Concurrency in Rust

Alex Crichton

What’s Rust?

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

Concurrency?

Rust?

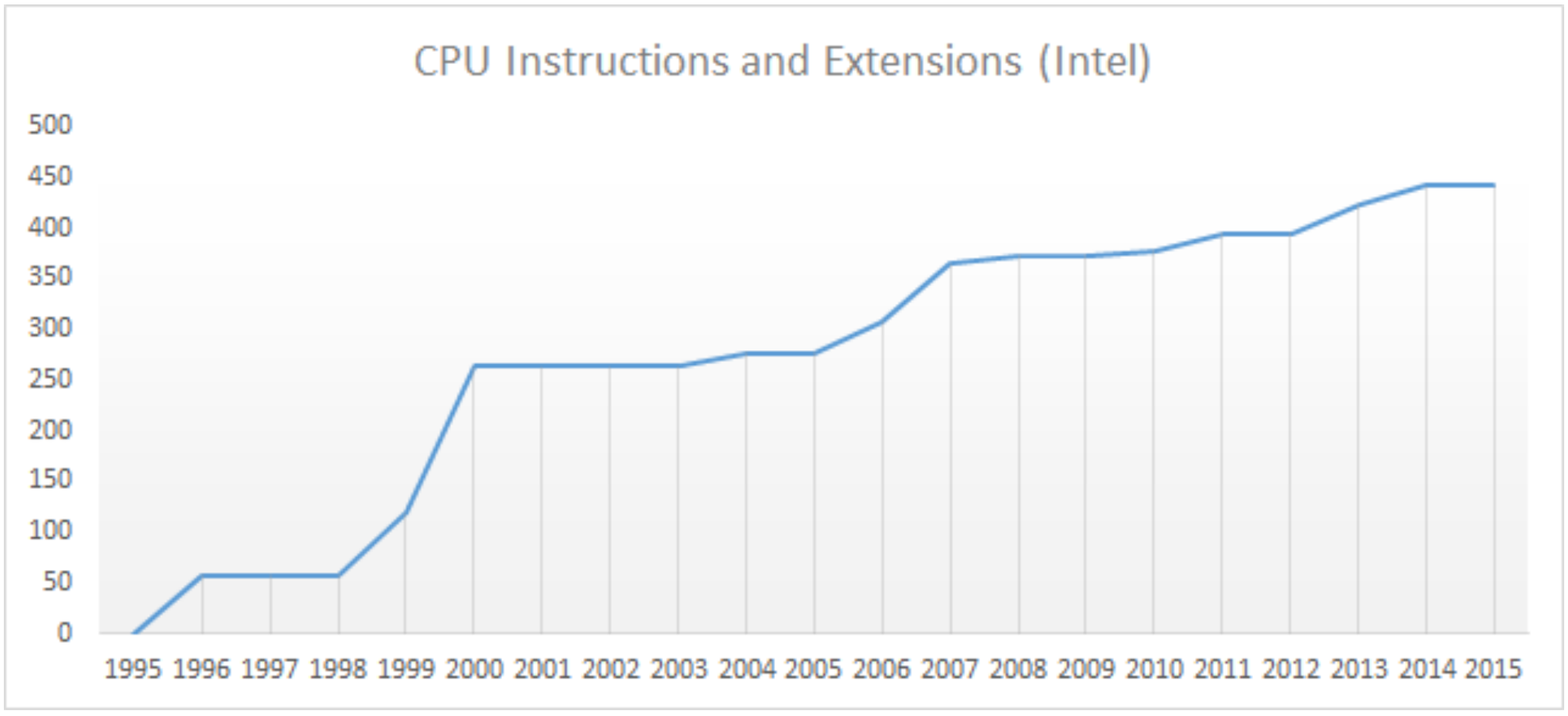
Libraries

Futures

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.

Why concurrency?

