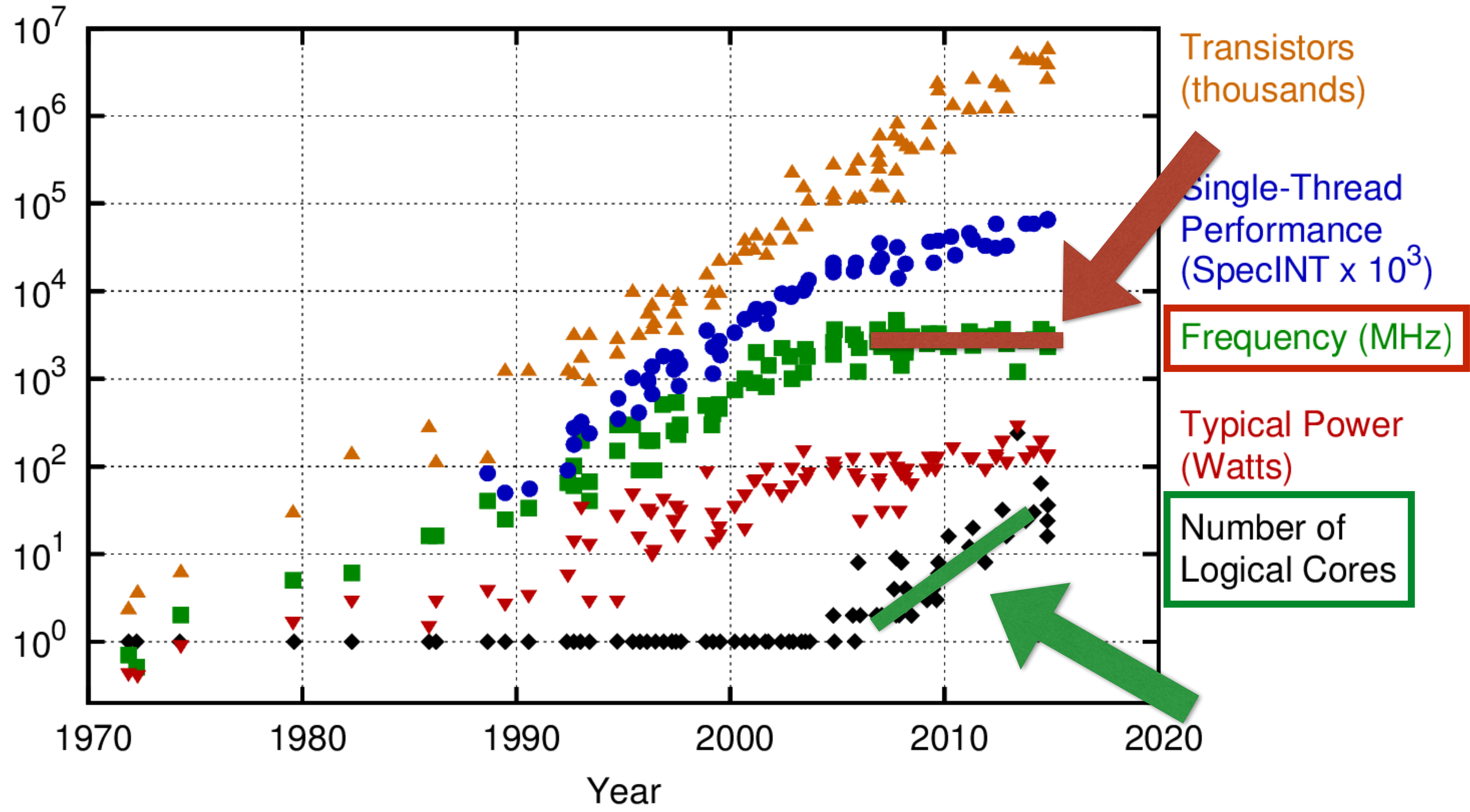




Concurrency in Rust

Alex Crichton

40 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Bug 631527

Parallelize selector matching



[Get help with this page](#)

NEW Assigned to [dzbarsky](#)

▼ **Status** (NEW bug with no priority)

Product: ▶ Core

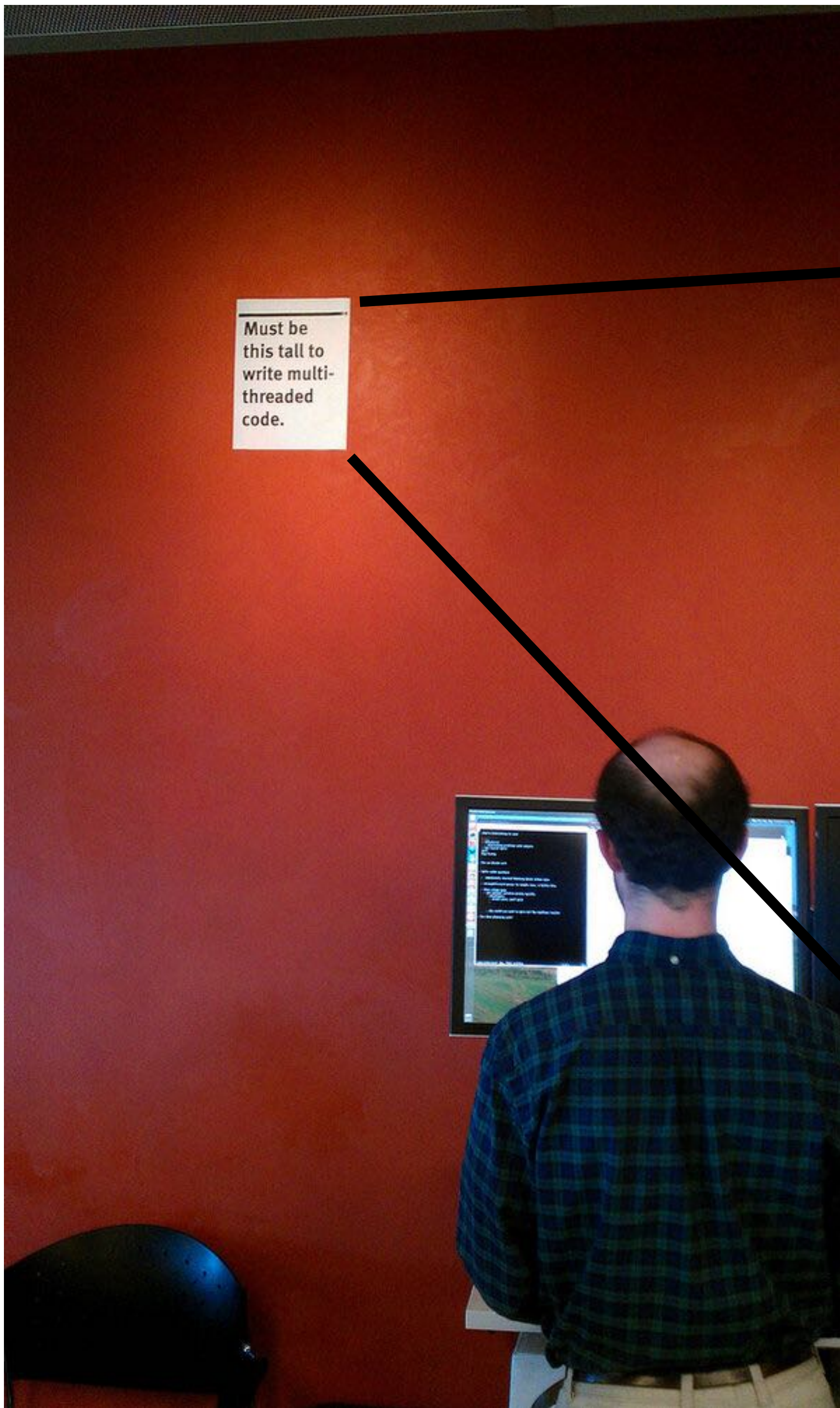
Component: ▶ CSS Parsing and Computation

