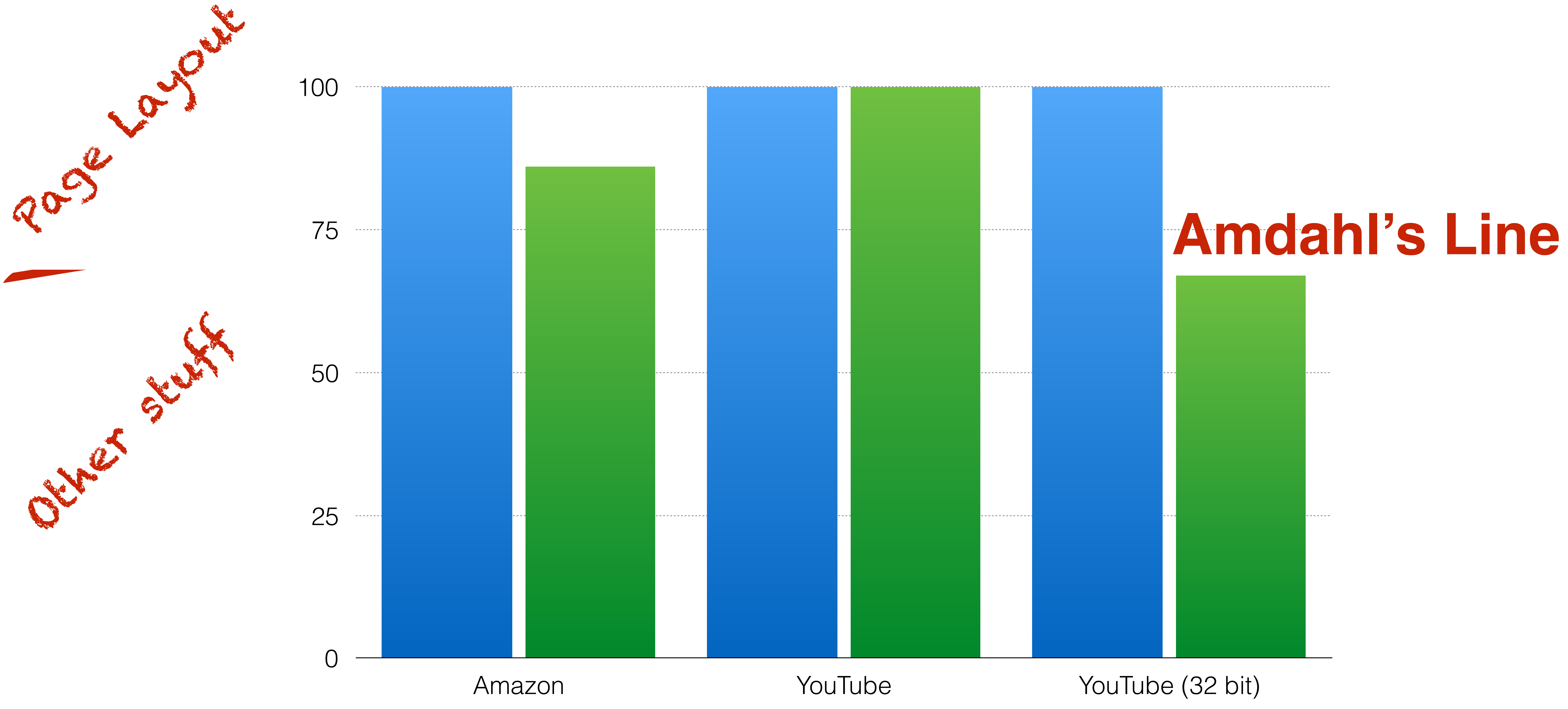
Hi, I'm @linclark...

