Rust

Nicholas Matsakis
Mozilla Research

What makes Rust different?

C++ Java ML Haskell

More Control **More Safety**

Rust: Control *and* safety

Why Mozilla?

Browsers need **control**.

Browsers need **safety**.

Servo: Next-generation
browser built in Rust.

C++

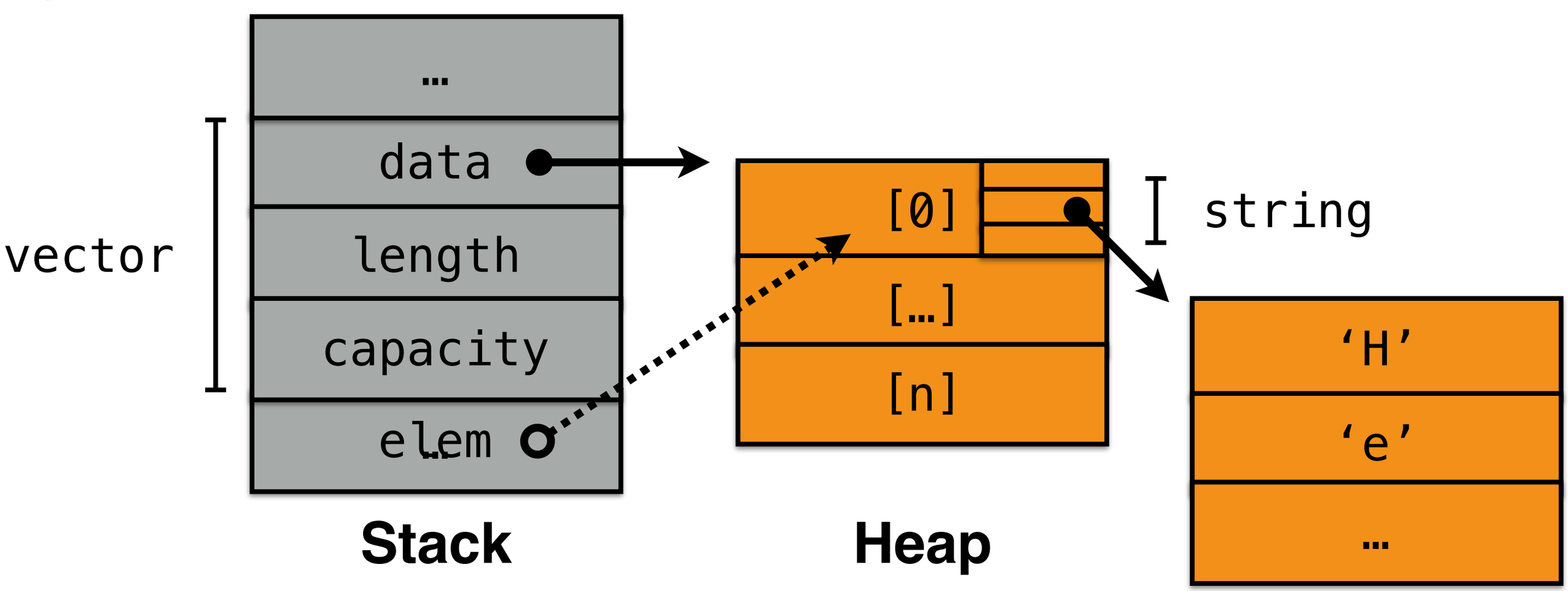
What is **control?**

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

← Stack and inline layout.

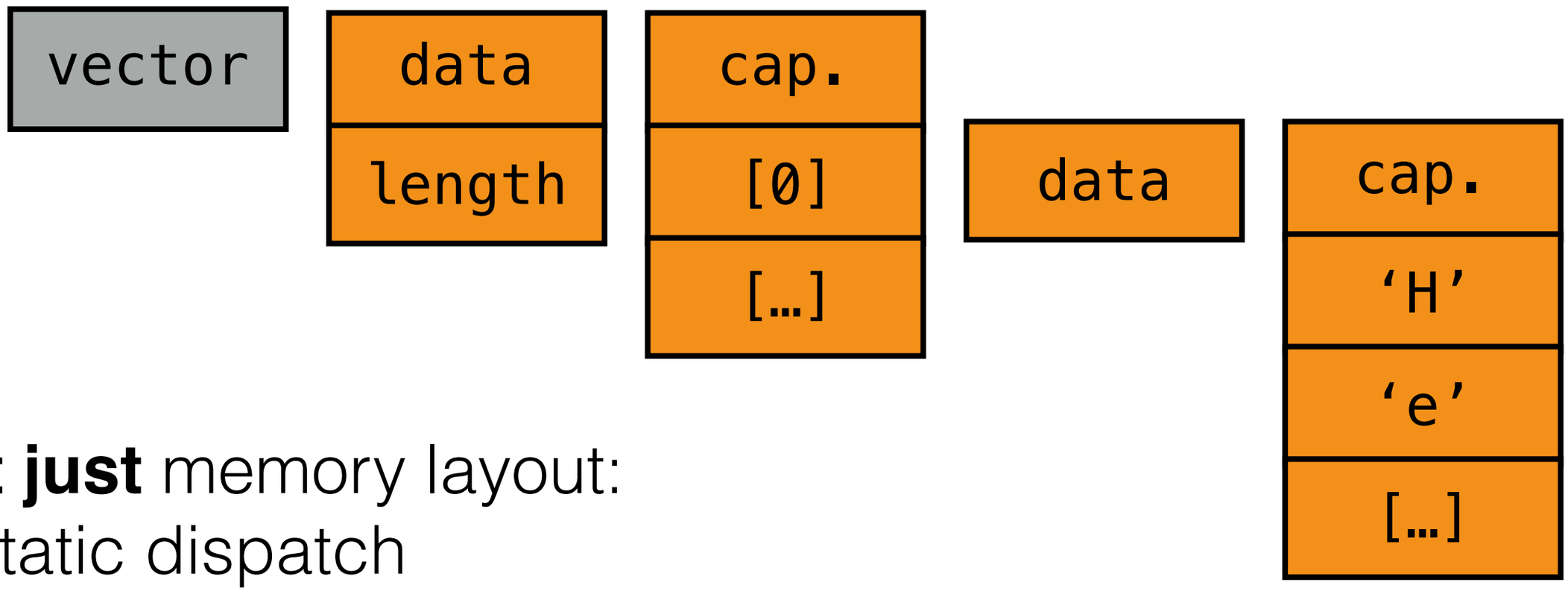
← Lightweight references

← Deterministic destruction



Zero-cost abstraction

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



Not just memory layout:

- Static dispatch
- Template expansion
- …

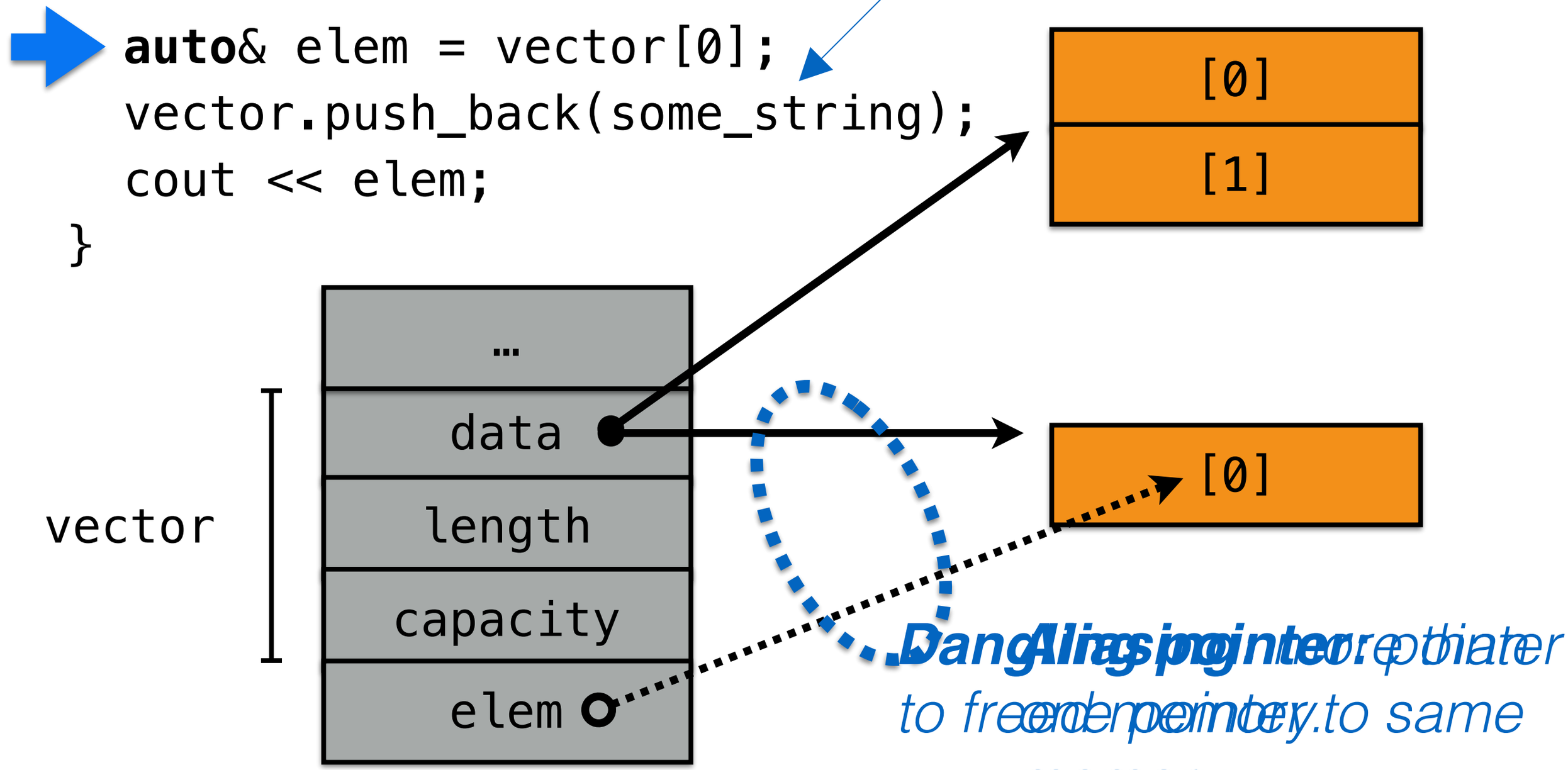
Java

C++

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.



What about **GC?**

No **control**.

Requires a **runtime**.

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

C++

Read it:

```
void foo(const Foo &f, …) {
    use(f.bar);
}
```

Write it:

```
void foo(Foo &f, …) {
    f.bar = …;
}
```

Gut it:

```
void foo(Foo &&f, …) {
    vec.push_back(f);
}
```

Keep it:

```
void foo(unique_ptr<Foo> f, …) {
    …
}
```

Definitely **progress**.

But definitely **not *safe***:

- Iterator invalidation.
- Double moves.
- Pointers into stack or freed memory.
- Data races.
- … *all that stuff I talked about before …*

Ultimately, C++ mechanisms are **unenforced**.

The Rust Solution

Codify and enforce **safe patterns** using the type system:

1. Always have a clear owner.
2. While iterating over a vector, don’t change it.
3. …

No runtime required.

Credit where it is due

Rust **borrows liberally** from other great languages:

C++, Haskell, ML/Ocaml, Cyclone,
Erlang, Java, C#, …

Rust has an **active, amazing community**.

The Rust type system

Observation

Danger arises from…

Aliasing + **Mutation**

Hides dependencies.

Causes memory to be freed.

```
{   auto& e = v[0];   …   v.push_back(…);   }
```

Three basic patterns

Ownership

```
fn foo(v: T) {
    …
}
```

Shared borrow

```
fn foo(v: &T) {
    …
}
```

Mutable borrow

```
fn foo(v: &mut T) {
    …
}
```

Ownership

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

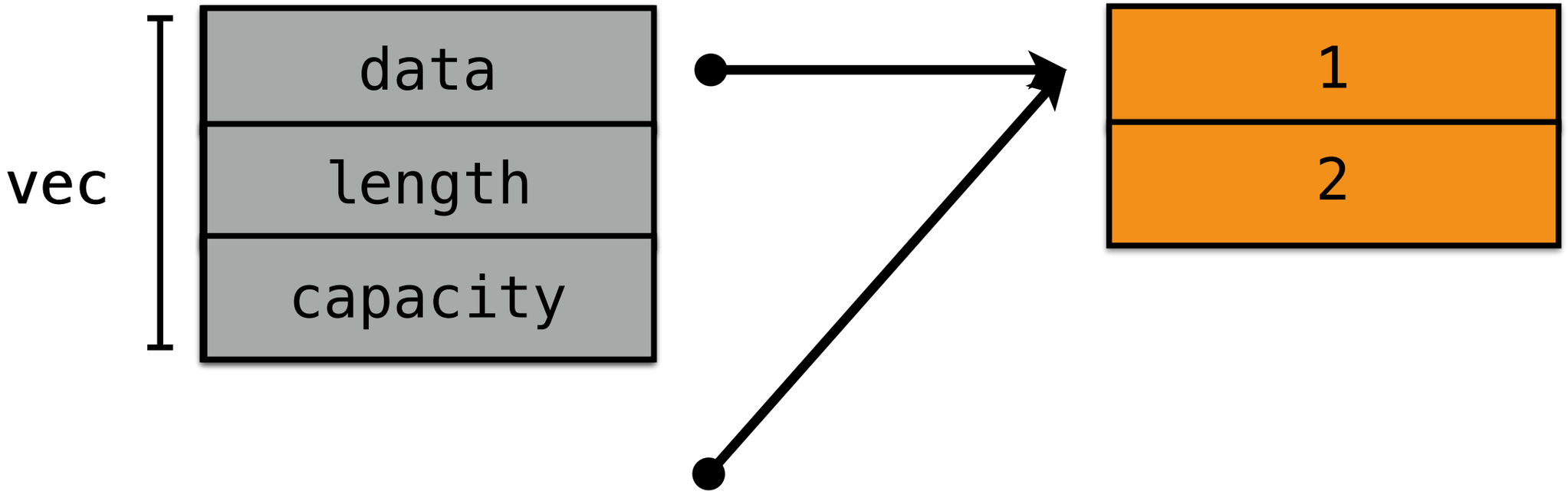
~~Aliasing~~ + Mutation

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

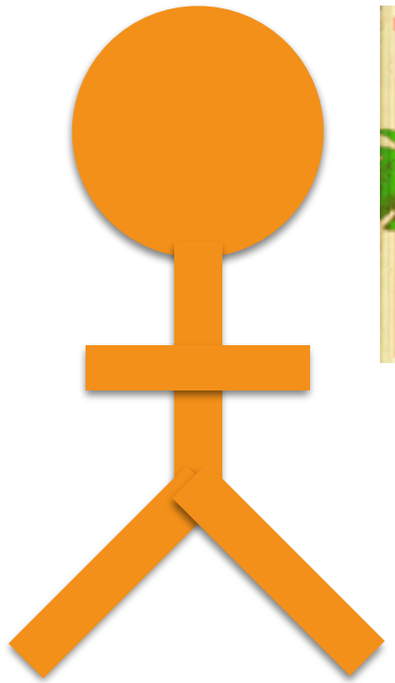
Prevents:

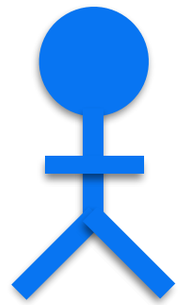
- use after free
- double moves
- …

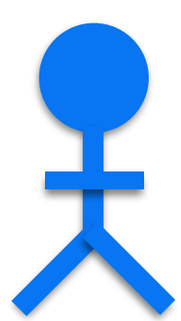
Borrow

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

Aliasing + ~~Mutation~~







Shared borrow (&T)

