Systems programming for all

# What Rust has to offer

**Strong safety guarantees…**

No seg-faults, no data-races, expressive type system.

**…without compromising on performance.**

No garbage collector, no runtime.

**Goal:**

**Confident, productive systems programming**

# 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/

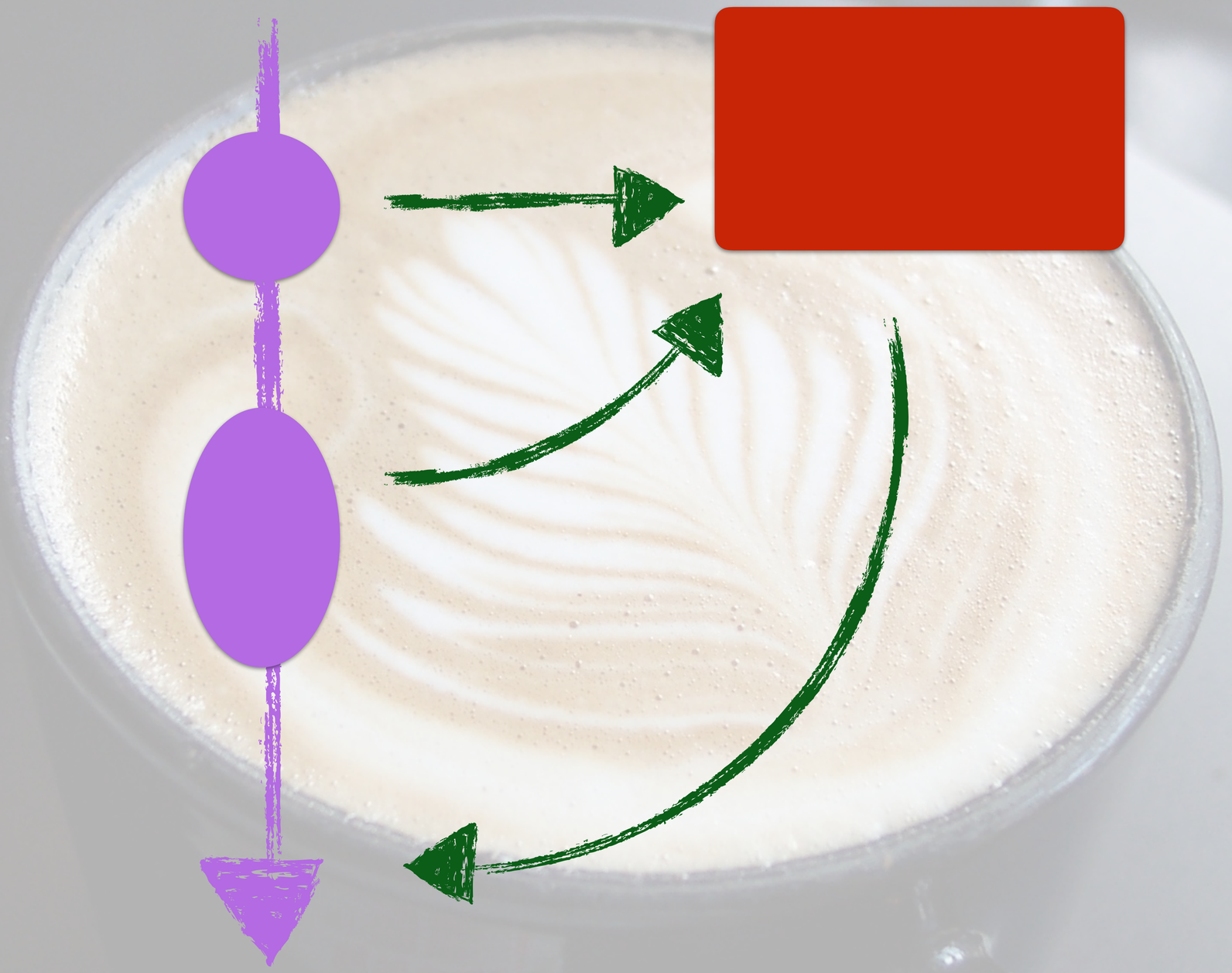


Photo credit: Juan de Dios Santander Vela
https://www.flickr.com/photos/juandesant/209110989/