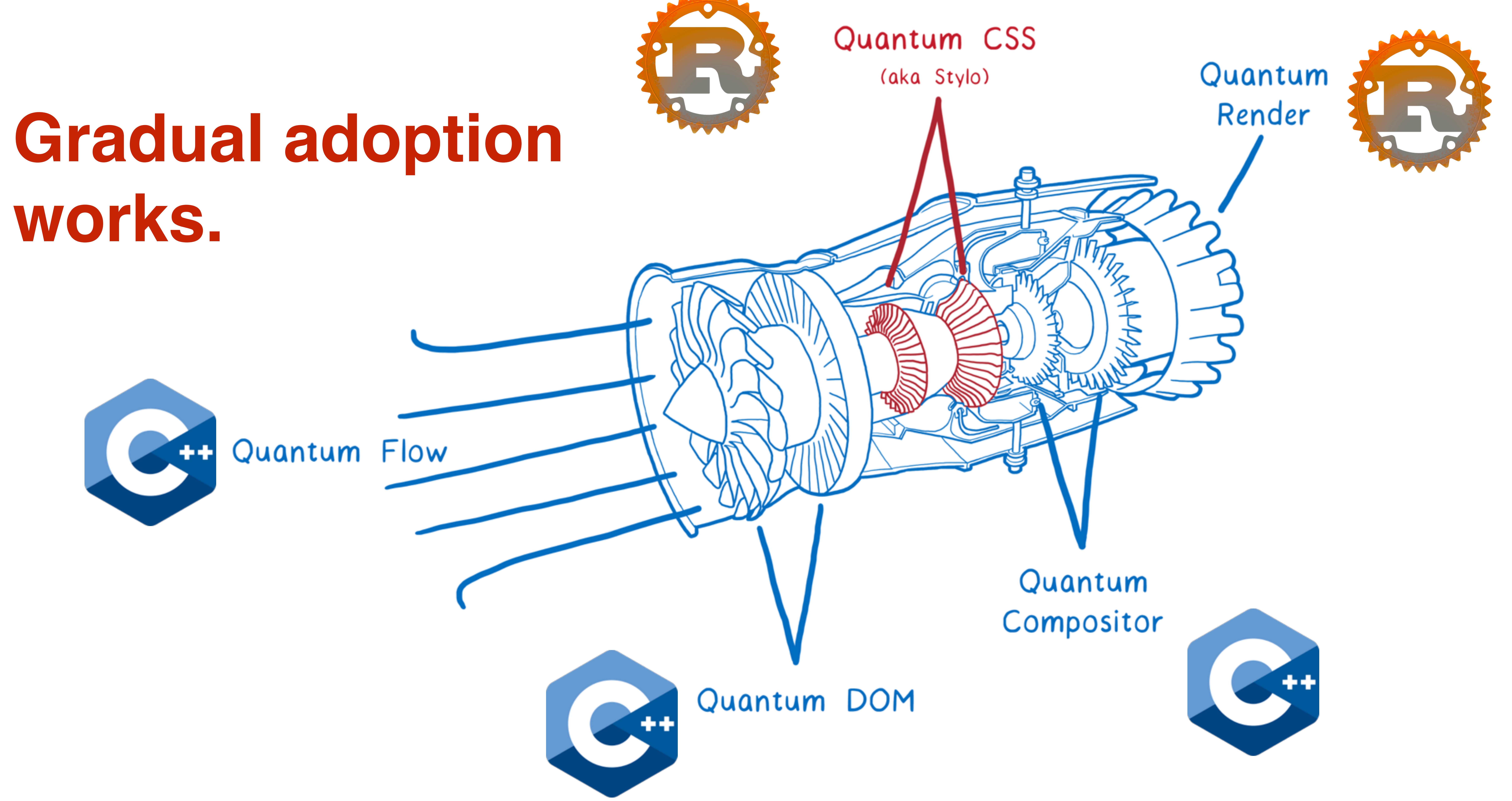
Bug 631527
Parallelize selector matching
Get help with this page
NEW Assigned to dzbarsky
Status (NEW bug with no priority)
Product: Core
Component: CSS Parsing and Computation
Status: NEW
Reported: 7 years ago
Reported: 7 years ago

Safety = Eat your spinach!