~~Aliasing~~ + Mutation

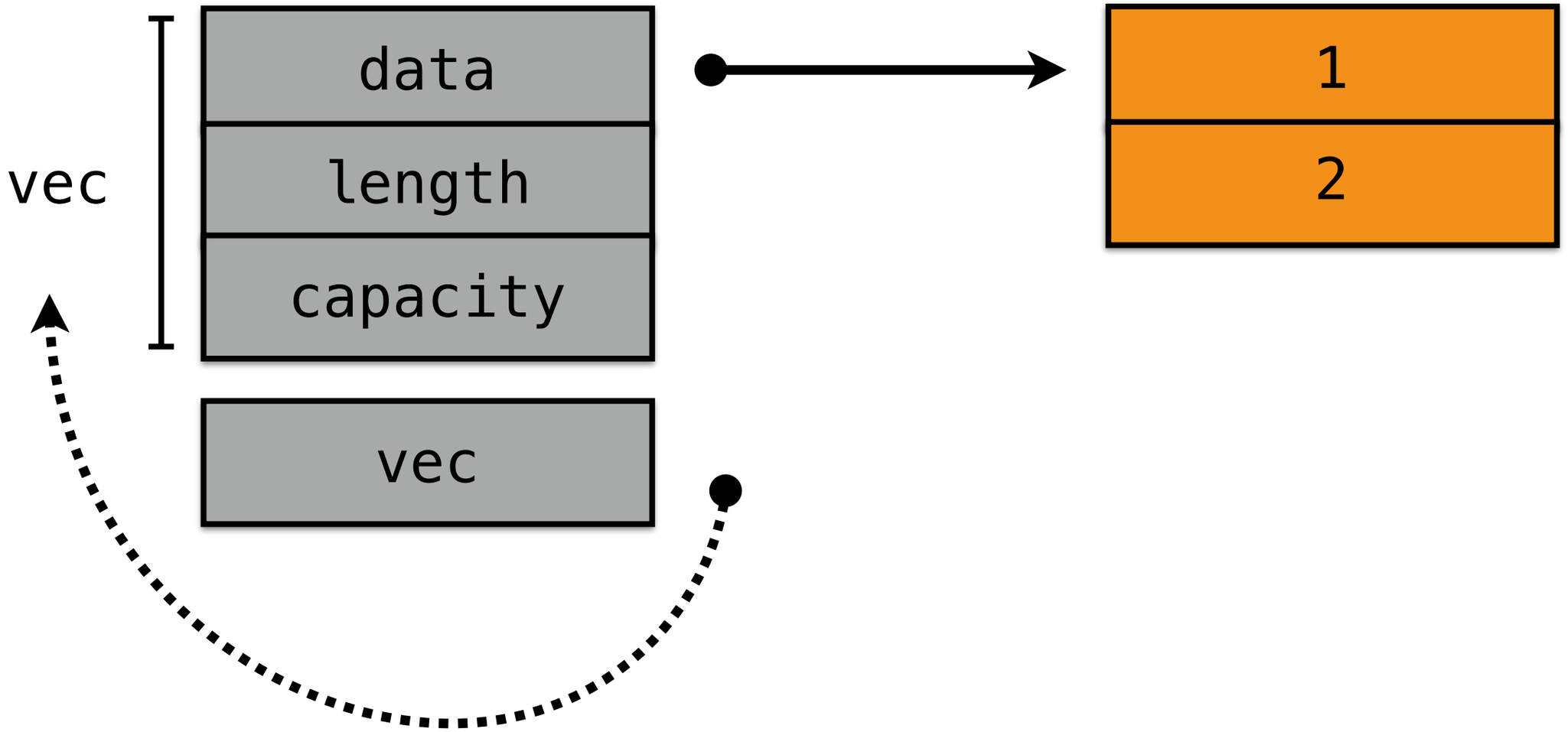
Mutable borrow (&mut T)

```
fn lender() {
  let mut vec = Vec::new();
  vec.push(1);
  vec.push(2);
  use(&vec);
  …
}
```

Loan out vec

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

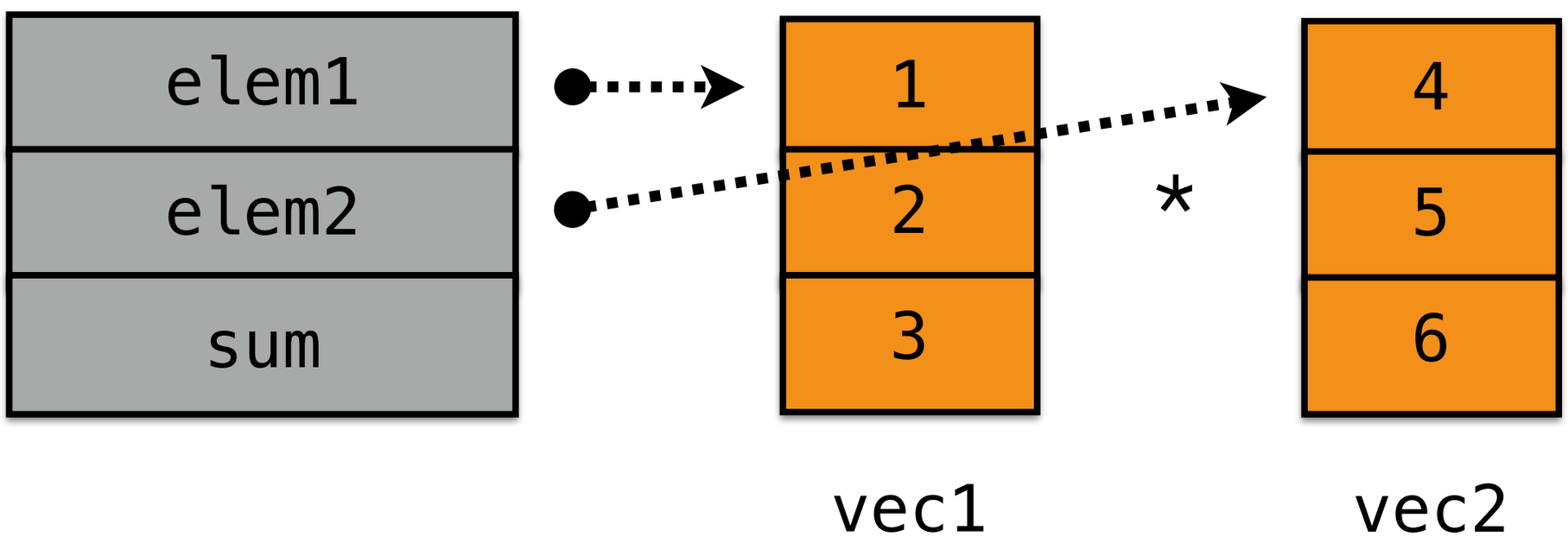
“Shared reference to Vec<int>”



```
fn dot_product(vec1: &Vec<int>, vec2: &Vec<int>)
                  -> int {
 let mut sum = 0;
 for (elem1, elem2) in vec1.iter().zip(vec2.iter()) {
  sum += (*elem1) * (*elem2);
 }
 return sum;
}
```

walk down matching indices

elem1, elem2 are references into the vector



Why “**shared**” reference?

```
fn dot_product(vec1: &Vec<int>, vec2: &Vec<int>)
                  -> int
{…}
```

```
fn magnitude(vec: &Vec<int>) -> int {
    sqrt(dot_product(vec, vec))
}
```

two shared references to the same vector — OK!