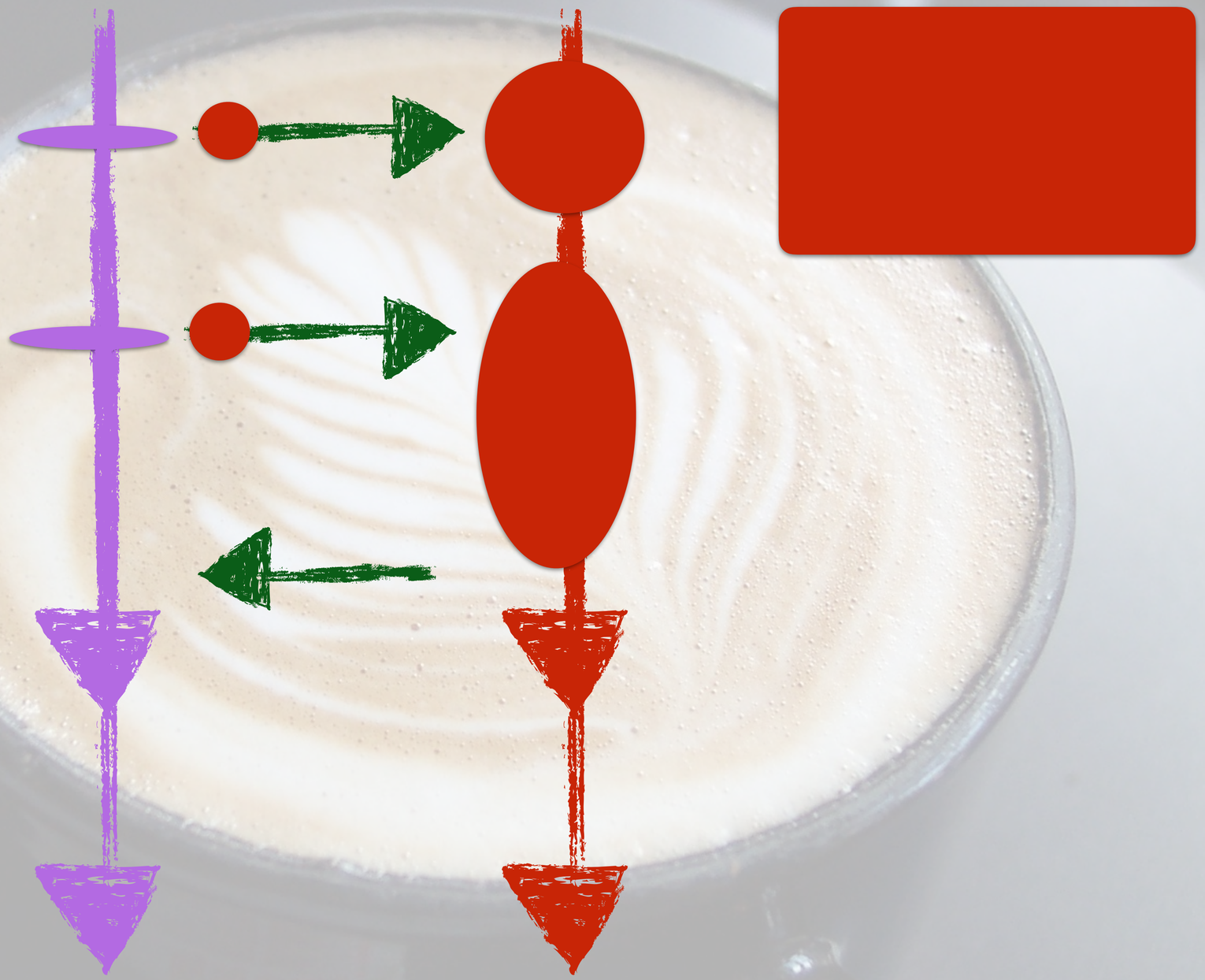
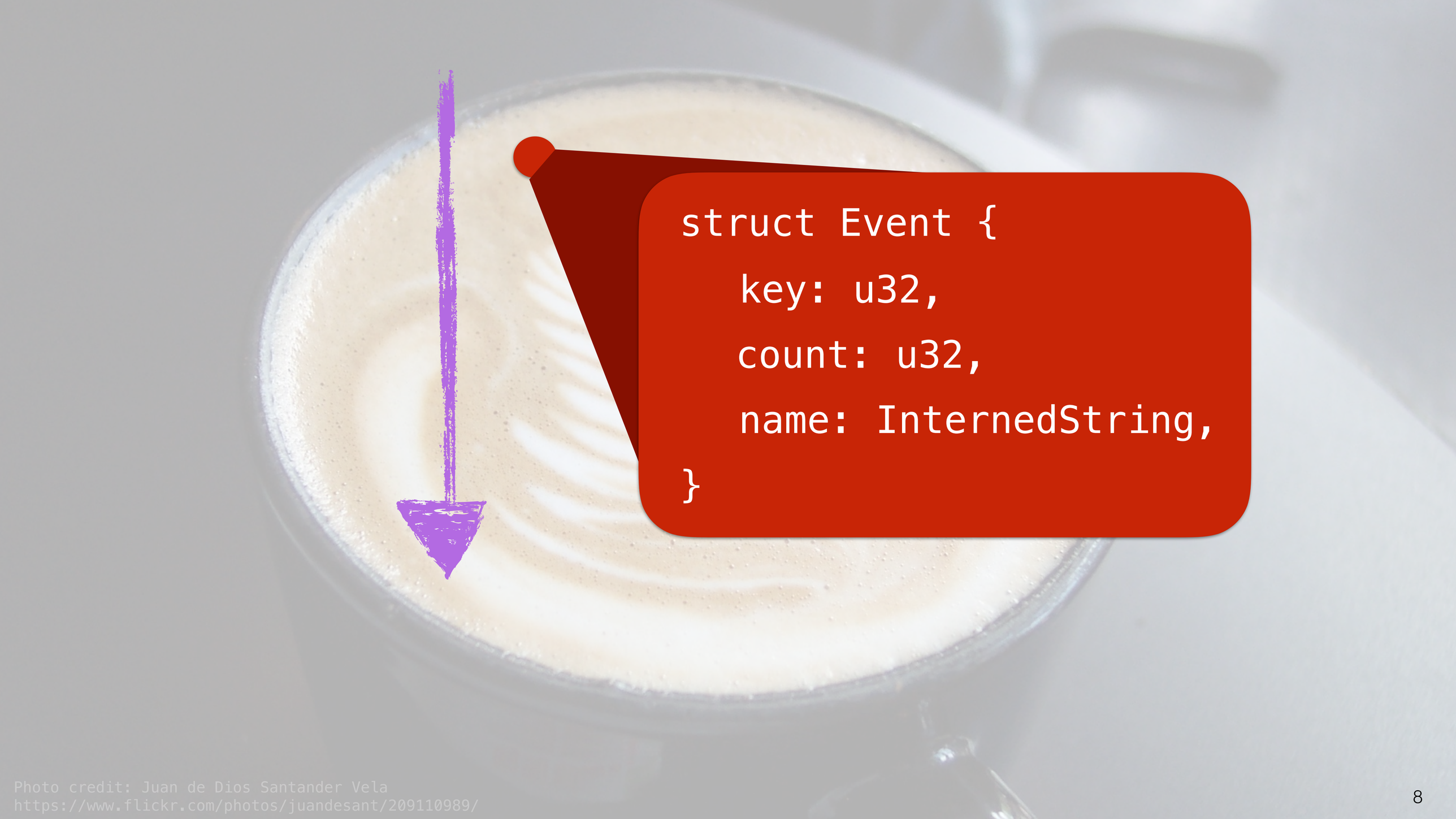


Photo credit: Juan de Dios Santander Vela
https://www.flickr.com/photos/juandesant/209110989/

# six months later…



```
struct Event {
    key: u32,
    count: u32,
    name: InternedString,
}
```

Photo credit: Juan de Dios Santander Vela
https://www.flickr.com/photos/juandesant/209110989/

```
struct Event {
    key: u32,
    count: u32,
    name: InternedString,
}
```

```
struct InternedString {
    value: Rc<String>,
```

"reference-counted string"

**non-atomically**
reference-counted string

Photo credit: Gratisography
https://www.pexels.com/photo/nature-walking-animal-strong-4075/

```
struct Event {
    key: u32,
    count: u32,
    name: InternedString,
}
```

**error:** the type `InternedString` does not implement `Send`

Photo credit: Juan de Dios Santander Vela
https://www.flickr.com/photos/juandesant/209110989/

# “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**

crates.io
Rust Package Registry

Click or press 'S' to search...

Browse All Crates | Docs ▾ | Log in with GitHub

Fork me on GitHub

# The Rust community's crate registry

Install Cargo

Getting Started

Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work.

234,806,125 Downloads

11,846 Crates in stock

## New Crates

- etcommon-block-core (0.1.0)
- dsdl_parser (0.0.1)
- tokenlock (0.1.1)

## Most Downloaded

- libc (0.2.32)
- bitflags (1.0.0)
- winapi (0.2.8)

## Just Updated

- sputnikvm (0.7.8)
- rusty-blockparser (0.6.1)
- rustracing_jaeger (0.1.2)

## Rust the language

Memory safety
Traits
Parallelism
Unsafe

## Rust the community

Rust in Firefox
Community processes

# Memory safety

Photo credit: Amelia Wells
https://www.flickr.com/photos/speculummundi/6424242877/