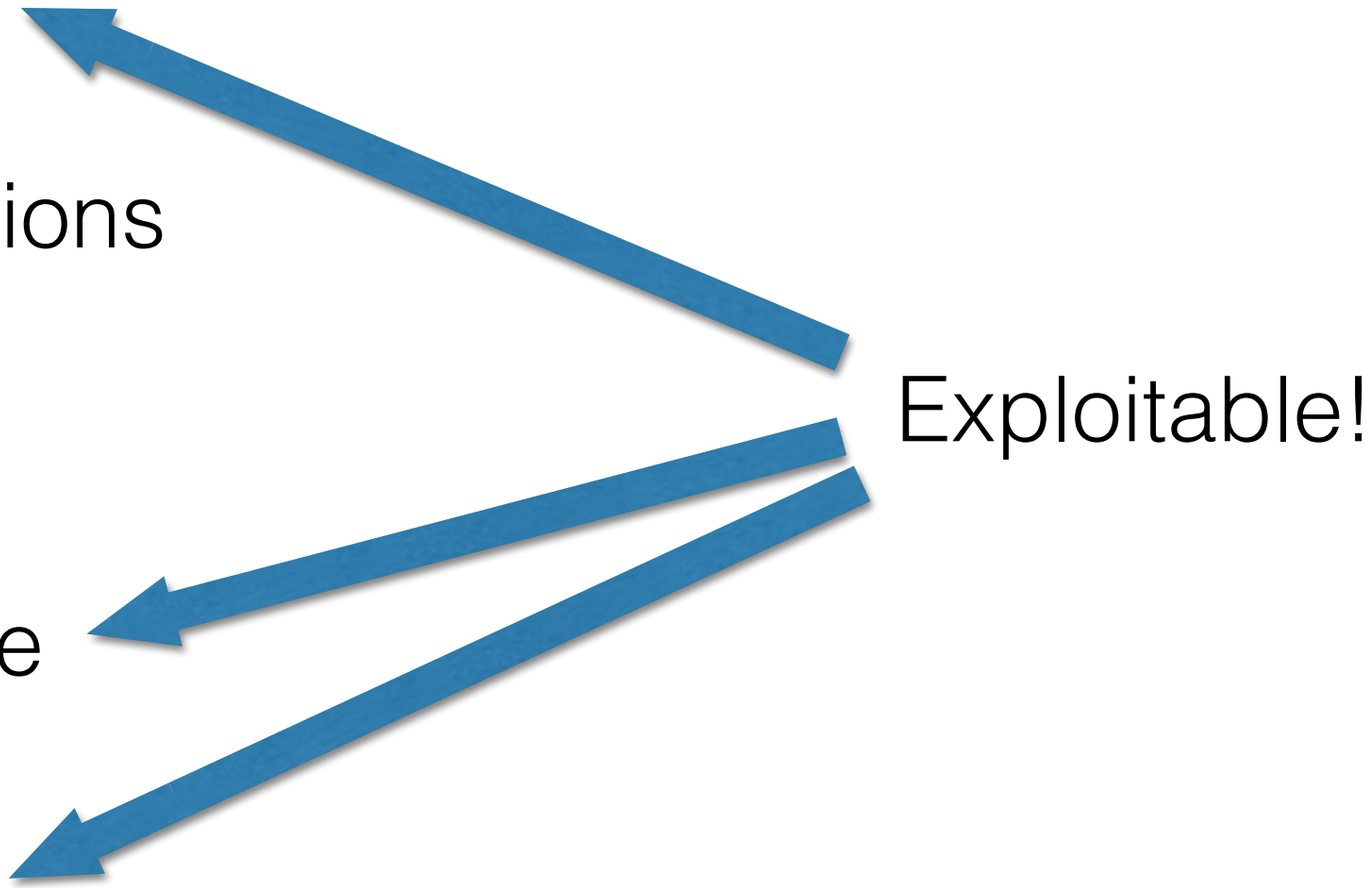


Getting our feet wet

```
// 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



Concurrency?

Rust?

Libraries

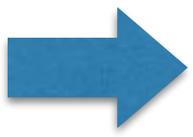
Futures

What's Rust?

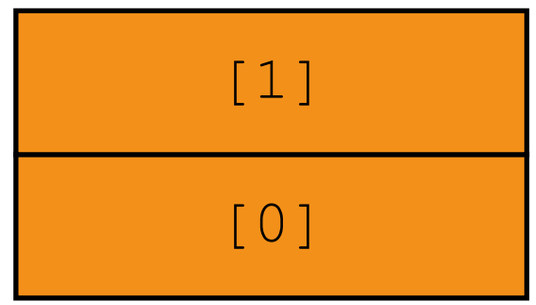