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

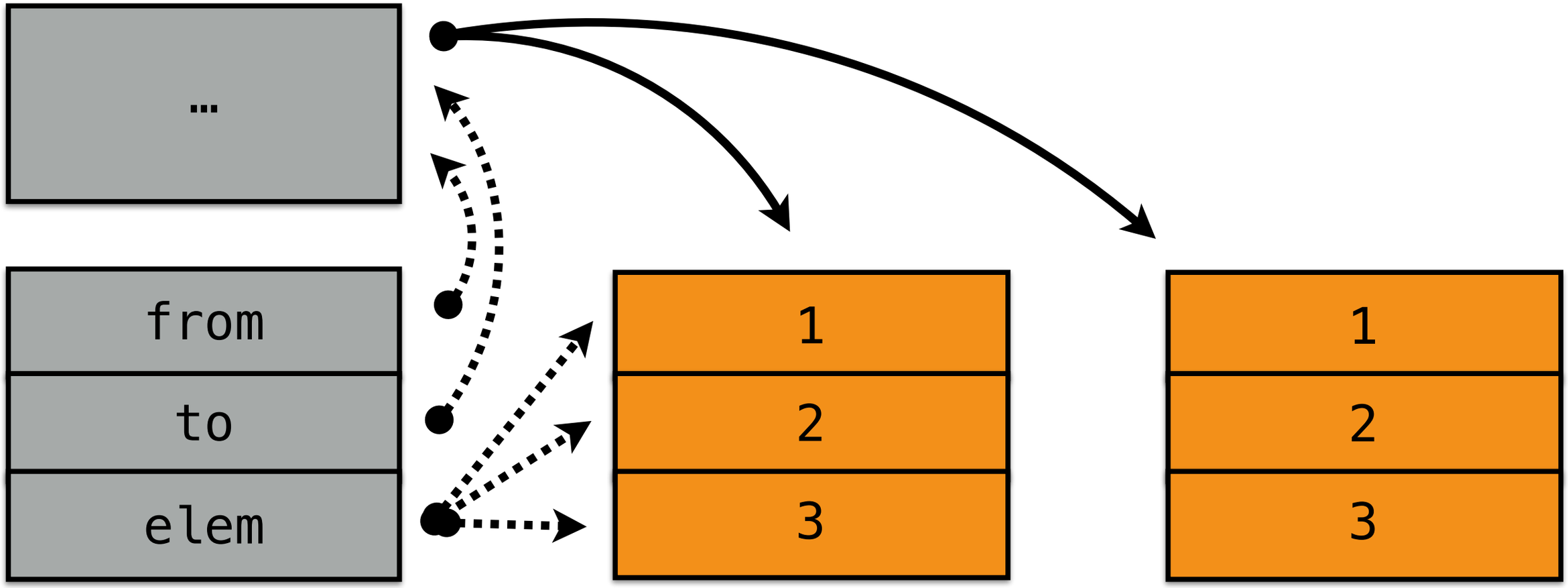
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {
   for elem in from.iter() {
    to.push(*elem);
   }
}
```

mutable reference to Vec<int>

push() is legal

Mutable references

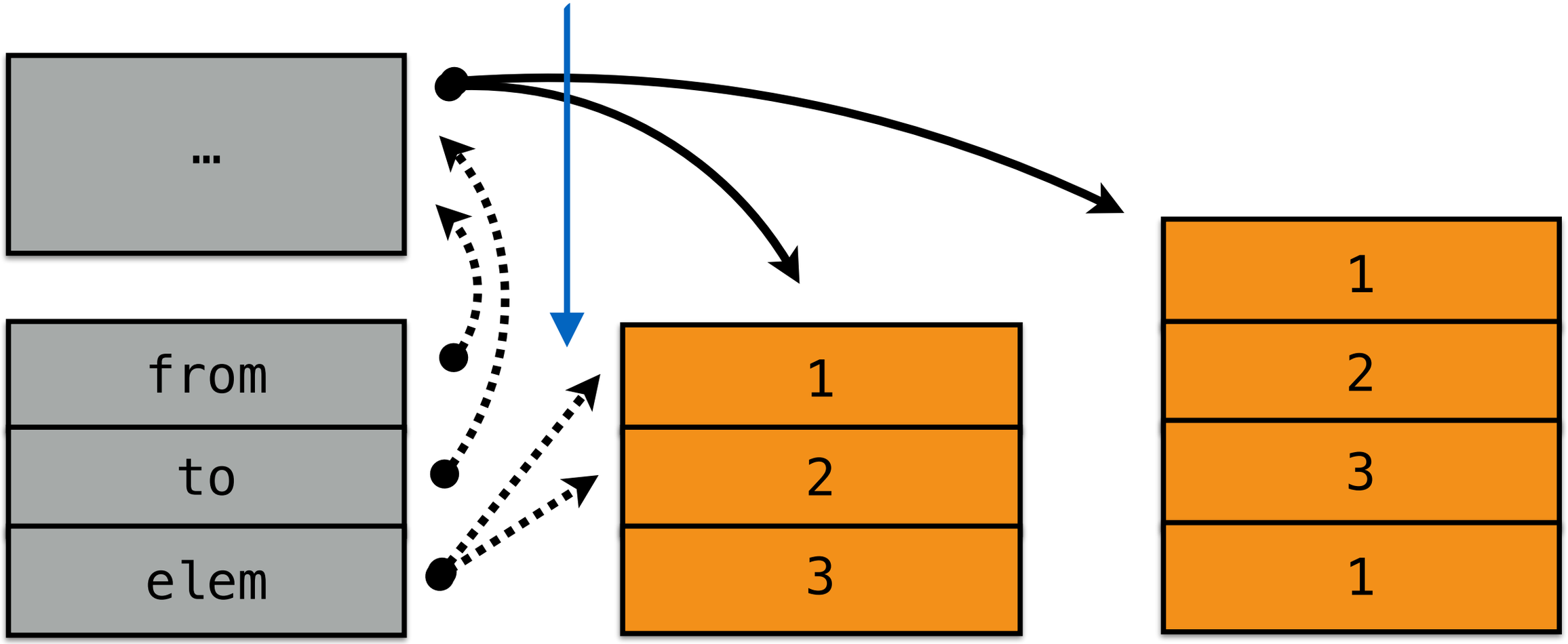
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {
   for elem in from.iter() {
    to.push(*elem);
   }
}
```



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

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

dangling pointer



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

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

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 range(0, vec.len()) {
        let elem: &int = &vec[i];
        …
        vec.push(…);
    }
    …
    vec.push(…);
}
```

Error: vec[i] is borrowed, cannot mutate

OK. loan expired.

Borrows restrict access to the original path for their duration.

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

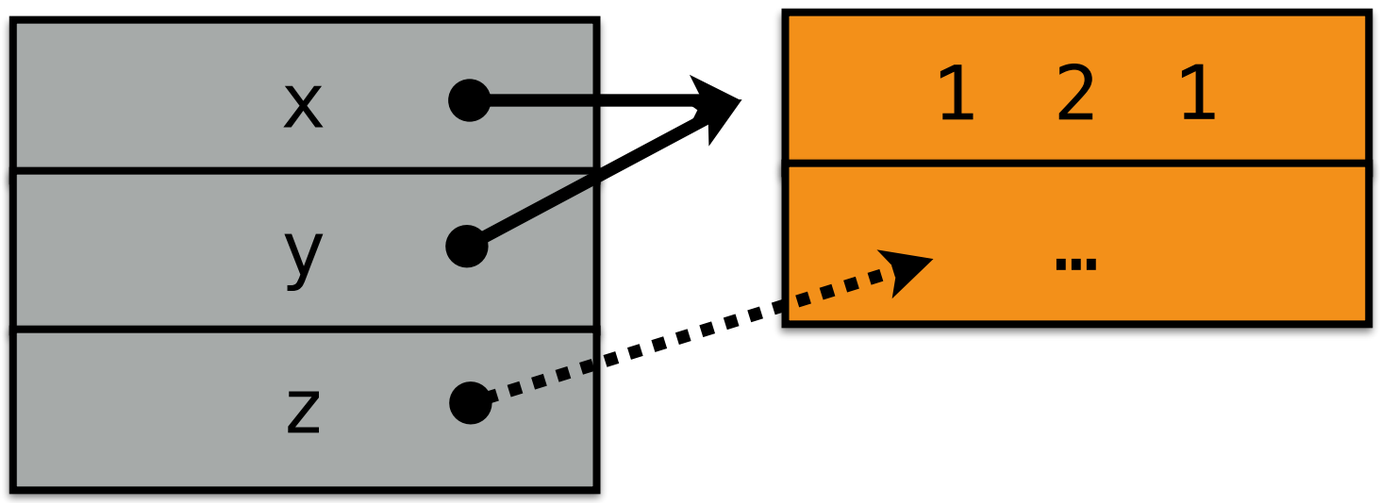
Abstraction

n. freedom from representational qualities in art.

Rust is an extensible language

- **Zero-cost** abstraction
- Rich libraries:
  - Containers
  - Memory management
  - Parallelism
  - ...
- Ownership and borrowing let libraries **enforce safety**.

