Rust: Systems Programming for Everyone

Felix Klock (`@pnkfelix`), Mozilla

space: next slide; esc: overview; arrows navigate

`http://bit.ly/1LQM3PS`

Why ...?

Why use Rust?

Fast code, low memory footprint

Go from bare metal (assembly; C FFI) ...

... to high-level (collections, closures, generic containers) ...

with *zero cost* (no GC, unboxed closures, monomorphization of generics)

Safety and *Parallelism*

Safety and Parallelism

Safety

No segmentation faults

No undefined behavior

No data races

(Multi-paradigm) Parallelism

msg passing via channels

shared state via `Arc` and atomics, `Mutex`, etc

use native threads... or scoped threads... or work-stealing...

Why would you (Felix) work on Rust?

It's awesome!

(Were prior slides really not a sufficient answer?)

oh, maybe you meant ...

Why would Mozilla sponsor Rust?

Hard to prototype research-y browser changes atop C++ code base

Rust ⇒ Servo, WebRender

Want Rust for next-gen infrastructure (services, IoT)

> *"Our mission is to ensure the Internet is a global public resource, open and accessible to all. An Internet that truly puts people first, where individuals can shape their own experience and are empowered, safe and independent."*

"accessible to all"

Where is Rust now?

1.0 release was back in May 2015

Rolling release cycle (up to Rust 1.7 as of March 2nd 2016)

Open source from the begining
`https://github.com/rust-lang/rust/`

Open model for future change (RFC process)
`https://github.com/rust-lang/rfcs/`

Awesome developer community (~1,000 people in `#rust`, ~250 people in `#rust-internals`, ~1,300 unique commiters to rust.git)

Talk plan

"Why Rust" Demonstration

"Ownership is easy" (... or is it?)

Sharing	Stuff
Sharing *capabilities*	(Language stuff)
Sharing *work*	(Parallelism stuff)
Sharing *code*	(Open source distribution stuff)

Lightning Demo

Demo: sequential web page fetch

```
fn sequential_web_fetch() {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method

    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
        "http://imgur.com", "http://mozilla.org"];

    for &site in sites { // step through the array...
        let client = Client::new();
        let res = client.get(site).send().unwrap();
        assert_eq!(res.status, hyper::Ok);
        let char_count = res.chars().count();
        println!("site: {} chars: {}", site, char_count);
    }
}
```

(lets get rid of the Rust-specific pattern binding in `for`; this is not a tutorial)

Demo: sequential web page fetch

```rust
fn sequential_web_fetch() {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method

    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
        "http://imgur.com", "http://mozilla.org"];

    for site_ref in sites { // step through the array...
        let site = *site_ref; // (separated for expository purposes)

        { // (and a separate block, again for expository purposes)
            let client = Client::new();

            let res = client.get(site).send().unwrap();
            assert_eq!(res.status, hyper::Ok);
            let char_count = res.chars().count();
            println!("site: {} chars: {}", site, char_count);
        }
    }
}
```

Demo: concurrent web page fetch

```rust
fn concurrent_web_fetch() -> Vec<::std::thread::JoinHandle<()>> {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method

    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
        "http://imgur.com", "http://mozilla.org"];
    let mut handles = Vec::new();
    for site_ref in sites {
        let site = *site_ref;
        let handle = ::std::thread::spawn(move || {
            // block code put in closure: ~~~~~~~
            let client = Client::new();

            let res = client.get(site).send().unwrap();
            assert_eq!(res.status, hyper::Ok);
            let char_count = res.chars().count();
            println!("site: {} chars: {}", site, char_count);
        });

        handles.push(handle);
    }

    return handles;
}
```

Print outs

Sequential version:

```
site: http://www.eff.org/ chars: 42425
site: http://rust-lang.org/ chars: 16748
site: http://imgur.com chars: 152384
site: http://mozilla.org chars: 63349
```

(on every run, when internet, and sites, available)

Concurrent version:

```
site: http://imgur.com chars: 152384
site: http://rust-lang.org/ chars: 16748
site: http://mozilla.org chars: 63349
site: http://www.eff.org/ chars: 42425
```

(on at least one run)

"what is this 'soundness' of which you speak?"

Demo: soundness I

```
fn sequential_web_fetch_2() {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method

    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
    //  ~~~~~ `sites`, an array (slice) of strings, is stack-local
        "http://imgur.com", "http://mozilla.org"];

    for site_ref in sites {
    //  ~~~~~~~~ `site_ref` is a *reference to* elem of array.
        let client = Client::new();
        let res = client.get(*site_ref).send().unwrap();
        // moved deref here  ~~~~~~~~~
        assert_eq!(res.status, hyper::Ok);
        let char_count = res.chars().count();
        println!("site: {} chars: {}", site_ref, char_count);
    }
}
```

Demo: soundness II

```
fn concurrent_web_fetch_2() -> Vec<::std::thread::JoinHandle<()>> {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method

    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
    //  ~~~~~ `sites`, an array (slice) of strings, is stack-local
        "http://imgur.com", "http://mozilla.org"];
    let mut handles = Vec::new();
    for site_ref in sites {
    //  ~~~~~~~~ `site_ref` still a *reference* into an array
        let handle = ::std::thread::spawn(move || {
            let client = Client::new();
            let res = client.get(*site_ref).send().unwrap();
            // moved deref here  ~~~~~~~~~
            assert_eq!(res.status, hyper::Ok);
            let char_count = res.chars().count();
            println!("site: {} chars: {}", site_ref, char_count);
            // Q: will `sites` array still be around when above runs?
        });
        handles.push(handle);
    }
    return handles;
}
```

some (white) lies:
"Rust is just about ownership"

"Ownership is intuitive"

"Ownership is intuitive"

Let's buy a car

```
let money: Money = bank.withdraw_cash();
let my_new_car: Car = dealership.buy_car(money);
```

```
let second_car = dealership.buy_car(money); // <-- cannot reuse
```

money transferred into `dealership`, and car transferred to us.

"Ownership is intuitive"

Let's buy a car

```
let money: Money = bank.withdraw_cash();
let my_new_car: Car = dealership.buy_car(money);
// let second_car = dealership.buy_car(money); // <-- cannot reuse
```

money transferred into **`dealership`**, and car transferred to us.

```
my_new_car.drive_to(home);
garage.park(my_new_car);
```

```
my_new_car.drive_to(...) // now doesn't work
```

(can't drive car without access to it, e.g. taking it out of the garage)

"Ownership is intuitive"

Let's buy a car

```
let money: Money = bank.withdraw_cash();
let my_new_car: Car = dealership.buy_car(money);
// let second_car = dealership.buy_car(money); // <-- cannot reuse
```

money transferred into **`dealership`**, and car transferred to us.

```
my_new_car.drive_to(home);
garage.park(my_new_car);
// my_new_car.drive_to(...) // now doesn't work
```

(can't drive car without access to it, e.g. taking it out of the garage)

```
let my_car = garage.unpark();
my_car.drive_to(work);
```

...reflection time...

Correction: Ownership is intuitive, except for programmers ...

(copying data like integers, and characters, and .mp3's, is "free")

... and anyone else who *names* things

Über Sinn und Bedeutung

("On sense and reference" -- Gottlob Frege, 1892)

If ownership were all we had, car-purchase slide seems nonsensical

```
my_new_car.drive_to(home);
```

Does this transfer **`home`** into the car?

Do I lose access to my home, just because I drive to it?

We must distinguish an object itself from ways to name that object

Above, **`home`** cannot be (an owned) **`Home`**

`home` must instead be some kind of *reference* to a **`Home`**

So we will need references

We can solve any problem by introducing an extra level of indirection

-- David J. Wheeler

a truth: Ownership *is* important

Ownership is important

Ownership enables:	which removes:
RAII-style destructors	a source of memory leaks (or fd leaks, etc)
no dangling pointers	many resource management bugs
no data races	many multithreading heisenbugs

Do I need to take ownership here, accepting the associated resource management responsibility? Would temporary access suffice?

Good developers ask this already!

Rust forces function signatures to encode the answers

(and they are checked by the compiler)

Sharing Data: Ownership and References

Rust types

Move	Copy	Copy if `T:Copy`
`Vec<T>`, `String`, …	`i32`, `char`, …	`[T; n]`, `(T1,T2,T3)`, …

```
struct Car { color: Color, engine: Engine }

fn demo_ownership() {
    let mut used_car: Car = Car { color: Color::Red,
                                  engine: Engine::BrokenV8 };
    let apartments = ApartmentBuilding::new();
```

references to data (`&mut T`, `&T`):

```
    let my_home: &Home;       // <-- an "immutable" borrow
    let christine: &mut Car; // <-- a "mutable" borrow
    my_home = &apartments[6]; //       (read `mut` as "exclusive")
    let neighbors_home = &apartments[5];
    christine = &mut used_car;
    christine.engine = Engine::VintageV8;
}
```

Why multiple **&**-reference types?

Distinguish *exclusive* access from *shared* access

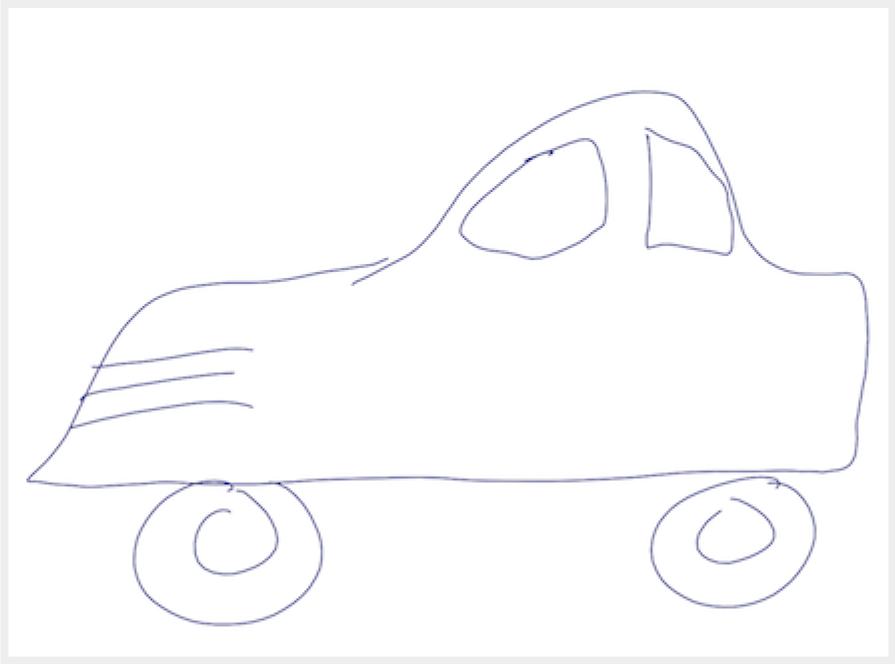
Enables **safe**, **parallel** API's

A Metaphor

(reminder: metaphors never work 100%)

```rust
let christine = Car::new();
```

This is "Christine"