Photo credit: Salim Virji
https://www.flickr.com/photos/salim/8594532469/

Initial load times (relative to today)




Gradual adoption works.
Quantum CSS
(aka Stylo)
Quantum Render
Quantum Flow
Quantum Compositor
Quantum DOM

Embedded
Elite
100uF
450V
Elite
100uF
450V
Elite
100uF
450V
High-Voltage
4H.L2E02.A35
BenQ
T2AH/250V

Discovery

> Discover the world of microcontrollers through Rust!

This book is an "introductory course" on microcontroller-based "embedded systems" that uses Rust as the teaching language rather than the usual C/C++.

`https://japaric.github.io/discovery/`

Documentation Community Papers Hardware Blog

Programmable IoT starts at the edge

An embedded operating system designed for running multiple concurrent, mutually distrustful applications on low-memory and low-power microcontrollers.

Get started | Join the community

Signpost

build passing

The Signpost project is a modular city-scale sensing platform that is designed to be installed on existing signposts. It is powered through solar energy harvesting, and provides six slots for generic sensing tasks. Modules have access to a set of shared platform resources including power, communication, gps-based location and time, storage, and higher-performance computation, and they use a Signpost-specific software API that enables not only use of these resources, but also supports the development of inter-module applications.

The project is driven by several core applications, but also strives to be an upgradeable and adaptable platform that supports new applications for

Community
Photo credit: David McSpadden
https://www.flickr.com/photos/familyclan/15535822737/

Rust Leadership Structure

Team	Members	Peers
Core	9	
Language	6	5
Libraries	7	1
Compiler	9	
Dev Tools	6	11
Cargo	6	
Infrastructure	10	
Community	13	
Documentation	4	
Moderation	5	

58 people

10 Mozilla (17%)

Rust 1.0: Stability as a deliverable

Since the early days of Rust, there have only been two things you could count on: safety, and change.

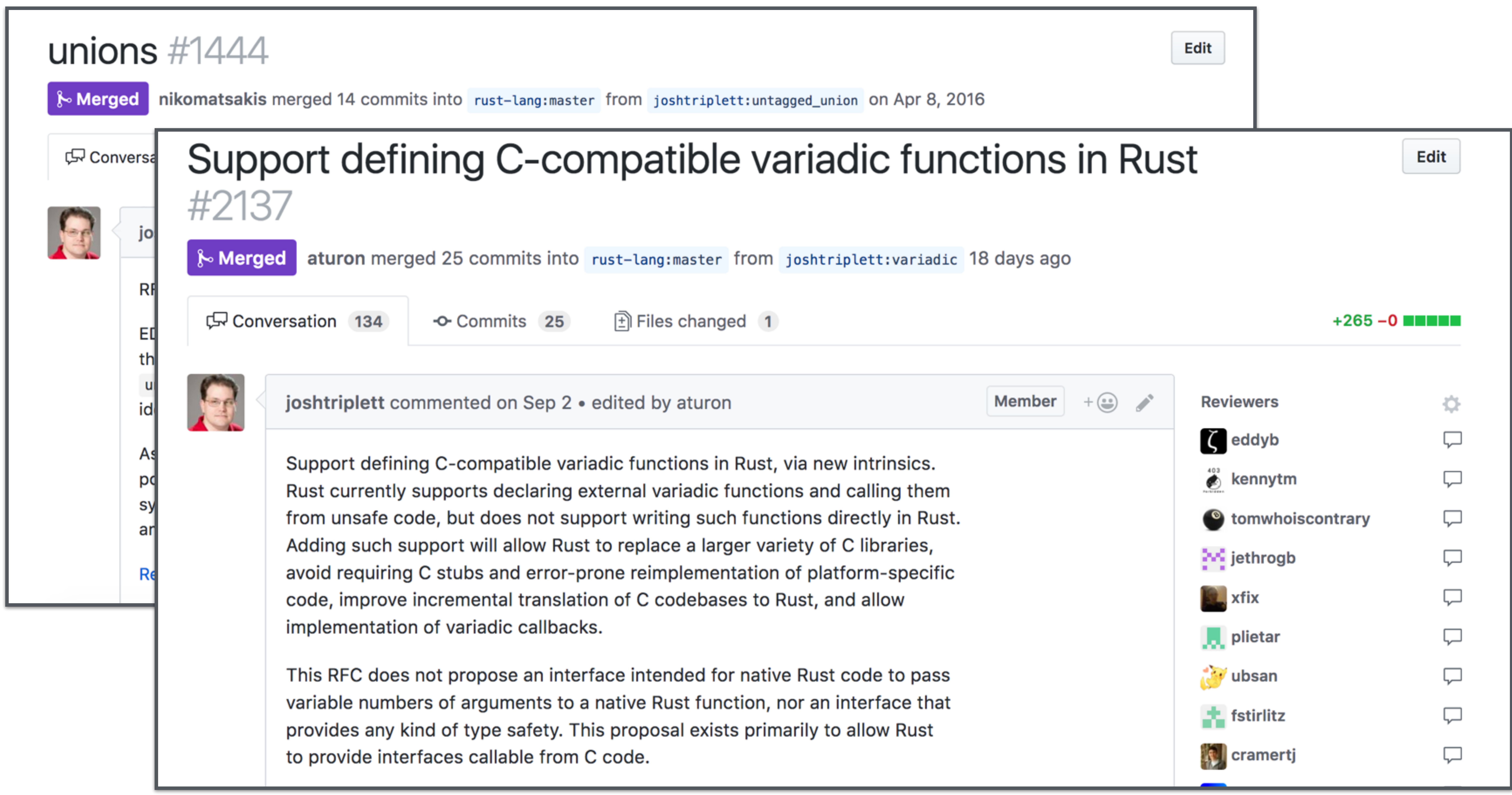
And sometimes not the first one.