Zero-cost abstractions

+

Memory safety & data-race freedom

=

**Confident, productive systems programming**

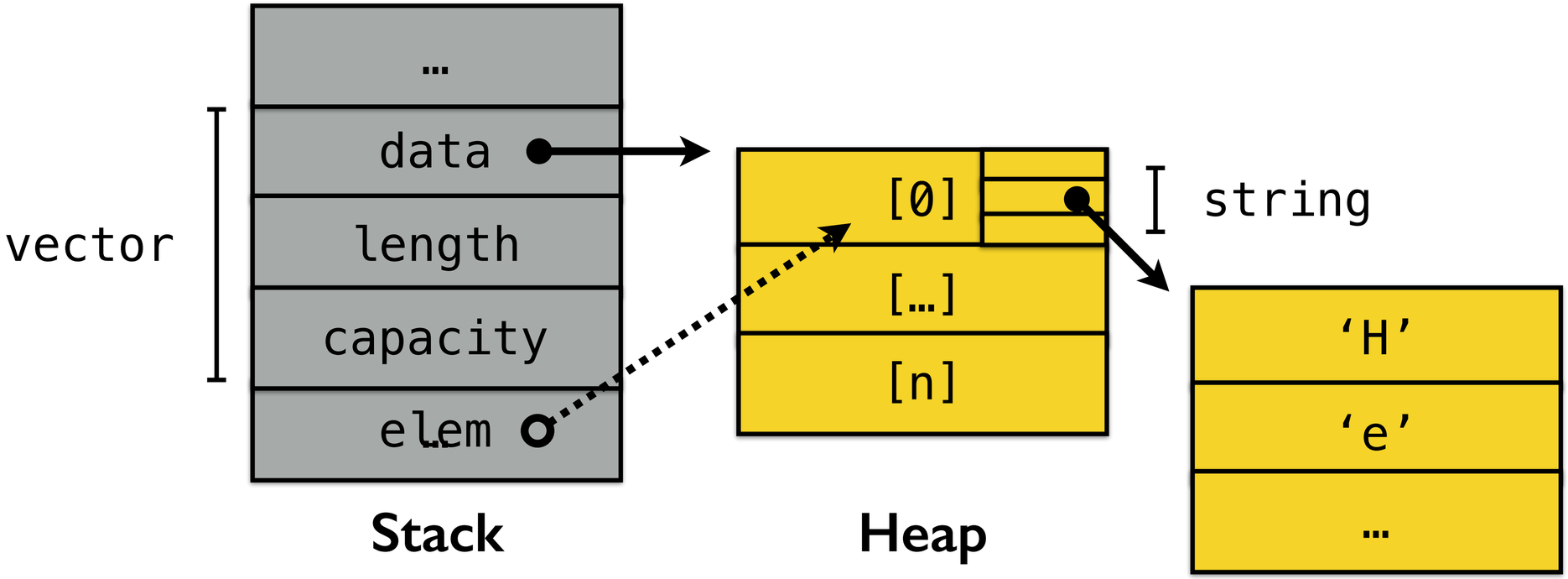
# Zero-cost abstractions

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

Stack and inline layout.

Lightweight references

Deterministic destruction

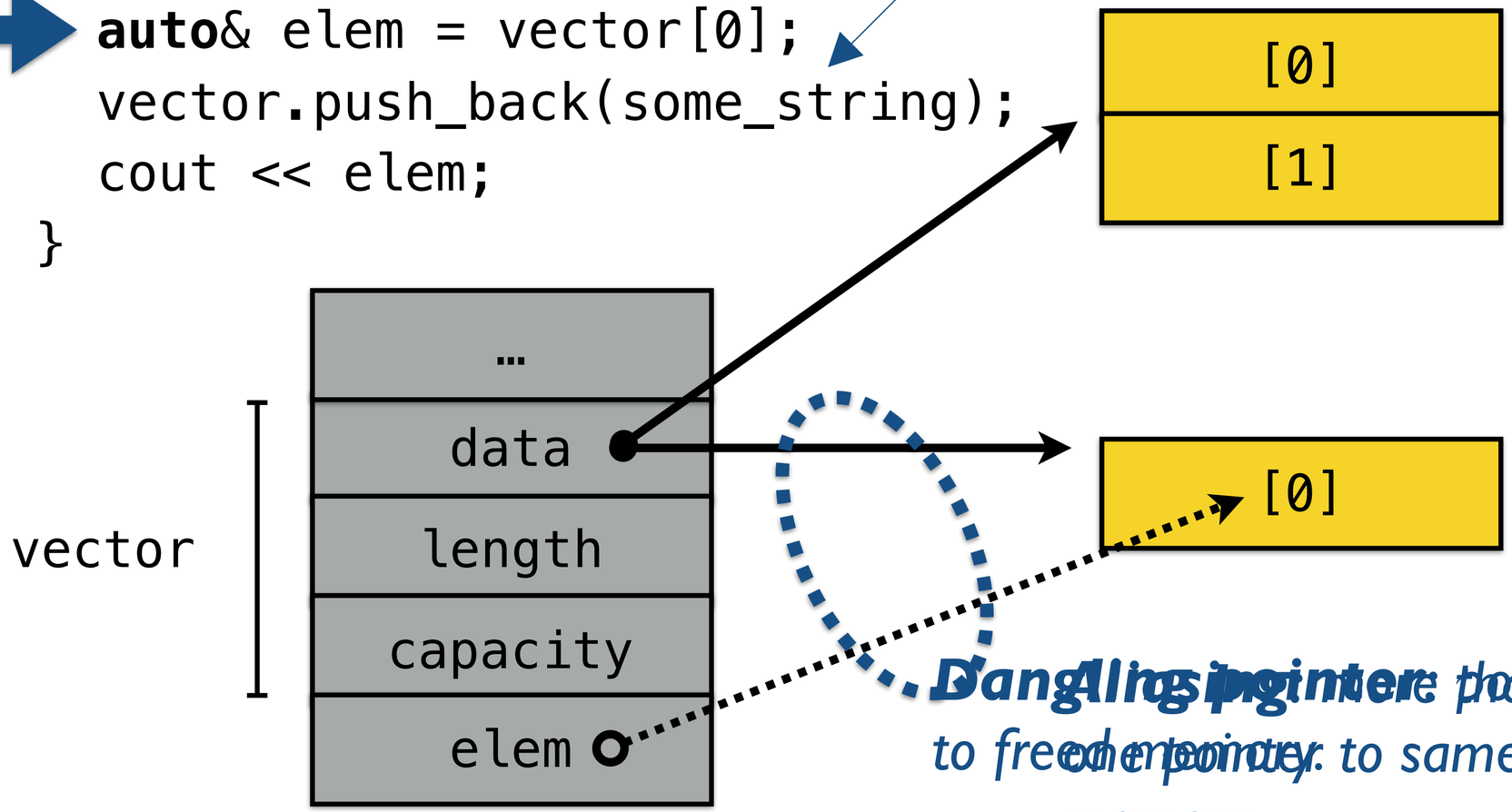



C++

# Memory safety

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

**Mutating** the vector freed old contents.




***Dangling pointer:*** *elem pointer to freed memory.*

***Aliasing pointer:*** *elem pointer and vector point to same memory.*

C++

# ~ *Ownership and Borrowing* ~

| Type | Ownership | Alias? | Mutate? |
|---|---|---|---|
| T | Owned | | ✓ |

# Ownership

*n.* The act, state, or right of possessing something.

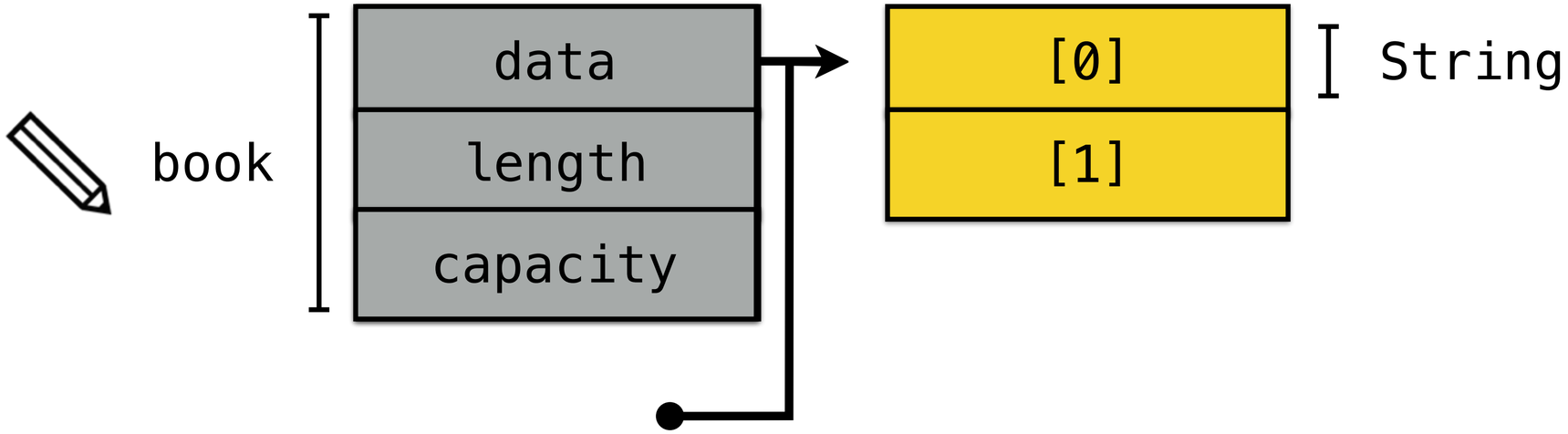
```
fn main() {
  let mut book = Vec::new();
  book.push(…);
  book.push(…);
  publish(book);
  publish(book);
}
```

Give ownership.