pristine unborrowed car

(apologies to Stephen King)

```
let read_only_borrow = &christine;
```

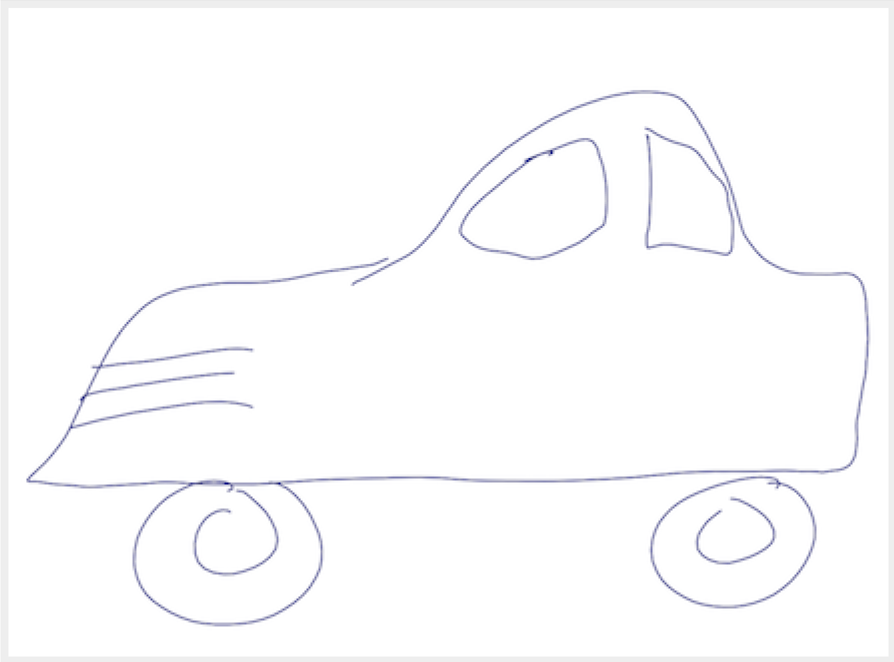
single inspector (immutable borrow)

(apologies to Randall Munroe)

```
read_only_borrows[2] = &christine;
read_only_borrows[3] = &christine;
read_only_borrows[4] = &christine;
```

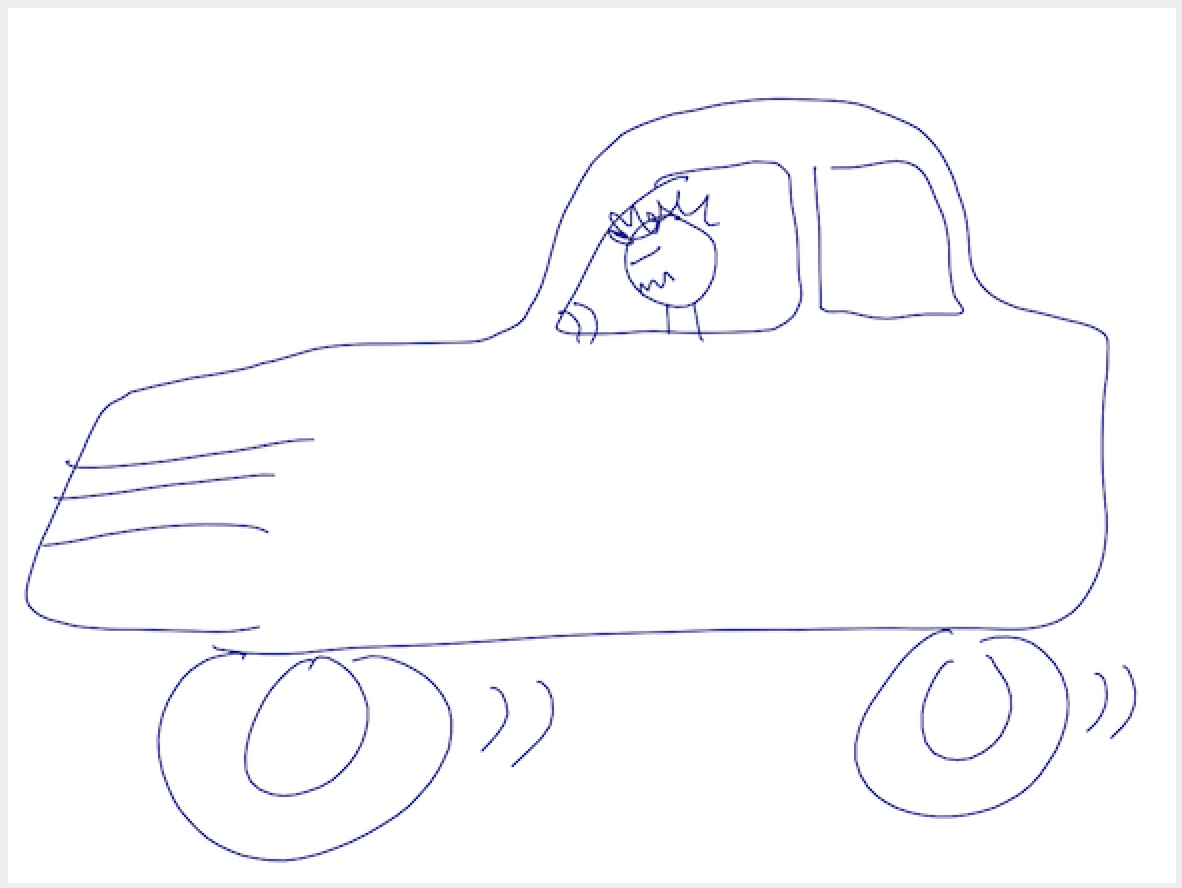
many inspectors (immutable borrows)

When inspectors are finished, we are left again with:



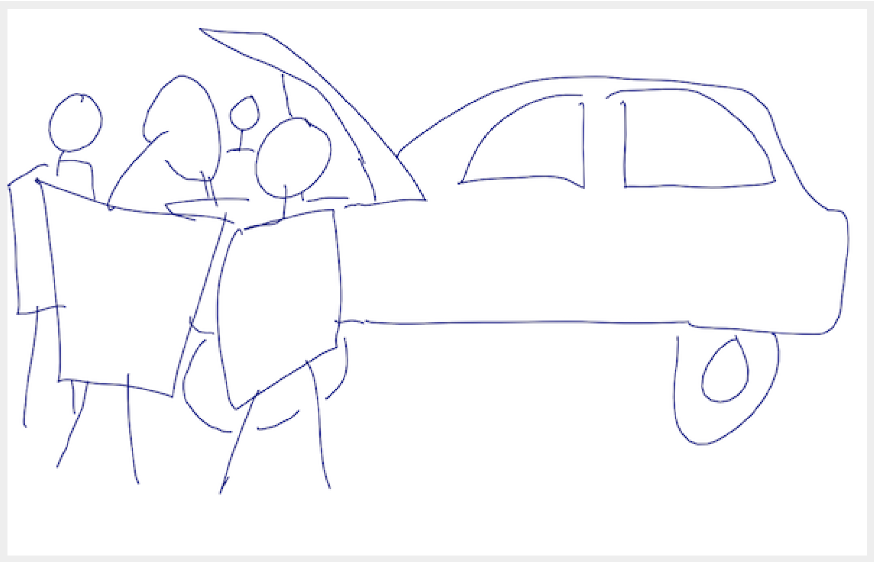
pristine unborrowed car

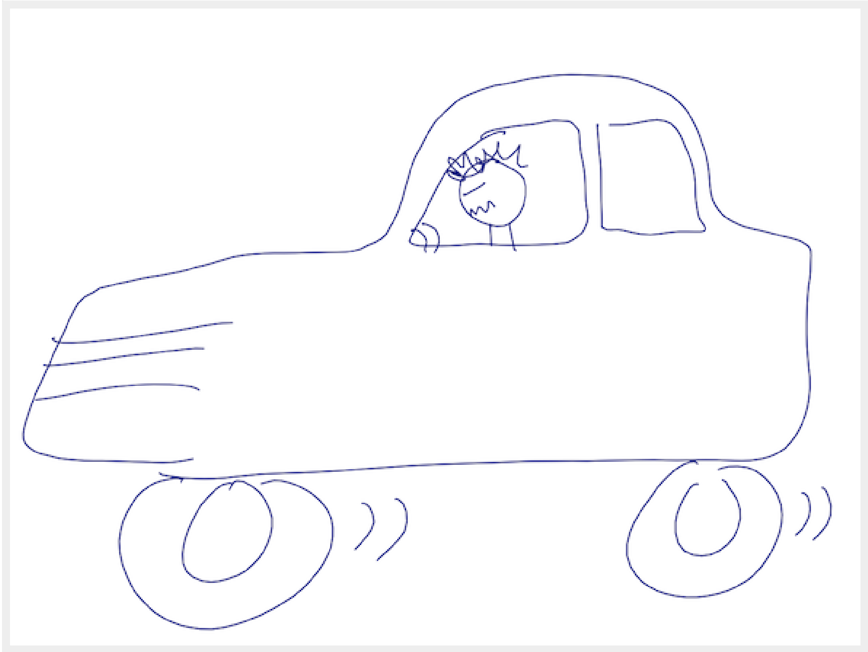
```rust
let mutable_borrow = &mut christine; // like taking keys ...
give_arnie(mutable_borrow); // ... and giving them to someone
```



driven car (mutably borrowed)