Our responsibility [after 1.0] is to ensure that you never dread upgrading Rust.

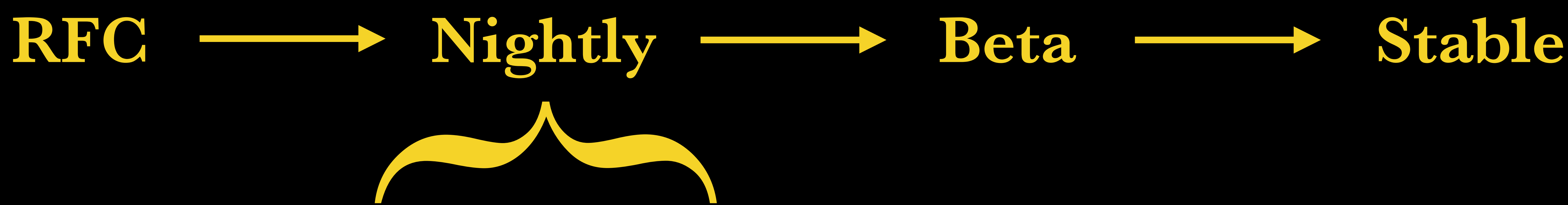
~ The feature pipeline ~

RFC ⟶ **Nightly** ⟶ **Beta** ⟶ **Stable**

RFC Process