**Error:** use of moved value: `book`

```
fn publish(book: Vec<String>) {
  …
}
```

Take ownership of the vector




# Ownership

# “Manual” memory management in Rust:

Values owned by **creator**.

Values **moved** via assignment.

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

**Feels invisible.**

# Borrow

*v.* To receive something with the promise of returning it.

# ~ *Ownership and Borrowing* ~

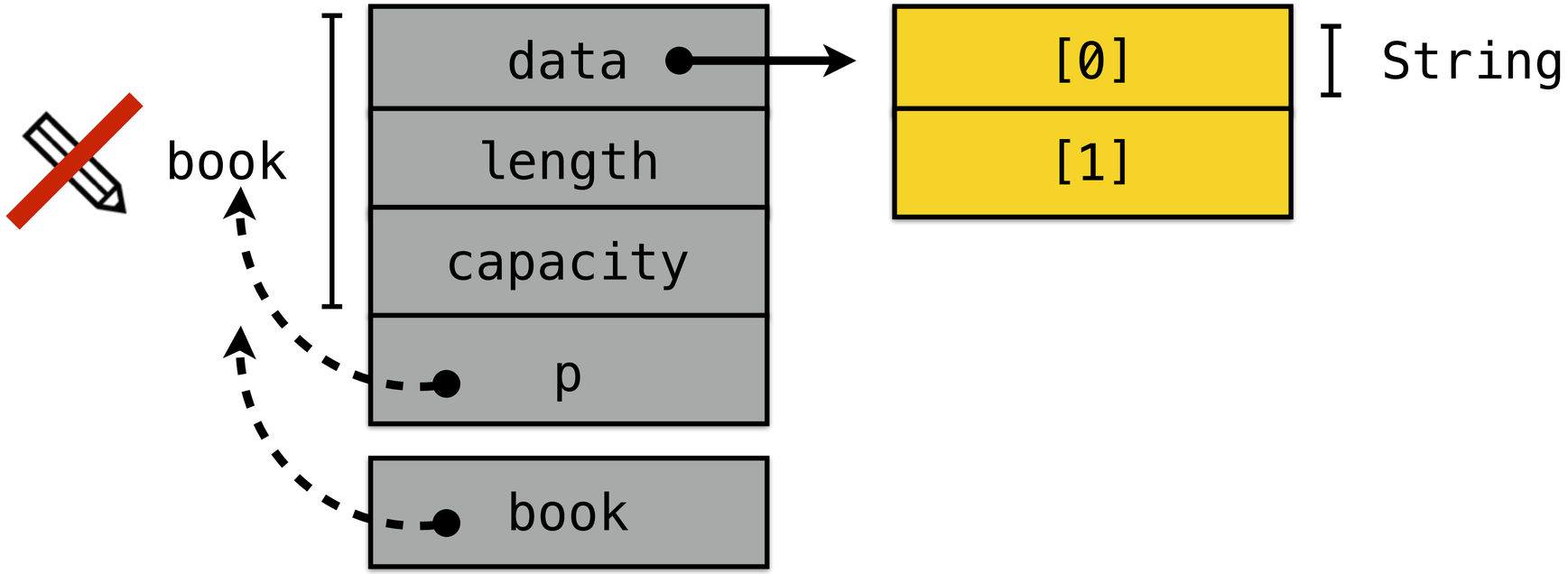
| Type | Ownership | Alias? | Mutate? |
|---|---|---|---|
| T | Owned | | ✓ |
| &T | Shared reference | ✓ | |

```
fn main() {
  let mut book = Vec::new();
  book.push(…);
  book.push(…);
  let p = &book;
  publish(p);
  publish(p);
}
```

**Borrow** the vector, creating a reference

```
fn publish(book: &Vec<String>) {
  …
}
```

Change type to a **reference** to a vector




## Shared borrow

# Shared data: immutable

```
let mut book = Vec::new();
book.push(…);
{
    let r = &book;
    book.push(…);
    r.push(…);
}
book.push(…);
```

- book **mutable** here
- book **borrowed** here
- cannot mutate while shared *
- r goes out of scope; borrow ends
- now book is **mutable** again

**** Actually: mutation only under controlled circumstances***

# ~ *Ownership and Borrowing* ~

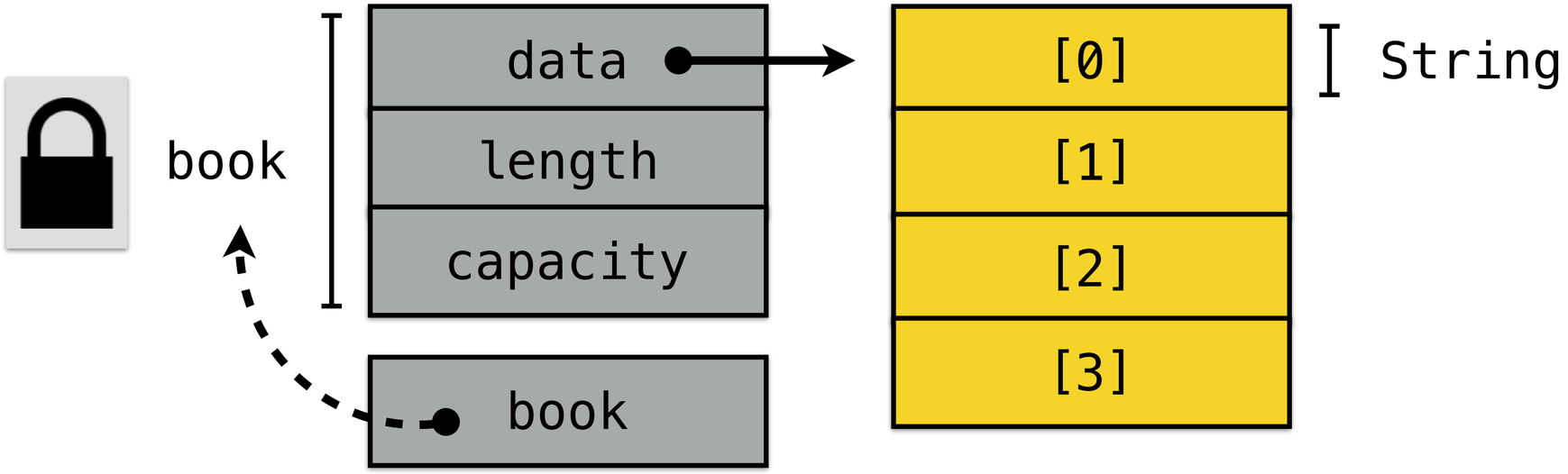
| Type | Ownership | Alias? | Mutate? |
|---|---|---|---|
| T | Owned | | ✓ |
| &T | Shared reference | ✓ | |
| &mut T | Mutable reference | | ✓ |

```rust
fn main() {
  let mut book = Vec::new();
  book.push(…);
  book.push(…);
  edit(&mut book);
  edit(&mut book);
}
```