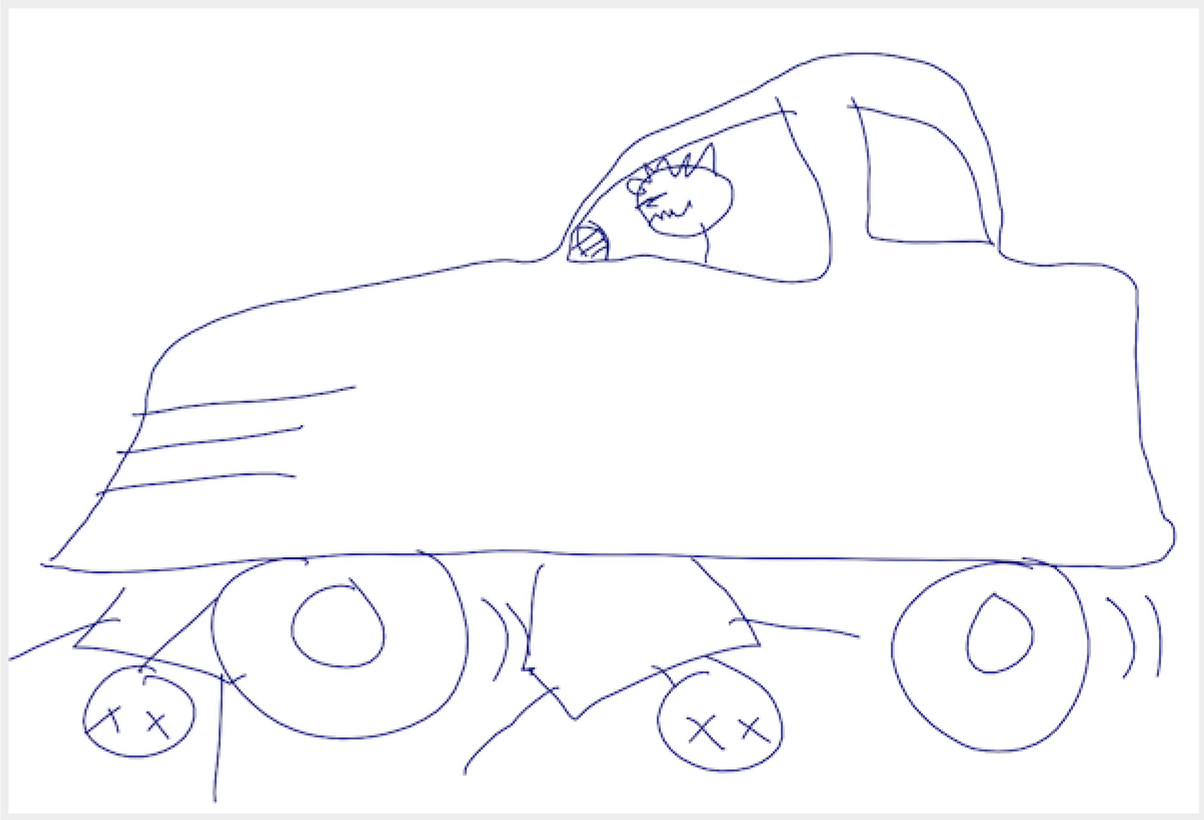
Can't mix the two in safe code!





Otherwise: (data) races!

```rust
read_only_borrows[2] = &christine;
let mutable_borrow = &mut christine;
read_only_borrows[3] = &christine;
// ⇒ CHAOS!
```



mixing mutable and immutable is illegal

Ownership	`T`	
Exclusive access	`&mut T`	("mutable")
Shared access	`&T`	("read-only")

Exclusive access

`&mut`: can I borrow the car?

```
fn borrow_the_car_1() {
    let mut christine = Car::new();
    {
        let car_keys = &mut christine;
        let arnie = invite_friend_over();
        arnie.lend(car_keys);
    } // end of scope for `arnie` and `car_keys`
    christine.drive_to(work); // I still own the car!
}
```

But when her keys are elsewhere, I cannot drive `christine`!

```
fn borrow_the_car_2() {
    let mut christine = Car::new();
    {
        let car_keys = &mut christine;
        let arnie = invite_friend_over();
        arnie.lend(car_keys);
        christine.drive_to(work); // <-- compile error
    } // end of scope for `arnie` and `car_keys`
}
```

Extending the metaphor

Possessing the keys, Arnie could take the car for a new paint job.

```rust
fn lend_1(arnie: &Arnie, k: &mut Car) { k.color = arnie.fav_color; }
```

Or lend keys to someone else (*reborrowing*) before paint job

```rust
fn lend_2(arnie: &Arnie, k: &mut Car) {
    arnie.partner.lend(k); k.color = arnie.fav_color;
}
```

Owner loses capabilities attached to `&mut`-borrows only *temporarily* (*)

(*): "Car keys" return guaranteed by Rust; sadly, not by physical world

End of metaphor

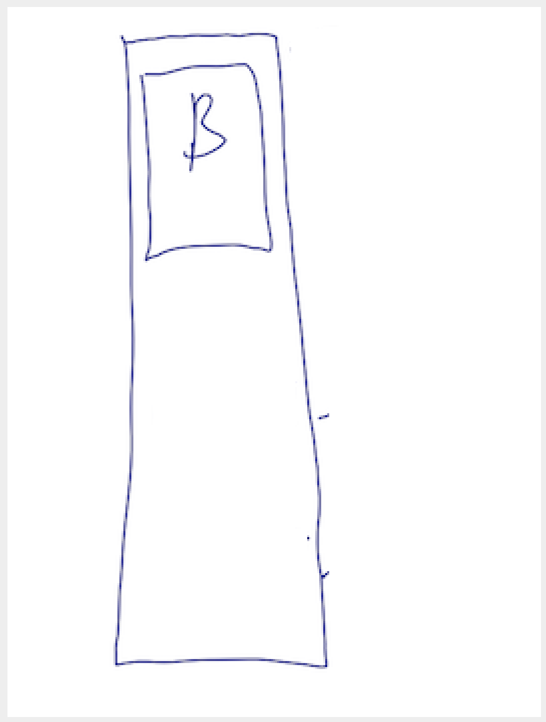
(on to models)

Pointers, Smart and Otherwise

(More pictures)

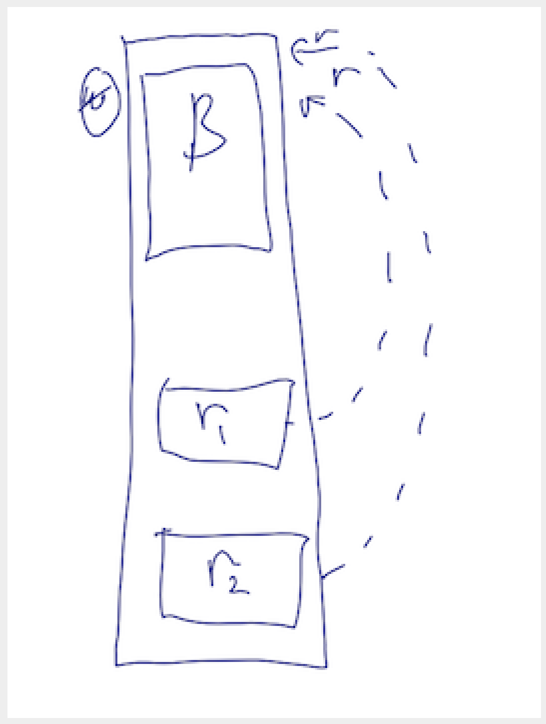
Stack allocation

```
let b = B::new();
```



stack allocation

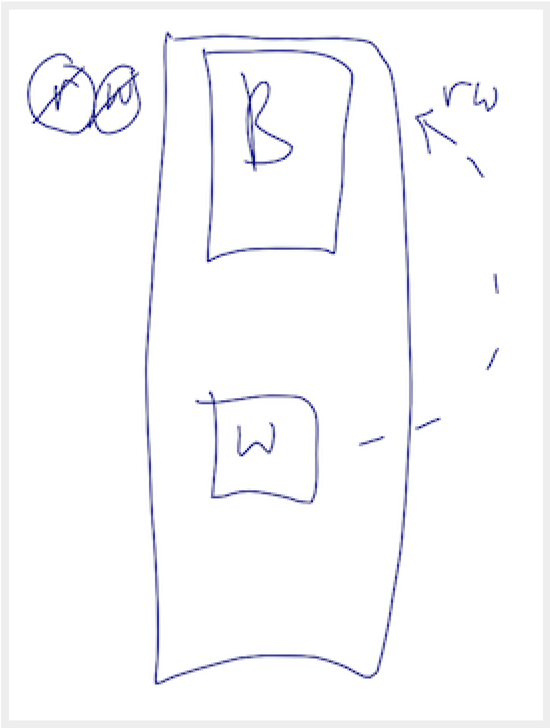
```rust
let b = B::new();

let r1: &B = &b;
let r2: &B = &b;
```



stack allocation and immutable borrows

(**b** has lost write capability)

```
let mut b = B::new();

let w: &mut B = &mut b;
```

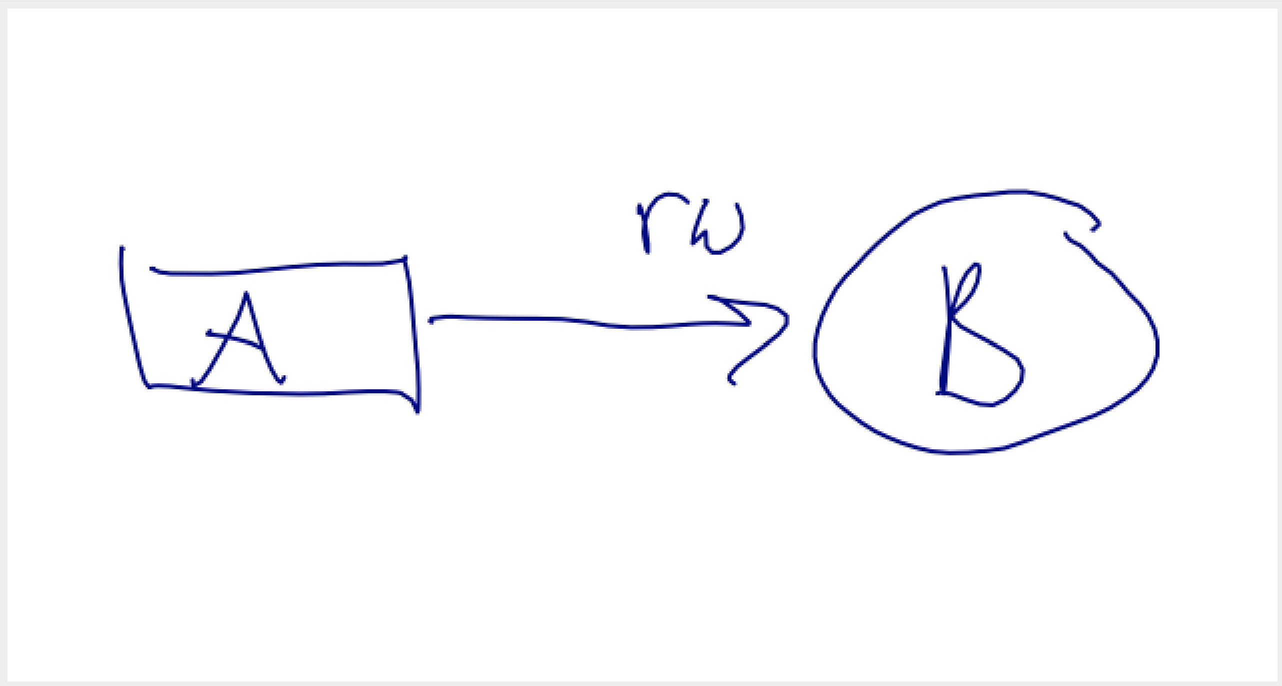


stack allocation and mutable borrows

(**b** has temporarily lost both read *and* write capabilities)