Status: NEW

Reported: 7 years ago

Reported: 7 years ago

Modified: 3 months ago



Must be
this tall to
write multi-
threaded
code.

Must be
this tall to
write multi-
threaded
code.



Fearless Concurrency with Rust

Apr 10, 2015 • Aaron Turon

The Rust project was initiated to solve two thorny problems:

- How do you do safe systems programming?
- How do you make concurrency painless?

Initially these problems seemed orthogonal, but to our amazement, the solution turned out to be identical: **the same tools that make Rust safe also help you tackle concurrency head-on.**

Memory safety bugs and concurrency bugs often come down to code accessing data when it shouldn't. Rust's secret weapon is *ownership*, a discipline for access control that systems programmers try to follow but that Rust's compiler checks statically for you.

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

Concurrency?

Rust?

Libraries

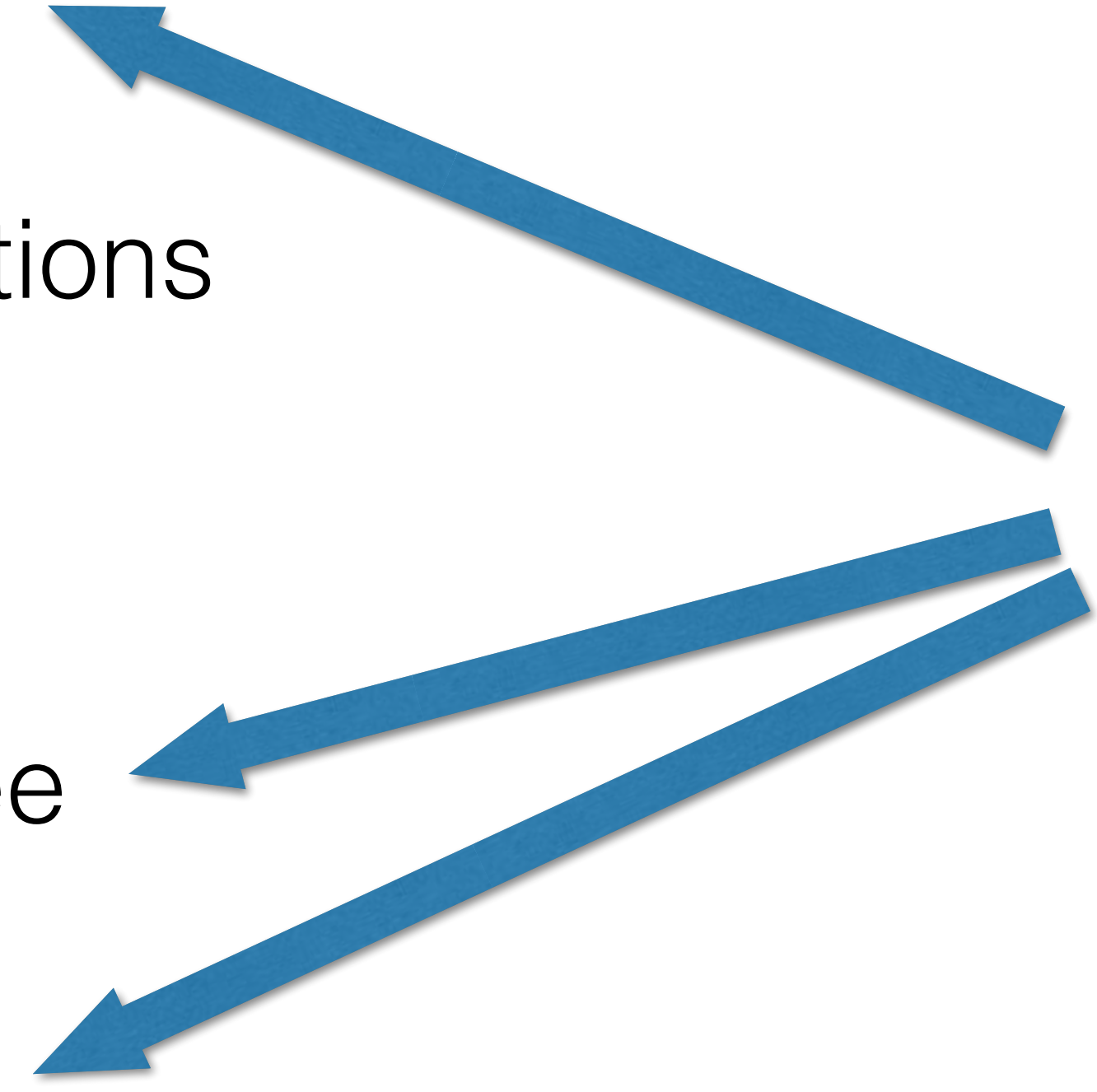
What's concurrency?

In computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

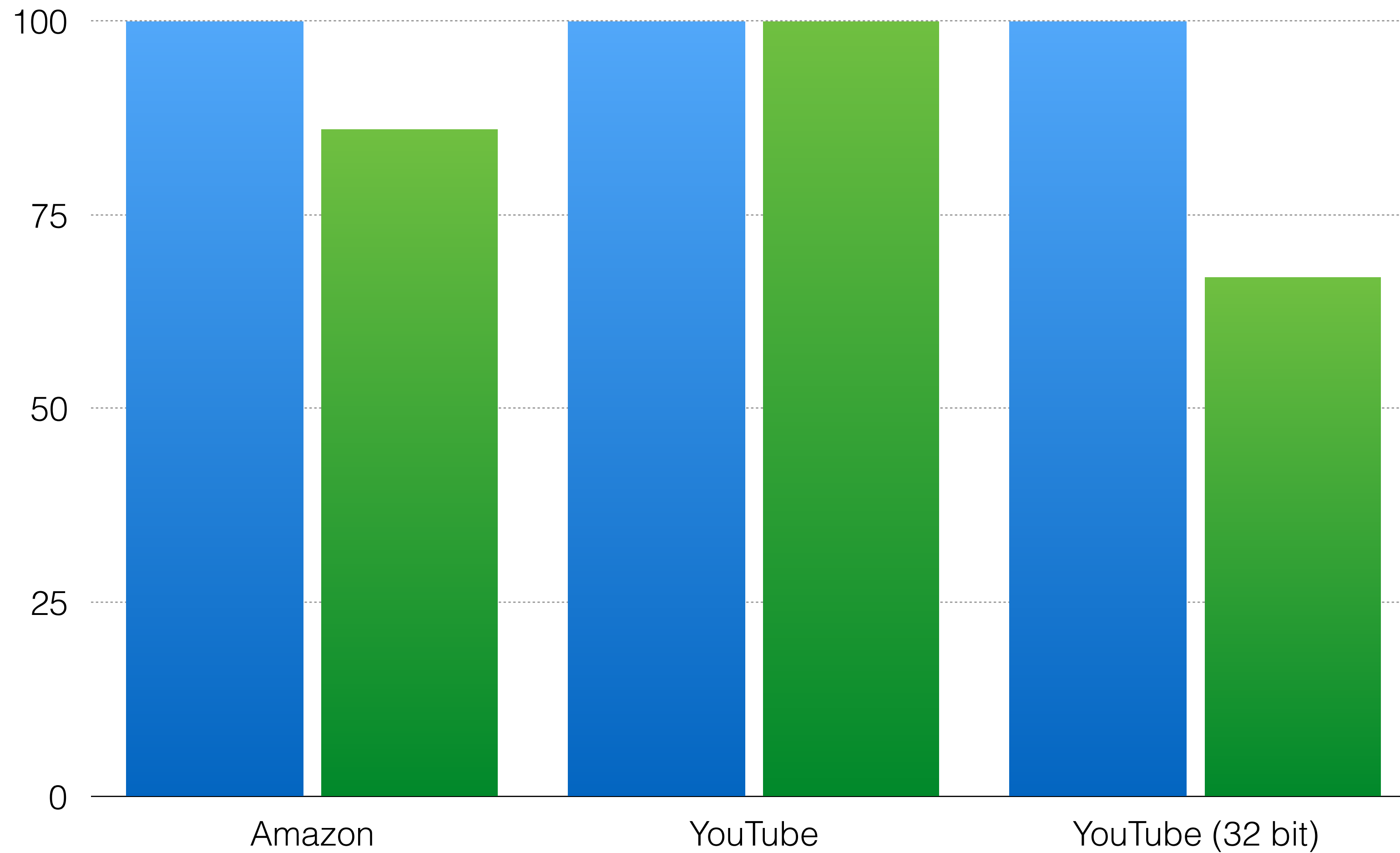
Getting our feet wet

```
// What does this print?  
int main() {  
    int pid = fork();  
    printf("%d\n", pid);  
}
```

Concurrency is hard!

- Data Races
 - Race Conditions
 - Deadlocks
 - Use after free
 - Double free
- Exploitable!
- 
- A diagram consisting of three blue arrows pointing from the word 'Exploitable!' on the right towards the first three items in the list: 'Data Races', 'Race Conditions', and 'Deadlocks'. The arrows are thick and have a slight shadow.

Concurrency is nice!



Concurrency?

Rust?