~ The feature pipeline ~



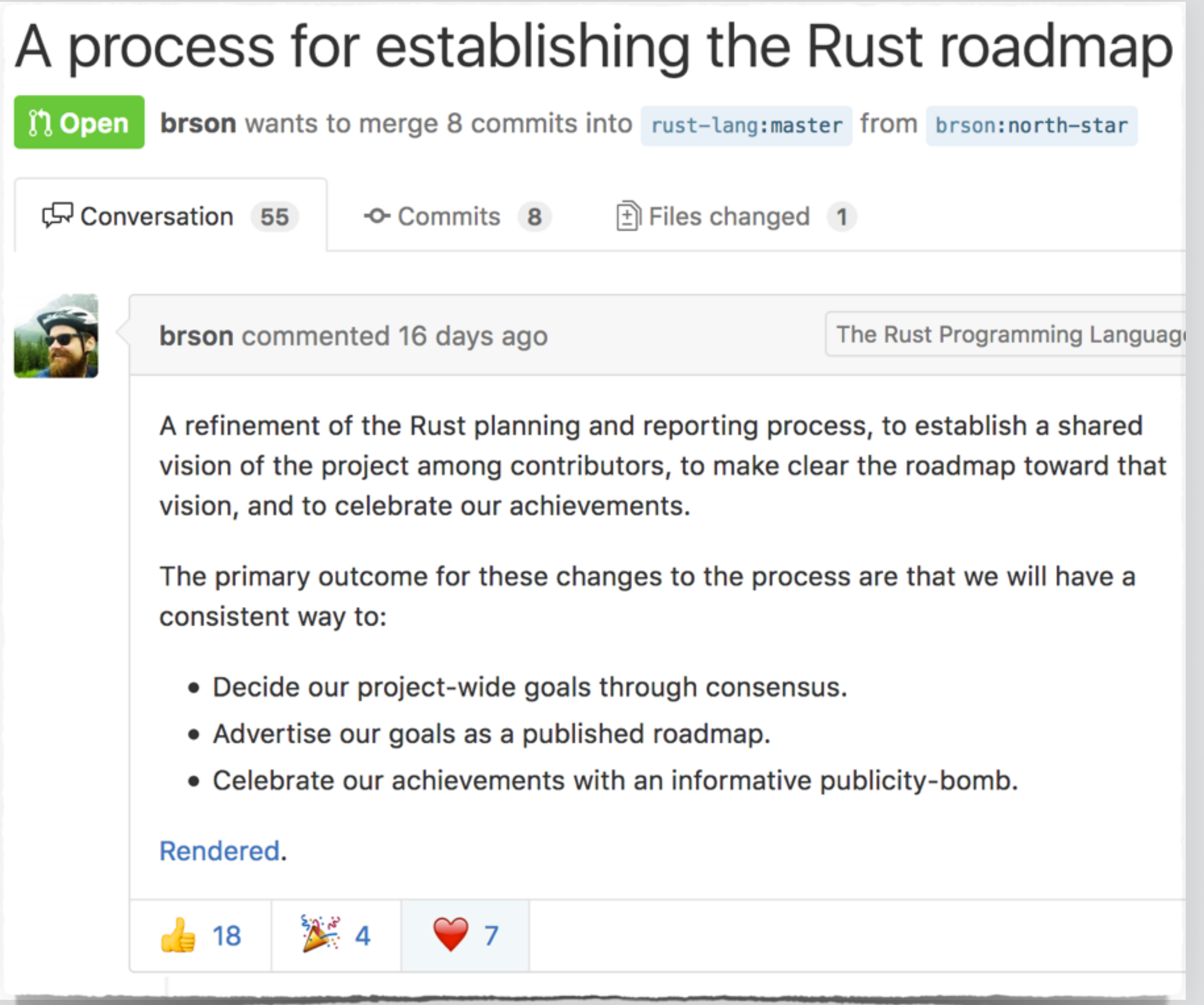
Unstable features
are available

~ The feature pipeline ~

RFC → **Nightly** → **Beta** → **Stable**

6 week release cycle;