```
fn example() {
  let x: Rc<T> = Rc::new(…);
  {
    let y = x.clone();
    let z = &*y;
    …
  } // runs destructor for y
} // runs destructor for x
```



Borrowing and Rc

```
fn deref<'a,T>(r: &'a Rc<T>) -> &'a T {
    …
}
```

Given a borrowed reference
to an `Rc<T>`…

…return a reference to the
`T` inside with **same extent**

New reference can be thought of as a kind of **sublease**.
Returned reference cannot outlast the original.

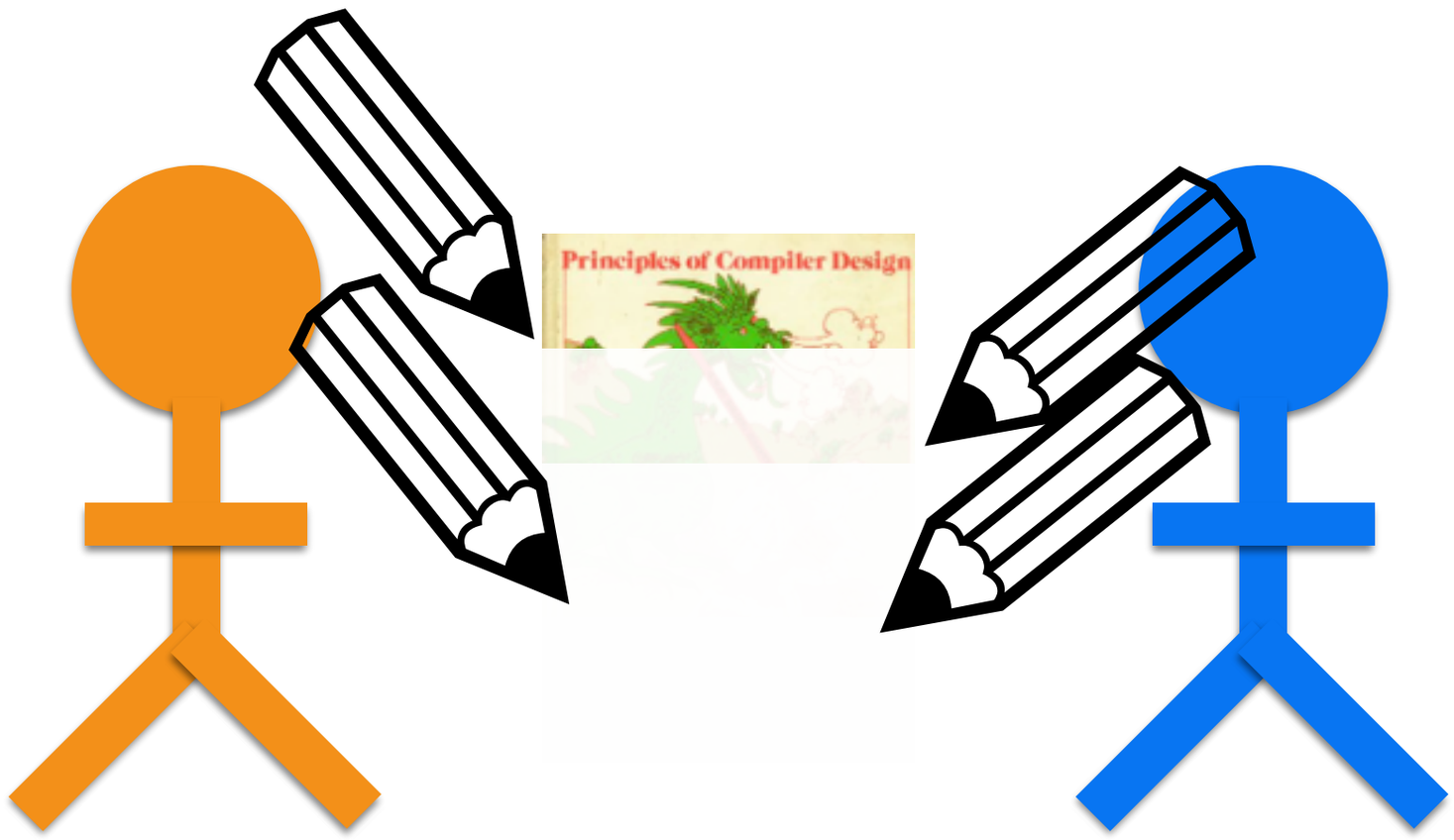
```
fn example() -> &T {
    let x: Rc<T> = Rc::new(…);
    return &*x;
} // runs destructor for x
```

Error: extent of borrow exceeds lifetime of `x`

Concurrency

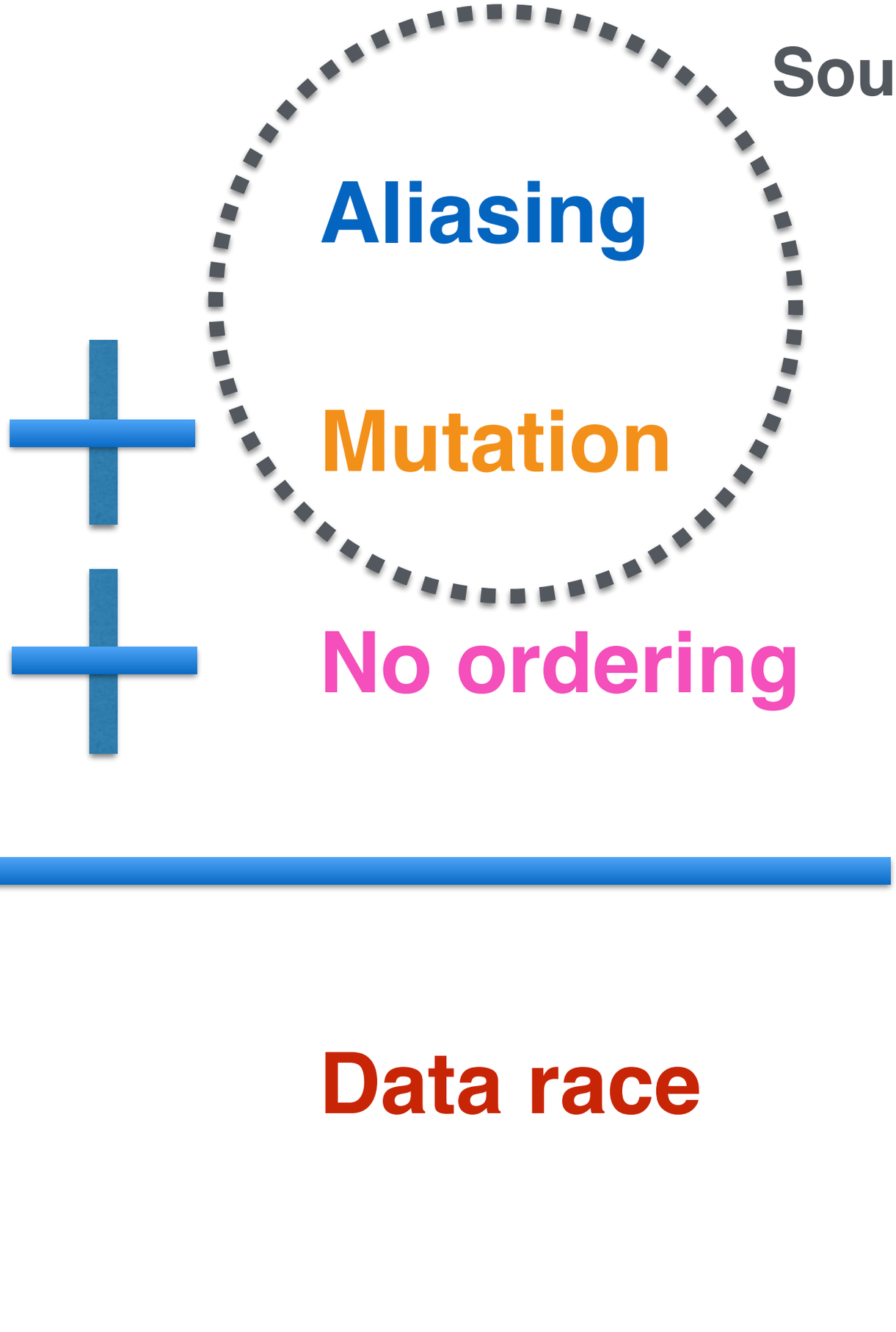
n. several computations executing simultaneously, and potentially interacting with each other