Libraries

Zero-cost abstractions

+

Memory safety & data-race freedom

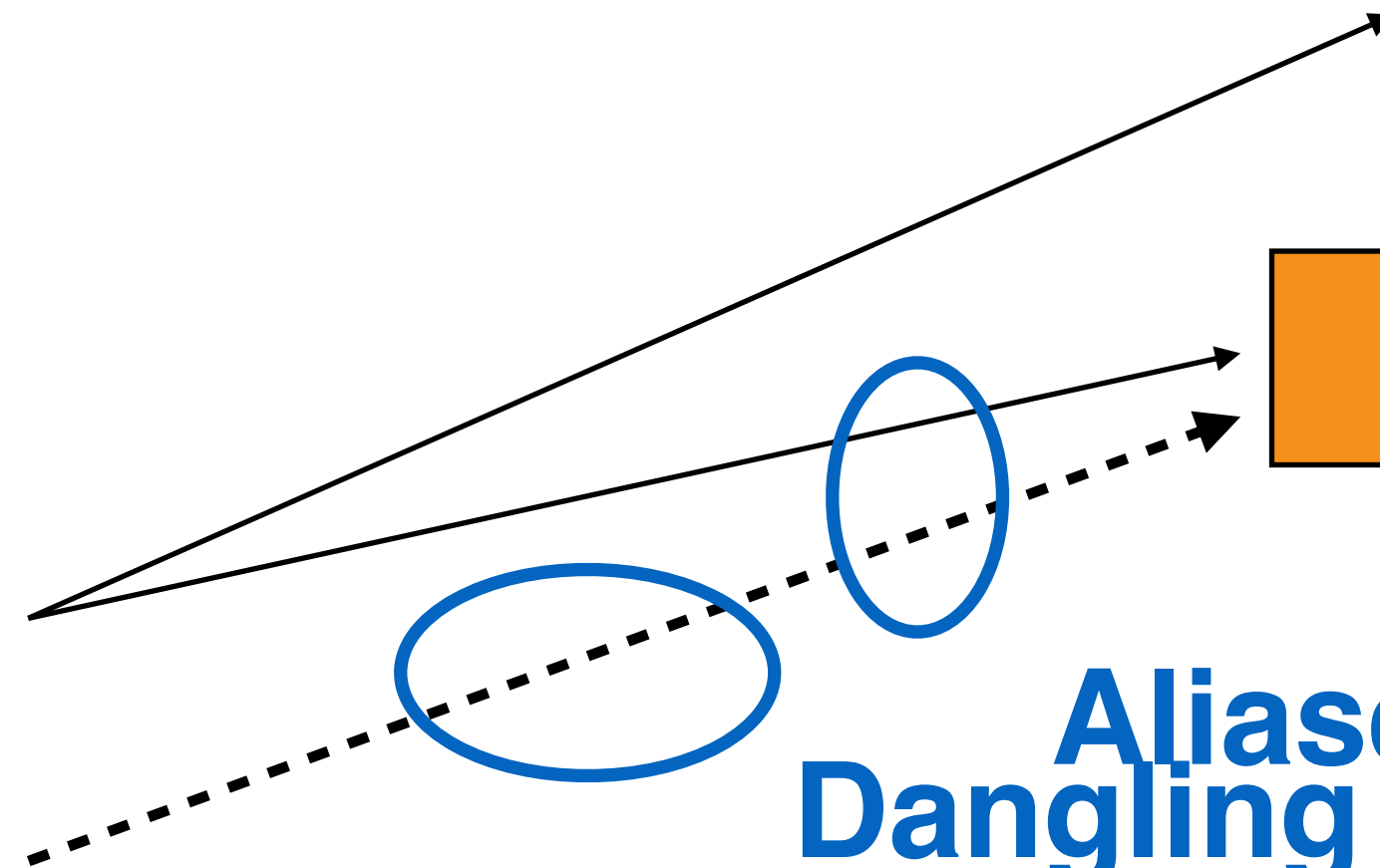
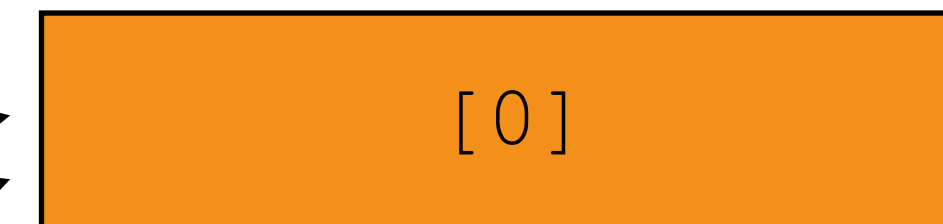
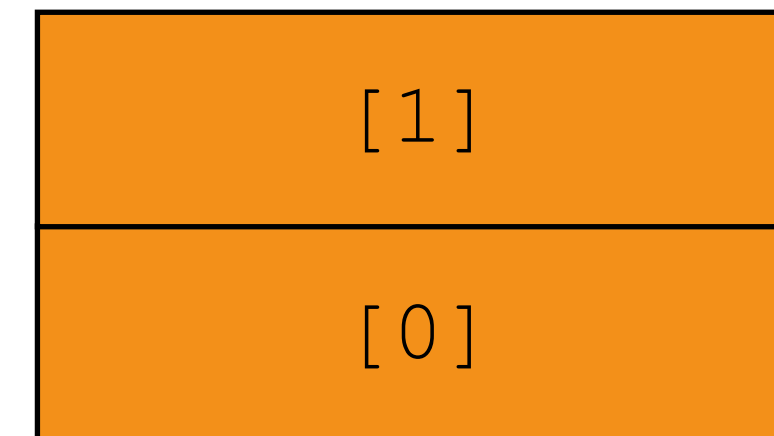
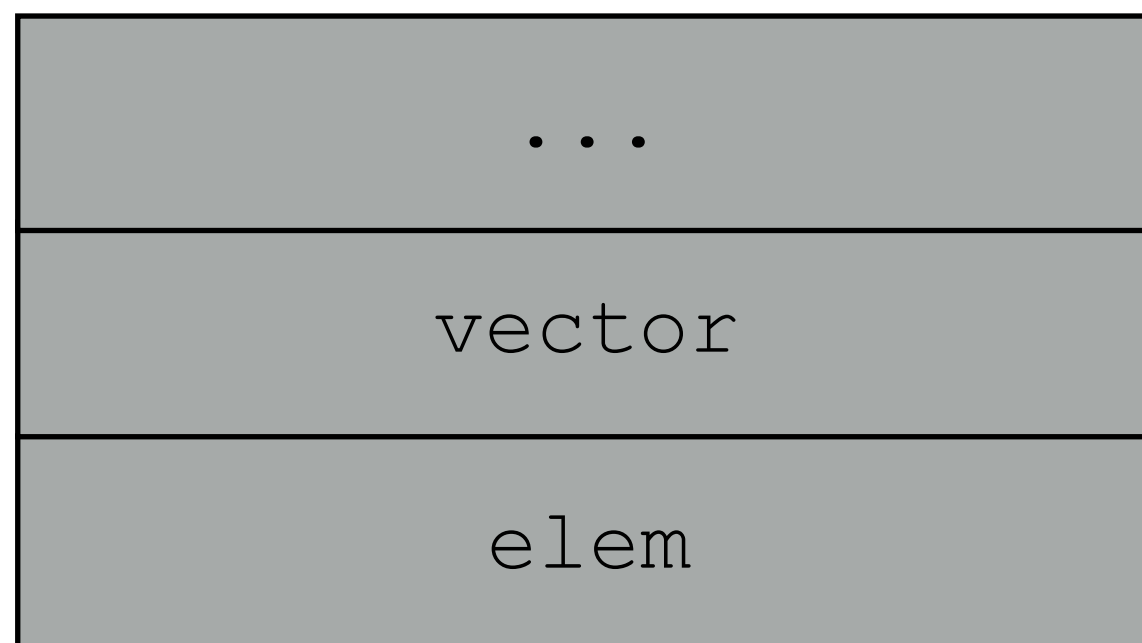
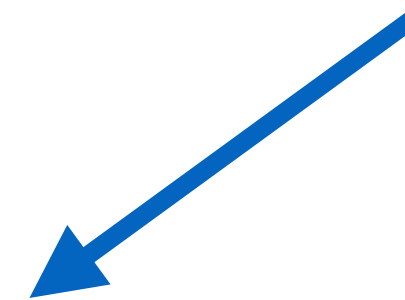
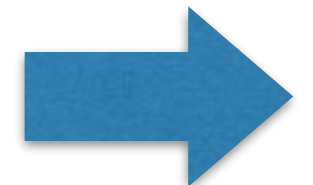
=

Confident, productive systems programming

What's **safety**?