Heap allocation: **Box<B>**

```
let a = Box::new(B::new());
```

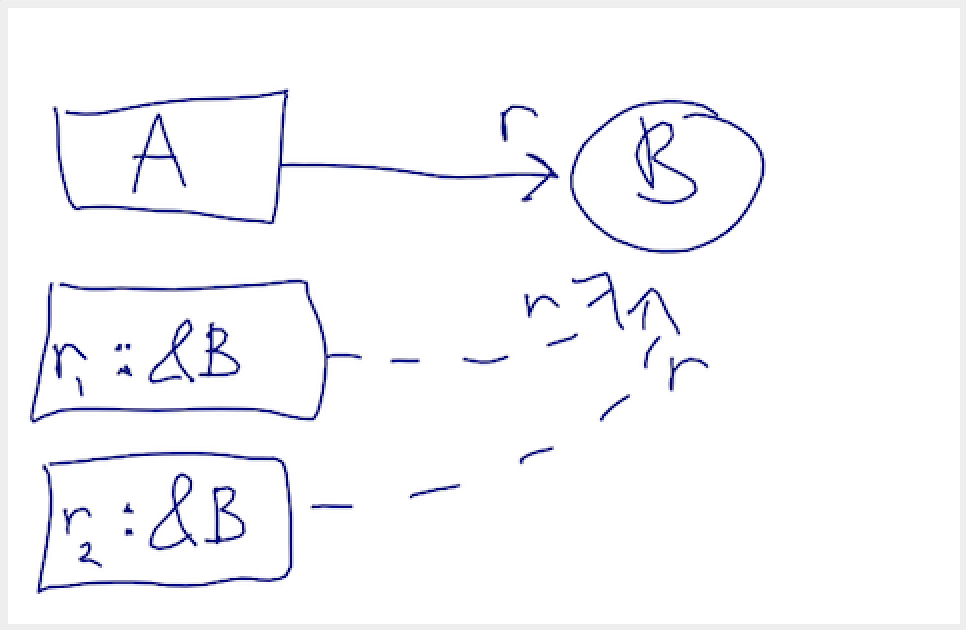


pristine boxed B

a (as owner) has both read and write capabilities

Immutably borrowing a box

```
let a = Box::new(B::new());
let r_of_box: &Box<B> = &a; // (not directly a ref of B)

let r1: &B = &*a;
let r2: &B = &a; // <-- coercion!
```

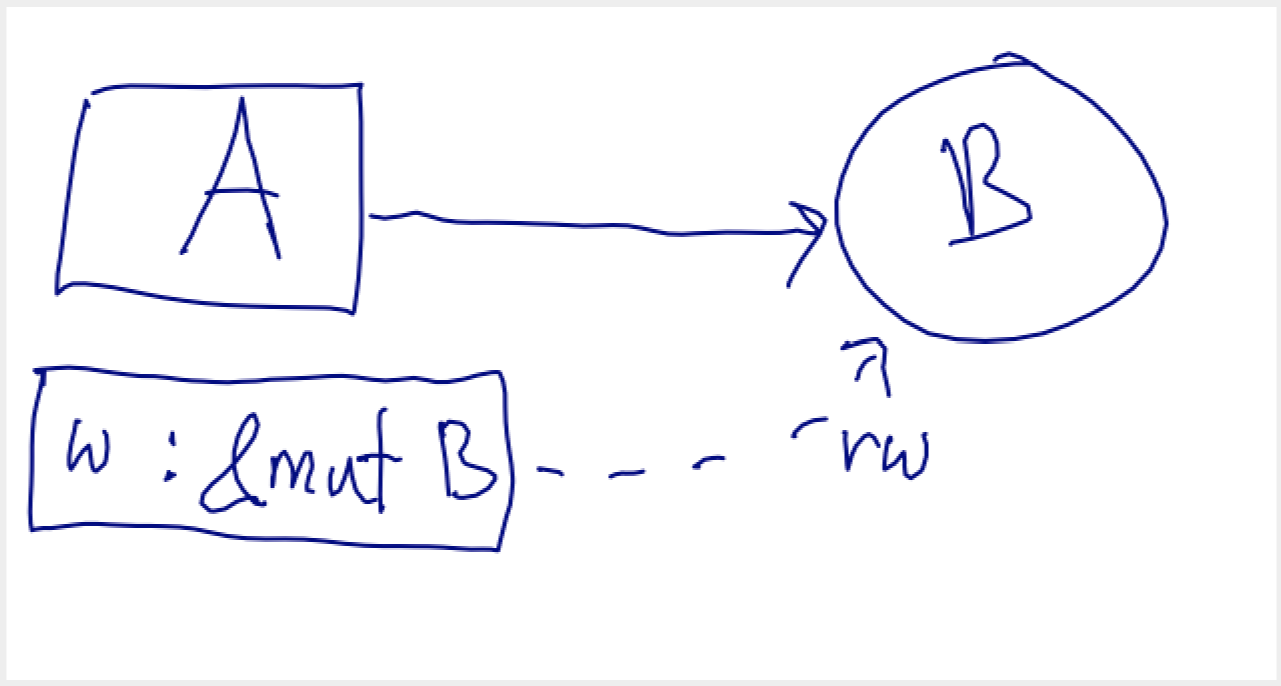


immutable borrows of heap-allocated B

a retains read capabilities (has temporarily lost write)

Mutably borrowing a box

```
let mut a = Box::new(B::new());

let w: &mut B = &mut a; // (again, coercion happening here)
```

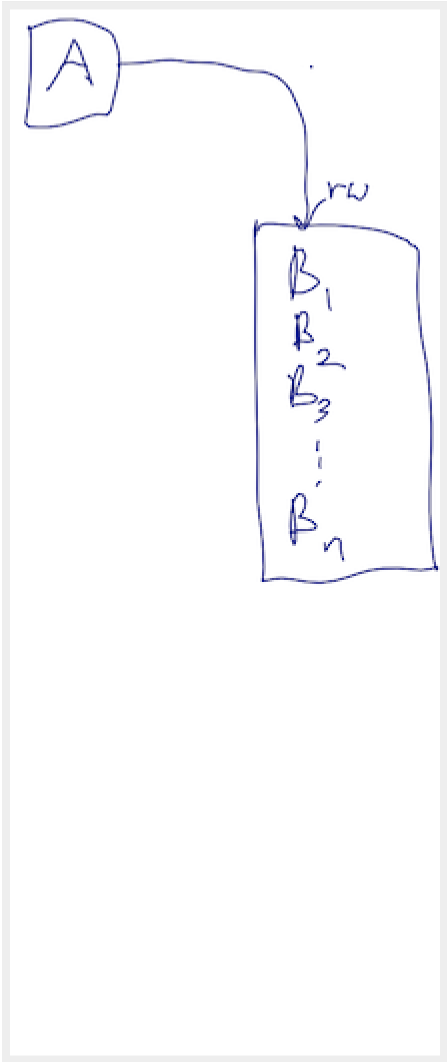


mutable borrow of heap-allocated B

a has temporarily lost *both* read and write capabilities

Heap allocation: **Vec<B>**

```
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
```



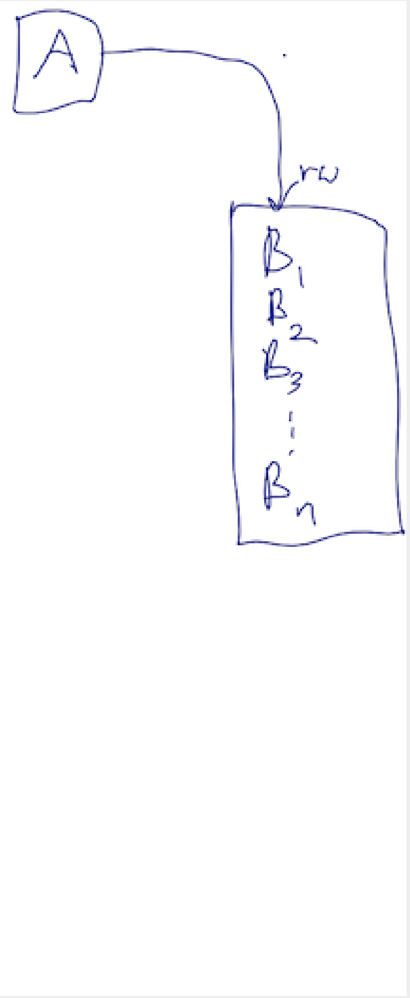
vec, filled to capacity

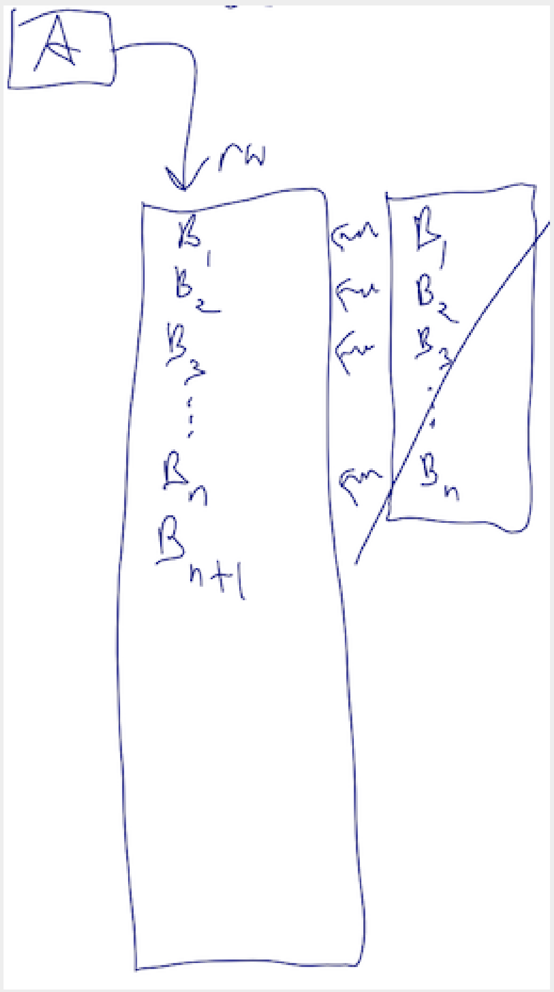
Vec Reallocation

```
...
a.push(B::new());
```

