Introduction to Rust

Ken Friis Larsen

`kflarsen@diku.dk`

Department of Computer Science, University of Copenhagen (DIKU)

(some material borrowed from Aaron Turon)

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

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

Today’s Program

- What is safe systems programming, in Rust
- Rust’s building blocks for concurrency

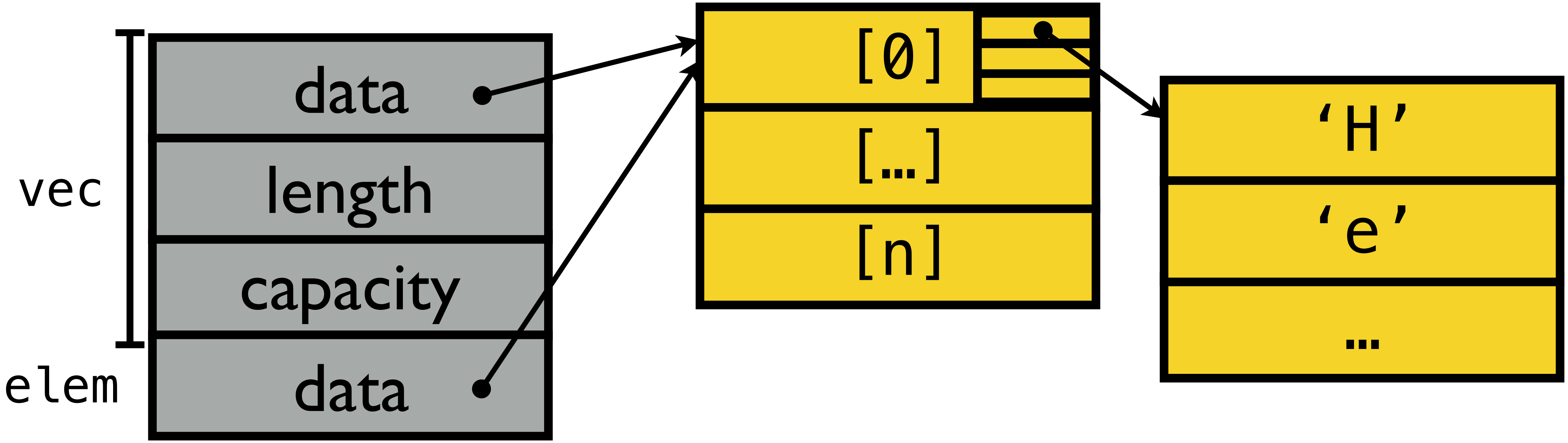
Influences

- Programming languages:
  - C/C++
  - ML and Haskell

- Driving application: Servo (browser engine)
  - Browsers need **control**.
  - Browsers need **safety**.

