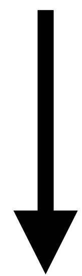
Memory Safety with Rust

Will Crichton

Memory management goal:

Allocate memory when you need it,
and free it when you’re done.

stack



heap

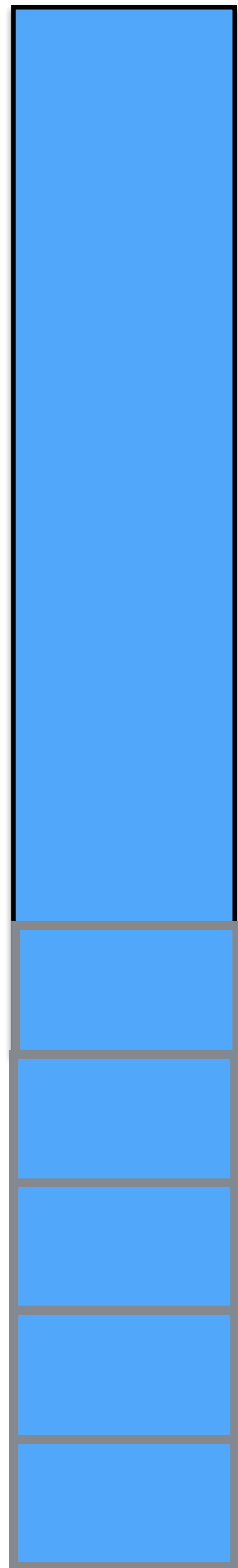
(uninitialized data) bss

(read-only data) rodata

data

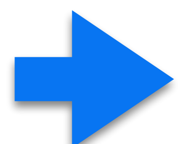
text

interrupt vectors

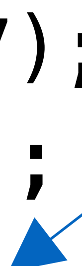


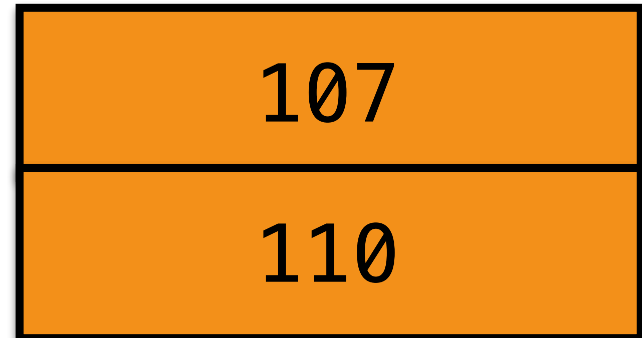
code/badc

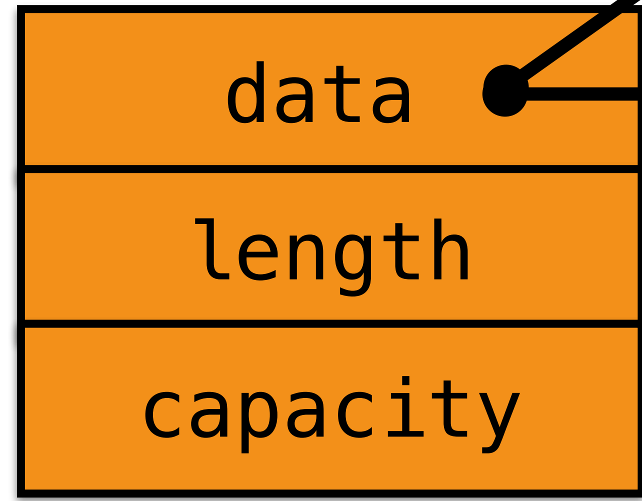
```
void main() {
  list* l = list_new();
  list_append(list, 107);
  int* n = &l->array[0];
  list_append(l, 110);
  printf(“%d”, *n);
}
```

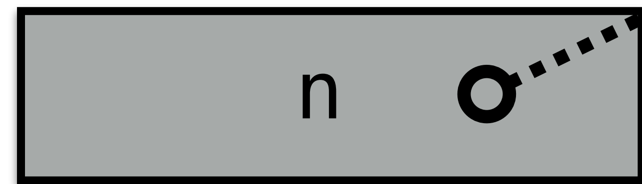


Mutating the vector freed old contents.