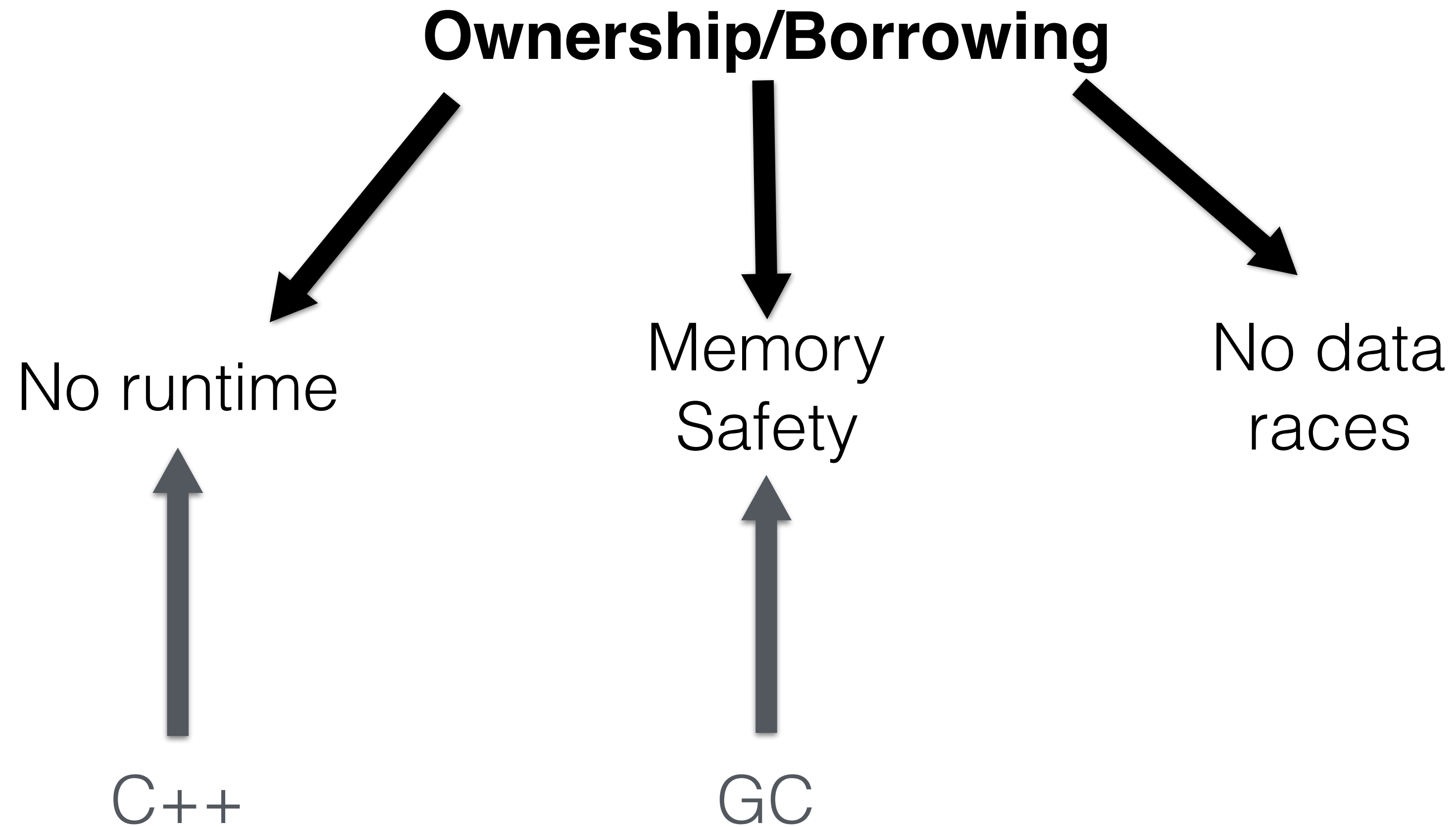
```
void example() {  
    vector<string> vector;  
    // ...  
    auto& elem = vector[0];  
    vector.push_back(some_string);  
    cout << elem;  
}
```

Mutation



**Aliased pointers
Dangling pointer!**

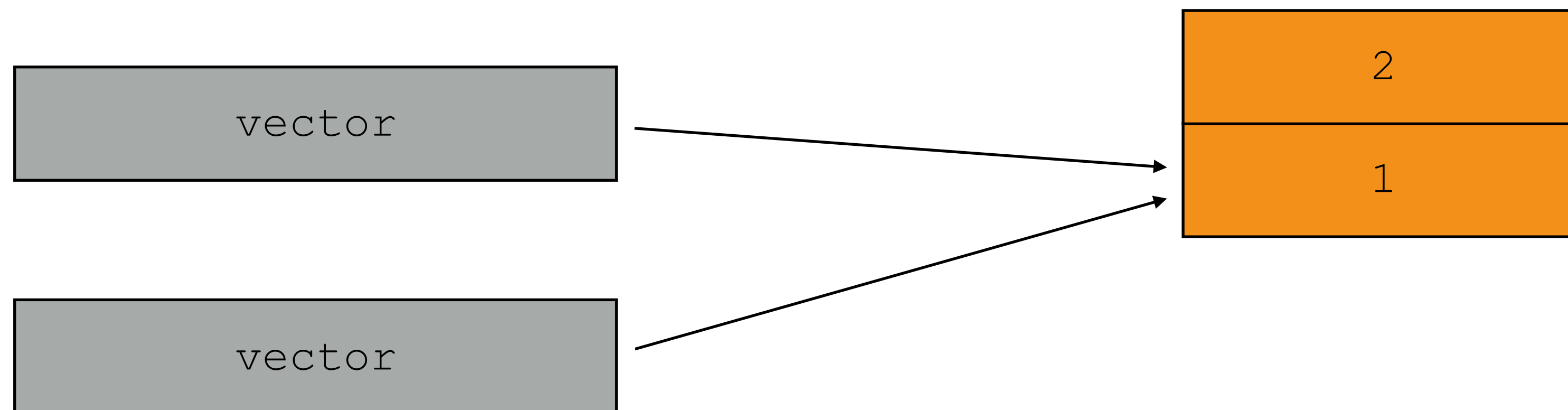
Rust's Solution



Ownership

```
fn main() {  
    let mut v = Vec::new();  
    v.push(1);  
    v.push(2);  
    take(v);  
    // ...  
}
```

```
fn take(v: Vec<i32>) {  
    // ...  
}
```