only stable features

The Rust Roadmap



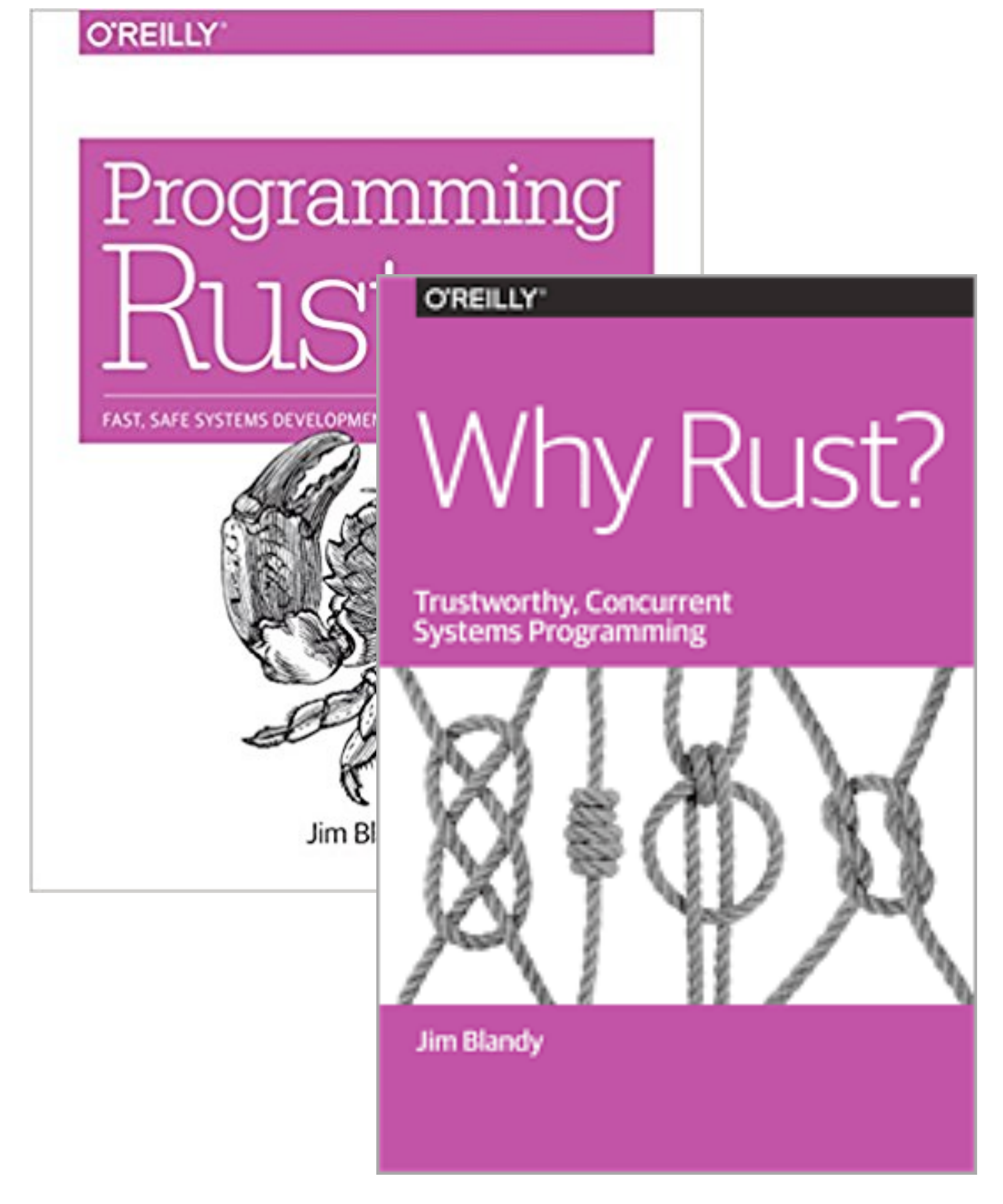
The Rust Programming Language Blog

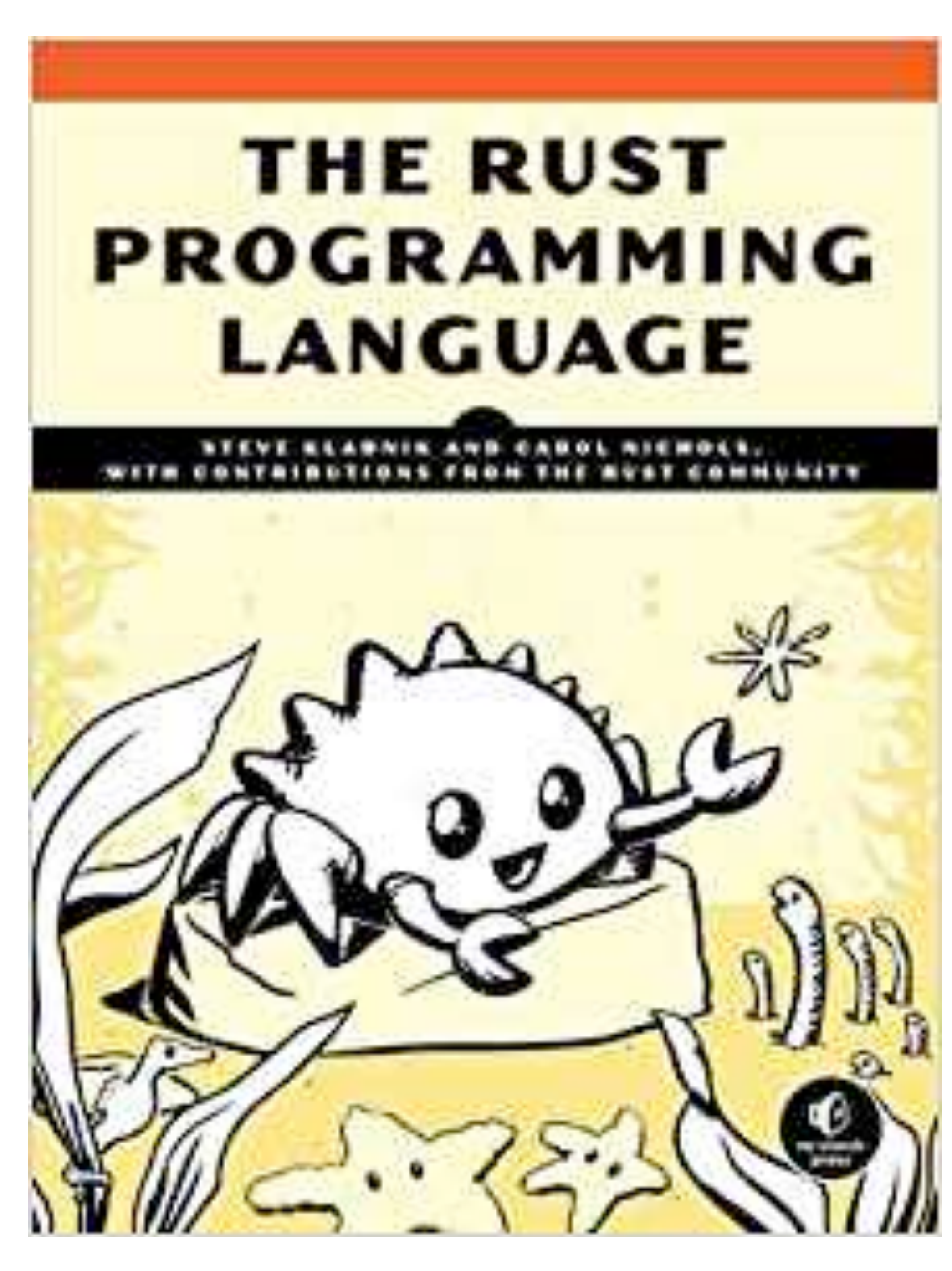
New Year's Rust: A Call for Community Blogposts