before

after

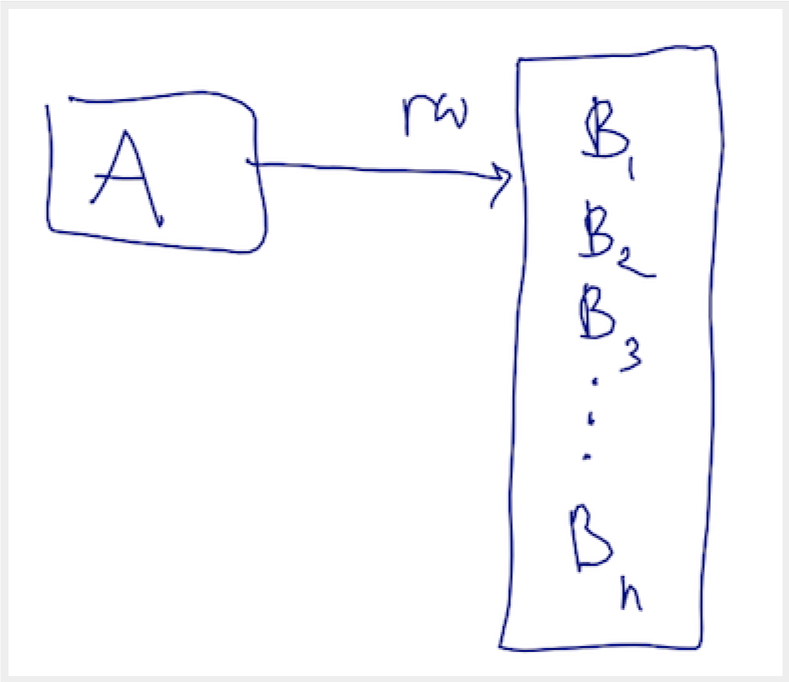

A
rw
B1
B2
B3
Bn


A
rw
B1
B2
B3
Bn
Bn+1
B1
B2
B3
Bn

Slices: borrowing *parts* of an array

Basic **Vec<B>**

```
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
```

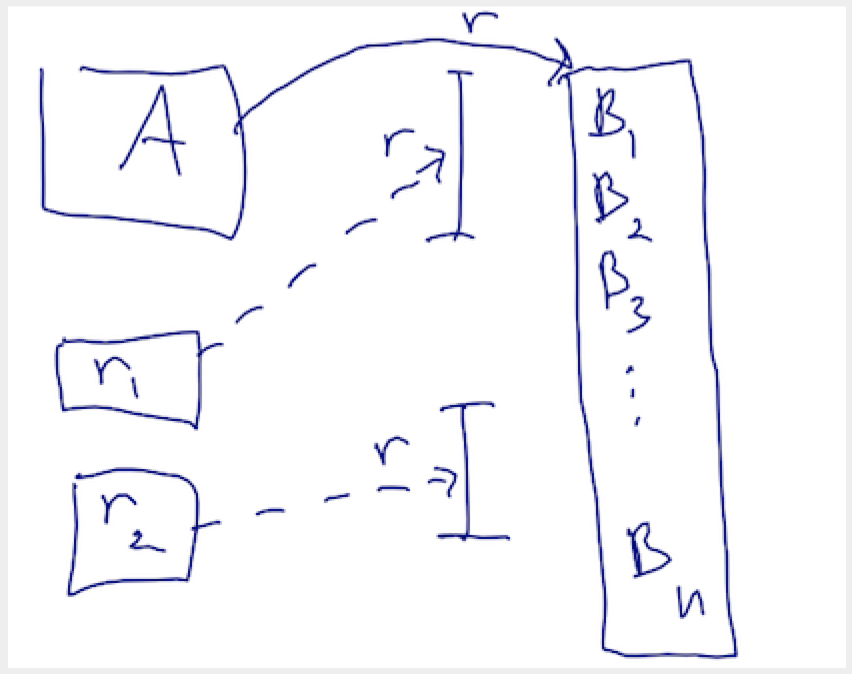


pristine unborrowed vec

(**a** has read and write capabilities)

Immutable borrowed slices

```
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
let r1 = &a[0..3];
let r2 = &a[7..n-4];
```

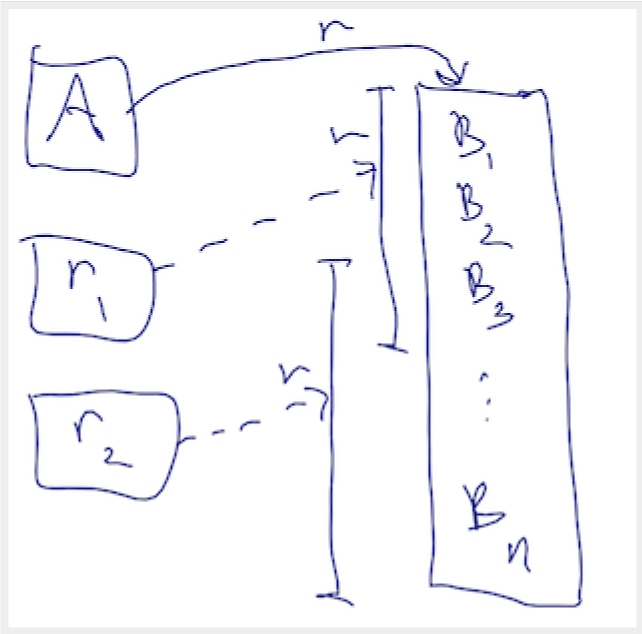


mutiple borrowed slices vec

(**a** has only read capability now; shares it with **r1** and **r2**)

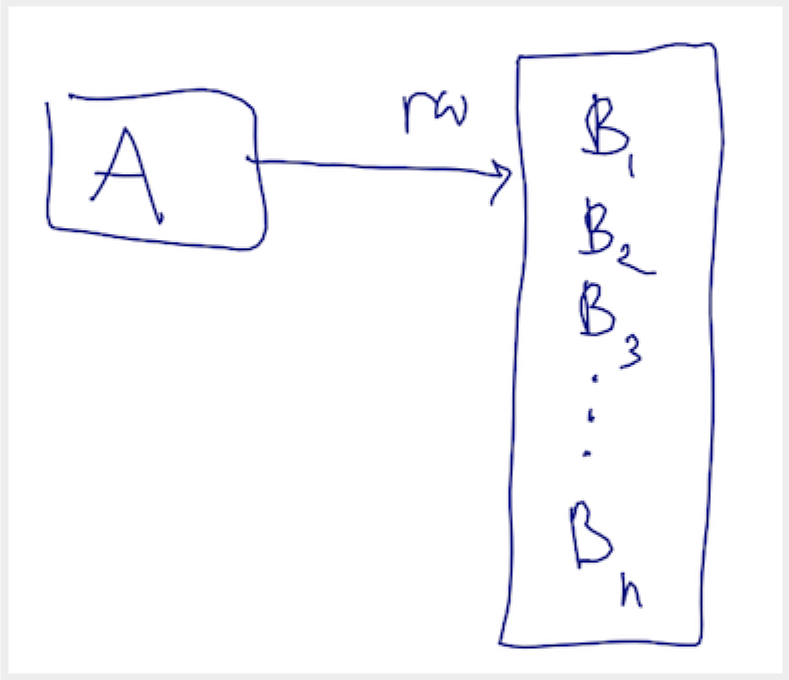
Safe overlap between &[..]

```
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
let r1 = &a[0..7];
let r2 = &a[3..n-4];
```



overlapping slices

Basic **Vec<B>** again

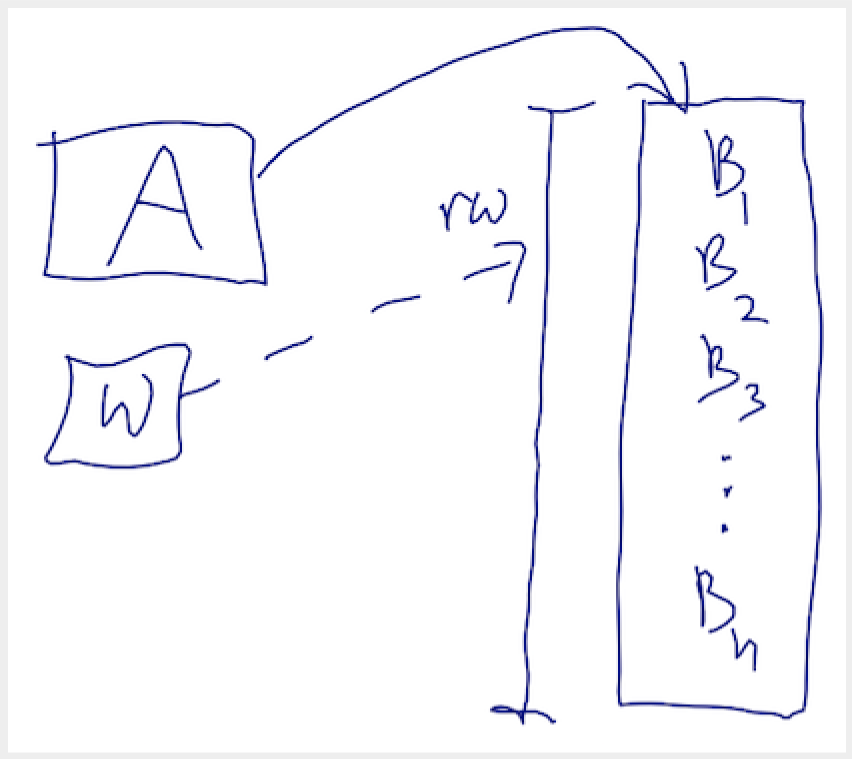


pristine unborrowed vec

(**a** has read and write capabilities)