Rust is a systems programming language that runs blazingly fast, **prevents segfaults**, and guarantees thread safety.

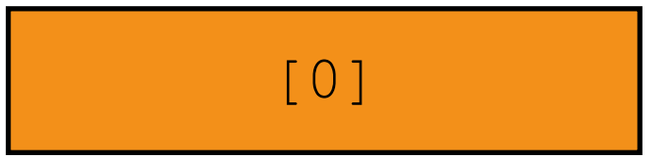
What's **safety**?

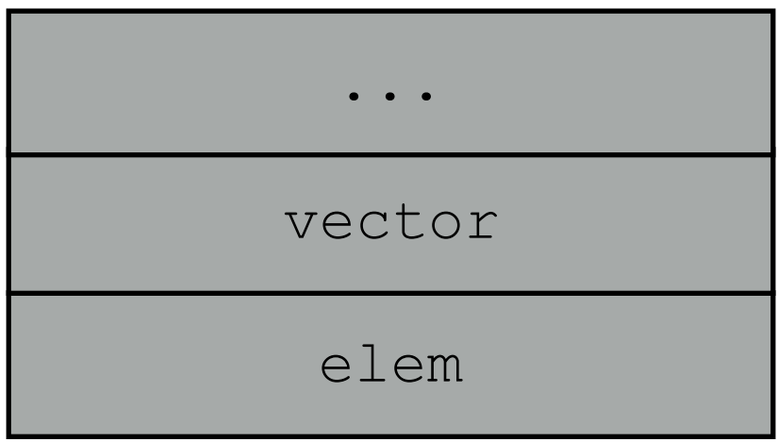
```
void example() {
   vector<string> vector;
   // ...
   auto& elem = vector[0];
   vector.push_back(some_string);
