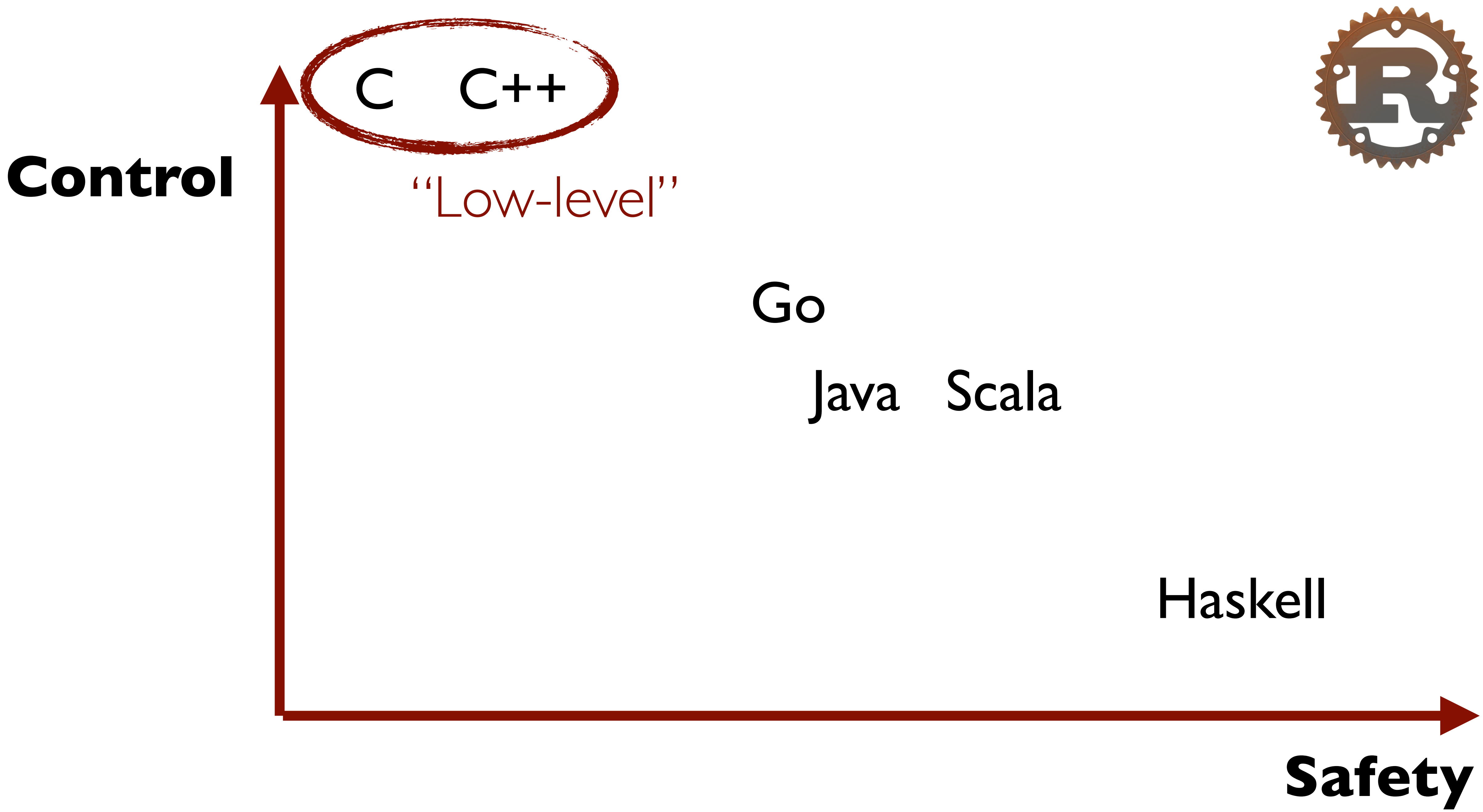
Aaron Turon
Mozilla Research

Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety.

- *https://www.rust-lang.org/*


C C++
"Low-level"
Control
Go
Java Scala
Haskell
Safety

Conventional Wisdom

The Essence of Rust

Low-level == Unsafe

Low-level ≠ Safe

Why Rust?

- You're already doing systems programming, want safety or expressiveness.

- You *wish* you could do some systems work
  - Maybe as an embedded piece in your Java, Python, JS, Ruby, …

Why Mozilla?

Browsers need **control**.

Browsers need **safety**.

Rust: New language for safe systems programming.

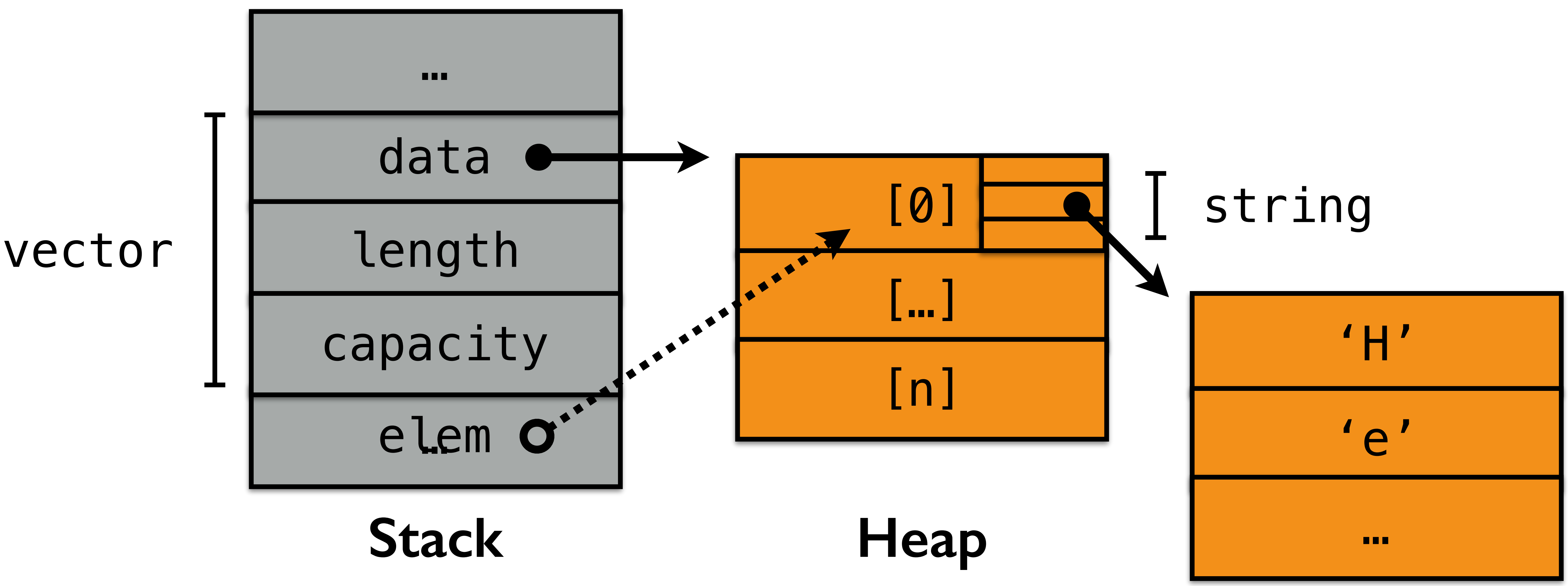
Servo: Next-generation browser built in Rust.

