Memory Safety with Rust

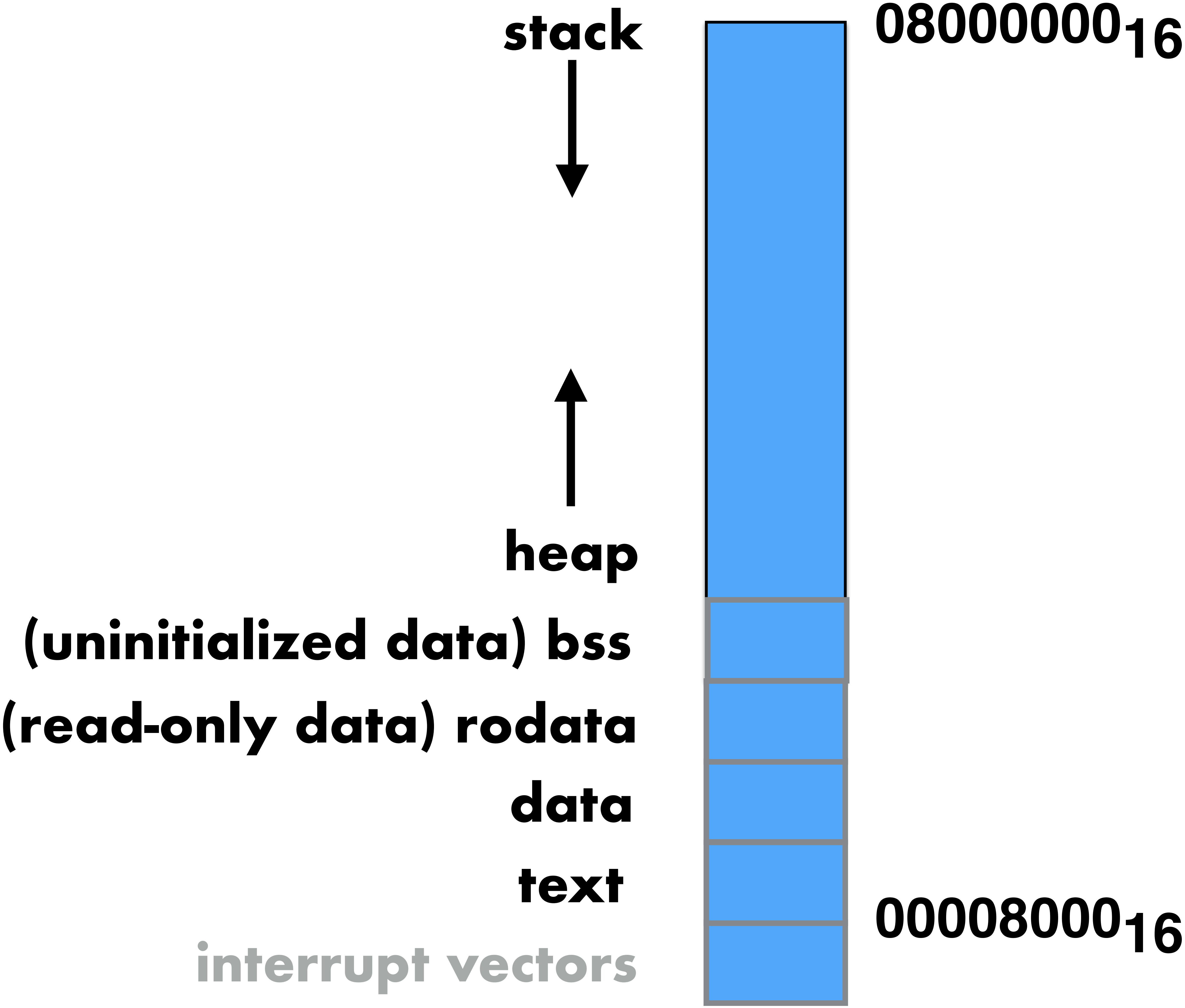
Will Crichton

Today’s goals

- When is memory allocated and deallocated?
- Where does memory live?
- What kinds of pointers does Rust have?