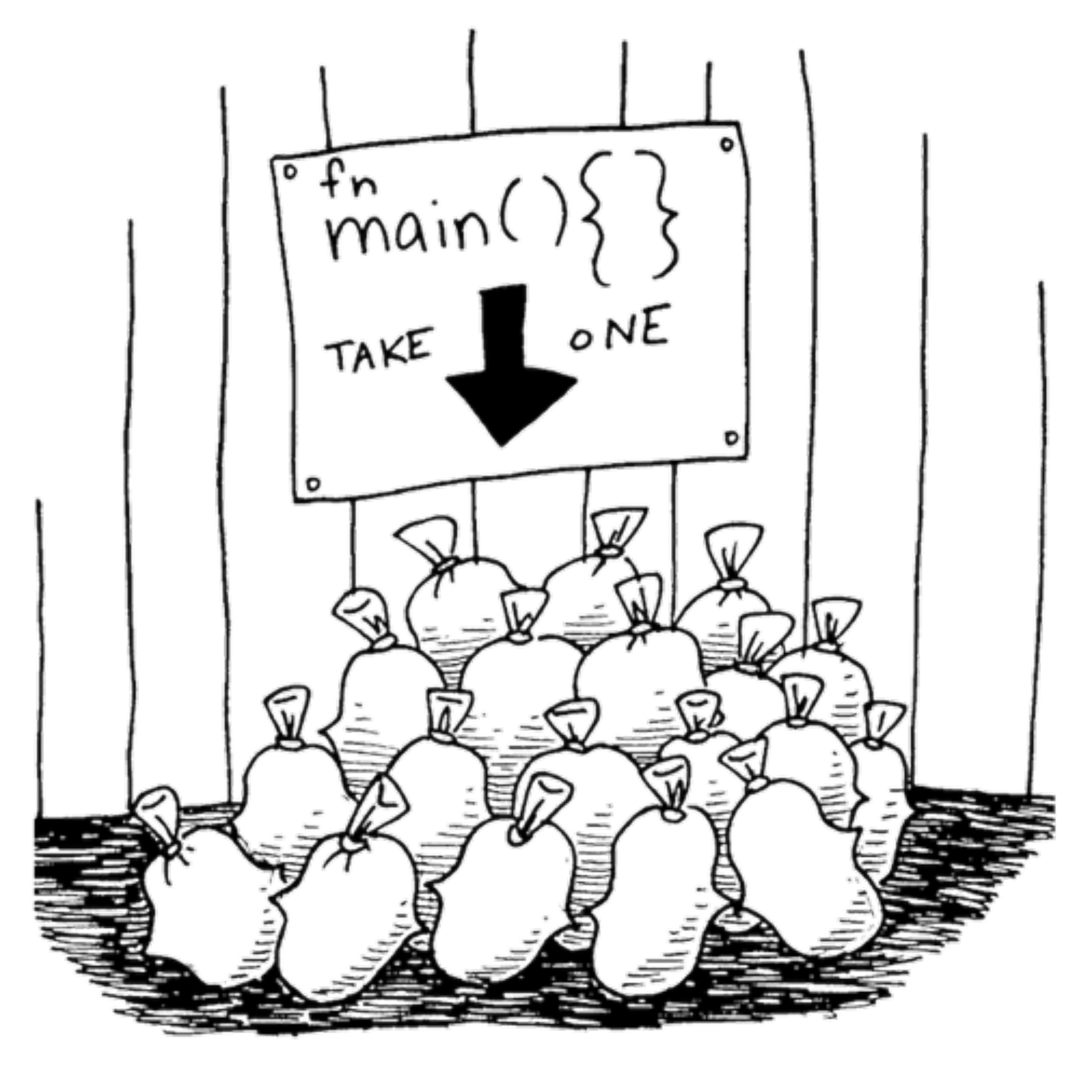
You can write up these posts and email them to community@rust-lang.org or tweet them with the hashtag `#Rust2018`. We'll aggregate any blog posts sent via email or with the hashtag in one big blog post here.

Last year, the Rust team started a new tradition: defining a roadmap of goals for the upcoming year. We leveraged our RFC process to solicit community feedback. While we got a lot of awesome feedback on that RFC, we'd like to try something new **in addition to** the RFC process: a call for community blog posts for ideas of what the goals should be.

Want to learn more?







O'Reilly

Order now!

rust-lang.org

Book, 2nd ed.

intorust.com

Screencasts

Q & A

Rust - Hot-or-Not?

Thank you!

Get in touch

• January 16	Proefzitten | Seats to meet
• February 27/28 & March 1	Embedded World, booth: 3-637
• March 20	Proefzitten | Seats to meet

www.sioux.eu

DRINKS

Source of your technology