Ownership

```
fn main() {  
    let mut v = Vec::new();  
    v.push(1);  
    v.push(2);  
    take(v);  
    // ...  
}  
  
fn take(v: Vec<i32>) {  
    // ...  
}
```

Ownership

```
fn main() {  
    let mut v = Vec::new();  
    v.push(1);  
    v.push(2);  
    take(v);  
    v.push(3);  
}  
  
fn take(v: Vec<i32>) {  
    // ...  
}
```

Ownership

```
fn main() {  
    let mut v = Vec::new();  
    v.push(1);  
    v.push(2);  
    take(v);  
    v.push(3);  
}  
  
fn take(v: Vec<i32>) {  
    // ...  
}
```

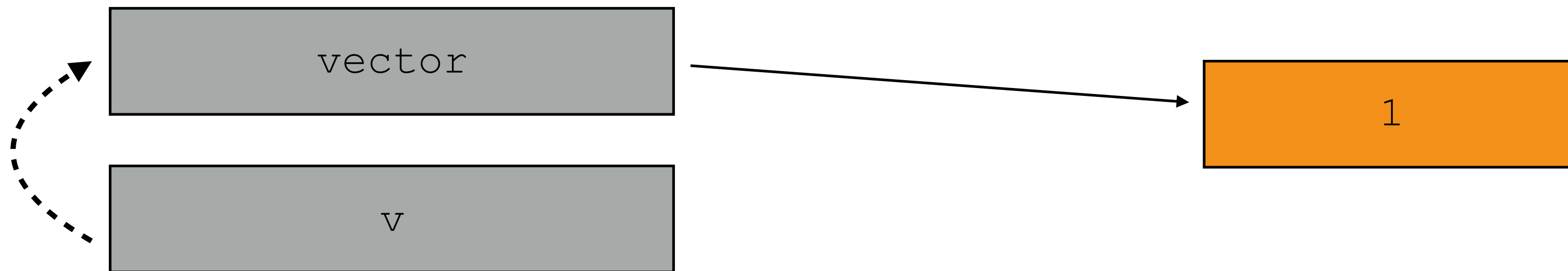
error: use of moved value `v`



Borrowing

```
fn main() {  
    let mut v = Vec::new();  
    push(&mut v);  
    read(&v);  
    // ...  
}
```

```
fn read(v: &mut Vec<u32>) {  
    v.push(1);  
}
```



Safety in Rust

- Rust *statically* prevents aliasing + mutation
- Ownership prevents double-free
- Borrowing prevents use-after-free
- Overall, no segfaults!

Concurrency?

Rust?

Libraries

Library-based concurrency

Originally: Rust had message passing built into the language.

Now: library-based, multi-paradigm.

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

std::thread

```
let loc = thread::spawn(|| {  
    "world"  
});  
println!("Hello, {}!",  
        loc.join().unwrap());
```


std::thread

```
let mut dst = Vec::new();  
thread::spawn(move || {  
    dst.push(3);  
});
```

```
dst.push(4);
```