Control

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

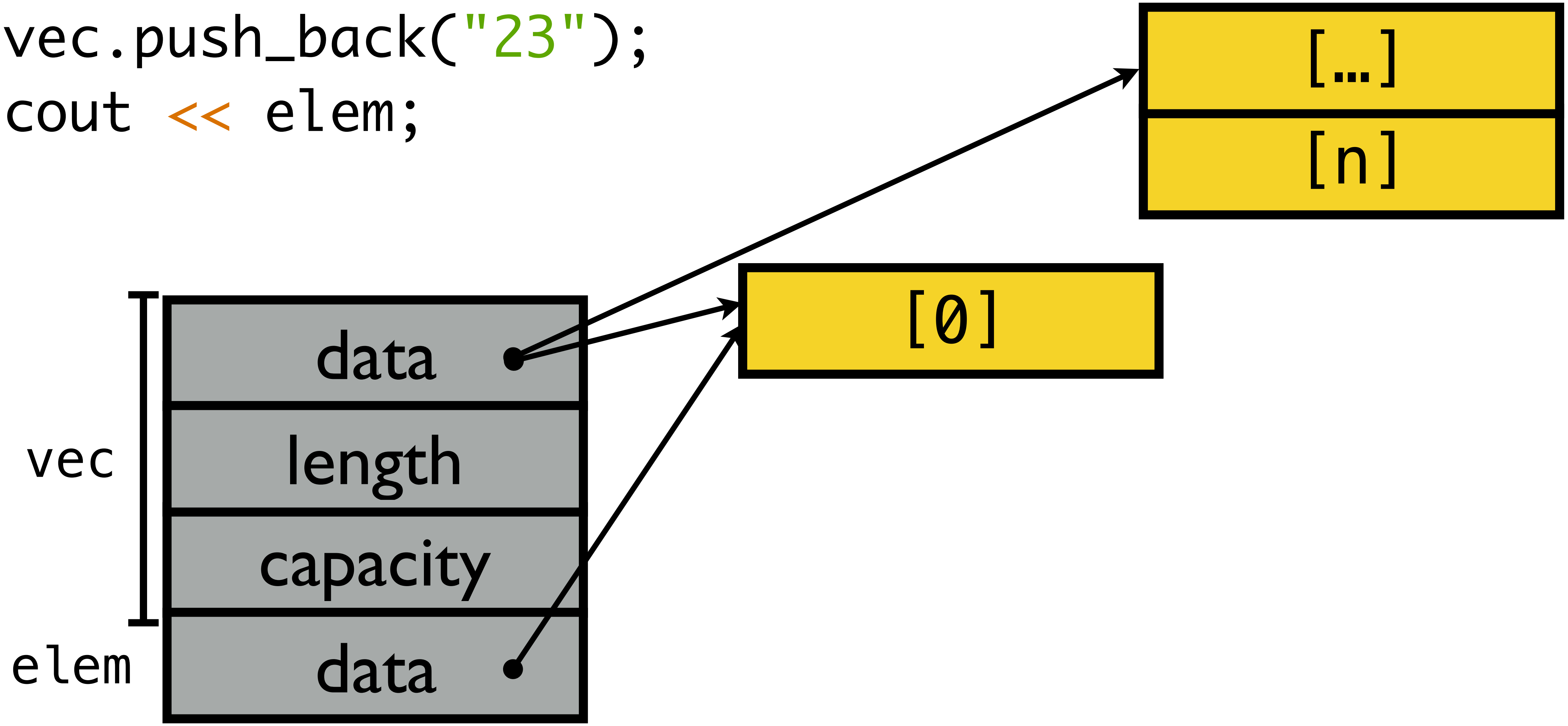


Zero-cost Abstraction

- Ability to define *abstractions* that *optimize away to nothing*.
- Example: Java's ArrayList vs. C++'s vectors

Safety

```
void example () {
  std::vector<std::string> vec;
  …
  auto& elem = vec[0];
  vec.push_back("23");
  cout << elem;
}
```



Memory Management

- *Garbage Collection* takes care of *memory management* and *memory accounting* for you
- *Memory management:* allocate memory and free it.
- *Memory accounting:* when is it OK to free memory

Memory Management in Different Languages

- *Garbage Collected:* Java, C#, Haskell, F#, Python
- *Manual management and accounting:* C, assembler, C++
- *Semi-automatic accounting:* C++, Swift, Objective-C
- *Static accounting:* Rust, MLKit, Cyclone

Why not a Garbage Collector?

- No **control**.
- Requires a **runtime**.
- **Insufficient** to prevent related problems: iterator invalidation, data races, and others.