C++

What is **control**?

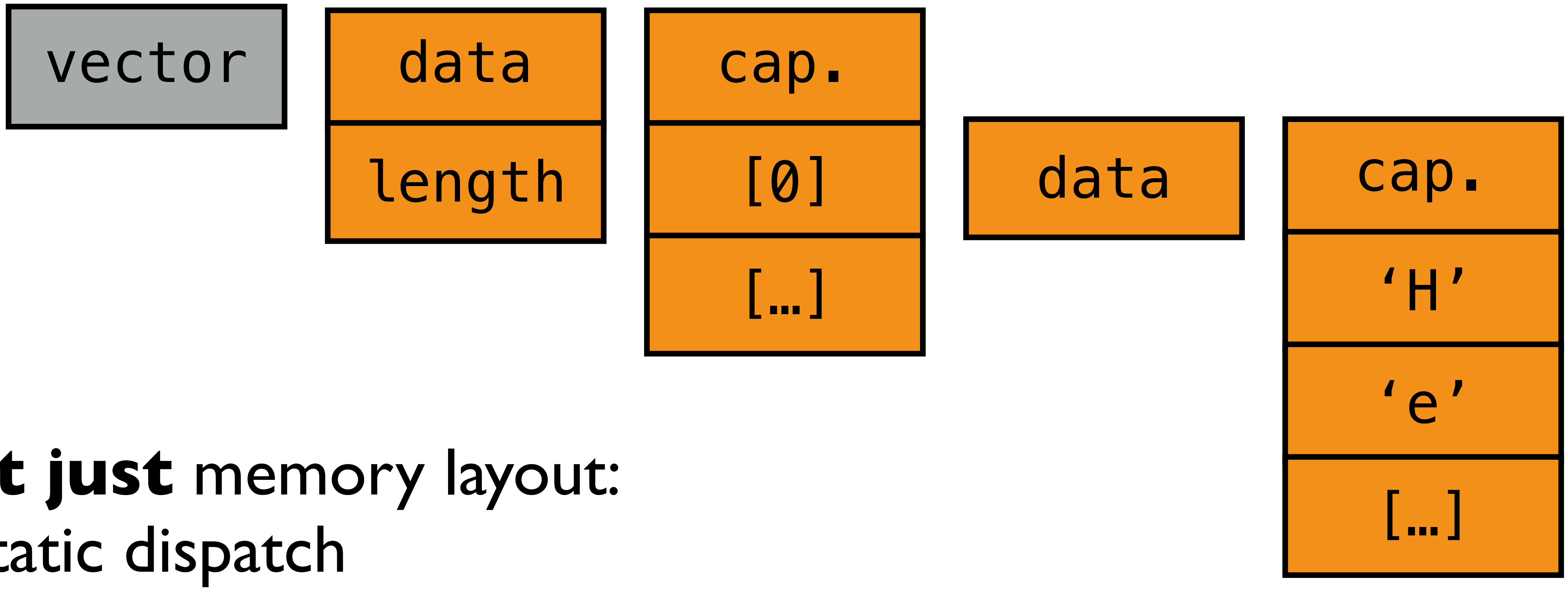
```
void example() {
   vector<string> vector;
   …
   auto& elem = vector[0];
   …
}
```

- Stack and inline layout. (`vector<string> vector;`)
- Interior references (`auto& elem = vector[0];`)
- Deterministic destruction (`}`)



Zero-cost abstraction

Ability to define **abstractions** that **optimize away to nothing**.



Not just memory layout:

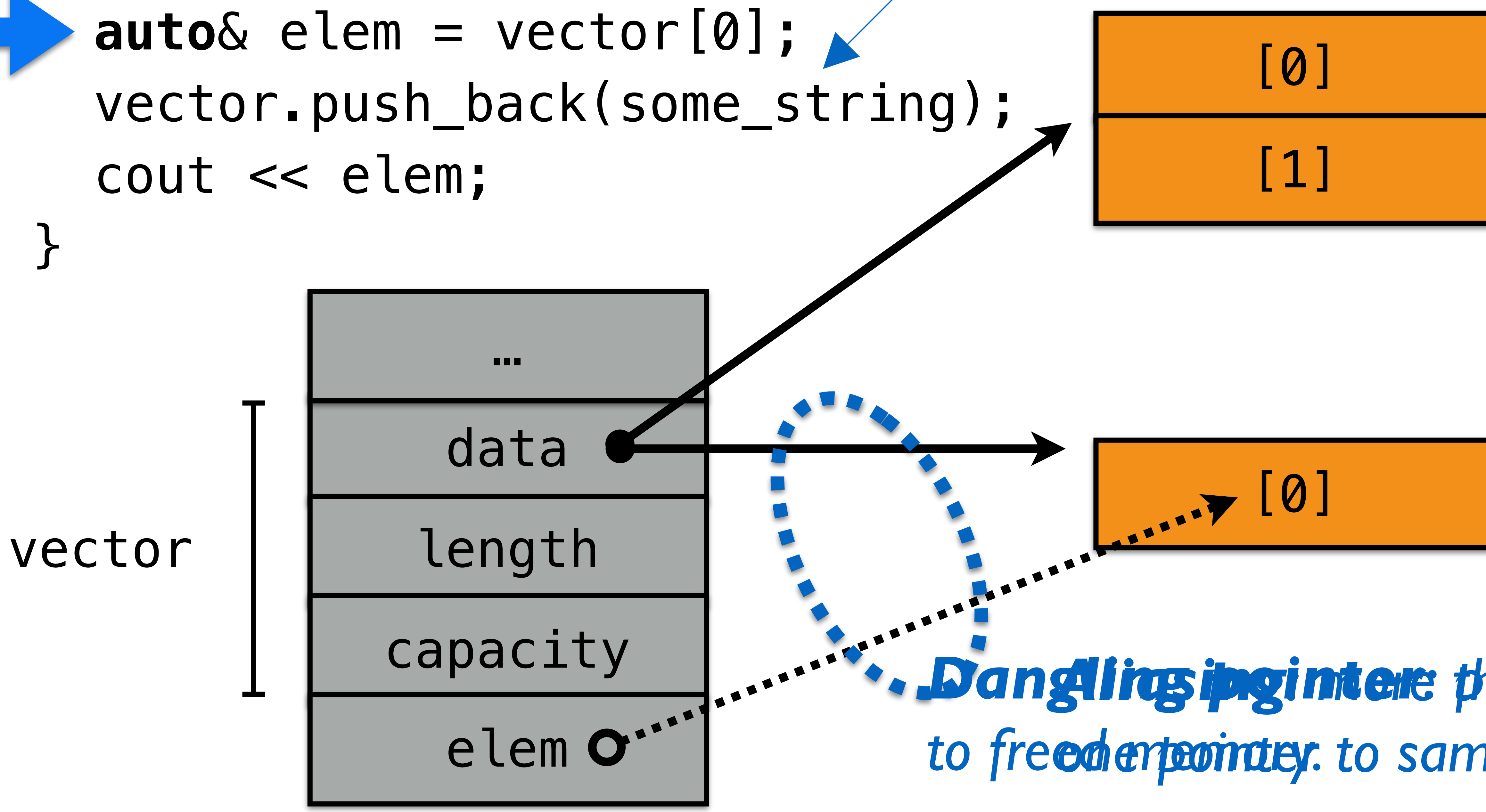
- Static dispatch
- Template expansion
- …