   cout << elem;
}
```

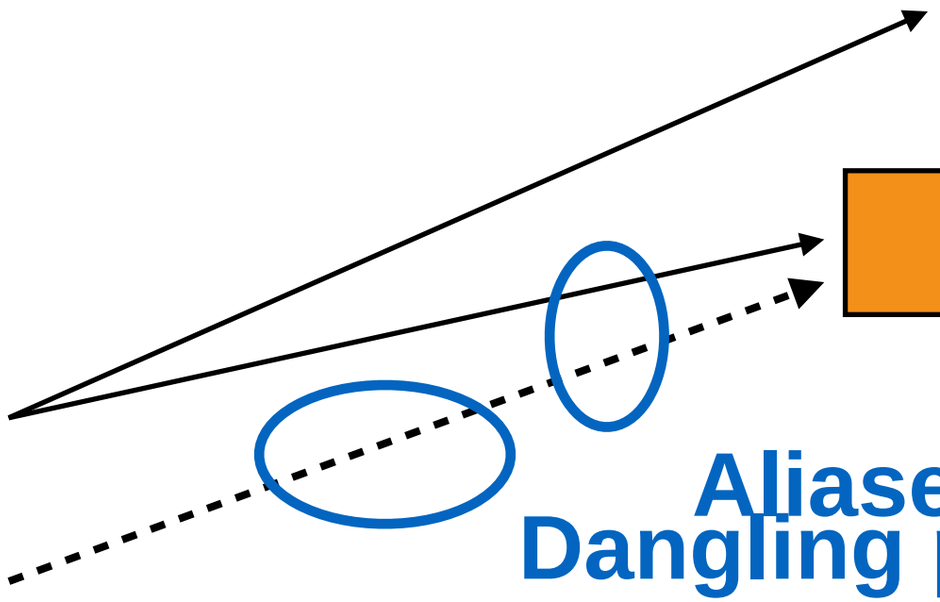


Mutation



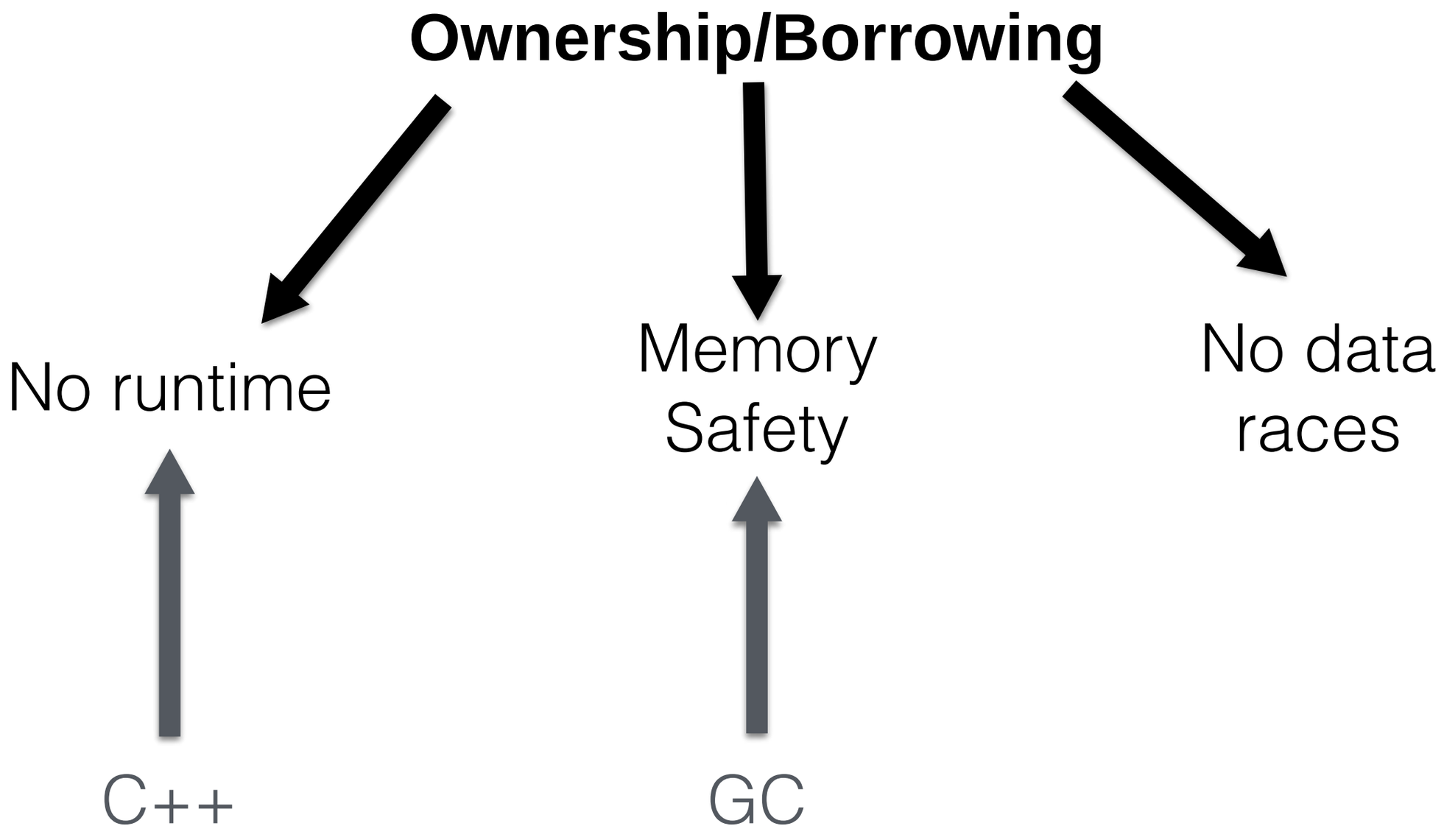






Aliased pointers
Dangling pointer!

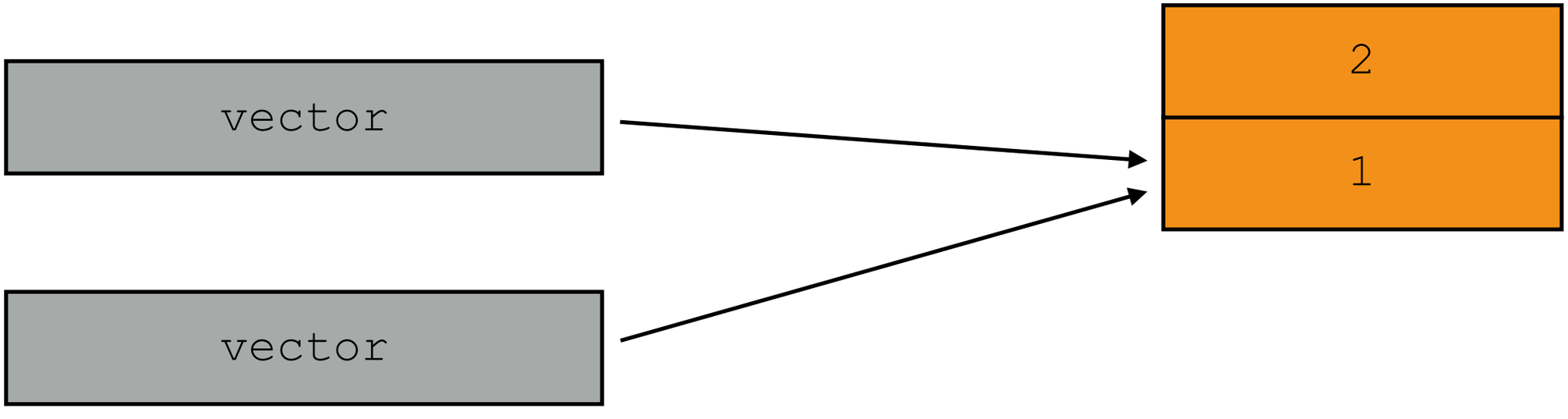
Rust's Solution



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);
}
```