error: use after move



std::thread

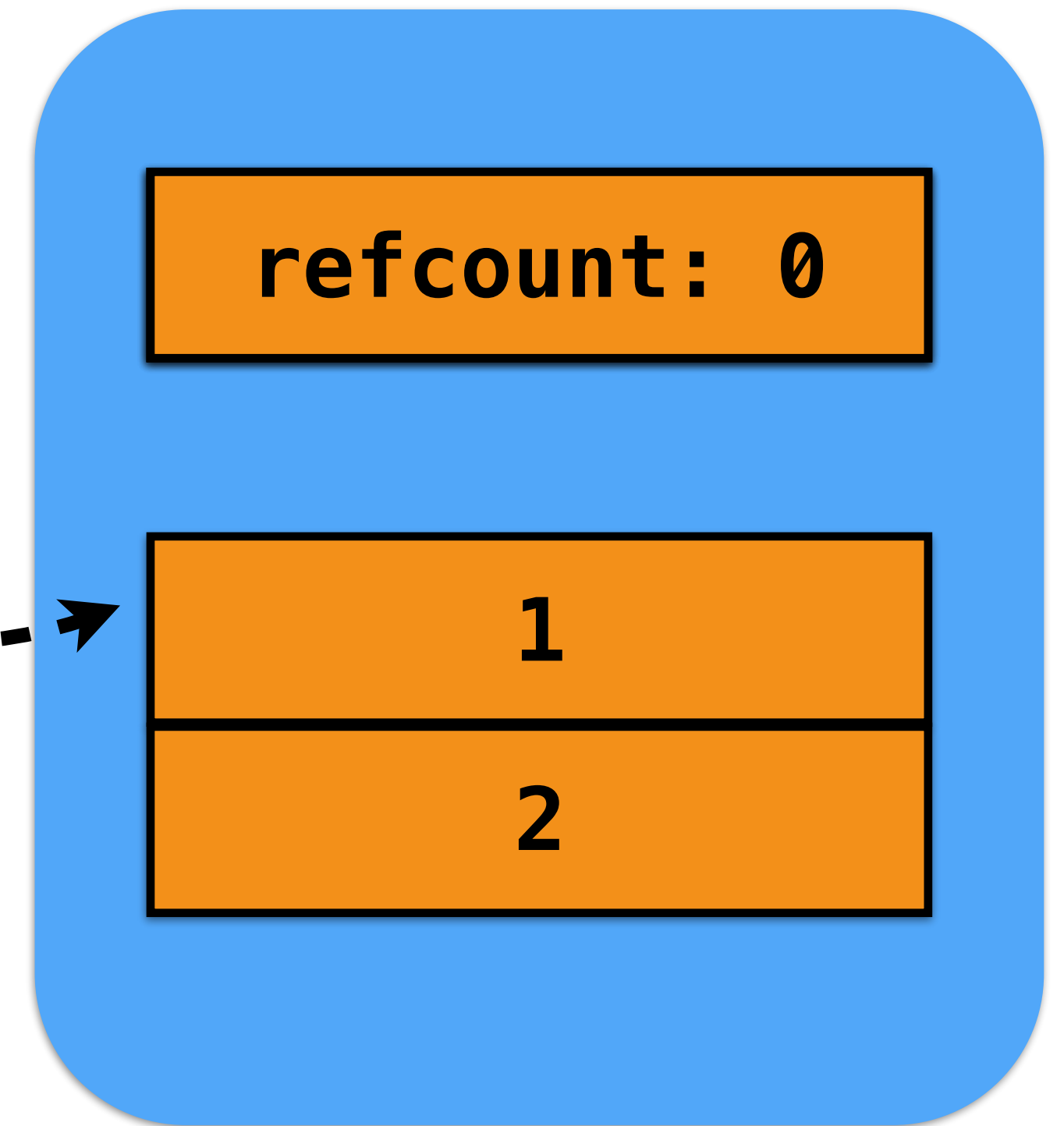
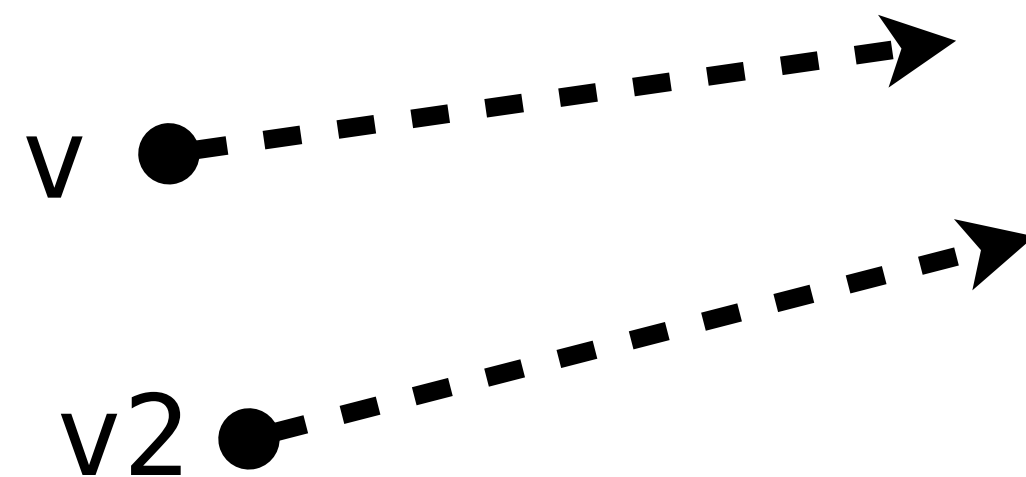
```
let mut dst = Vec::new();  
thread::spawn(|| {  
    dst.push(3);  
});  
dst.push(4);
```

error: value doesn't live long enough



std::sync::Arc

```
let v = Arc::new(vec![1, 2]);  
let v2 = v.clone();  
thread::spawn(move || {  
    println!("{}", v.len());  
});  
another_function(&v2);
```



std::sync::Arc

```
let v = Arc::new(vec![1, 2]);  
let v2 = v.clone();  
thread::spawn(move || {  
    v.push(3);  
});  
another_function(&v2);
```

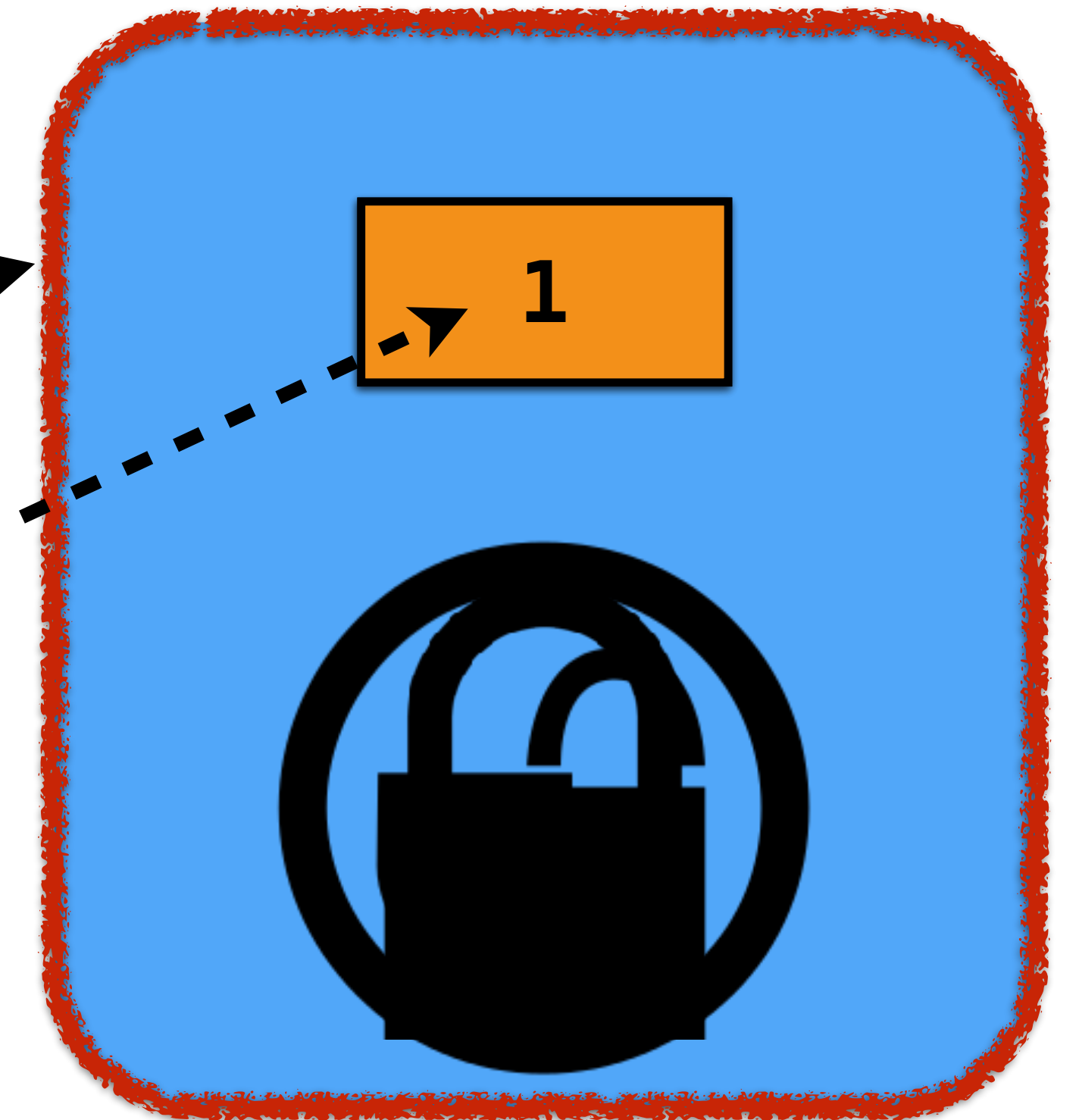
error: cannot mutate through shared reference