Dangling pointer: *pointer to freed memory.*

Aliasing: *more than one pointer to same memory.*

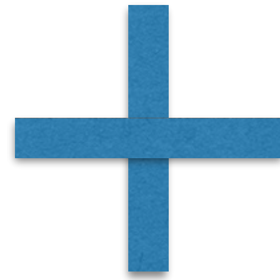
How can we solve this?

1. **Only delete objects when no references exist**
   - **Garbage collection**
   - **Java, Python, Javascript, Ruby, Haskell, …**

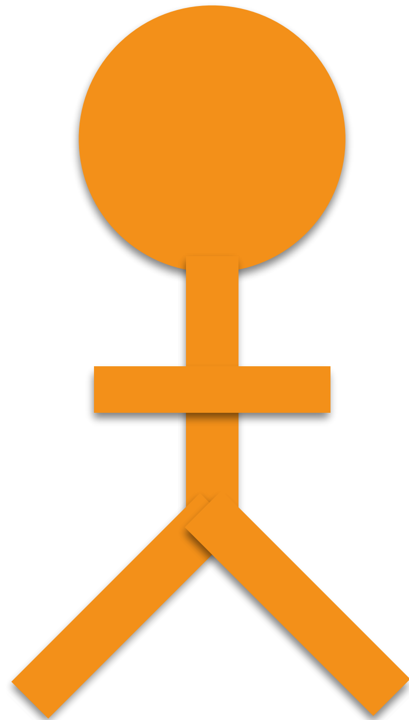
2. **Prevent simultaneous mutation and aliasing**