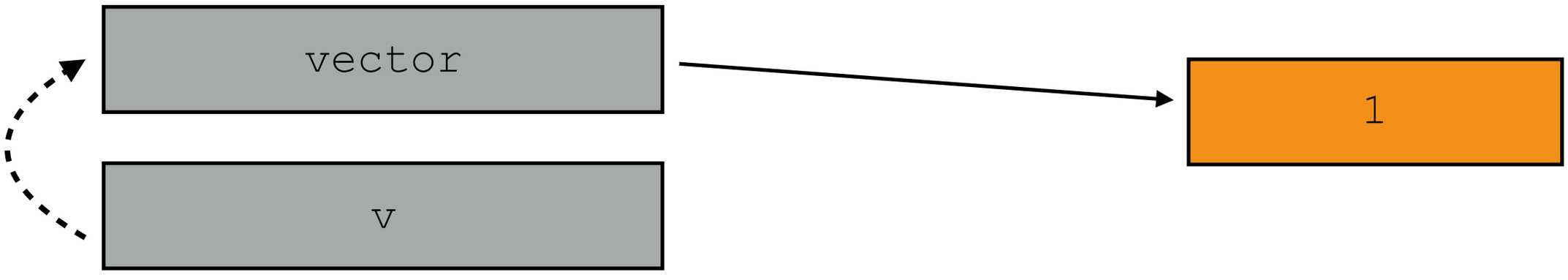
```
fn take(v: Vec<i32>) {
    // ...
}
```

error: use of moved value `v`

Borrowing

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

```
fn push(v: &mut Vec<i32>) {
    // v.push(1);
}
```



Safety in Rust

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

Data races

Aliasing!