std::sync::Mutex

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

counter

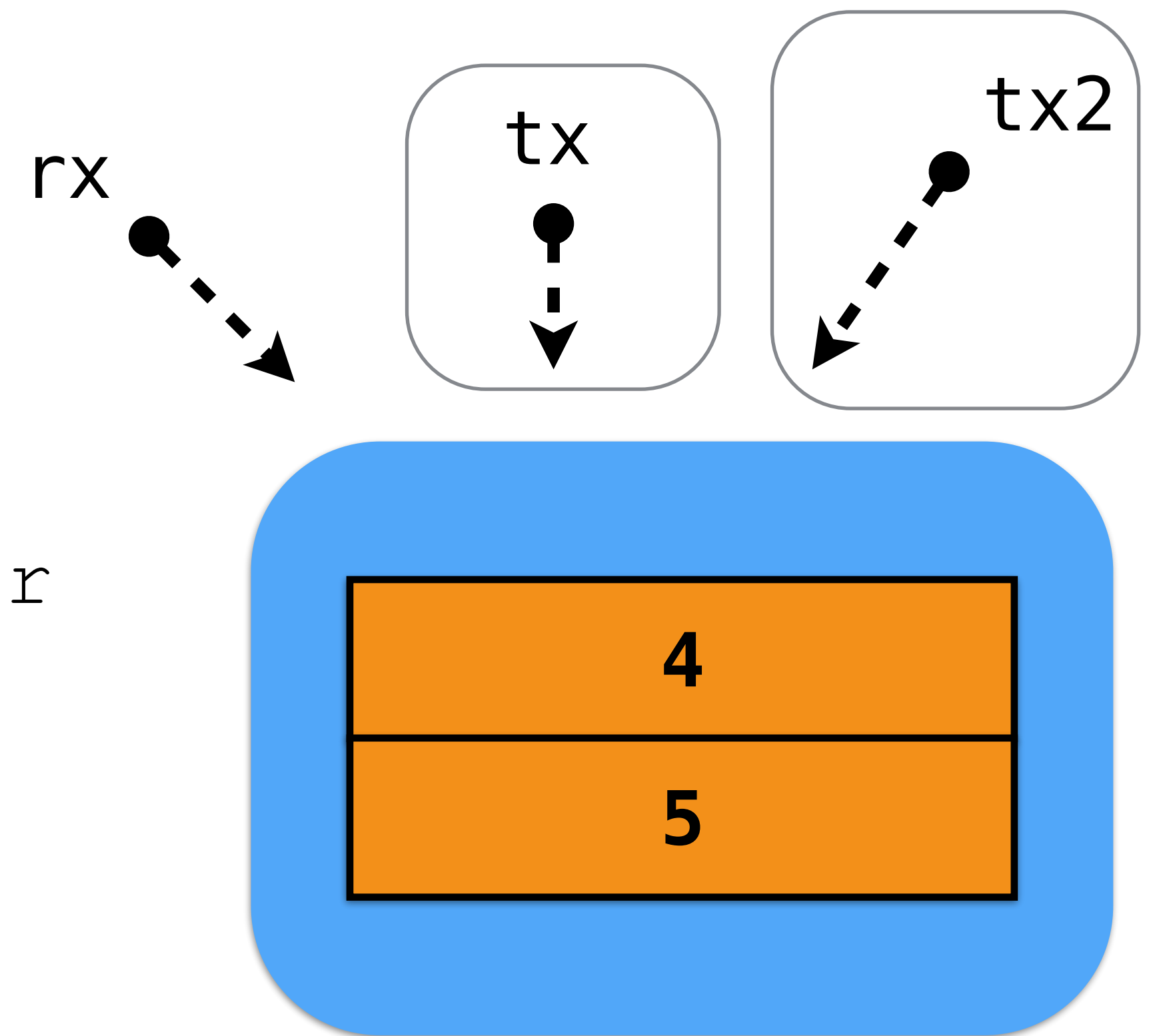
data



std::sync::mpsc

```
let (tx, rx) = mpsc::channel();  
let tx2 = tx.clone();  
thread::spawn(move || tx.send(5));  
thread::spawn(move || tx2.send(4));
```

```
// Prints 4 and 5 in an unspecified order  
println!("{:?}", rx.recv());  
println!("{:?}", rx.recv());
```



rayon

```
fn sum_of_squares(input: &[i32]) -> i32 {  
    input.iter()  
        .map(|&i| i * i)  
        .sum()  
}
```

rayon

```
use rayon::prelude::*;  
  
fn sum_of_squares(input: &[i32]) -> i32 {  
    input.par_iter()  
        .map(|&i| i * i)  
        .sum()  
}
```


rayon

```
use rayon::prelude::*;
```

```
fn sum_of_squares(input: &[i32]) -> i32 {  
    let mut cnt = 0;  
    input.par_iter()  
        .map(|&i| {  
            cnt += 1;  
            i * i  
        })  
        .sum()  
}
```

error: `cnt` cannot be shared concurrently



100% Safe

- Everything you just saw is foolproof
- No segfaults
- No data races
- No double frees...



doc.rust-lang.org/stable/book

users.rust-lang.org