Mutable slice of whole vec

```
let w = &mut a[0..n];
```

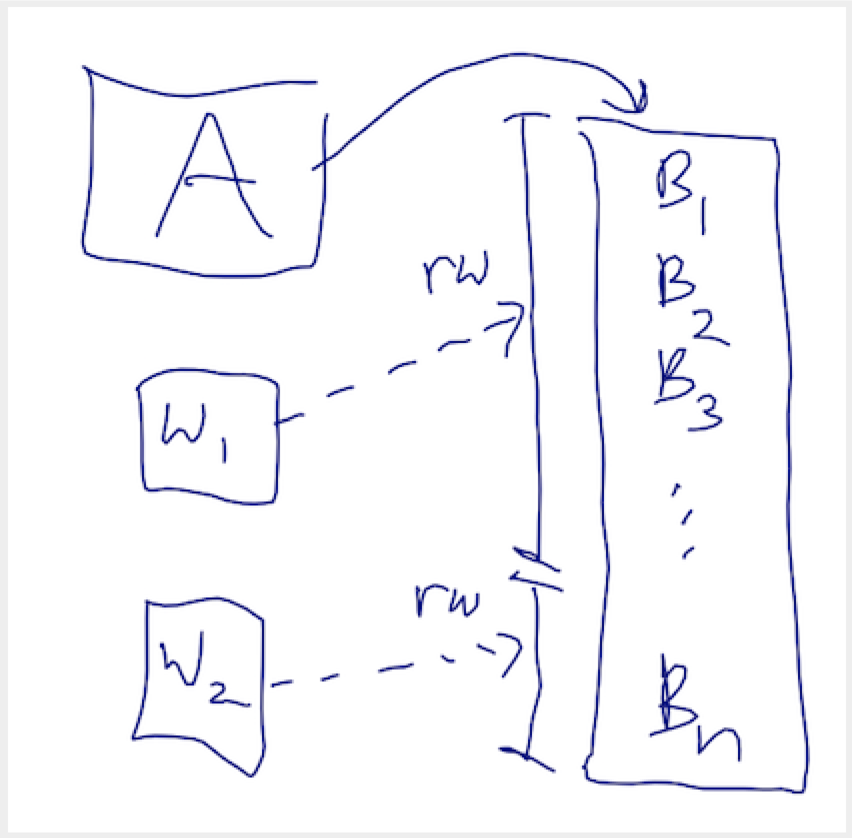


mutable slice of vec

(`a` has *no* capabilities; `w` now has read and write capability)

Mutable disjoint slices

```
let (w1,w2) = a.split_at_mut(n-4);
```



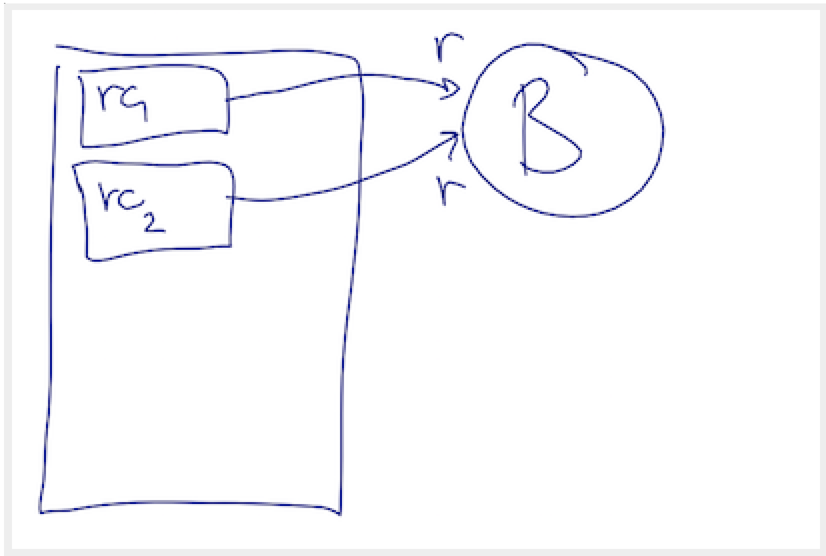
disjoint mutable borrows

(**w1** and **w2** share read and write capabilities for disjoint portions)

Shared *Ownership*

Shared Ownership

```rust
let rc1 = Rc::new(B::new());
let rc2 = rc1.clone(); // increments ref-count on heap-alloc'd value
```



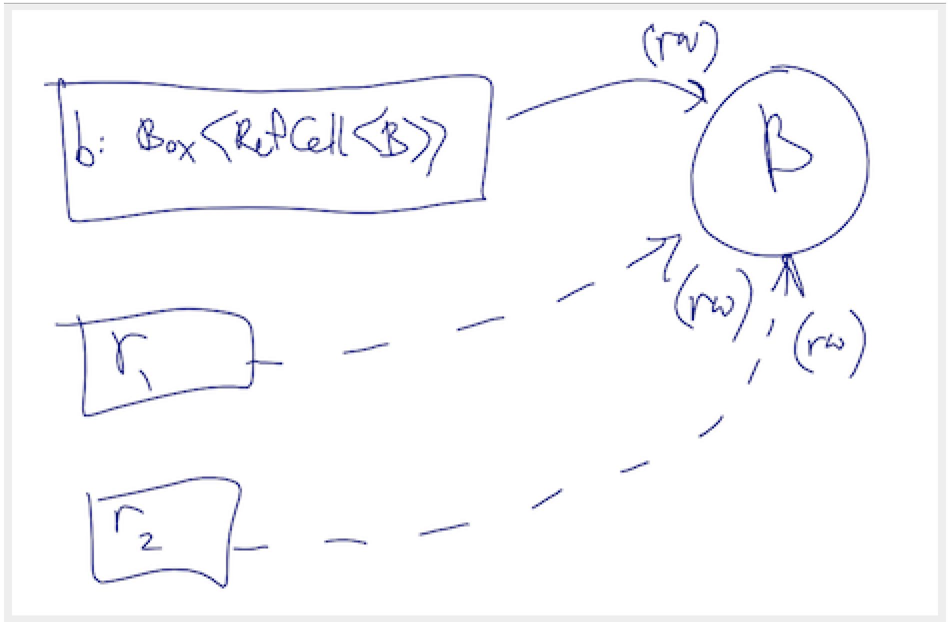
shared ownership via ref counting

(**`rc1`** and **`rc2`** each have read access; but neither can statically assume exclusive (**`mut`**) access, nor can they provide **`&mut`** borrows without assistance.)

Dynamic Exclusivity

`RefCell<T>`: Dynamic Exclusivity

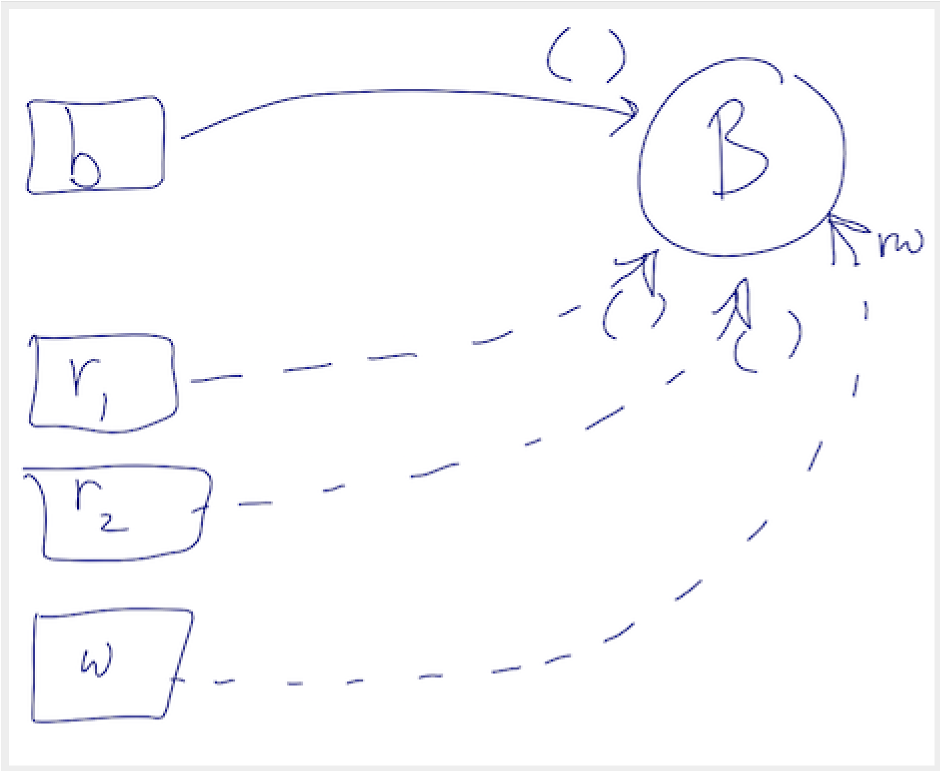
```
let b = Box::new(RefCell::new(B::new()));

let r1: &RefCell<B> = &b;
let r2: &RefCell<B> = &b;
```



box of refcell

`RefCell<T>`: Dynamic Exclusivity

```
let b = Box::new(RefCell::new(B::new()));
let r1: &RefCell<B> = &b;
let r2: &RefCell<B> = &b;
let w = r2.borrow_mut(); // if successful, `w` acts like `&mut B`
```



fallible mutable borrow

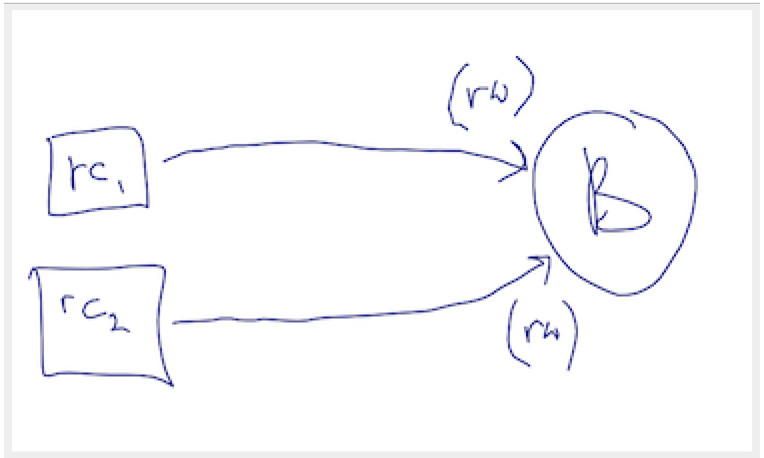
```
// below panics if `w` still in scope
```

```
// below panics if `w` still in scope
let w2 = b.borrow_mut();
```