Java

What is **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: *pointer to freed memory.*

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

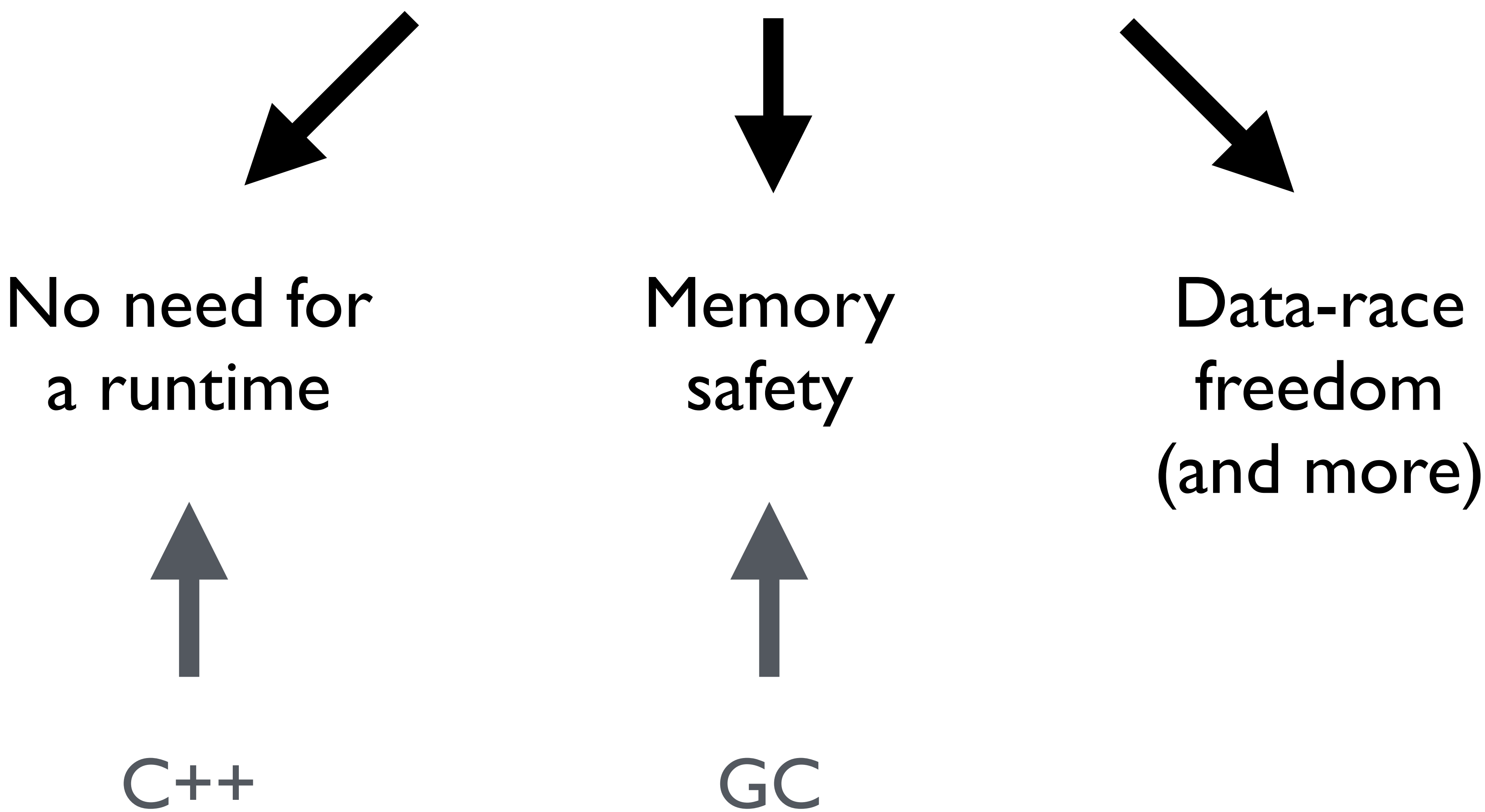
What about **GC**?

No **control**.

Requires a **runtime**.

Insufficient to prevent related problems: iterator invalidation, data races, many others.

Ownership & Borrowing



… Plus lots of goodies

- Pattern matching
- Traits
- “Smart” pointers
- Metaprogramming
- Package management (think Bundler)