- A data race happens when there are two concurrent memory accesses to the same location in a program where:
- at least one is unsynchronized
- at least one is a write

Mutation!

Concurrency?

Rust?

Libraries

Futures

Rust Concurrency Libs

- Language only provides ownership/borrowing
- Libraries implement common abstractions
- Flexible to cover wide range of paradigms

std::thread

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

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());
```

std::sync::Arc

```
let shared_numbers = Arc::new(vec![1, 2, 3]);
let child_numbers = shared_numbers.clone();
thread::spawn(move || {
    assert_eq!(child_numbers, [1, 2, 3]);
});
assert_eq!(shared_numbers, [1, 2, 3]);
```

std::sync::atomic::*

```
let number = AtomicUsize::new(10);
let prev = number.fetch_add(1, SeqCst);
assert_eq!(prev, 10);
let prev = number.swap(2, SeqCst);
assert_eq!(prev, 11);
assert_eq!(number.load(SeqCst), 2);
```

std::sync::Mutex

```
let lock = Mutex::new(vec![1, 2, 3]);
{
    let mut vector = lock.lock();
    vector.push(3);
}
// no more access to `vector`,
// lock is unlocked
```

crossbeam

- Epoch-based memory reclamation
- Easy translation of algorithms that require GC
- Work stealing deque
- MPMC queues

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()
}
```

100% Safe

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

Concurrency?

Rust?

Libraries

Futures

Async I/O in Rust (last year)

- *mio*, a “cross platform epoll” event loop library
- Servers were hand-written state machines
- Composition was quite difficult

facebook / wangle

What's a future?

- Database query
- RPC request
- Timeouts
- CPU intensive work
- Socket readiness

What's a future in Rust?

```
trait Future {
    type Item;
    type Error;

    fn poll(&mut self) -> Poll<Item, Error>;
    // ...
}
```

Composing futures

```
// Run one future, then another
f.and_then(|v| new_future(v))
```

```
// Wait for one of two futures
a.select(b)

// Wait for both futures
a.join(b)
```