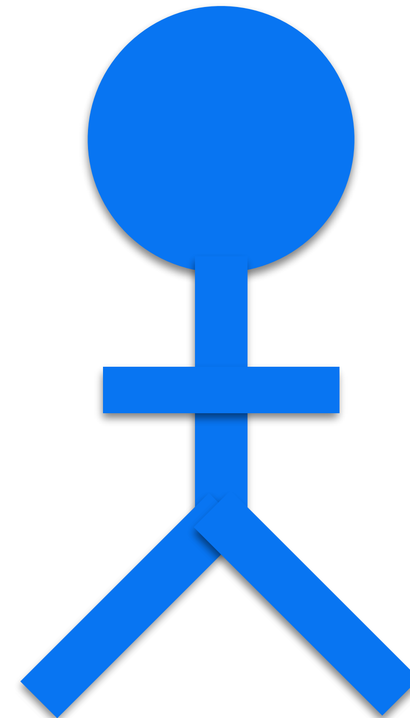
code/rust

~~Aliasing~~



Mutation



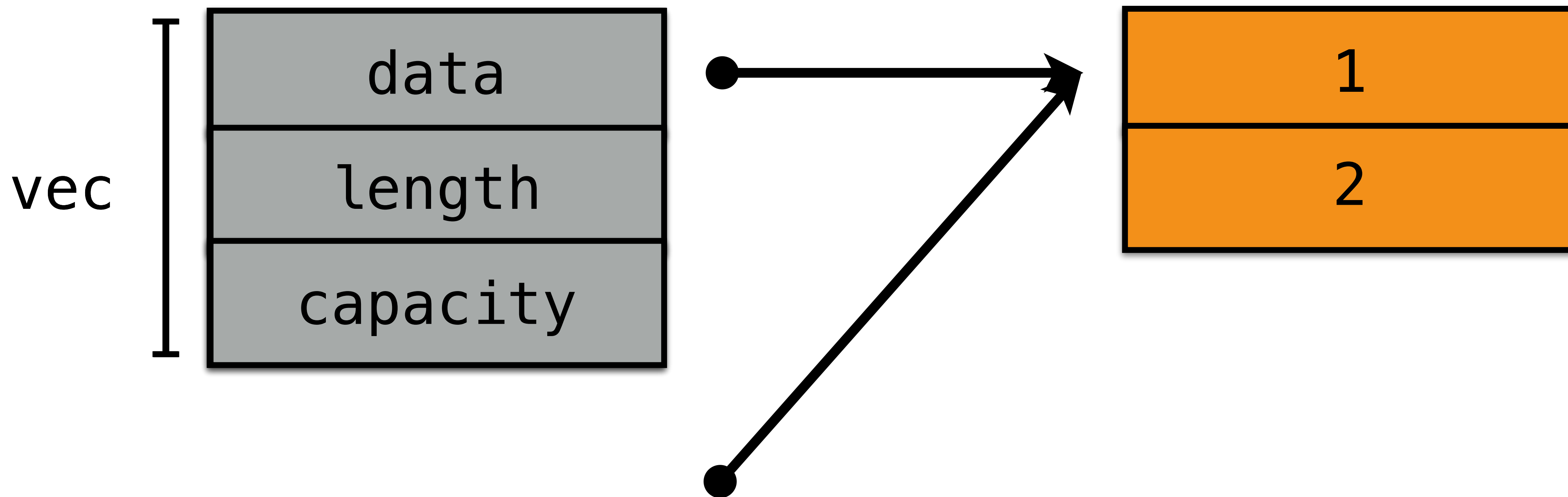


Ownership (T)

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

```rust
fn take(vec: Vec<i32>) {
    // …
}
```

Take ownership of a Vec<i32>



Compiler **enforces** moves

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

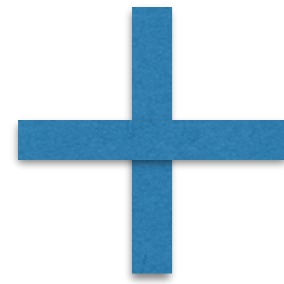
```
fn take(vec: Vec<i32>) {
    // …
}
```

Error: vec has been moved

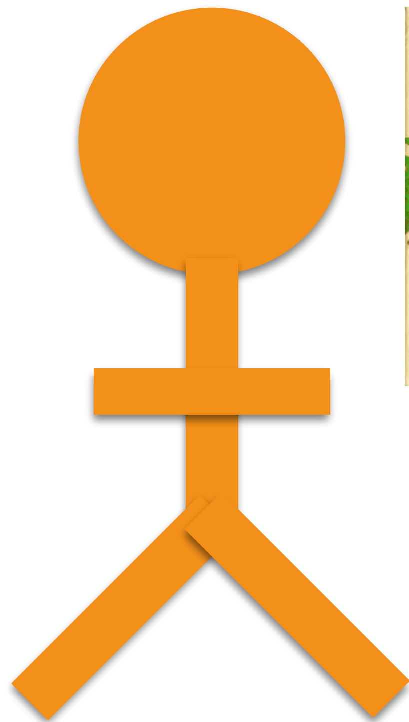
Prevents:

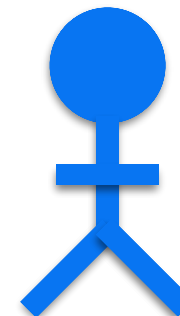
- use after free
- double moves
- …

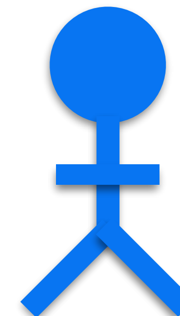
Aliasing



~~**Mutation**~~

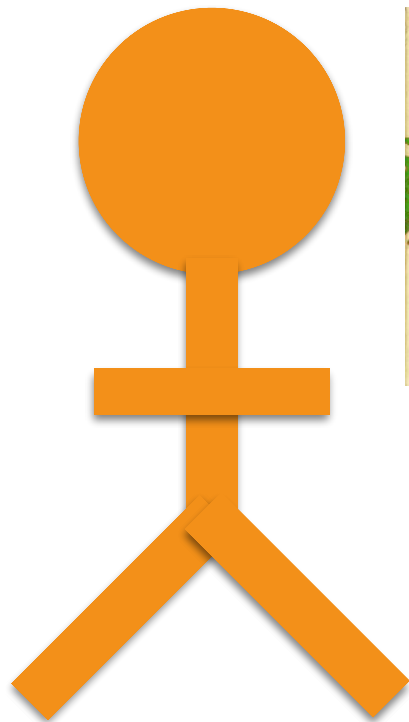


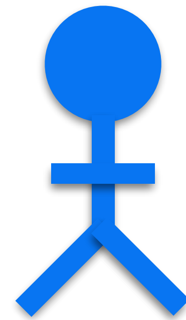


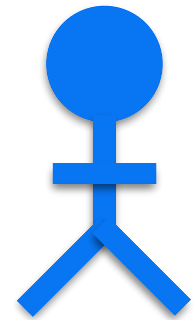


Shared borrow (&T)

~~Aliasing~~ Mutation







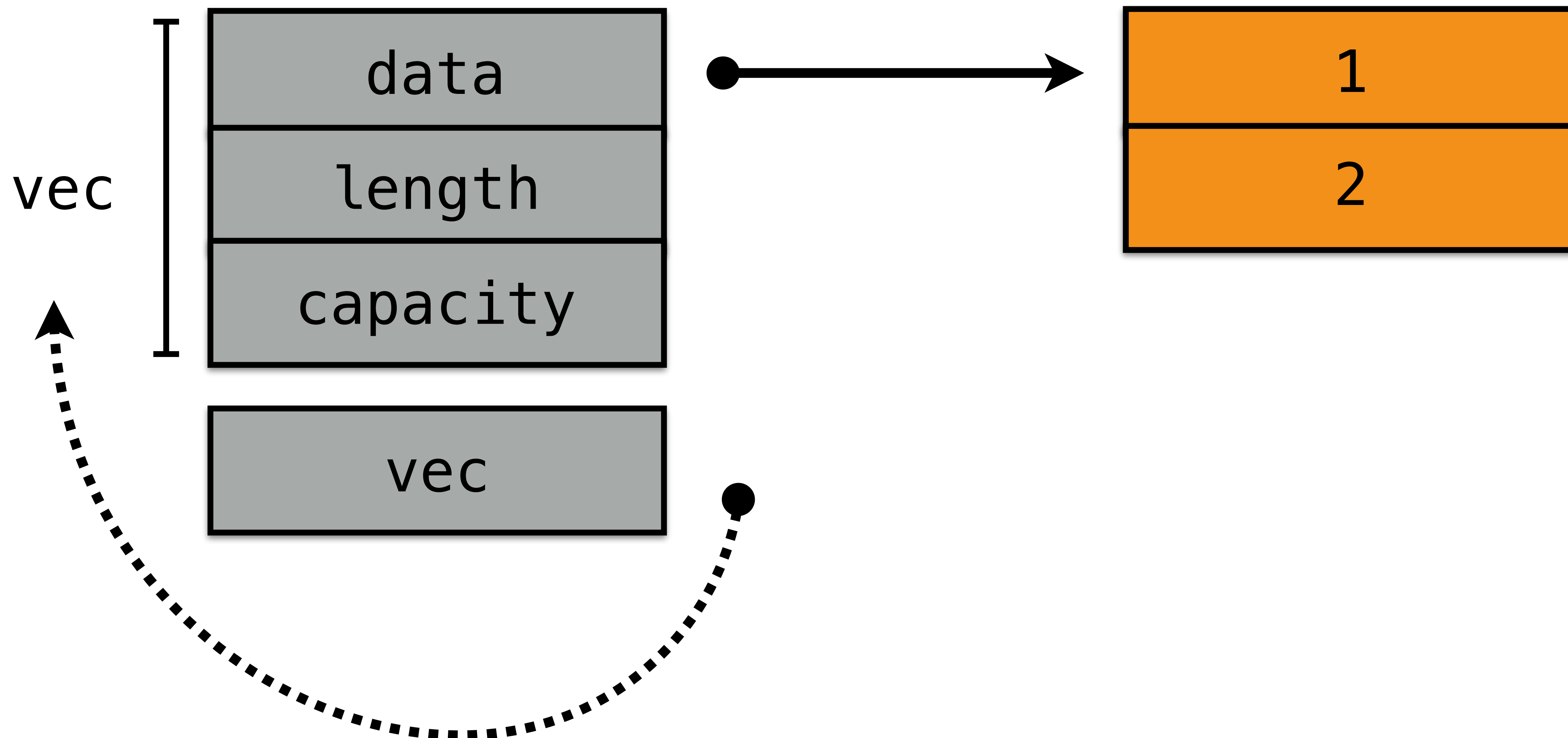
Mutable borrow (&mut T)