Previous generalizes to shared ownership

Rc<RefCell<T>>

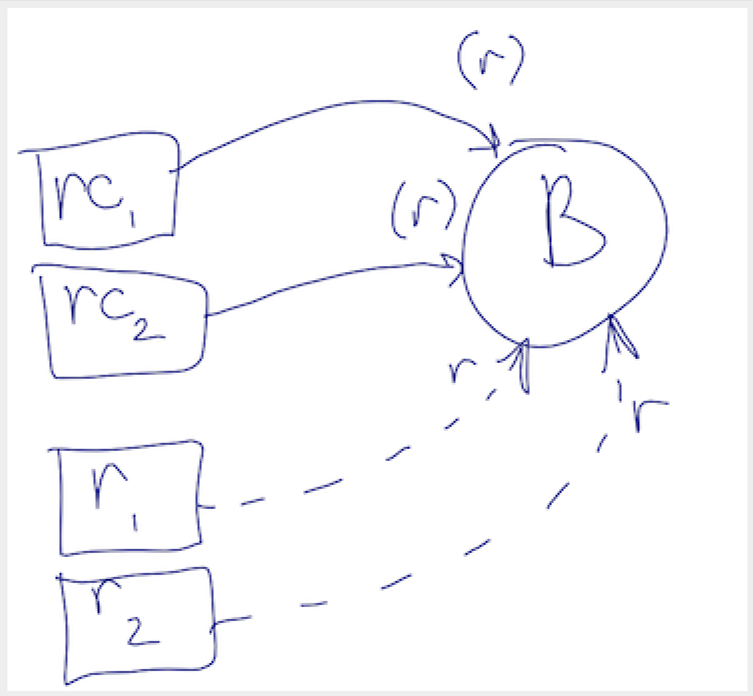
```
let rc1 = Rc::new(RefCell::new(B::new()));
let rc2 = rc1.clone(); // increments ref-count on heap-alloc'd value
```



shared ownership of refcell

Rc<RefCell<T>>

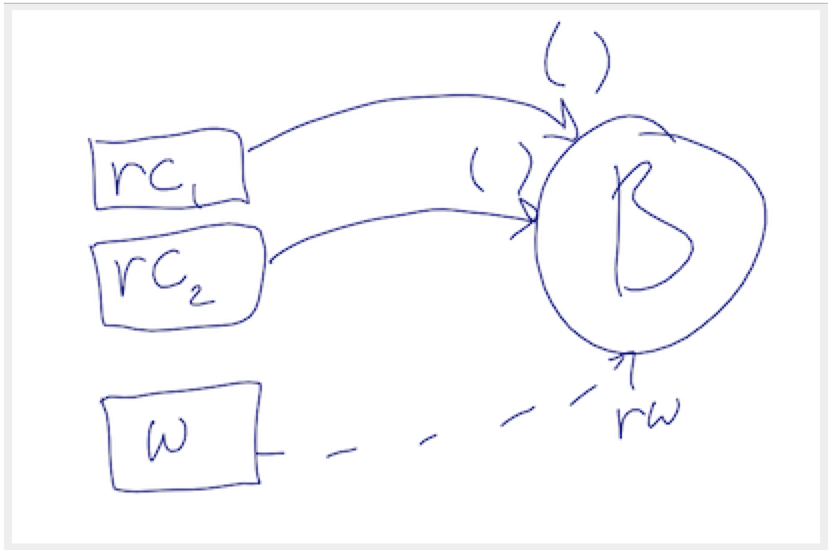
```
let rc1 = Rc::new(RefCell::new(B::new()));
let rc2 = rc1.clone();
let r1: &RefCell<B> = &rc1;
let r2: &RefCell<B> = &rc2; // (or even just `r1`)
```



borrows of refcell can alias

Rc<RefCell<T>>

```
let rc1 = Rc::new(RefCell::new(B::new()));
let rc2 = rc1.clone();
let w = rc2.borrow_mut();
```



there can be only one!

What static guarantees does `Rc<RefCell<T>>` have?

Not much!

If you want to port an existing *imperative* algorithm with all sorts of sharing, you *could* try using `Rc<RefCell<T>>`.

You then might spend much less time wrestling with Rust's type (+borrow) checker.

The point: `Rc<RefCell<T>>` is nearly an anti-pattern. It limits static reasoning. You should avoid it if you can.

Other kinds of shared ownership

`TypedArena<T>`

`Cow<T>`

`Rc<T>` vs `Arc<T>`

Sharing Work: Parallelism / Concurrency

Threading APIs (plural!)

`std::thread`

`dispatch` : OS X-specific "Grand Central Dispatch"

`crossbeam` : Lock-Free Abstractions, Scoped "Must-be" Concurrency

`rayon` : Scoped Fork-join "Maybe" Parallelism (inspired by Cilk)

(Only the *first* comes with Rust out of the box)

std::thread

```
fn concurrent_web_fetch() -> Vec<::std::thread::JoinHandle<()>> {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method


    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
        "http://imgur.com", "http://mozilla.org"];
    let mut handles = Vec::new();
    for site_ref in sites {
        let site = *site_ref;
        let handle = ::std::thread::spawn(move || {
            // block code put in closure: ~~~~~~~
            let client = Client::new();

            let res = client.get(site).send().unwrap();
            assert_eq!(res.status, hyper::Ok);
            let char_count = res.chars().count();
            println!("site: {} chars: {}", site, char_count);
        });

        handles.push(handle);
    }

    return handles;
}
```

dispatch

```
fn concurrent_gcd_fetch() -> Vec<::dispatch::Queue> {
    use hyper::{self, Client};
    use std::io::Read; // pulls in `chars` method
    use dispatch::{Queue, QueueAttribute};

    let sites = &["http://www.eff.org/", "http://rust-lang.org/",
        "http://imgur.com", "http://mozilla.org"];
    let mut queues = Vec::new();
    for site_ref in sites {
        let site = *site_ref;
        let q = Queue::create("qcon2016", QueueAttribute::Serial);
        q.async(move || {
            let client = Client::new();

            let res = client.get(site).send().unwrap();
            assert_eq!(res.status, hyper::Ok);
            let char_count = res.chars().count();
            println!("site: {} chars: {}", site, char_count);
        });

        queues.push(q);
    }

    return queues;
}
```