Rust’s Solution

Type system enforces *ownership* and *borrowing*:

1. All resources have a clear owner
2. Others can borrow from the owner
3. Owner cannot free or mutate the resource while it is borrowed

~~Alias~~ + ~~Mutation~~

Ownership

Ownership in Action

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

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

Ownership in Action

```
fn give() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    take(vec);
    vec.push(3);
}

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

Error!

Borrow in Action

```
fn lender() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    use(&vec);
    vec.push(3);
}

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

Mutable Borrow in Action

```
fn lender() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    use(&vec);
    vec.push(3);
}

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

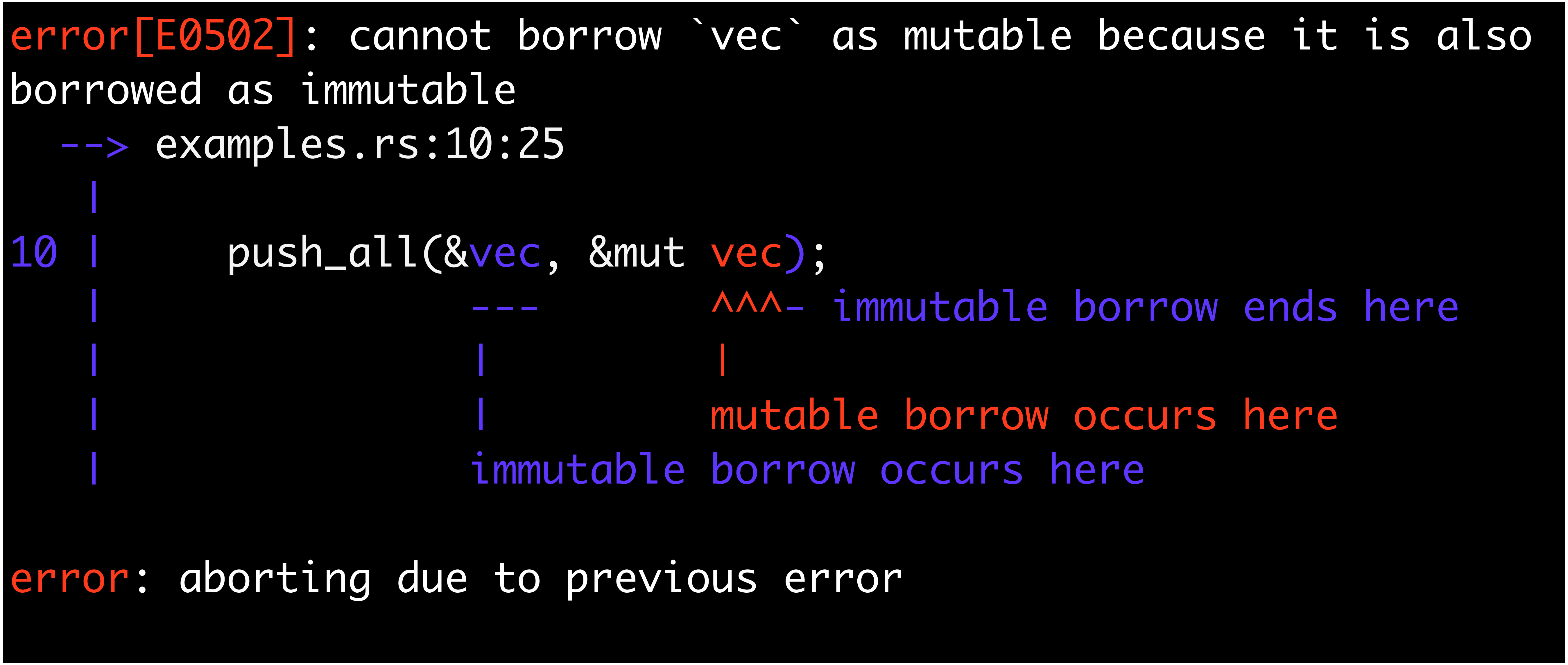
More Mutable Action

```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
  for elem in from {
    to.push(*elem);
  }
}
```

More Mutable Action

```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
  for elem in from {
    to.push(*elem);
  }
}

fn caller() {
    let mut vec = (0..10).collect();
    push_all(&vec, &mut vec);
}
```



```
error[E0502]: cannot borrow `vec` as mutable because it is also
borrowed as immutable
  --> examples.rs:10:25
   |
10 |     push_all(&vec, &mut vec);
   |                ---       ^^^- immutable borrow ends here
   |                |         |
   |                |         mutable borrow occurs here
   |                immutable borrow occurs here

error: aborting due to previous error
```

A `&mut` reference to a value is the only alias to that value

Concurrency

Concurrency in Rust

- **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**.
- In other words we have two threads that both have an **alias** to some memory, at least one of them is **mutating** the memory, and we don’t know the **ordering**.

Message Parsing