Memory management goal:

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



stack

08000000_{16}

heap

(uninitialized data) bss

(read-only data) rodata

data

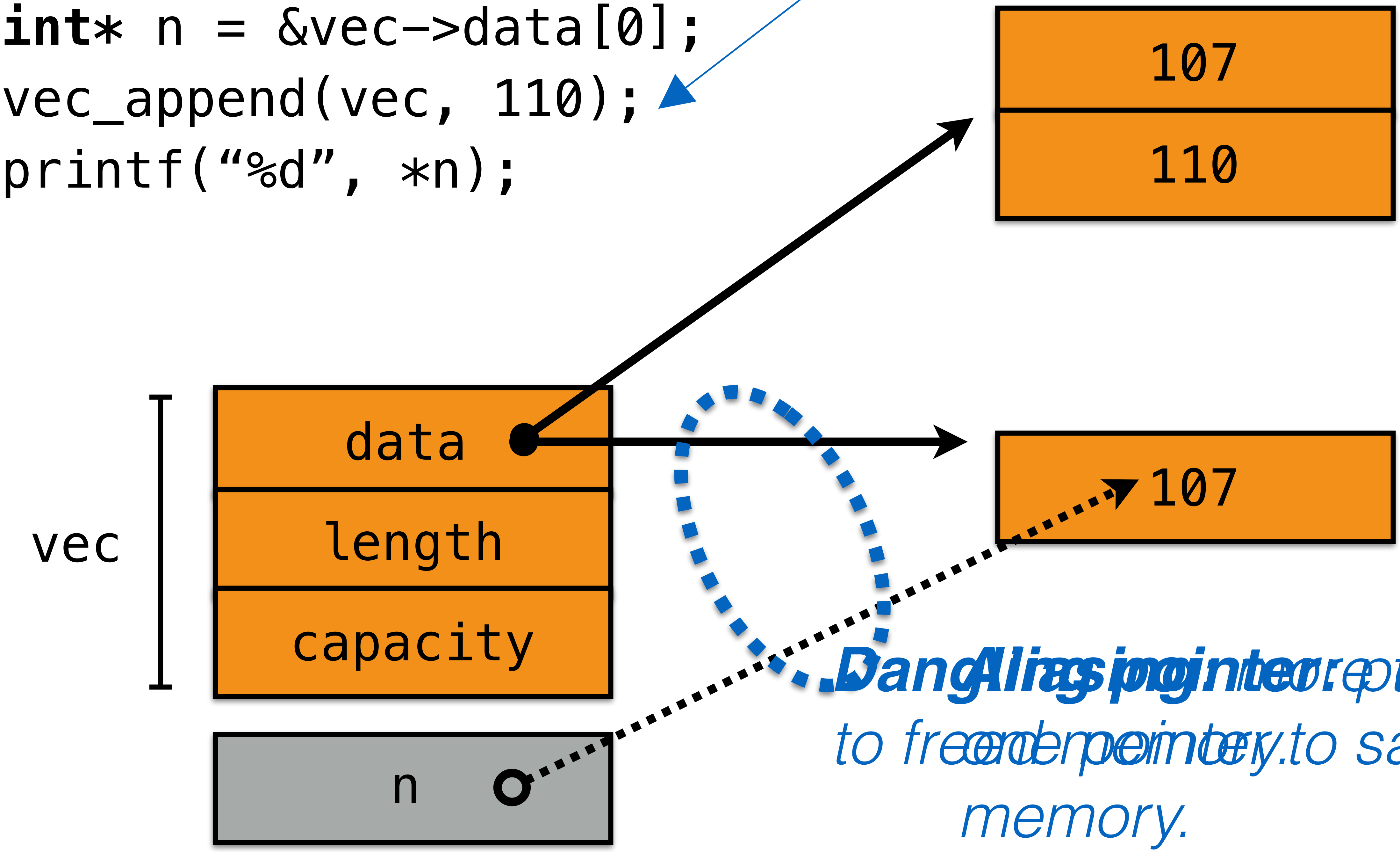
text

00008000_{16}

interrupt vectors

```
void main() {
   Vec* vec = vec_new();
   vec_append(vec, 107);
   int* n = &vec->data[0];