TL;DR: Rust is a modern language

Ownership

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

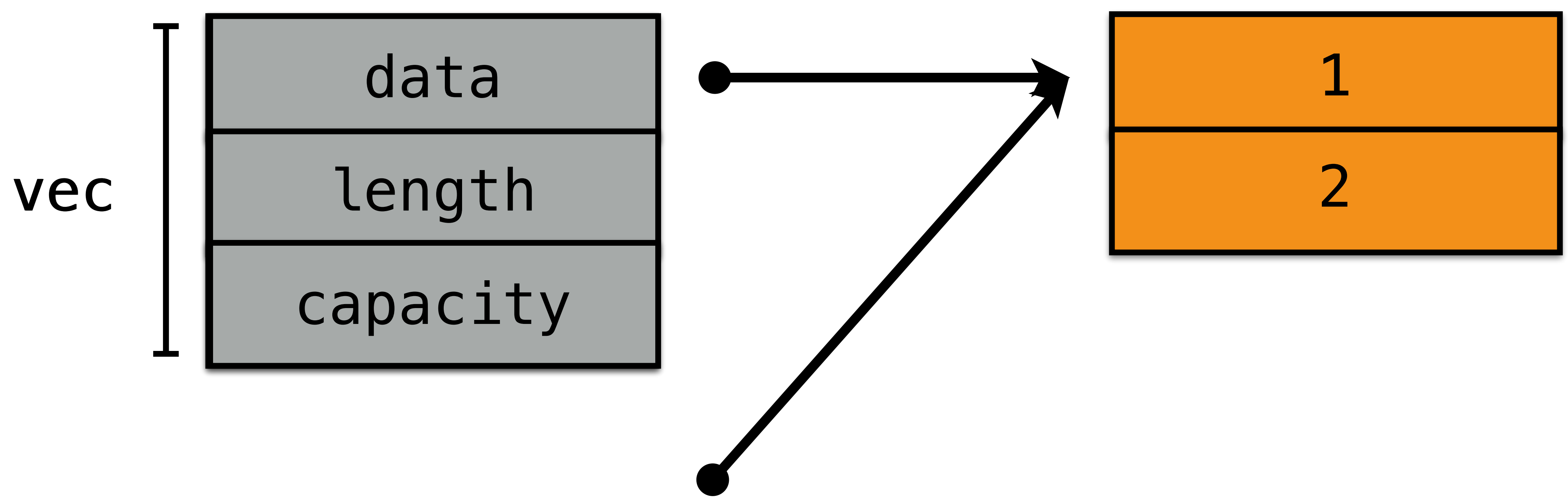
~~Aliasing~~ + Mutation

Ownership (T)

```
fn give() {
  let mut vec = Vec::new();
  vec.push(1);
  vec.push(2);
  take(vec);
  …
}
```

```
fn take(vec: Vec<int>) {
  // …
}
```

Take ownership of a Vec<int>



Compiler **enforces** moves

```
fn give() {
  let mut vec = Vec::new();
  vec.push(1);
  vec.push(2);
  take(vec);
  vec.push(2);
}
```

```
fn take(vec: Vec<int>) {
  // …
}
```

Error: vec has been moved

Prevents:

- use after free
- double moves
- …

Borrow

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

Aliasing + ~~Mutation~~

Shared borrow (&T)

~~Aliasing~~ + Mutation

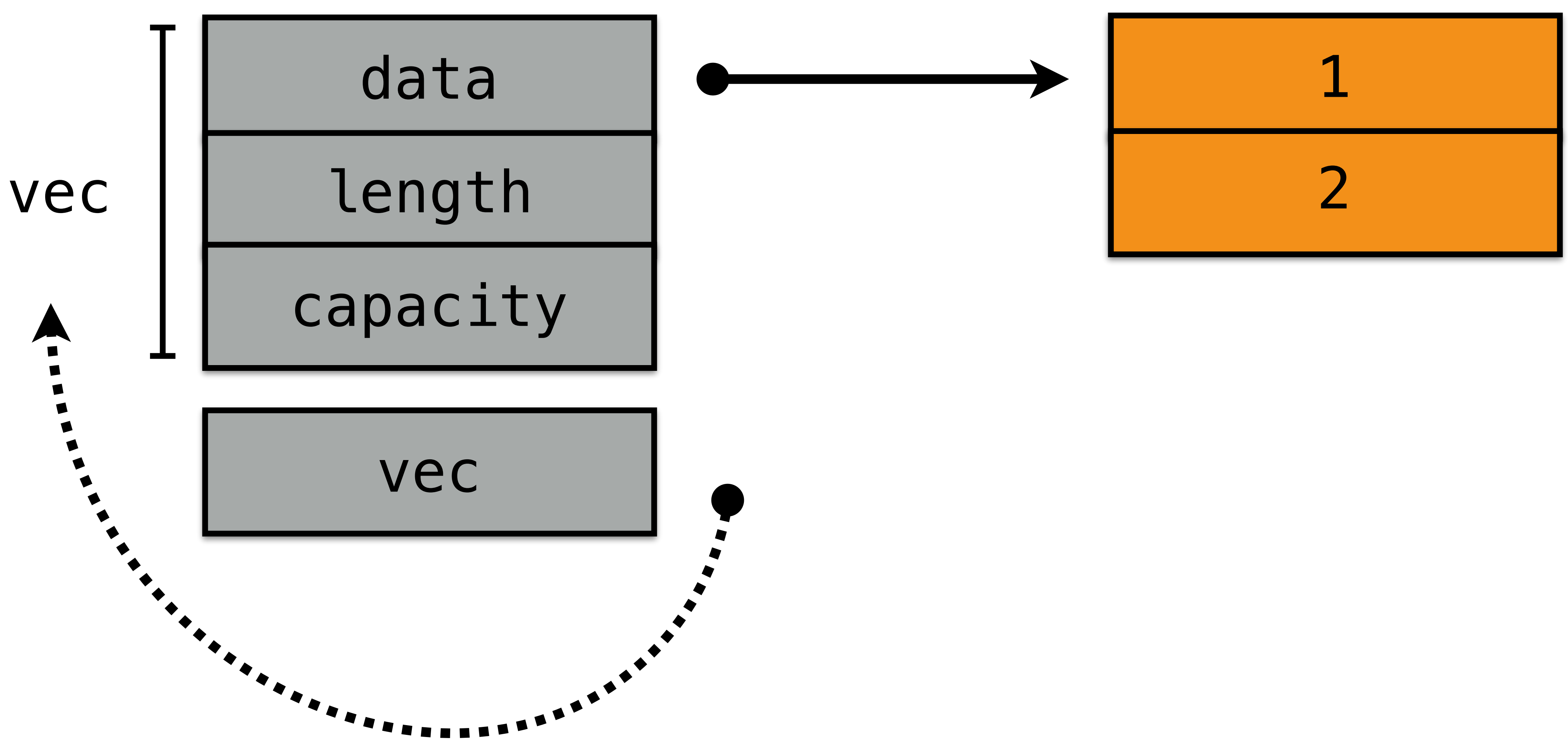
Mutable borrow (&mut T)

```
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>”



Aliasing + ~~Mutation~~

Shared references are **immutable**: *