```
fn spawn_child() {
    let (tx, rx) = channel();
    thread::spawn(move || {
        let result = …;
        tx.send(result);
    });

    let res = rx.recv();
}
```

Just plain ownership transfer

Locked Mutable Access

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

You get the only active alias for data.

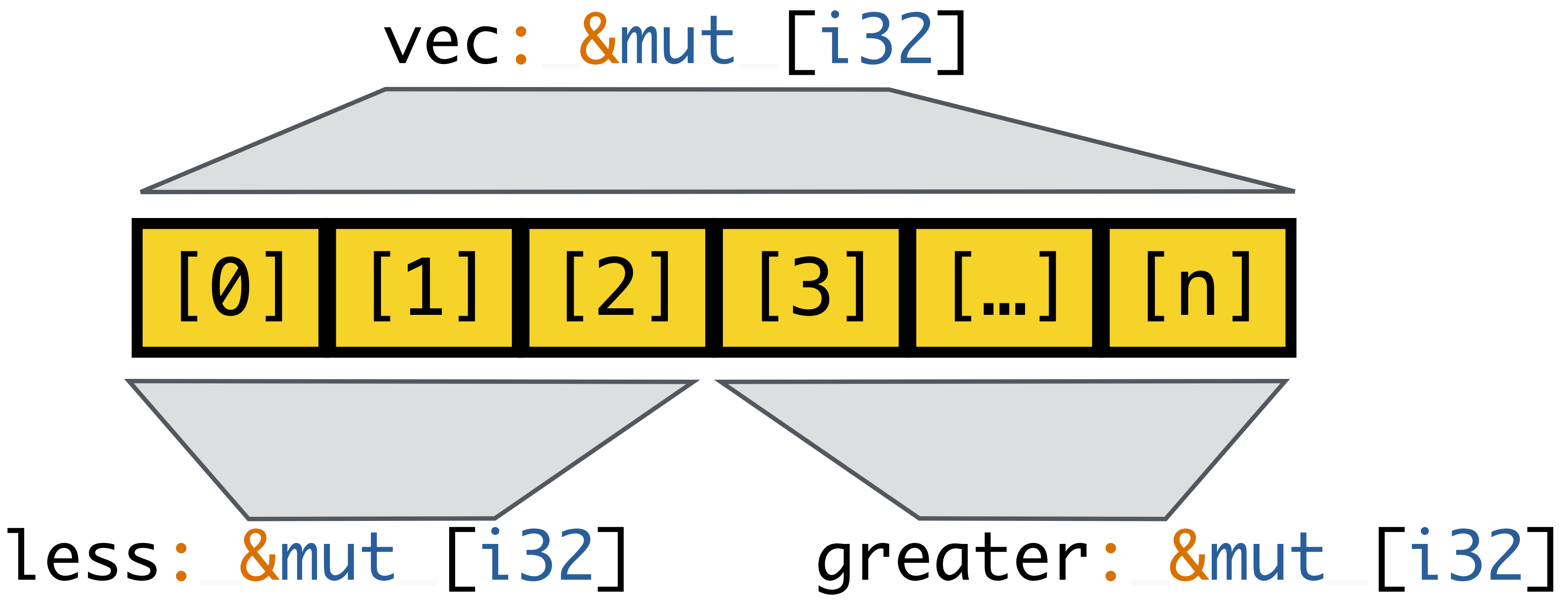
Mutex is unlocked at the end of scope.

Disjoint, Scoped Access

```
fn qsort(vec: &mut [i32]) {
    if vec.len() <= 1 { return; }
    let mid = partition(vec);
    let (less, greater) = vec.split_at_mut(mid);
    qsort(less);
    qsort(greater);
}
```

Disjoint, Scoped Access