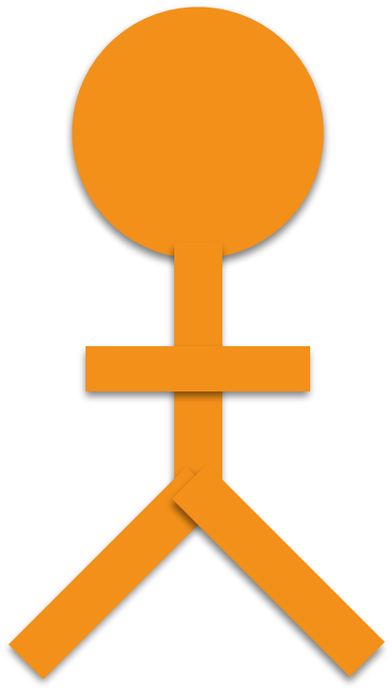
Data race

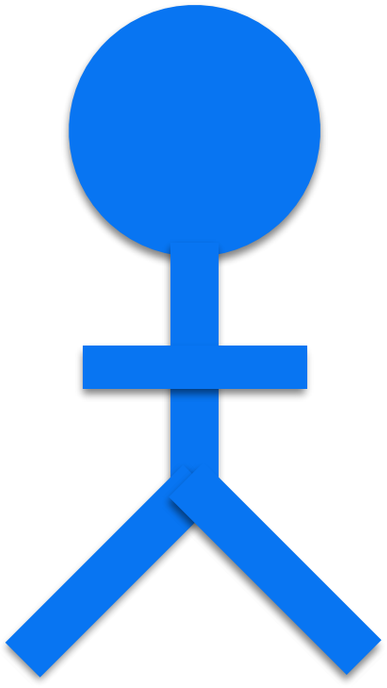


Two **unsynchronized** threads
accessing **same data**
where **at least one writes**.

Sound familiar?





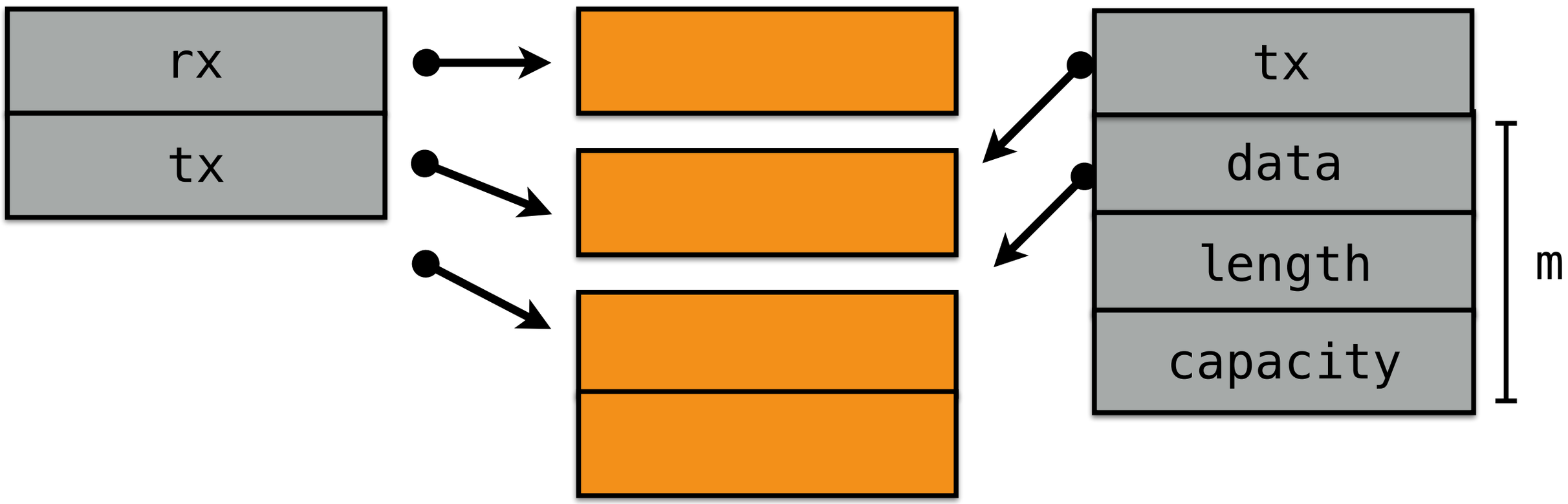


Messaging

(ownership)

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

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

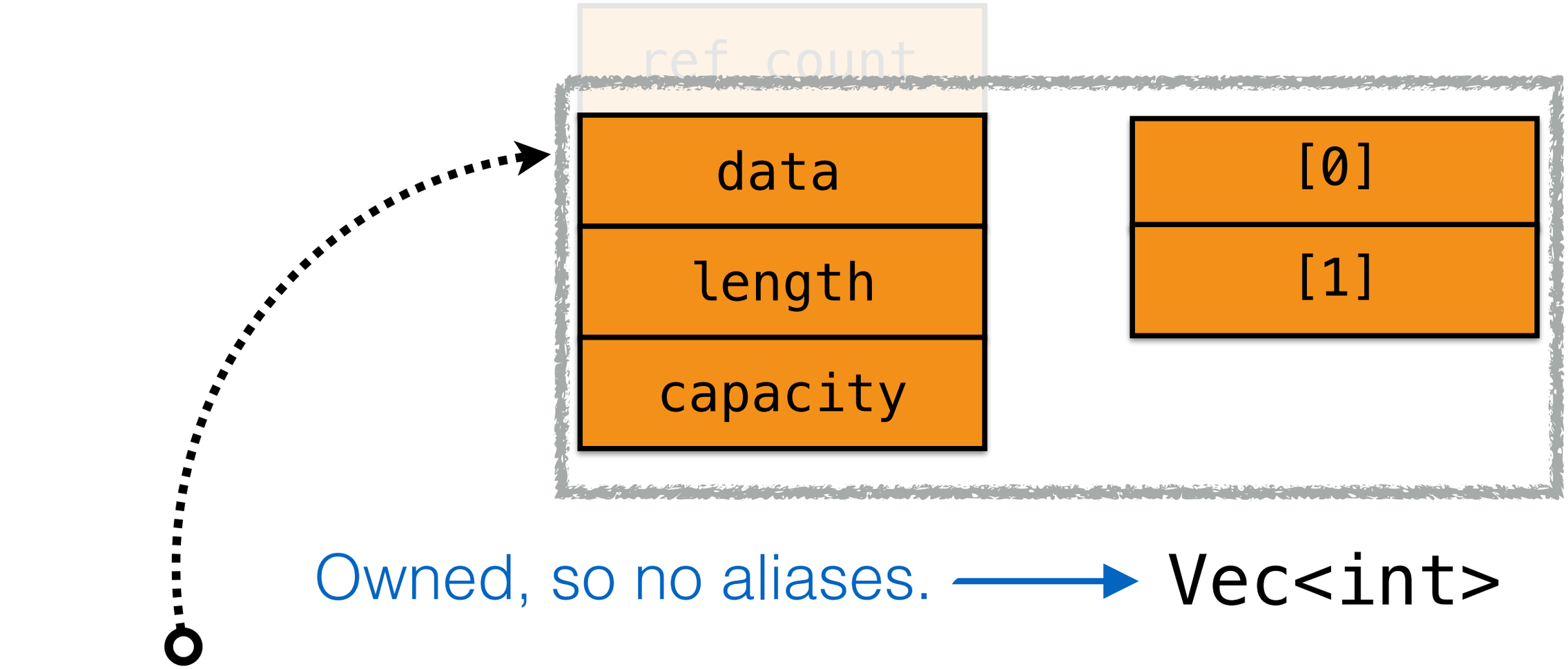


Shared read-only access

(ownership, borrowing)

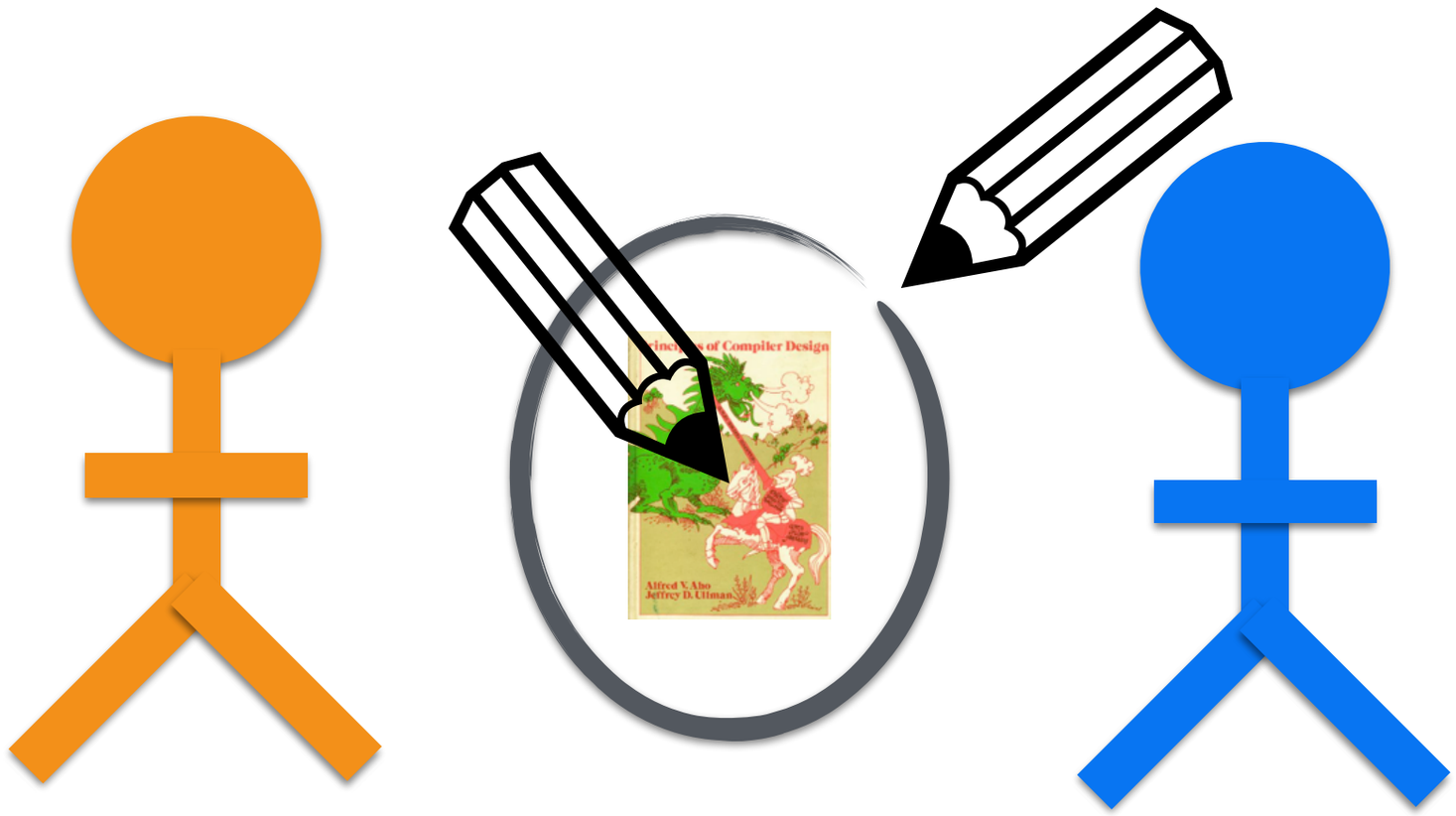
`Arc<Vec<int>>`

(ARC = Atomic Reference Count)



Owned, so no aliases. ⟶ `Vec<int>`

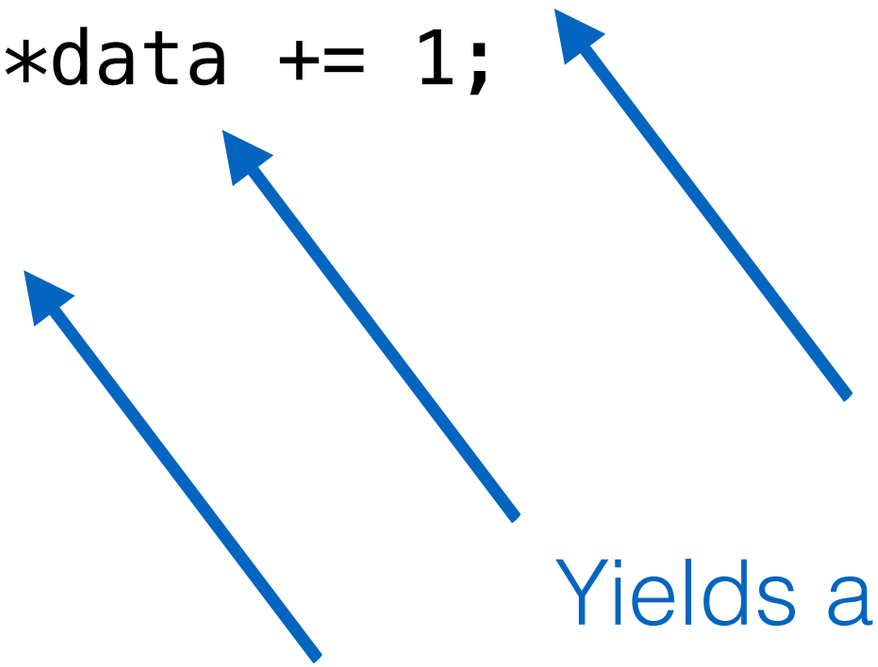
`&Vec<int>` ⟵ Shared reference, so Vec<int> is immutable.



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

And beyond…

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