   vec_append(vec, 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

~~Aliasing~~ + Mutation

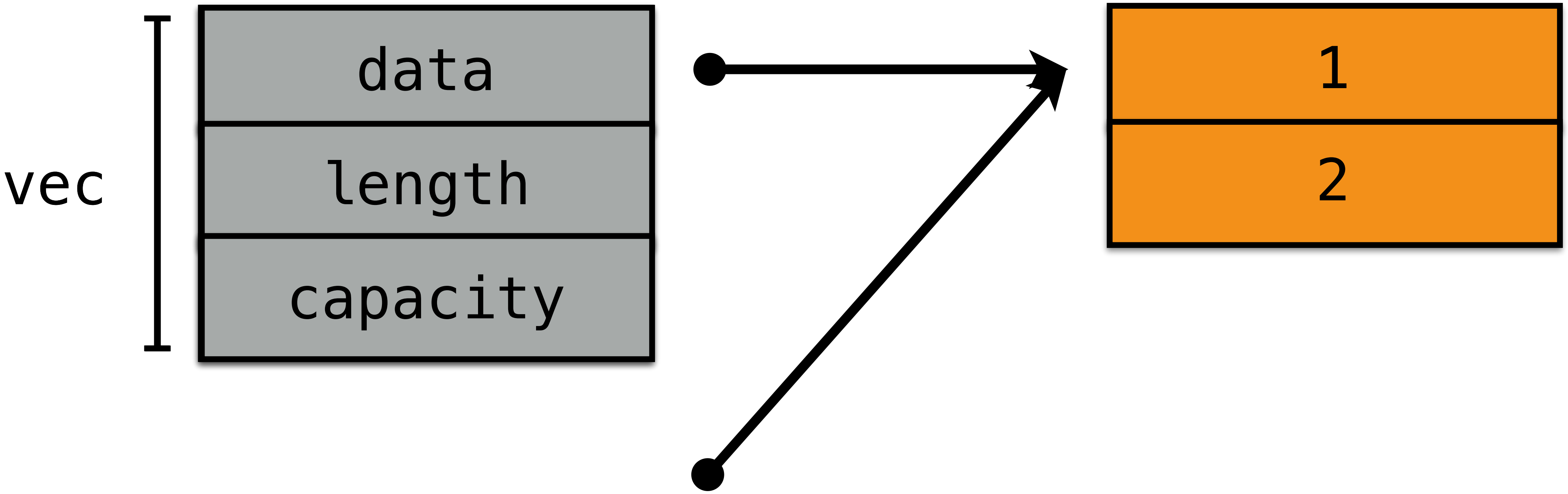
Principles of Compiler Design
Alfred V. Aho
Jeffrey D. Ullman

Ownership (T)

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

```
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:

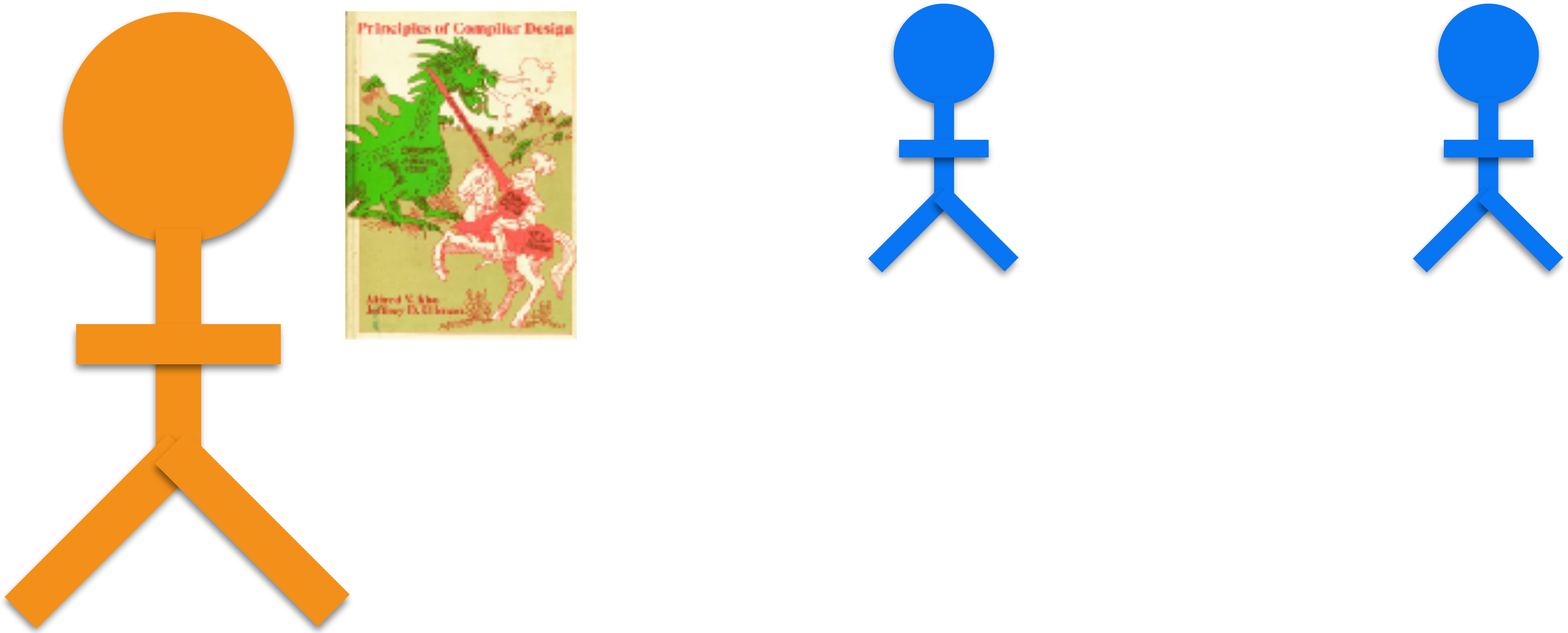
- use after free
- double moves
- …

Aliasing + ~~Mutation~~

Principles of Compiler Design
Alfred V. Aho
Jeffrey D. Ullman

Shared borrow (&T)