```
fn use(vec: &Vec<int>) {
  vec.push(3);
  vec[1] += 2;
}
```

Error: cannot mutate shared reference

* Actually: mutation only in controlled circumstances

Mutable references

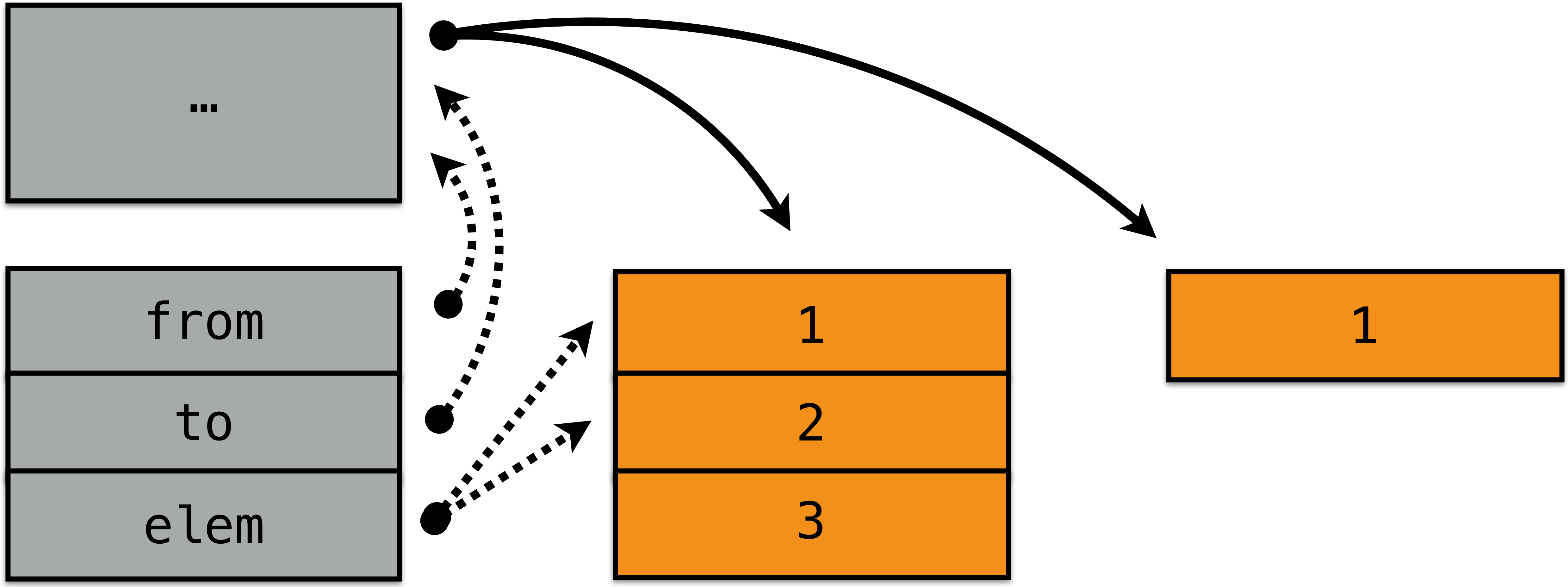
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {
  for elem in from {
   to.push(*elem);
  }
}
```

mutable reference to `Vec<int>`

`push()` is legal

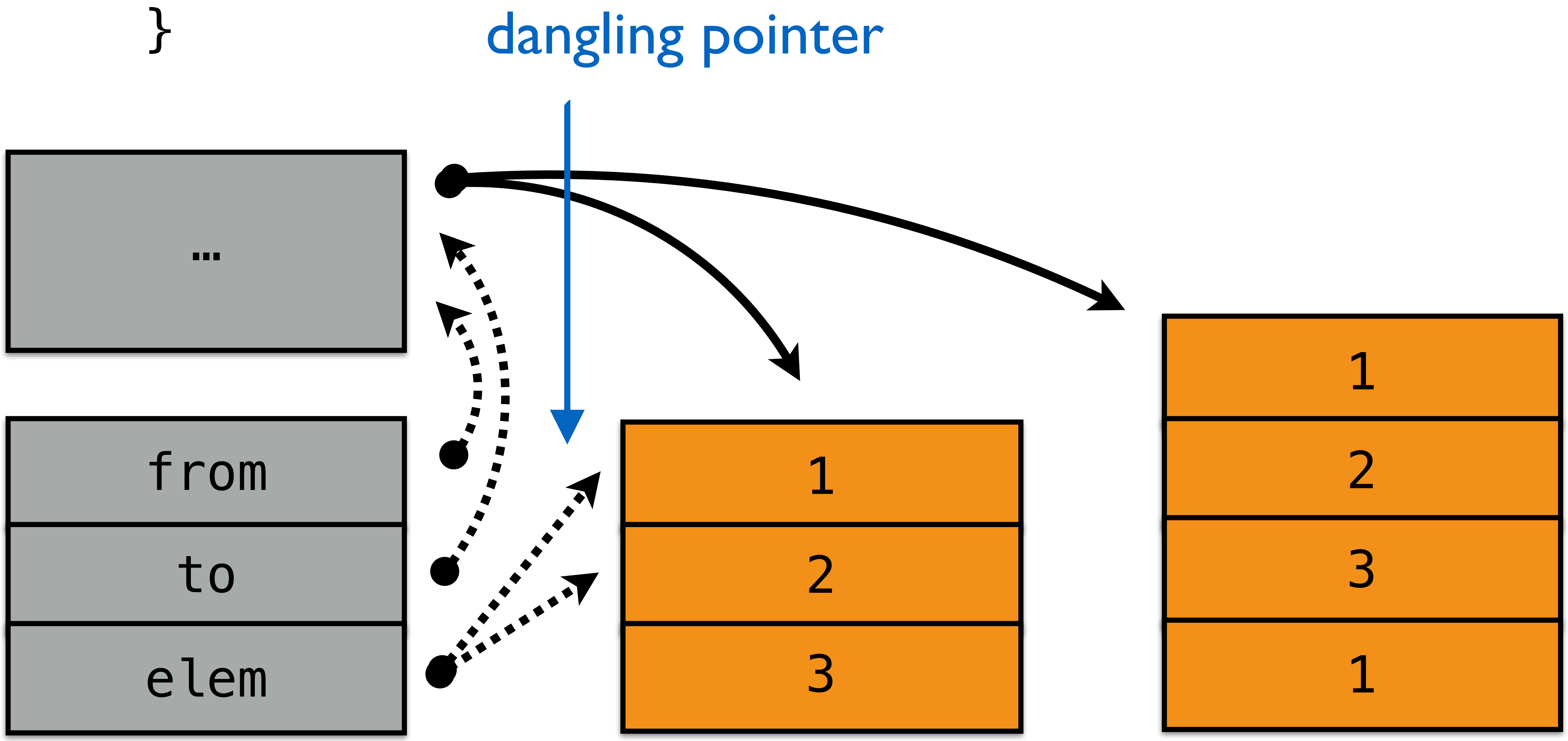
Mutable references

```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {
   for elem in from {
    to.push(*elem);
   }
}
```



What if **from** and **to** are equal?

```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {
  for elem in from {
    to.push(*elem);
  }
}
```



```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {…}

fn caller() {
    let mut vec = …;
    push_all(&vec, &mut vec);
}
```

(The text `&mut vec);` is struck through on the slide.)

shared reference

Error: cannot have both shared and mutable reference at same time

> A **&mut T** is the **only way** to access the memory it points at

```
{
  let mut vec = Vec::new();
  …
  for i in 0 .. vec.len() {
    let elem: &int = &vec[i];
    …
    vec.push(…);
  }
  …
  vec.push(…);
}
```

Error: vec[i] is borrowed, cannot mutate (at the struck-through `vec.push(…);` inside the loop)

OK. loan expired. (at the `vec.push(…);` after the loop)

Borrows restrict access to the original path for their duration.

&	no writes, no moves
&mut	no access at all

Concurrency

n. several computations executing simultaneously, and potentially interacting with each other.