**Mutable borrow**

```rust
fn edit(book: &mut Vec<String>) {
  book.push(…);
}
```

**Mutable reference**
to a vector




# Mutable borrow

# Mutable references: no other access

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

book.push(…);                  ← book mutable here

{
    let r = &mut book;         ← book borrowed here
    book.len();                ← cannot access while borrowed
    r.push(…);                 ← but &mut ref can mutate
}                              ← r goes out of scope; borrow ends

book.push(…);                  ← borrow ended; accessible again
```

# Lifetime of a reference

```
let mut book = Vec::new();
…
{
    let r: &String = &book[0];
    …
}
```

'l

```
let r: &String = &book;
```

```
let r: &'l String = &book;
```

```
fn first<'a>(v: &'a Vec<String>) -> &'a String {
    return &v[0];
}
```

"Returns a reference derived from `v`"

```
fn first(v: &Vec<String>) -> &String {
    return &v[0];
}
```

(more common shorthand)

```
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.**

```
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

# Traits

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

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone> Clone for Vec<T> {
  fn clone(&self) -> Vec<T> {
    let mut v = Vec::new();
    for elem in self {
        v.push(elem.clone());
    }
    return v;
  }
}
```

Implemented for a given type

Method that borrows its receiver

Implementation for vectors

Create new vector

Iterate (using references)…

…push clone of each element.

Return `v`

**Type-check model: check the generic definition once**

**Runtime model: monomorphized, like C++**

# Marker traits

| | ✅ | 🚫 |
|---|---|---|
| `// “Safe to send between threads”`<br>`trait Send { }` | `String`<br>`u32`<br>`Arc<String>` | `Rc<String>` |
| `// “Safe to memcpy”`<br>`trait Copy { }` | `u32`<br>`f32` | `String`<br>`Rc<String>` |

# Parallelism

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

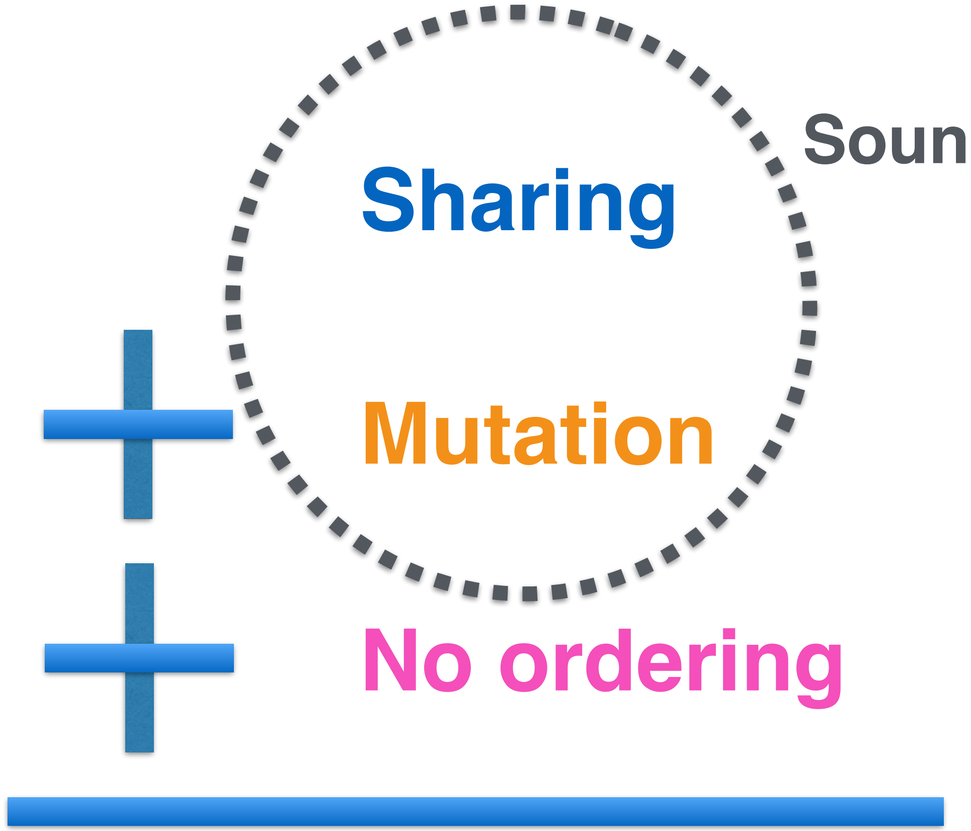
# Library-based concurrency

**Originally**: Rust had message passing built into the language.

**Now**: library-based, multi-paradigm.

Libraries leverage **ownership and traits** to avoid data races.

# Data races




Sound familiar?

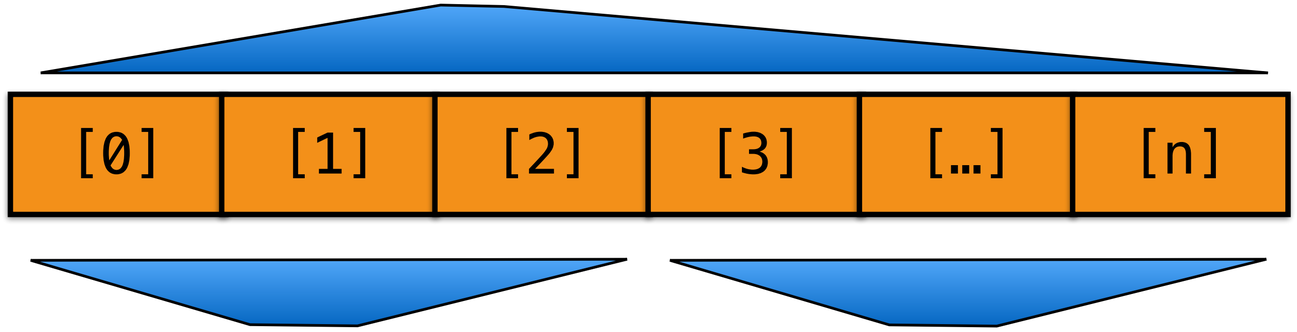
**Data race**

```
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);
  l
}
```

`&mut` can safely go

```
fn split_at_mut<'a>(v: &'a mut [T], …) -> (&'a mut [T], &'a mut [T])
```

vec: &mut [i32]