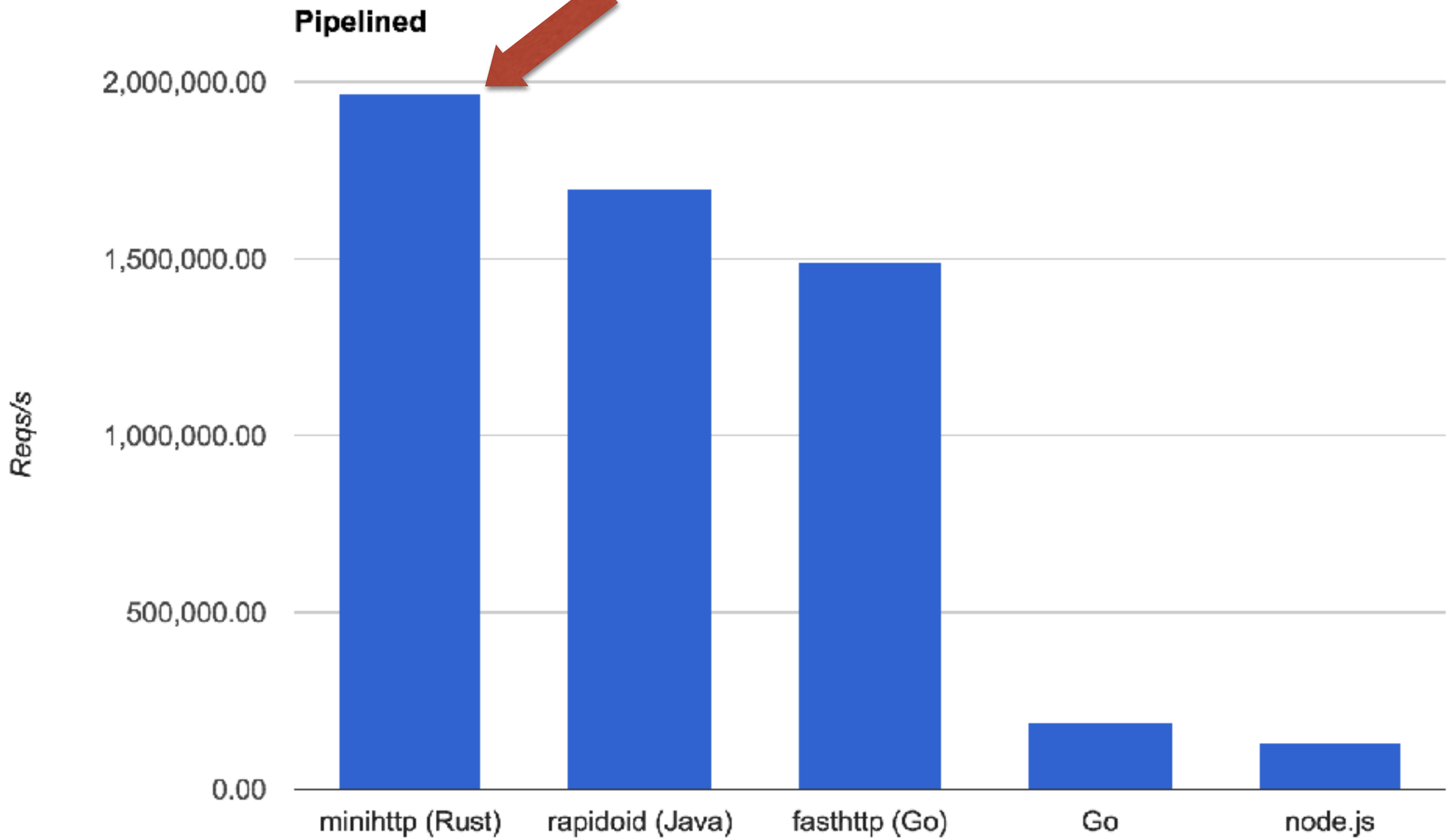
Async I/O in Rust (today)

- *futures*, a foundational abstraction for Async I/O
- Tokio, a runtime built on *mio* and *futures*
- Futures are **at all layers** of the stack

Zero-cost futures

- No allocations in combinators
- No synchronization in combinators
- Library is #![no_std] compatible
- One dynamic dispatch per event
- One allocation per connection

This is futures!



Don’t just take my word

Tokio

Framework	Best performance (higher is better)	Cls	Lng	Plt	FE	Aos	IA	Errors
octane	4,287,990 100.0%	Plt	C	Non	Non	Lin	Rea	112
tokio-minihttp	4,040,435 94.2%	Mcr	Rus	Rus	tok	Lin	Rea	
libreactor	3,912,028 91.2%	Plt	C	lib	Non	Lin	Rea	
rapidoid-http-fast	3,860,425 90.0%	Plt	Jav	Rap	Non	Lin	Rea	
rapidoid	3,832,927 89.4%	Plt	Jav	Rap	Non	Lin	Rea	
undertow	3,241,383 75.6%	Plt	Jav	Utw	Non	Lin	Rea	
colossus	3,173,481 74.0%	Mcr	Sca	Akk	Non	Lin	Rea	

`trait Future`

- Allows for most specialized implementation
- Enables inter-combinator optimizations
- Permits dynamic dispatch when required

`trait` Future

Zero-cost closures

```
fn and_then<F, B>(self, f: F) -> AndThen<Self, B, F>
  where F: FnOnce(Item) -> B,
        B: IntoFuture<Error=Self::Error>,
        Self: Sized,
```

Ergonomic conversion

Cancellation

- Cancel a future by dropping it
- Ownership implies one handle to a future
- Deterministic destruction so we know what drops

Questions?