Rust’s vision for concurrency

Originally: only isolated message passing

Now: libraries for many paradigms,
using ownership to avoid footguns,
guaranteeing no data races

Data race

Two **unsynchronized** threads
accessing **same data**
where **at least one writes**.

Sound familiar?

Aliasing

+ Mutation

+ No ordering

Data race

No data races =
No accidentally-shared state.

All sharing is explicit!

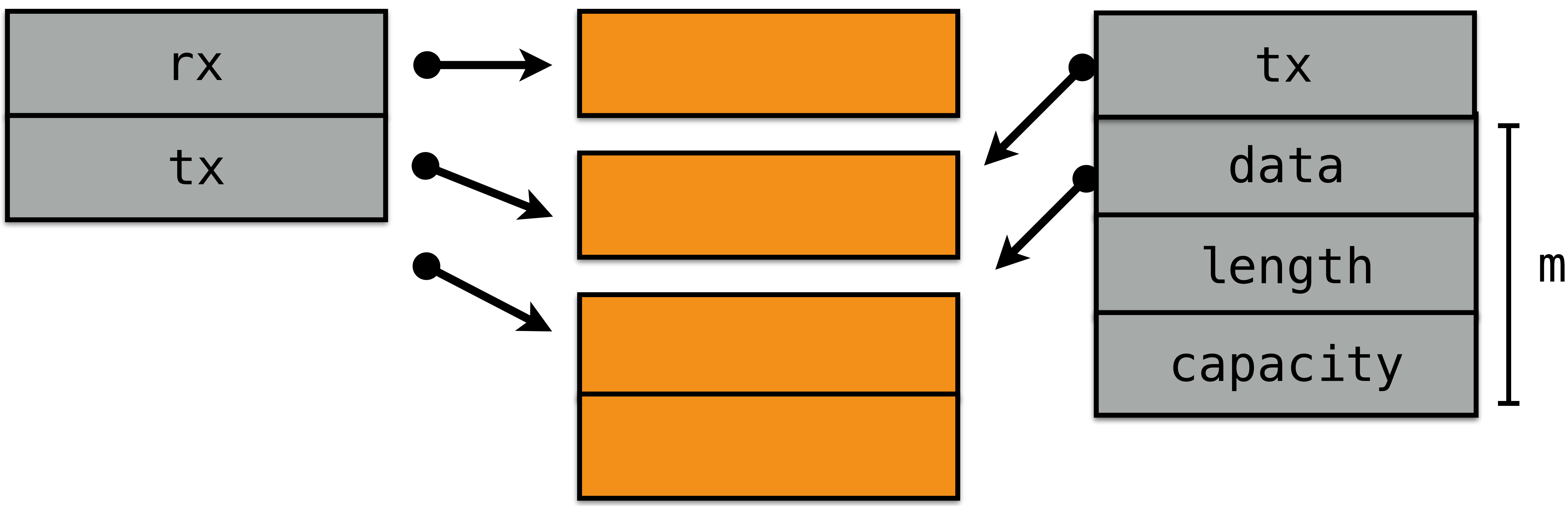
```
*some_value = 5;
return *some_value == 5;  // ALWAYS true
```

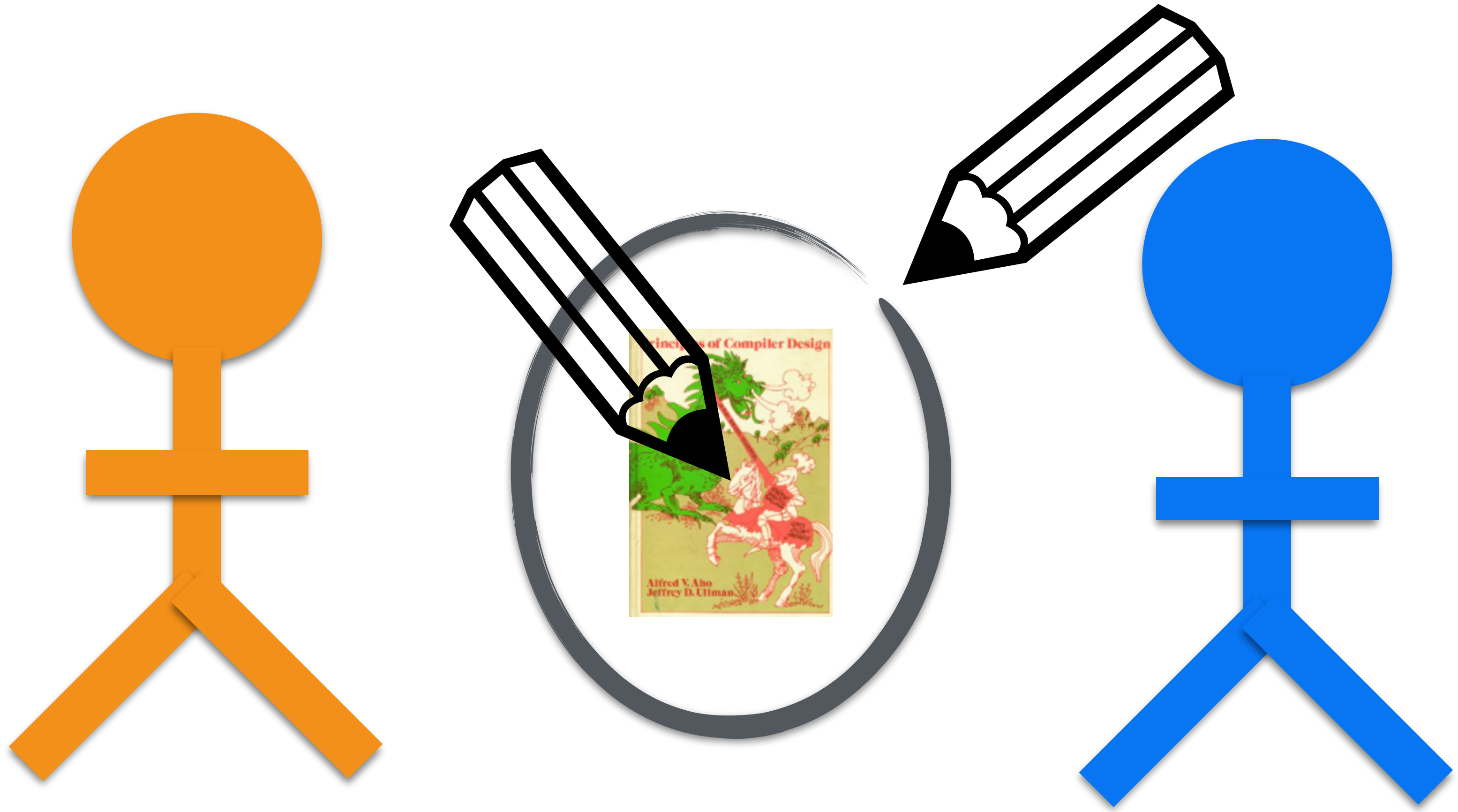
Messaging

(ownership)

```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv();
}
```

```
move || {
  let m = Vec::new();
  …
  tx.send(m);
}
```