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

# Parallel iterators

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

trait ParallelIterator {
    type Item;
    …
}
```

```
join(
  || /* seq iter */,
  || /* seq iter */)
```

# ~ *Concurrency paradigms* ~

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

# Unsafe

# Vision: An Extensible Language

**Core language:**

Ownership and borrowing

**Libraries:**

Reference-counting

Files

Parallel execution

…

**Use ownership/ borrowing to enforce correct usage.**

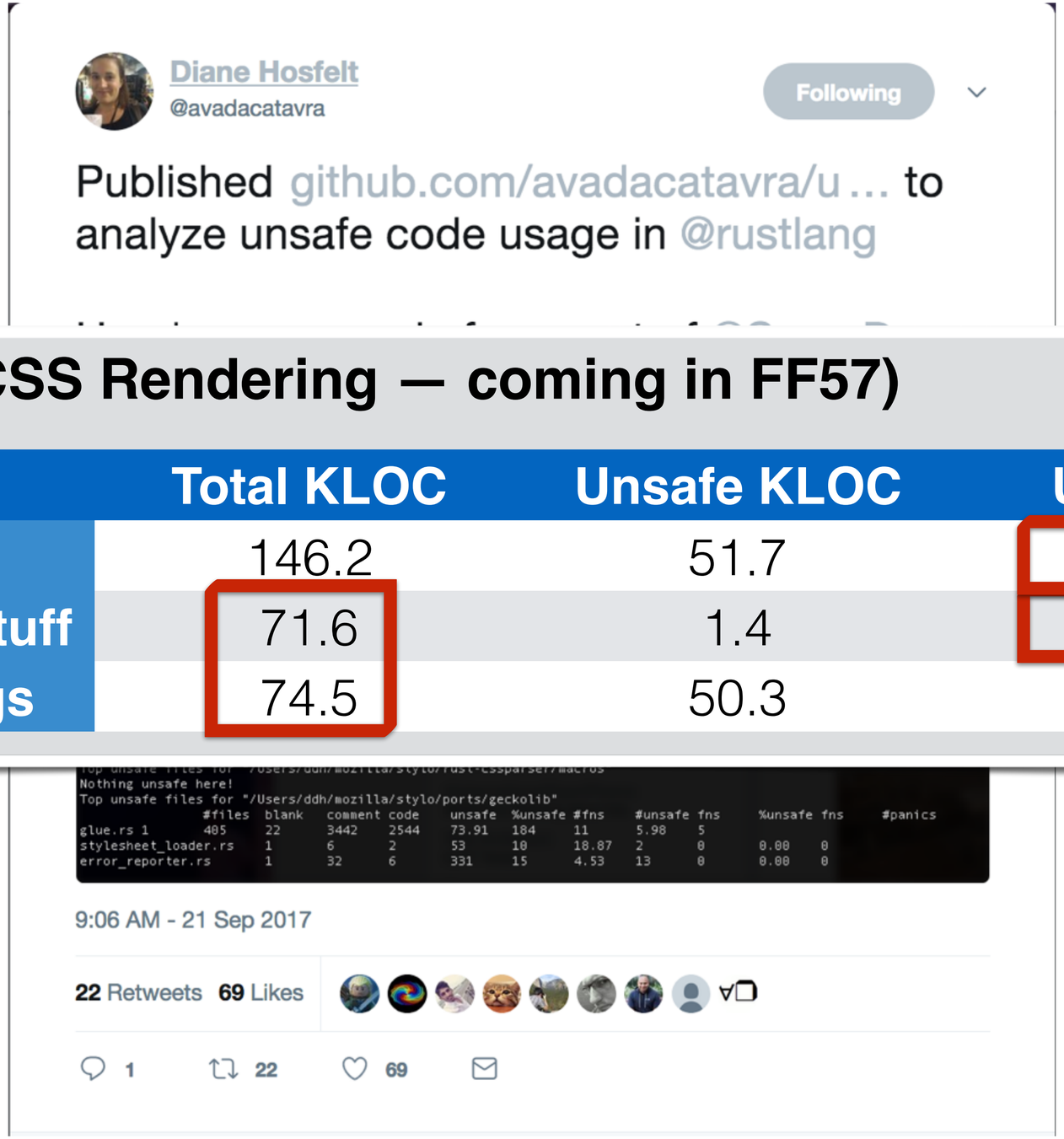
# Safe abstractions

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

Validates input, etc.

Trust me.

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




## 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% |

Basically all safe

**Parallel Iterators**

Safe interface

**join()**

Unsafe impl

Unsafe

**threadpool**

DOCS.RS | Releases | Rust

# crossbeam 0.2.10

Support for lock-free data structures, synchronizers, and parallel programming

Documentation | Crate | Source

## “Partially safe:”

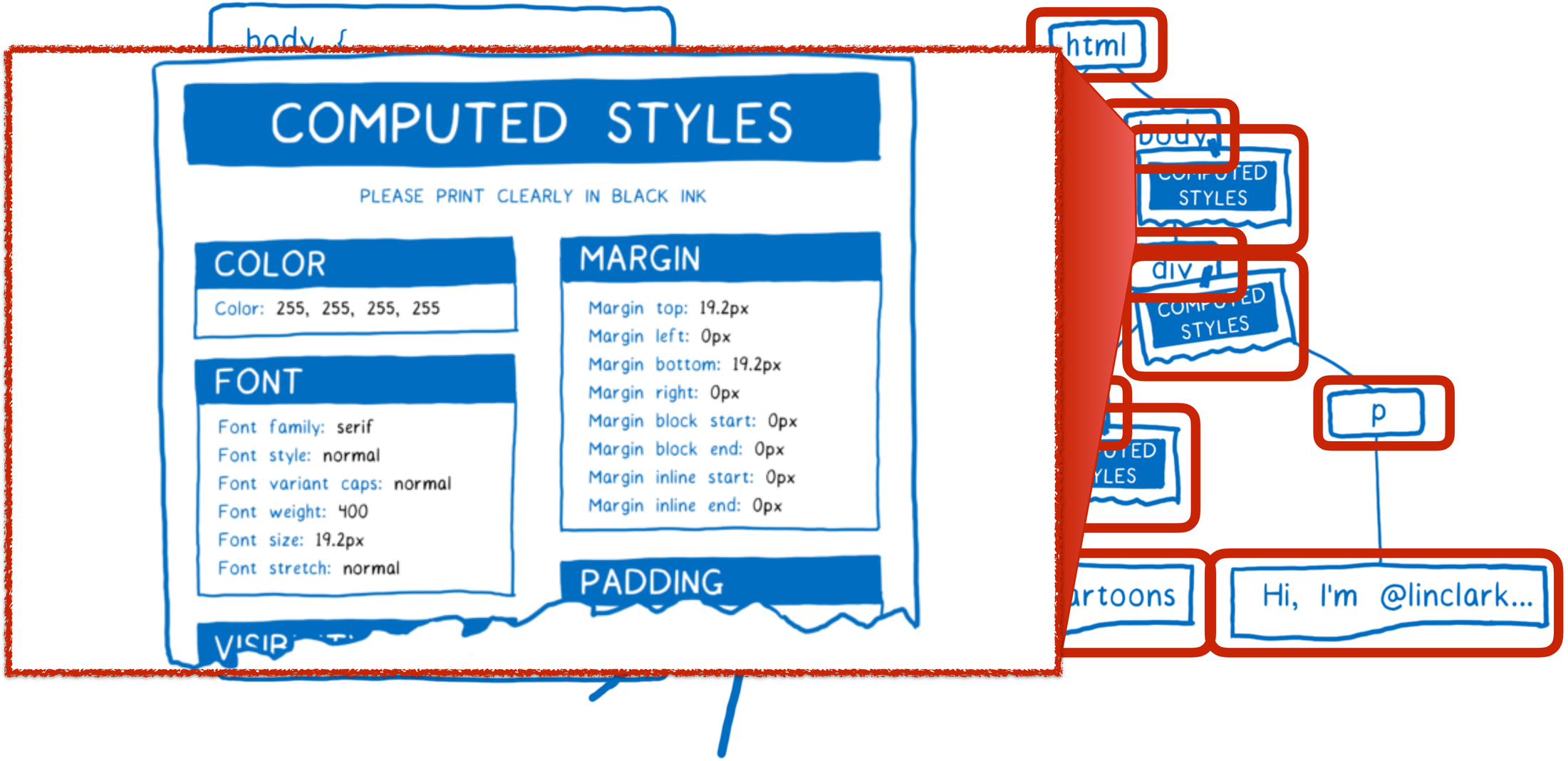
API contains just one unsafe method

Relatively simple correctness criteria

Library handles the rest

Mission accomplished
Rust in Firefox 48

# STYLE

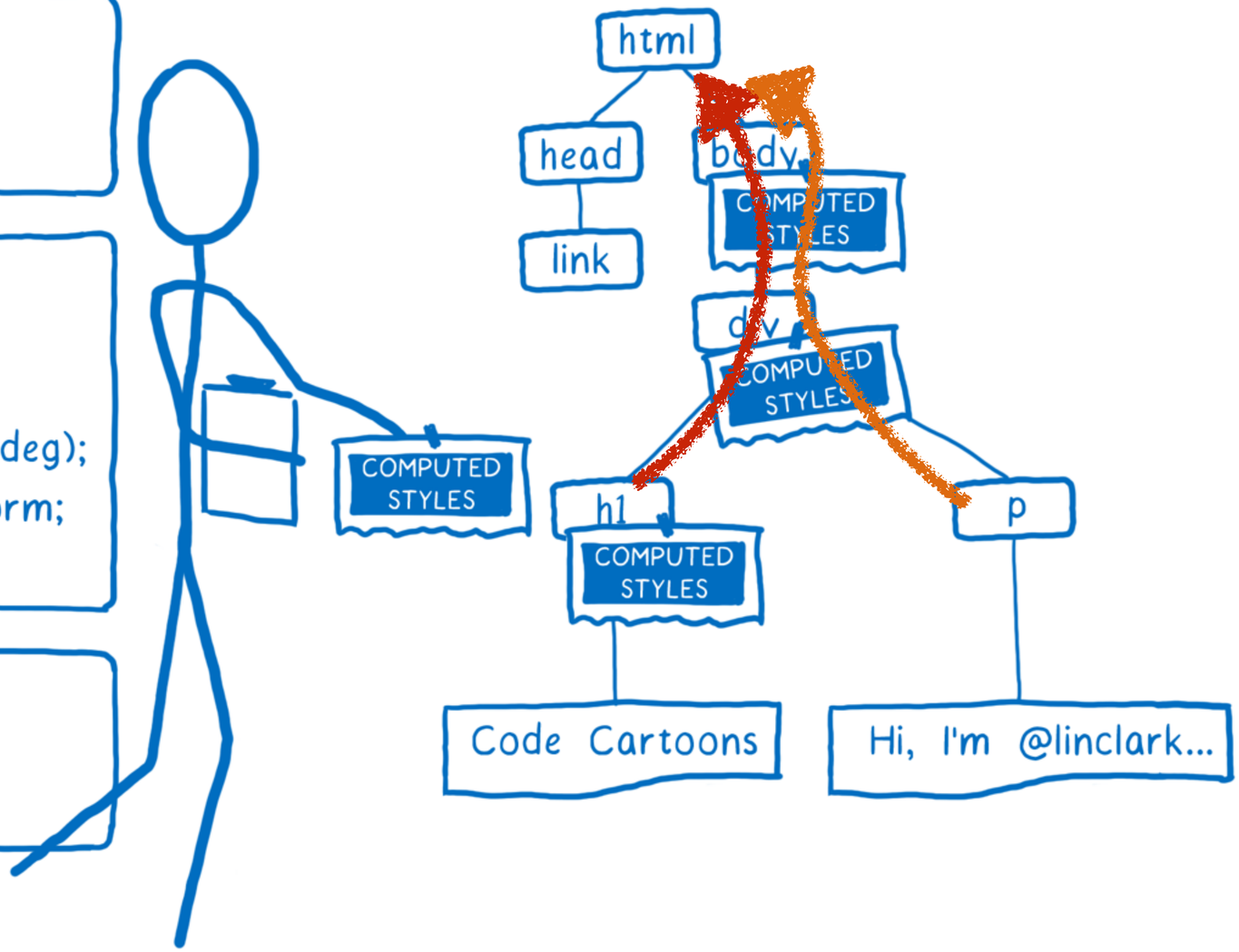

COMPUTED STYLES
PLEASE PRINT CLEARLY IN BLACK INK
COLOR
Color: 255, 255, 255, 255
FONT
Font family: serif
Font style: normal
Font variant caps: normal
Font weight: 400
Font size: 19.2px
Font stretch: normal
MARGIN
Margin top: 19.2px
Margin left: 0px
Margin bottom: 19.2px
Margin right: 0px
Margin block start: 0px
Margin block end: 0px
Margin inline start: 0px
Margin inline end: 0px
PADDING
html
STYLES
STYLES
p
Hi, I'm @linclark...

# STYLE

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

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

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



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

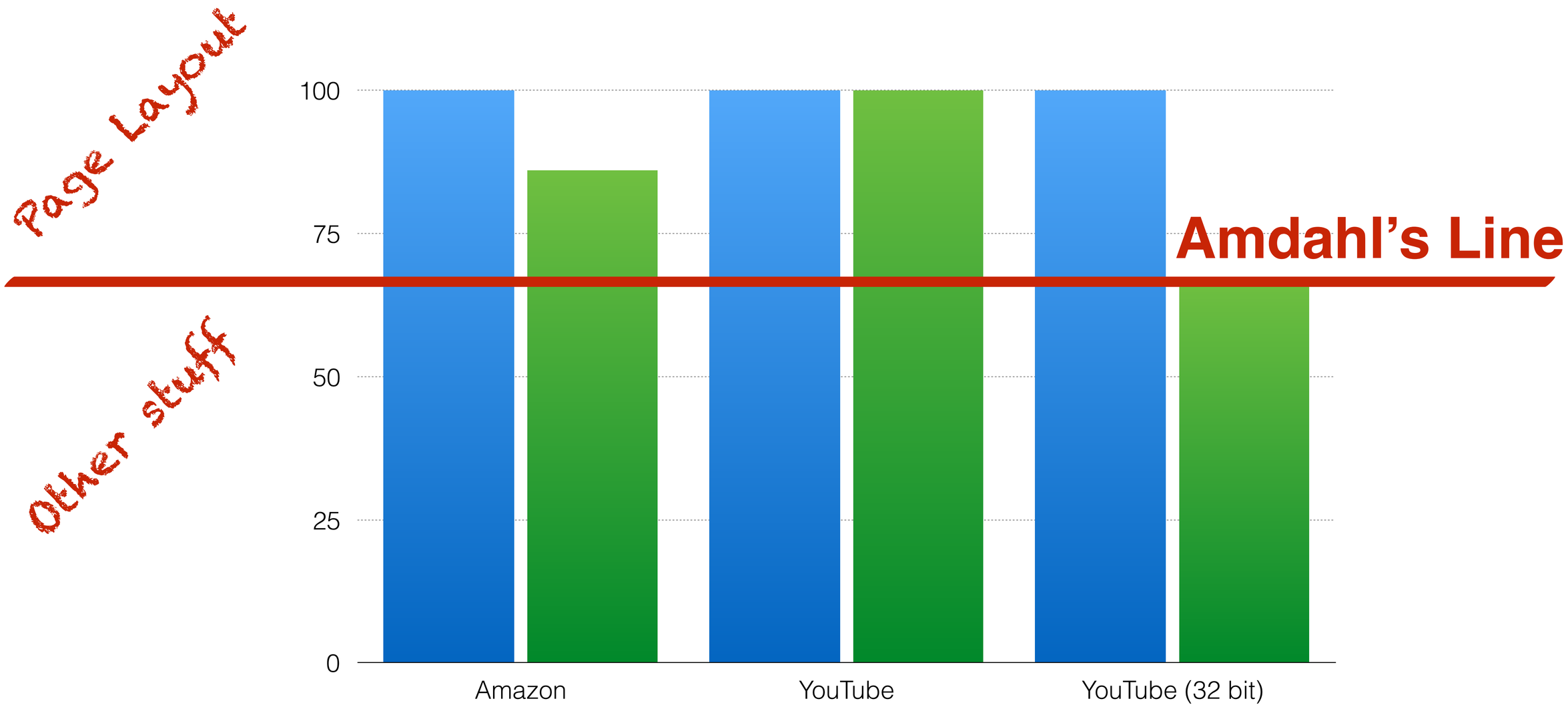
Reported: 7 years ago

Reported: 7 years ago

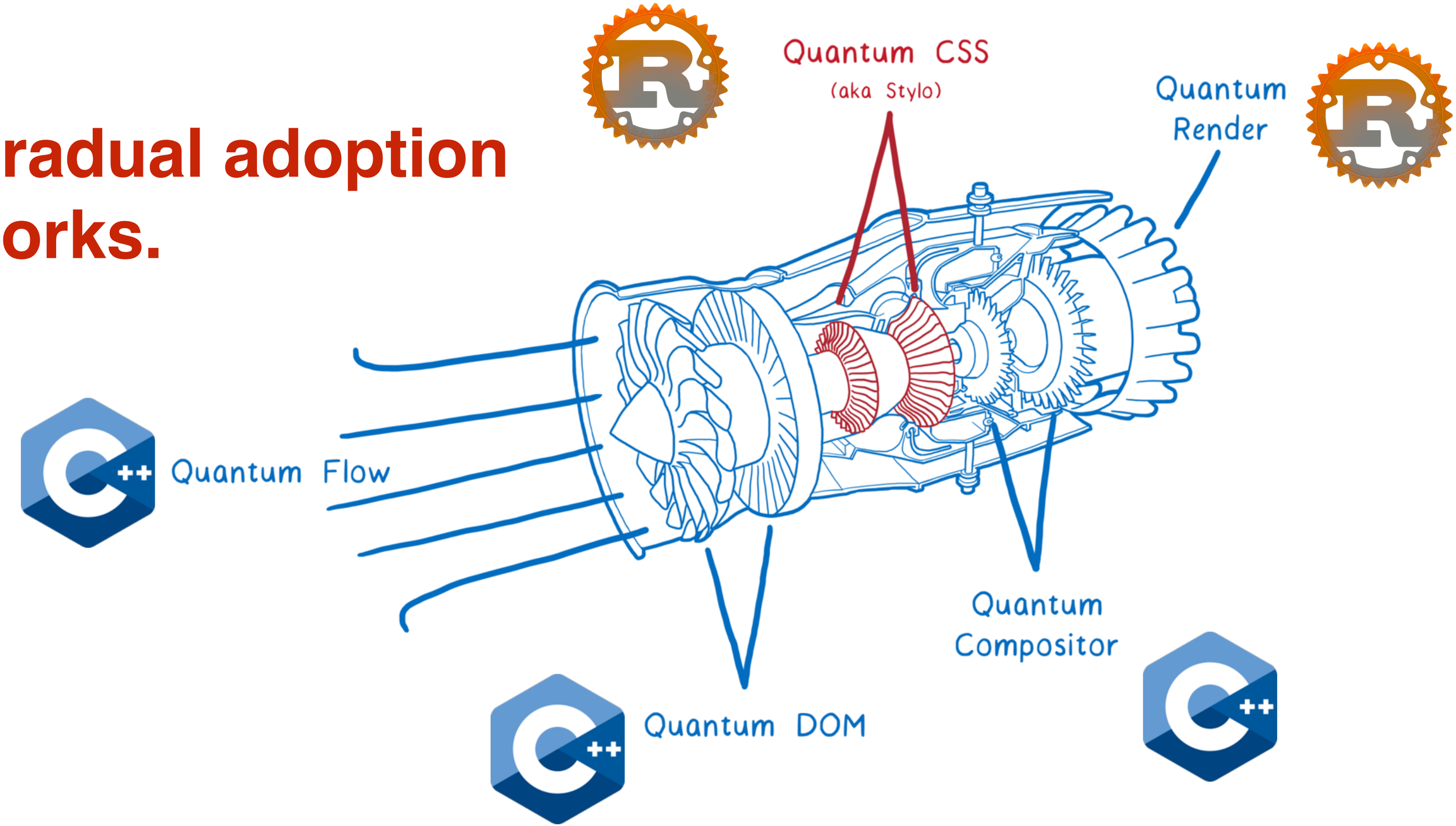
# Safety = Eat your spinach!

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