~~Aliasing~~ + Mutation



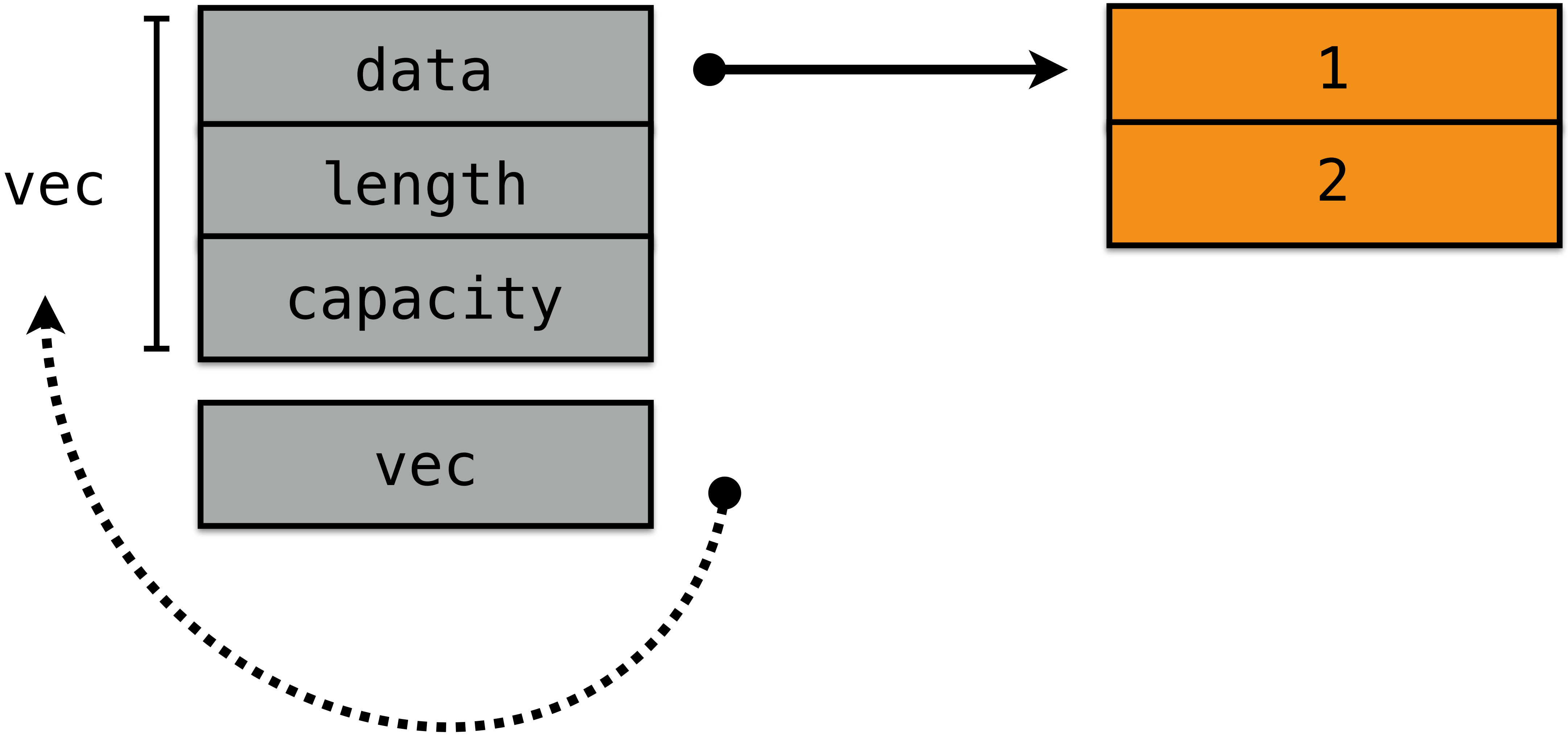
Mutable borrow (&mut T)

```
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

Mutable references

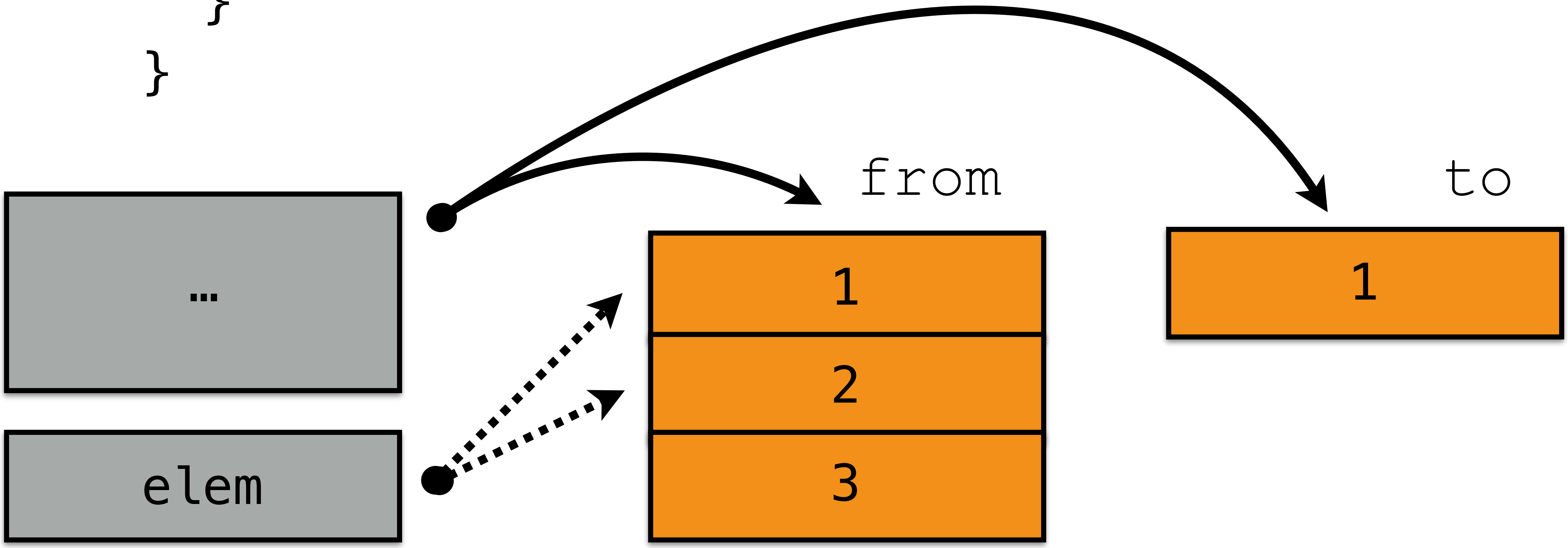