Locked mutable access

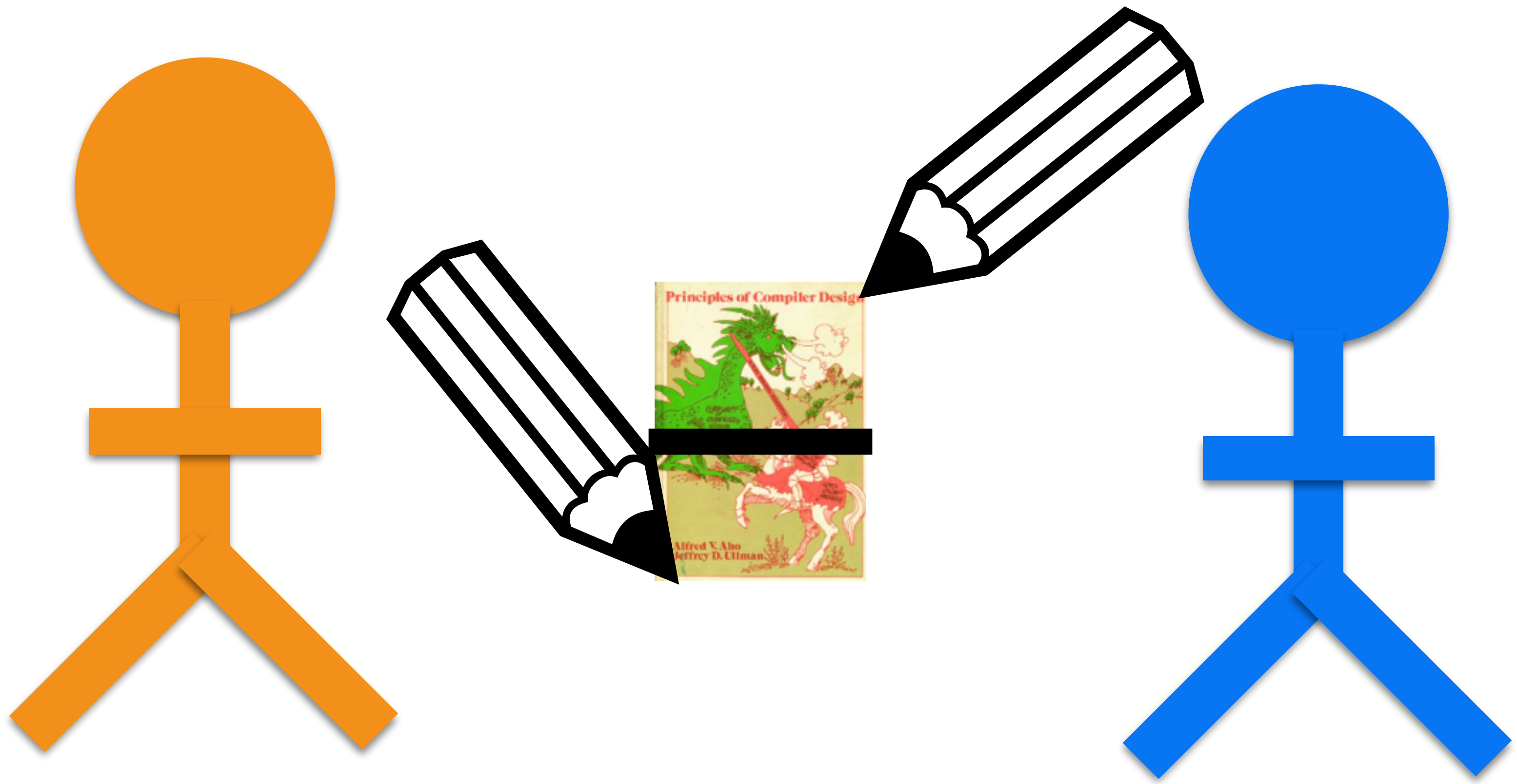
(ownership, borrowing)

```
fn sync_inc(mutex: &Mutex<int>) {
  let mut data = mutex.lock();
  *data += 1;
}
```

Destructor releases lock

Yields a mutable reference to data

Destructor runs here, releasing lock

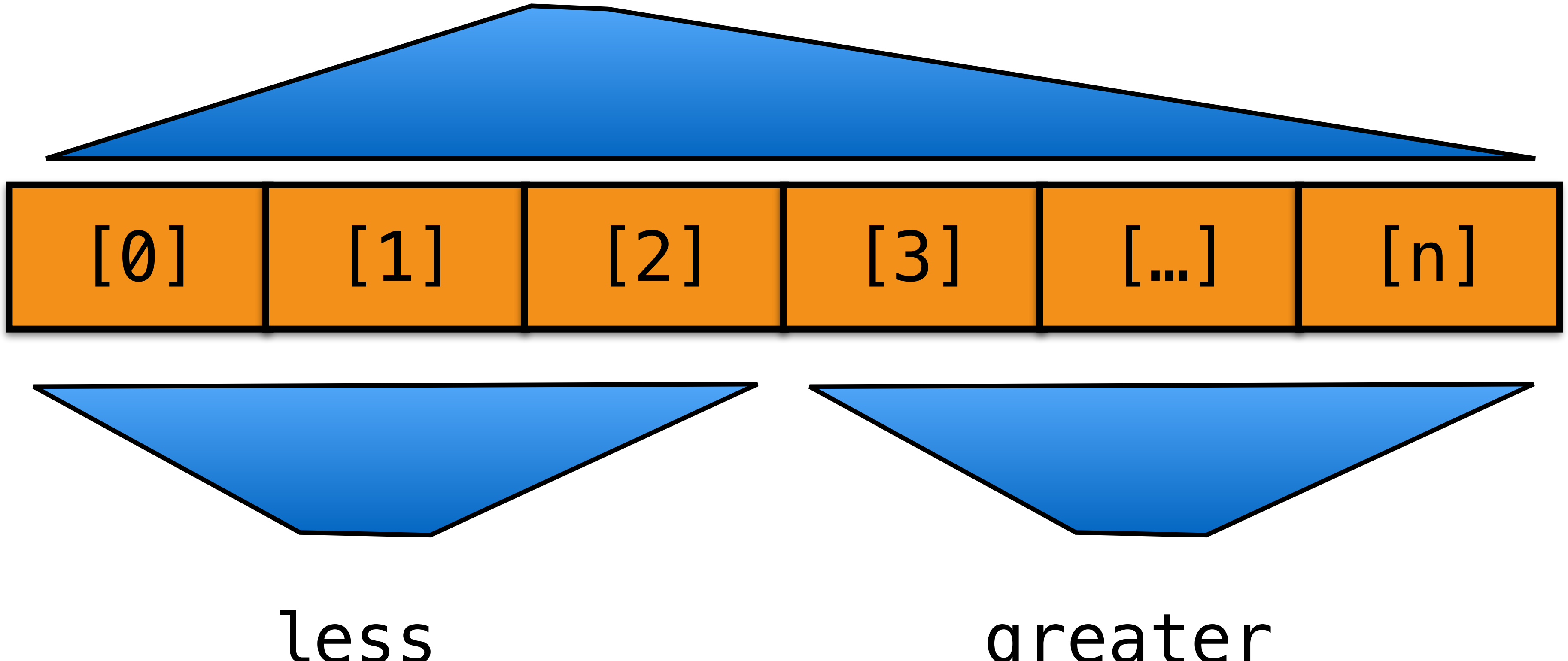


Disjoint, scoped access

(borrowing)

```
fn qsort(vec: &mut [int]) {
  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);
}
```