```
fn qsort(vec: &mut [i32]) {
    if vec.len() <= 1 { return; }
    let mid = partition(vec);
    let (less, greater) = vec.split_at_mut(mid);
    qsort(less);
    qsort(greater);
}
```



`vec: &mut [i32]`

[0] [1] [2] [3] [...] [n]

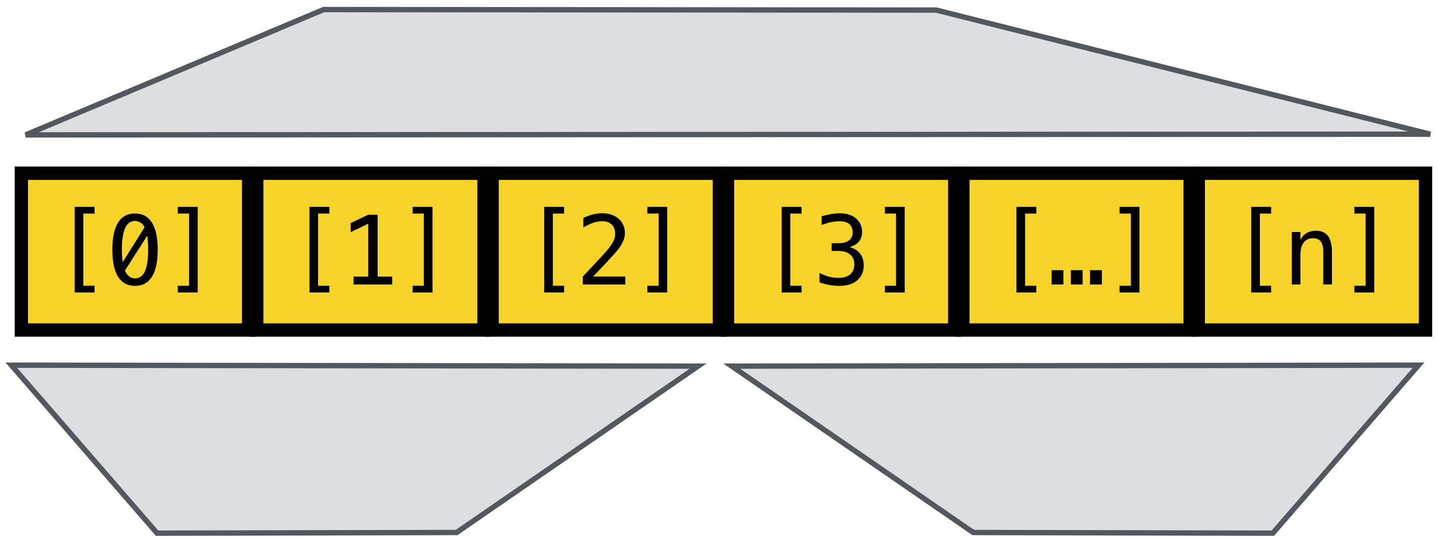
`less: &mut [i32]` `greater: &mut [i32]`

Checked by the Type System

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

Disjoint, Scoped Access in Parallel

```
fn parallel_qsort(vec: &mut [int]) {
    if vec.len() <= 1 { return; }
    let mid = partition(vec);
    let (less, greater) = vec.split_at_mut(mid);
    parallel::join(
        || parallel_qsort(less),
        || parallel_qsort(greater)
    );
}
```



`vec: &mut [i32]`

`less: &mut [i32]`

`greater: &mut [i32]`

Static checking for thread safety

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

```
Arc<Vec<i32>>: Send
Rc<Vec<i32>> : !Send
```

Takeways

- Rust's type system checks that you get memory account right, by keeping track of ownership and borrowing.
- Ownership and borrowing is also useful for concurrency. Good substrate for building new abstractions.
- Lots of things was not covered: data parallelism, atomic primitives, lock-free data structures, concurrent hashtables, futures, …

Foto af Colin Kinner

Questions