```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
  for elem in from.iter() {
    to.push(*elem);
  }
}
```

mutable reference to Vec<i32>

push() is legal

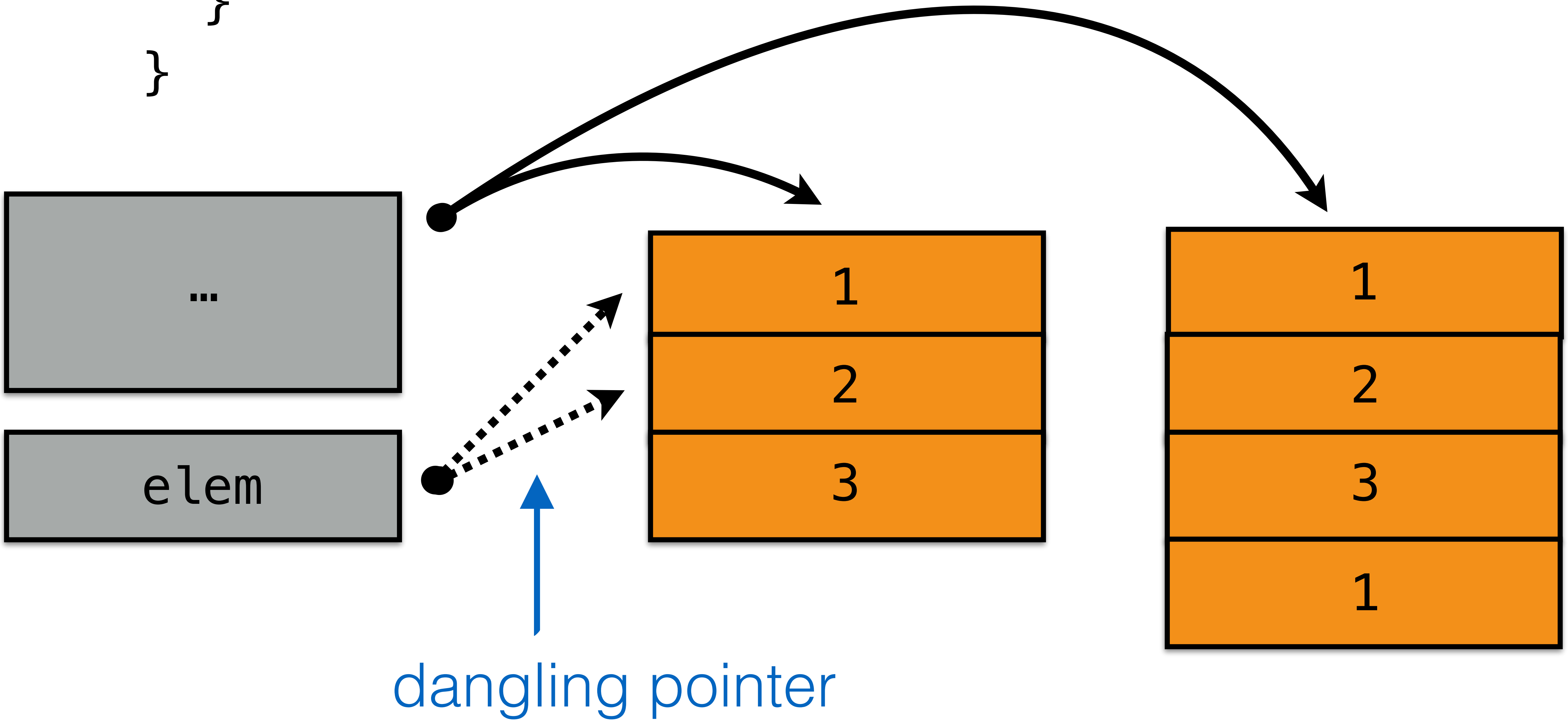
Iteration