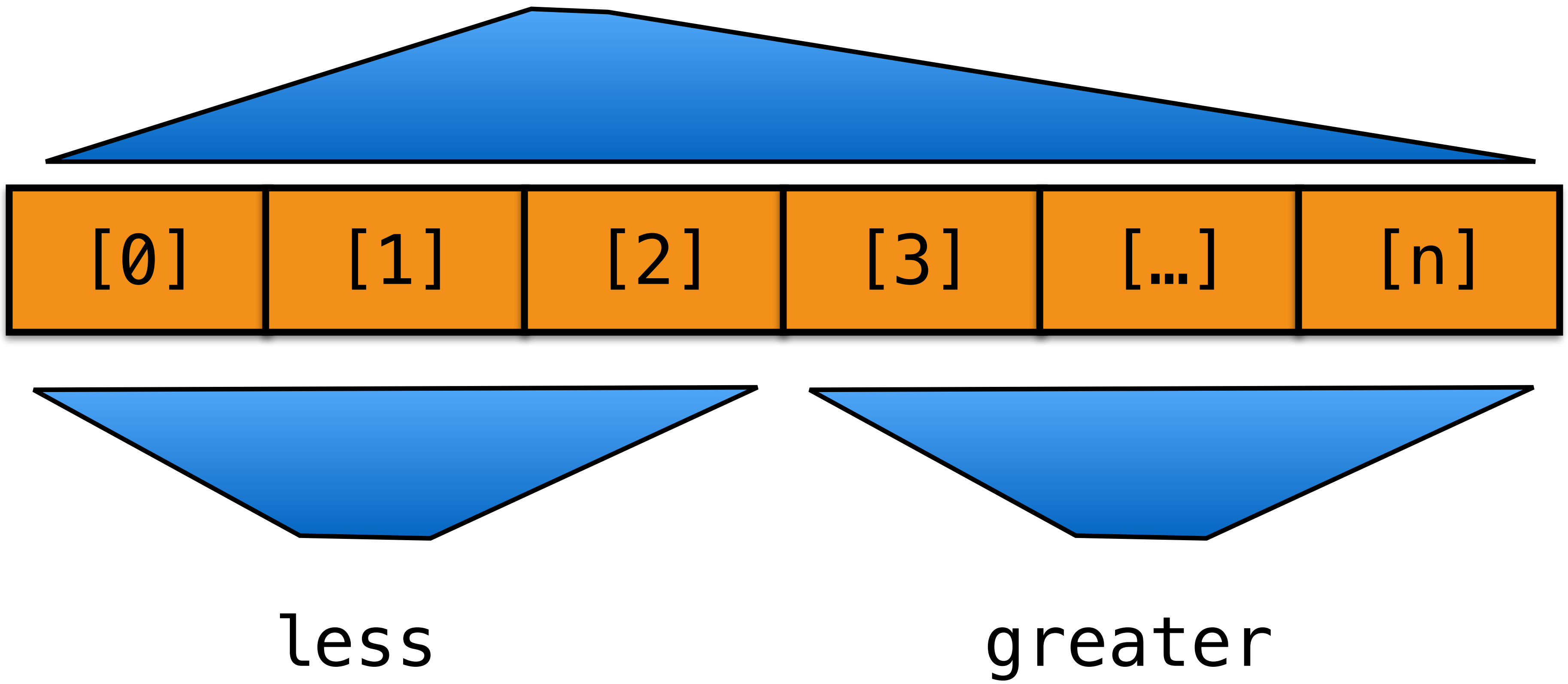
```
let vec: &mut [int] = …;
```



```
fn split_at_mut(&mut self, mid: usize)
              -> (&mut [T], & mut [T])
```

```
fn parallel_qsort(vec: &mut [int]) {
  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);
  parallel::join(
    || parallel_qsort(less),
    || parallel_qsort(greater)
  );
}
```

```
let vec: &mut [int] = …;
```



Static checking for thread safety

```
fn send<T: Send>(&self, t: T)
```

Only "sendable" types

```
Arc<Vec<int>>: Send
Rc<Vec<int>> : !Send
```

And beyond…

Concurrency is an area of **active development**.

Either already have or have plans for:

- Atomic primitives
- Non-blocking queues
- Concurrent hashtables
- Lightweight thread pools
- Futures
- CILK-style fork-join concurrency
- etc.

Always data-race free

Unsafe

adj. not safe; hazardous

Safe abstractions

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

Validates input, etc.

Trust me.

Useful for:

- Bending mutation/aliasing rules (split_at_mut)
- Interfacing with C code

Ownership enables **safe** abstraction boundaries.

Community

n. A feeling of fellowship with others sharing similar goals.

“The Rust community seems to be populated entirely by human beings. I have no idea how this was done.”

— *Jamie Brandon*

It takes a village…

Community focus from the start:

Rust 1.0 had > 1,000 contributors

Welcoming, pragmatic culture

Developed “in the open”

Much iteration; humility is key!

Clear leadership

Mix of academic and engineering backgrounds

“Keepers of the vision”

RFC: associated items and multidispatch #195

Merged **alexcrichton** merged 5 commits into `rust-lang:master` from `aturon:associated-items` on Sep 16, 2014

Conversation 69 | Commits 5 | Files changed 1

aturon commented on Aug 12, 2014 — Owner

This RFC extends traits with *associated items*, which make generic programming more convenient, scalable, and powerful. In particular, traits will consist of a set of methods, together with:

- Associated functions (already present as "static" functions)
- Associated statics
- Associated types
- Associated lifetimes

These additions make it much easier to group together a set of related types, functions, and constants into a single package.

This RFC also provides a mechanism for *multidispatch* traits, where the `impl` is selected based on multiple types. The connection to associated items will become clear in the detailed text below.

Rendered view

Articulating the vision

Memory safety		garbage collection
Concurrency	*without*	data races
Abstraction		overhead
Stability		stagnation

Hack without fear!