crossbeam

lock-free data structures

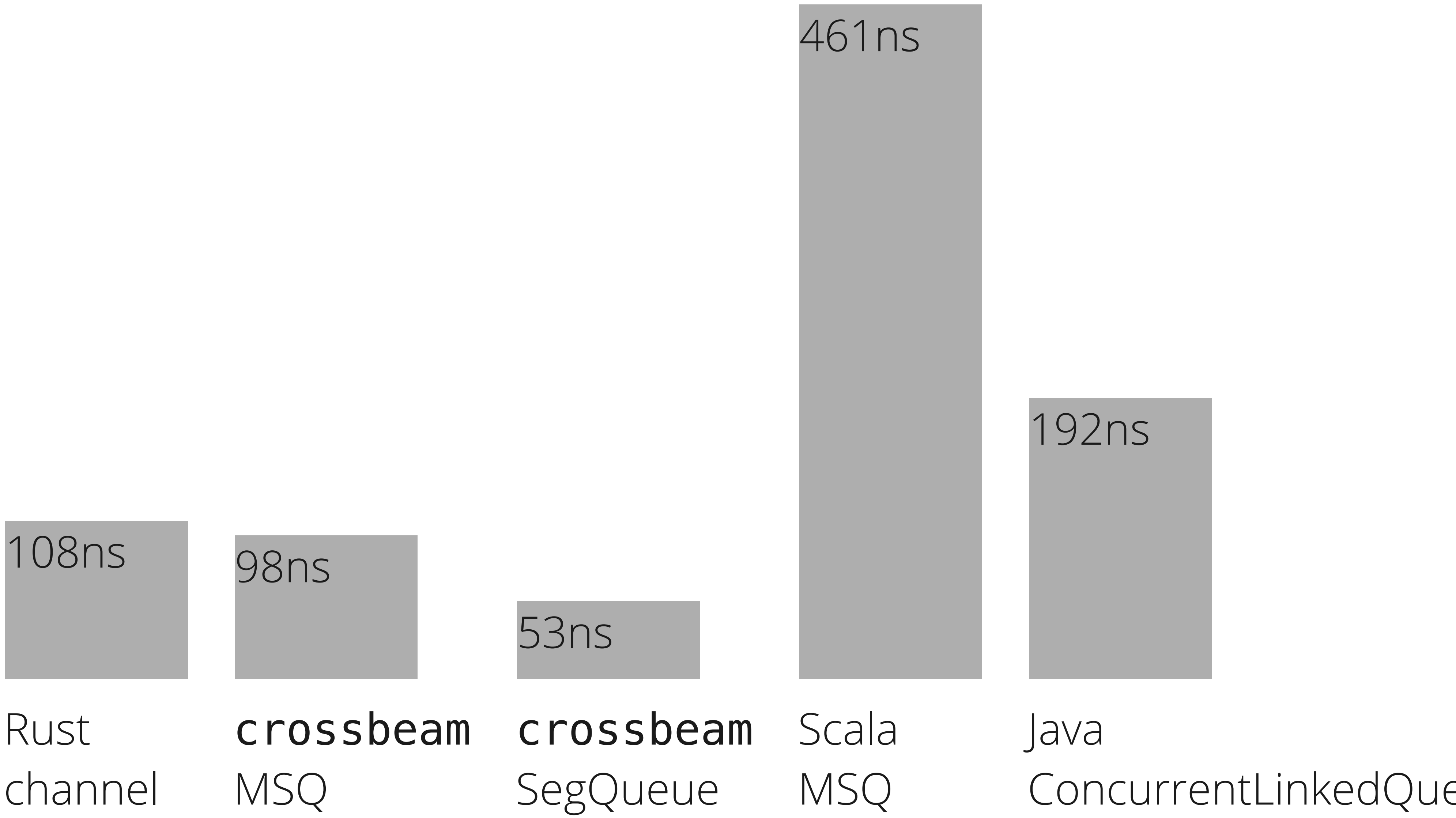
scoped threading abstraction

upholds Rust's safety (data-race freedom) guarantees

lock-free data structures

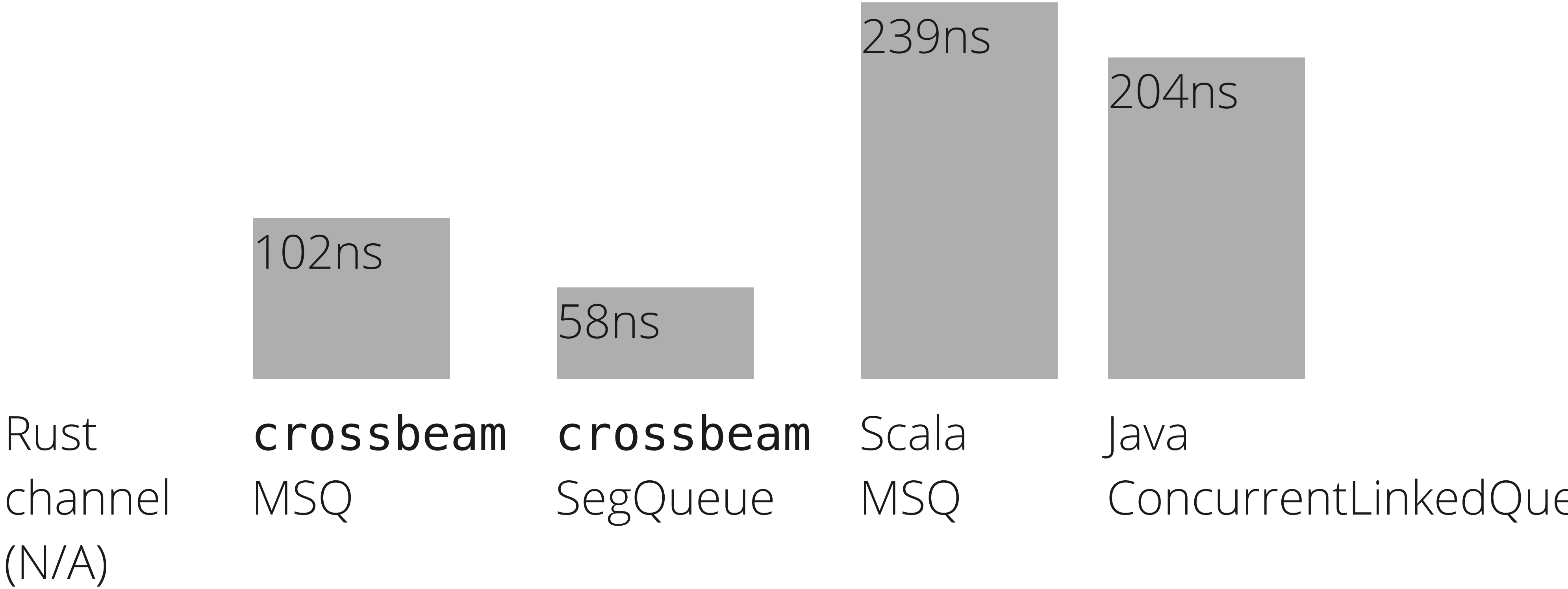
`crossbeam` MPSC benchmark

mean ns/msg (2 producers, 1 consumer; msg count 10e6; 1G heap)



`crossbeam` MPMC benchmark

mean ns/msg (2 producers, 2 consumers; msg count 10e6; 1G heap)



See "Lock-freedom without garbage collection"

`https://aturon.github.io/blog/2015/08/27/epoch/`

scoped threading?

`std::thead` does not allow sharing stack-local data

```
fn std_thread_fail() {
    let array: [u32; 3] = [1, 2, 3];

    for i in &array {
        ::std::thread::spawn(|| {
            println!("element: {}", i);
        });
    }
}
```

```
error: `array` does not live long enough
```

`crossbeam` scoped threading

```
fn crossbeam_demo() {
    let array = [1, 2, 3];

    ::crossbeam::scope(|scope| {
        for i in &array {
            scope.spawn(move || {
                println!("element: {}", i);
            });
        }
    });
}
```

`::crossbeam::scope` enforces parent thread joins on all spawned children before returning

ensures that it is sound for children to access local references passed into them.

crossbeam `scope`: "must-be concurrency"

Each `scope.spawn(..)` invocation fires up a fresh thread

(Literally just a wrapper around `std::thread`) `rayon`: "maybe parallelism"

rayon demo 1: map reduce

Sequential

```
fn demo_map_reduce_seq(stores: &[Store], list: Groceries) -> u32 {
    let total_price = stores.iter()
                            .map(|store| store.compute_price(&list))
                            .sum();
    return total_price;
}
```

Parallel (*potentially*)

```
fn demo_map_reduce_par(stores: &[Store], list: Groceries) -> u32 {
    let total_price = stores.par_iter()
                            .map(|store| store.compute_price(&list))
                            .sum();
    return total_price;
}
```

Rayon's Rule

the decision of whether or not to use parallel threads is made dynamically, based on whether idle cores are available

i.e., solely for offloading work, *not* for when concurrent operation is necessary for correctness

(uses work-stealing under the hood to distribute work among a fixed set of threads)

rayon demo 2: quicksort

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                    || quick_sort(hi));
    }
}
```

```
fn partition<T:PartialOrd+Send>(v: &mut [T]) -> usize {
    // see https://en.wikipedia.org/wiki/
    //     Quicksort#Lomuto_partition_scheme
    ...
}
```

rayon demo 3: buggy quicksort

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                    || quick_sort(hi));
    }
}
```

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                    || quick_sort(lo));
        //                       ~~ data race!
    }
}
```

(See blog post "Rayon: Data Parallelism in Rust" `bit.ly/1IZcku4`)

Big Idea

3rd parties identify (and provide) *new abstractions* for concurrency and parallelism unanticipated in std lib.

Soundness and 3rd Party Concurrency

The Secret Sauce

`Send`

`Sync`

lifetime bounds

Send and Sync

`T: Send` means an instance of `T` can be *transferred* between threads

(i.e. move or copied as appropriate)

`T: Sync` means two threads can safely *share* a reference to an instance of `T`

Examples

`T: Send` : `T` can be *transferred* between threads

`T: Sync` : two threads can *share* refs to a `T`

`String` is `Send`

`Vec<T>` is `Send` (if `T` is `Send`)

(double-check: why not require `T: Sync` for `Vec<T>: Send`?)

`Rc<T>` is *not* `Send` (for any `T`)

but `Arc<T>` *is* `Send` (if `T` is `Send` and `Sync`)

(to ponder: why require `T:Send` for `Arc<T>`?)

`&T` is `Send` if `T: Sync`

`&mut T` is `Send` if `T: Send`

Send and Sync are only half the story

other half is lifetime bounds; come see me if curious

Sharing Code: Cargo

Sharing Code

`std::thread` is provided with std lib

But `dispatch`, `crossbeam`, and `rayon` are 3rd party

(not to mention `hyper` and a host of other crates used in this talk's construction)

What is Rust's code distribution story?

Cargo

`cargo` is really simple to use

```
cargo new     -- create a project
cargo test    -- run project's unit tests
cargo run     -- run binaries associated with project
cargo publish -- push project up to crates.io
```

Edit the associated `Cargo.toml` file to:

- add dependencies
- specify version / licensing info
- conditionally compiled features
- add build-time behaviors (e.g. code generation)

"What's this about `crates.io`?"

crates.io

Open-source crate distribution site

Has every version of every crate

Cargo adheres to *semver*

Semver

The use of Semantic Versioning in `cargo` basically amounts to this:

Major versions (MAJOR.minor.patch) are free to break whatever they want.

New public API's can be added with minor versions updates (major.MINOR.patch), as long as they do not impose breaking changes.

In Rust, breaking changes *includes* data-structure representation changes.

Adding fields to structs (or variants to enums) can cause their memory representation to change.

Why major versions can include breaking changes

Cargo invokes the Rust compiler in a way that salts the symbols exported by a compiled library.

This ends up allowing two distinct (major) versions of a library to be used *simultaneously* in the same program.

This is important when pulling in third party libraries.

Fixing versions

`cargo` generates a `Cargo.lock` file that tracks the versions you built the project with

Intent: application (i.e. final) crates should check their `Cargo.lock` into version control

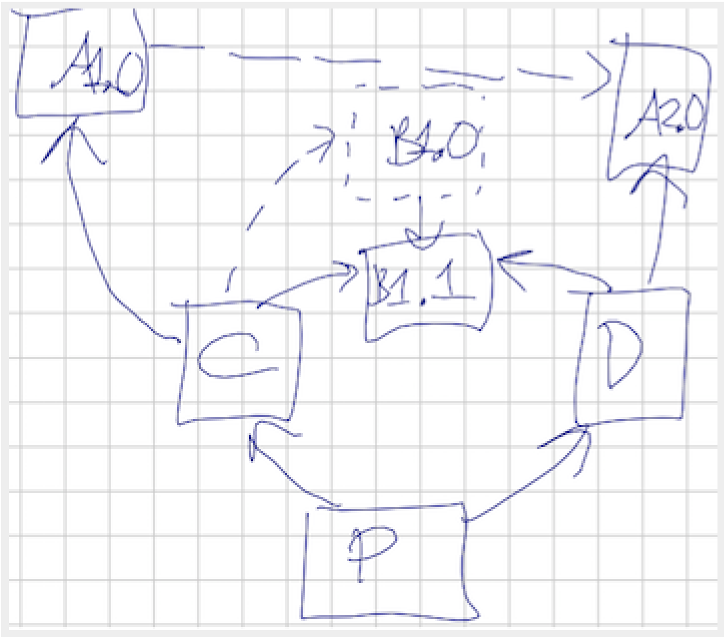
Ensures that future build attempts will choose the *same* versions

However: library (i.e. intermediate) crates should *not* check their `Cargo.lock` into version control.

Instead, everyone should follow sem.ver.; then individual applications can mix different libraries into their final product, upgrading intermediate libraries as necessary

Crate dependency graph

Compiler ensures one cannot pass struct defined via **X** version 2.x.y into function expecting **X** version 1.m.n, or vice versa.



A: Graph Structure **B**: Token API

C: Lexical Scanner **D**: GLL Parser **P**: Linked Program

In Practice

If you (*) follow the sem.ver. rules, then you do not usually have to think hard about those sorts of pictures.

"you" is really "you and all the crates you use"

You may not believe me, but `cargo` is really simple to use
Coming from a C/C++ world, this feels like magic
(probably feels like old hat for people used to package dependency managers)

Final Words

Final Words

(and no more pictures)

Interop

Rust to C

easy: `extern { ... }` and `unsafe { ... }`

C to Rust

easy: `#[no_mangle] extern "C" fn foo(...) { ... }`

Ruby, Python, etc to Rust

see e.g. `https://github.com/wycats/rust-bridge`

Customers

Mozilla (of course)

Skylight

MaidSafe

… others

Pivot from C/C++ to Rust

Maidsafe is one example of this

Rust as enabler of individuals

From "mere script programmer" to "lauded systems hacker"

Or if you prefer:

Enabling *sharing* systems hacking knowledge with everyone

Programming in Rust has made me look at C++ code in a whole new light

---

Thanks

`www.rust-lang.org`

Hack Without Fear