Either already have or have plans for:

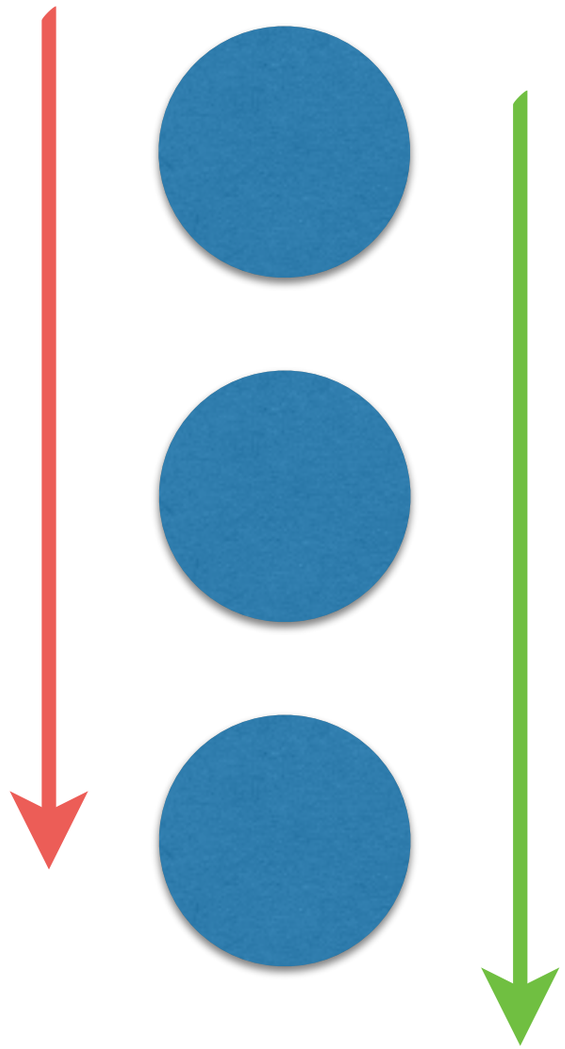
- Atomic primitives
- Non-blocking queues
- Concurrent hashtables
- Lightweight thread pools
- Futures
- CILK-style fork-join concurrency
- etc.

Always data-race free

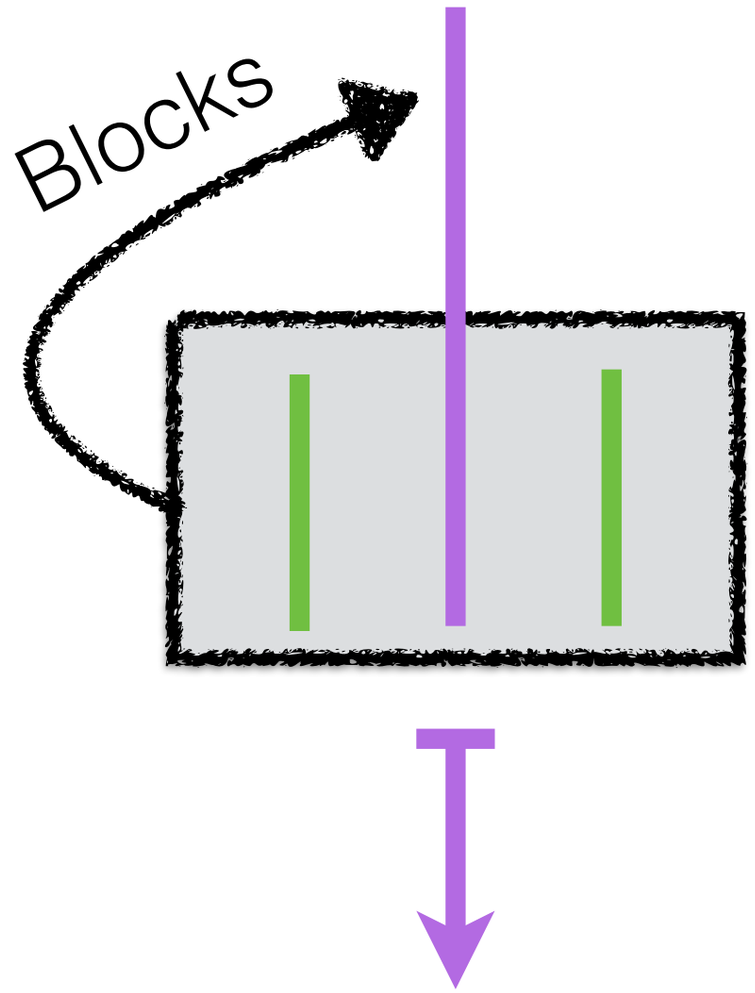
Parallel

adj. occurring or existing at the same time

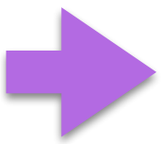
Concurrent vs parallel



Concurrent threads

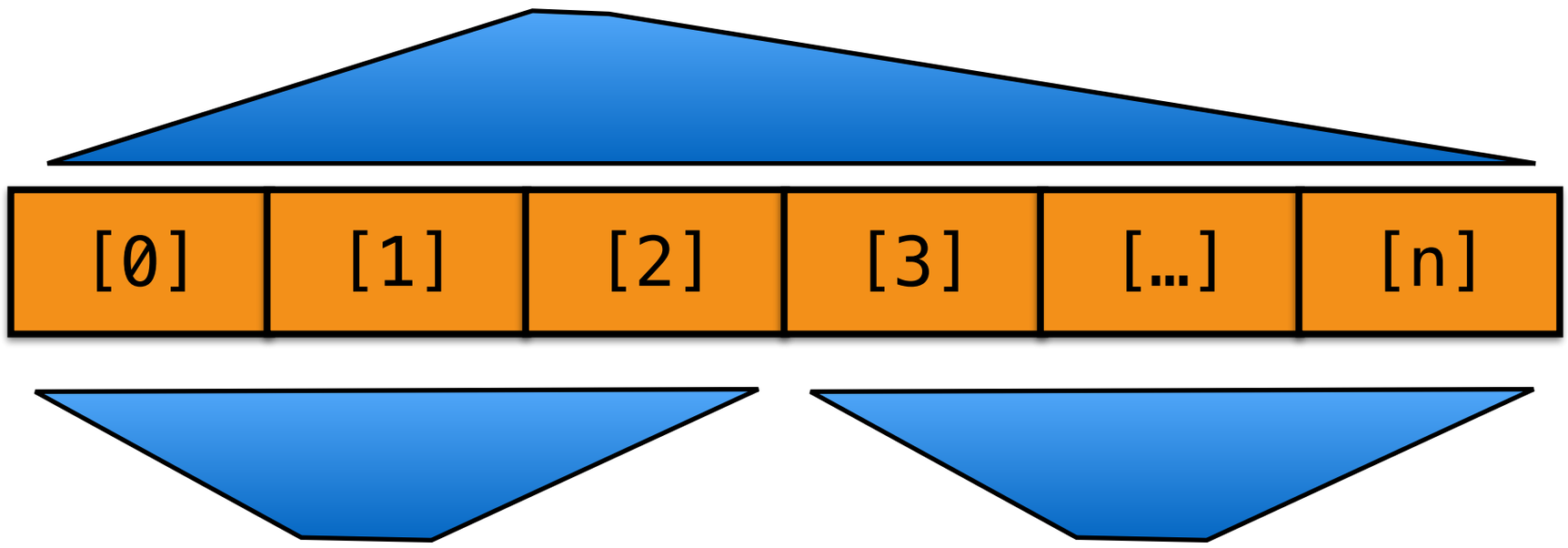


Parallel jobs

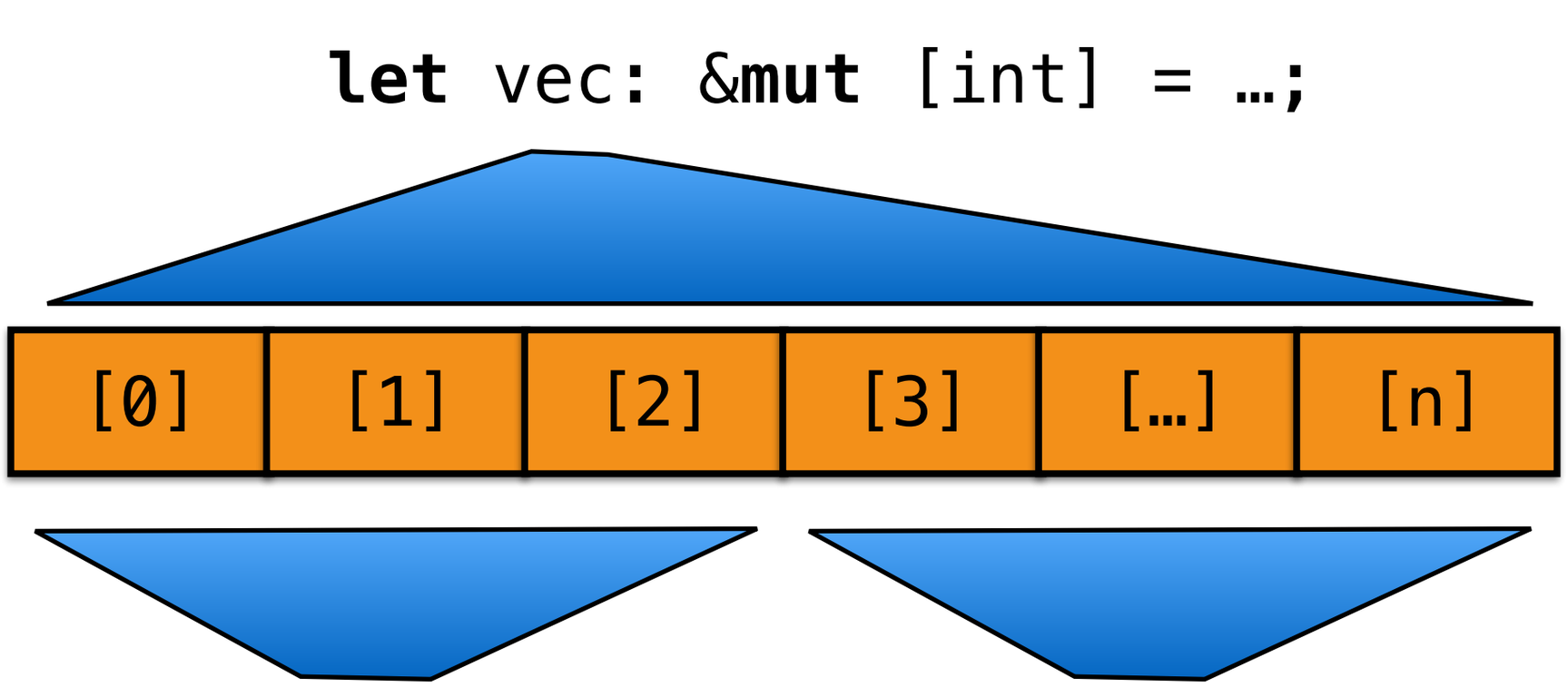


```
fn qsort(vec: &mut [int]) {
  let pivot = vec[random(vec.len())];
  let mid = vec.partition(vec, pivot);
  let (less, greater) = vec.mut_split_at(mid);
  qsort(less);
  qsort(greater);
}
```

```
let vec: &mut [int] = …;
```



```
fn parallel_qsort(vec: &mut [int]) {
  let pivot = vec[random(vec.len())];
  let mid = vec.partition(vec, pivot);
  let (less, greater) = vec.mut_split(mid);
  parallel::do(&[
    || parallel_qsort(less),
    || parallel_qsort(greater)
  ]);
}
```



Unsafe

adj. not safe; hazardous

Safe abstractions

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

Validates input, etc.

Trust me.

- Uninitialized memory
- Interfacing with C code
- Building parallel abstractions like ARC
- …

Status of Rust

“Rapidly stabilizing.”

Goal for 1.0:

- Stable syntax, core type system
- Minimal set of core libraries

Conclusions

- Rust gives **control** without compromising **safety**:
  - Zero-cost abstractions
  - Zero-cost safety

- Guarantees **beyond dangling pointers**:
  - Iterator invalidation in a broader sense
  - Data race freedom