# Initial load times (relative to today)



# Gradual adoption works.



# Community

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

# Open and welcoming

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

The Rust Programming Language Blog

# impl Future for Rust

Sep 18, 2017 • Aaron Turon

The Rust community has been hard at work on our 2017 roadmap, but as we come up on the final quarter of the year, we're going to kick it into high gear—and we want you to join us!

**Almost 40 working groups spread across 7 Rust "teams"**

# RFC Process

unions #1444

Edit

Merged nikomatsakis merged 14 commits into `rust-lang:master` from `joshtriplett:untagged_union` on Apr 8, 2016

Conversa

## Support defining C-compatible variadic functions in Rust #2137

Edit

Merged aturon merged 25 commits into `rust-lang:master` from `joshtriplett:variadic` 18 days ago

Conversation 134 | Commits 25 | Files changed 1 | +265 −0

joshtriplett commented on Sep 2 • edited by aturon — Member

Support defining C-compatible variadic functions in Rust, via new intrinsics. Rust currently supports declaring external variadic functions and calling them from unsafe code, but does not support writing such functions directly in Rust. Adding such support will allow Rust to replace a larger variety of C libraries, avoid requiring C stubs and error-prone reimplementation of platform-specific code, improve incremental translation of C codebases to Rust, and allow implementation of variadic callbacks.

This RFC does not propose an interface intended for native Rust code to pass variable numbers of arguments to a native Rust function, nor an interface that provides any kind of type safety. This proposal exists primarily to allow Rust to provide interfaces callable from C code.

Reviewers

- eddyb
- kennytm
- tomwhoiscontrary
- jethrogb
- xfix
- plietar
- ubsan
- fstirlitz
- cramertj

# Interesting ongoing research problems

**chalk: formalizing the trait system**

Translating our trait semantics into logical predicates
Applying work on logic programming to solve them

**Runtimes: improving performance of Rayon**

Work-stealing runtime, integrating with futures and I/O

**Optimizing compilers**

In principle, Rust can optimize like crazy
In practice, there are some obstacles we have to resolve

**Caller knows that `vec` does not escape**