```
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);
  }
}
```



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

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

shared reference

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

Lifetime of a value = lifetime of a name

```
fn main() {
  let x = 1;
  {
    let y = 2;
    let z = &x;
    // y and z deallocated, 2 gone
  }
  // x deallocated, 1 gone
}
```

What if I don’t know how long an object should live?

Where are my objects allocated?

C/C++ rules

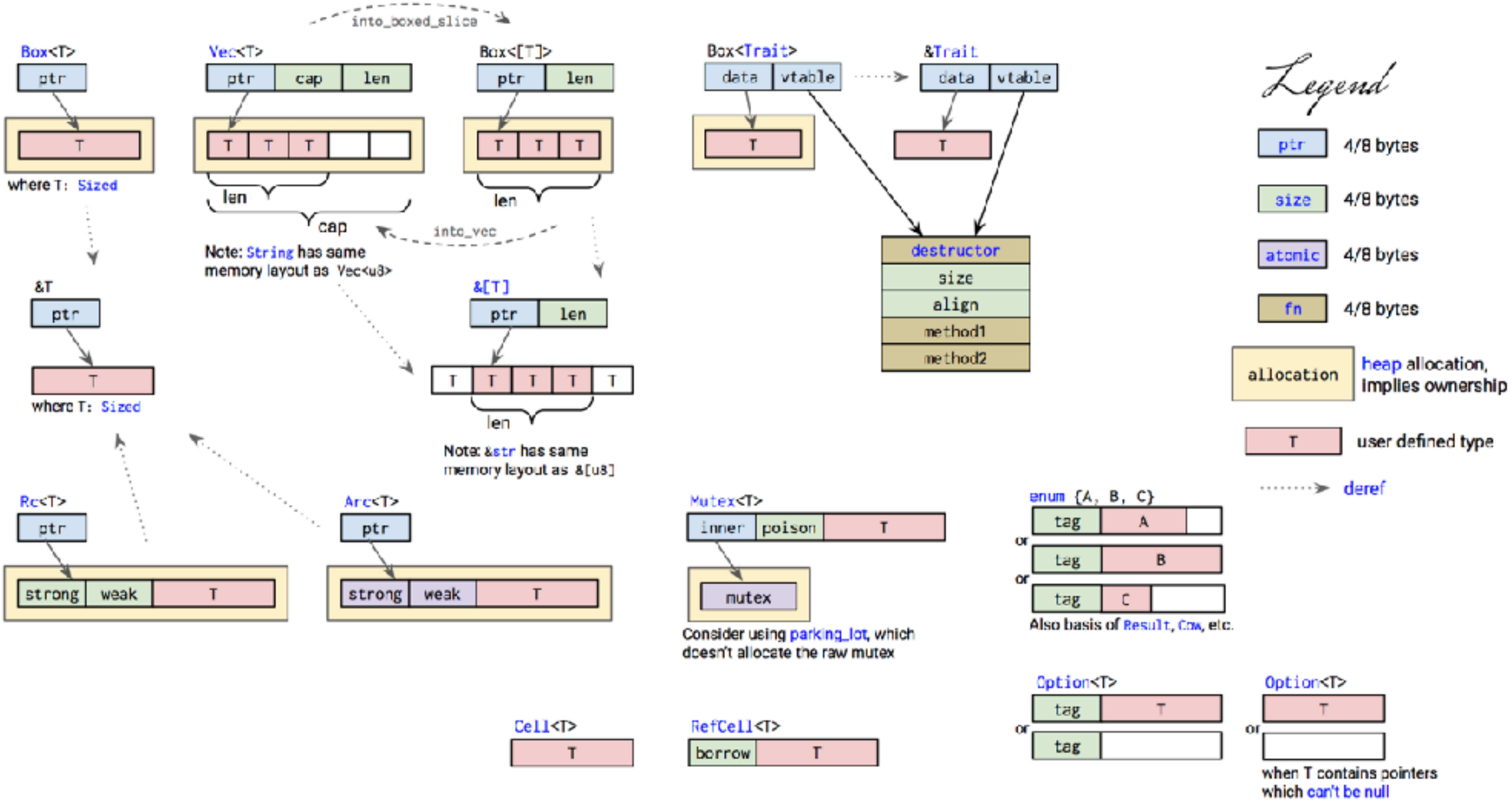
- **Variables are always on the stack**

- **Values on the stack by default**

- **malloc/new allocates on the heap**

Stack vs. heap

- Variables always reside on the stack, just like C
- Normal owned data (T) also on the stack
- Box<T>: pointer on stack to heap
- &T: pointer on stack to wherever T is



Note: String has same memory layout as Vec<u8>

Note: &str has same memory layout as &[u8]

Consider using parking_lot, which doesn't allocate the raw mutex

Also basis of Result, Cow, etc.

when T contains pointers which can't be null

Rust container cheat sheet, by Raph Levien, Copyright 2017 Google Inc., released under Creative Commons BY, 2017-04-21, version 0.0.3

Structs and closures