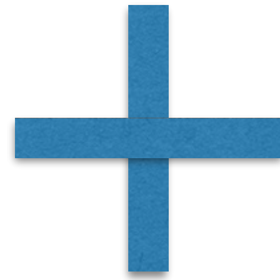
```
fn lender() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    use(&vec);
    …
}
```

Loan out vec

```
fn use(vec: &Vec<i32>) {
    // …
}
```

“Shared reference to Vec<i32>”



Aliasing  ~~Mutation~~

Shared references are **immutable**: *

```
fn use(vec: &Vec<i32>) {
  vec.push(3);
  vec[1] += 2;
}
```

Error: cannot mutate shared reference

* Actually: mutation only in controlled circumstances

Mutable references

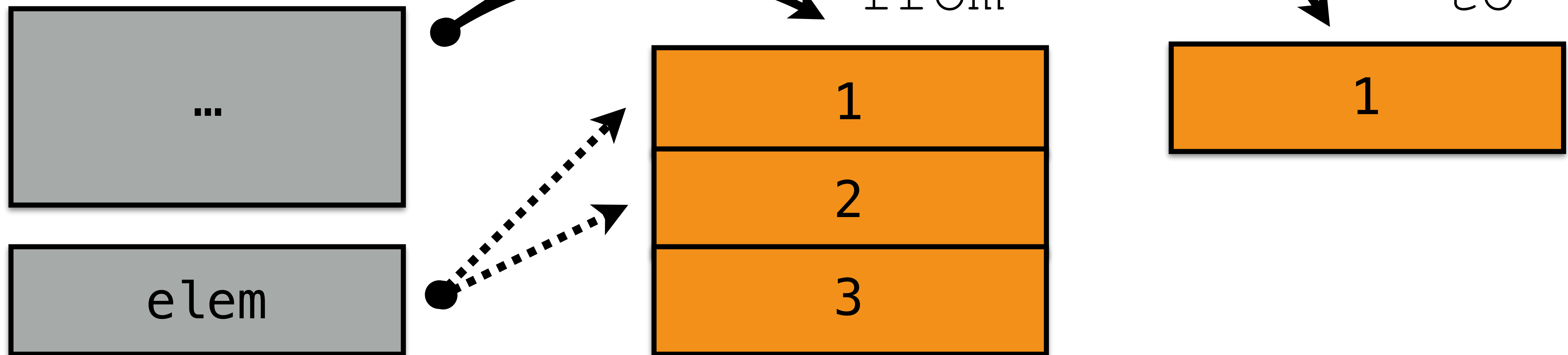
```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
  for elem in from.iter() {
    to.push(*elem);
  }
}
```

mutable reference to Vec<i32>

push() is legal

Iteration

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



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

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

…

elem

1
2
3

1
2
3
1

dangling pointer

```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {…}

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

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

Rust = less footguns

- **Memory errors can be subtle (hard to debug)**
- **Being principled pays off**
- **Take cs242 to learn more!**