```
fn some_function(vec: &mut [i32]) {
 …
}
```

**Callee knows that `vec` is unaliased**

```
static mut HAHA: *mut i32;

fn some_function(vec: &mut [i32]) {
 unsafe {
   let p: *mut i32 = vec;
   let q: *mut i32 = p;
   HAHA = q;
 }
}
```

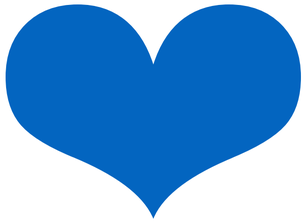
# Interested in Rust?

rust-lang.github.io/book

users.rust-lang.org

#rust-beginners on irc.mozilla.org

intorust.com



# 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
```

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

# “Zero cost” abstraction

```
vec1.iter()                    // Iter
    .zip(vec2.iter())          // Zip<Iter, Iter>
    .map(|(i, j)| i * j)       // Map<Zip<Iter, Iter>, …>
    .sum()
```

**=> invokes `next()` repeatedly and adds**

```
.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
```

```
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,
    }
  }
}
```

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

// Zip<Iter, Iter>
```

**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.**

# Marker traits

| | ✅ | 🚫 |
|---|---|---|
| `// “Safe to send between threads”`<br>`trait Send { }` | `String`<br>`u32`<br>`Arc<String>` | `Rc<String>` |
| `// “Safe to memcpy”`<br>`trait Copy { }` | `u32`<br>`f32` | `String`<br>`Rc<String>` |