

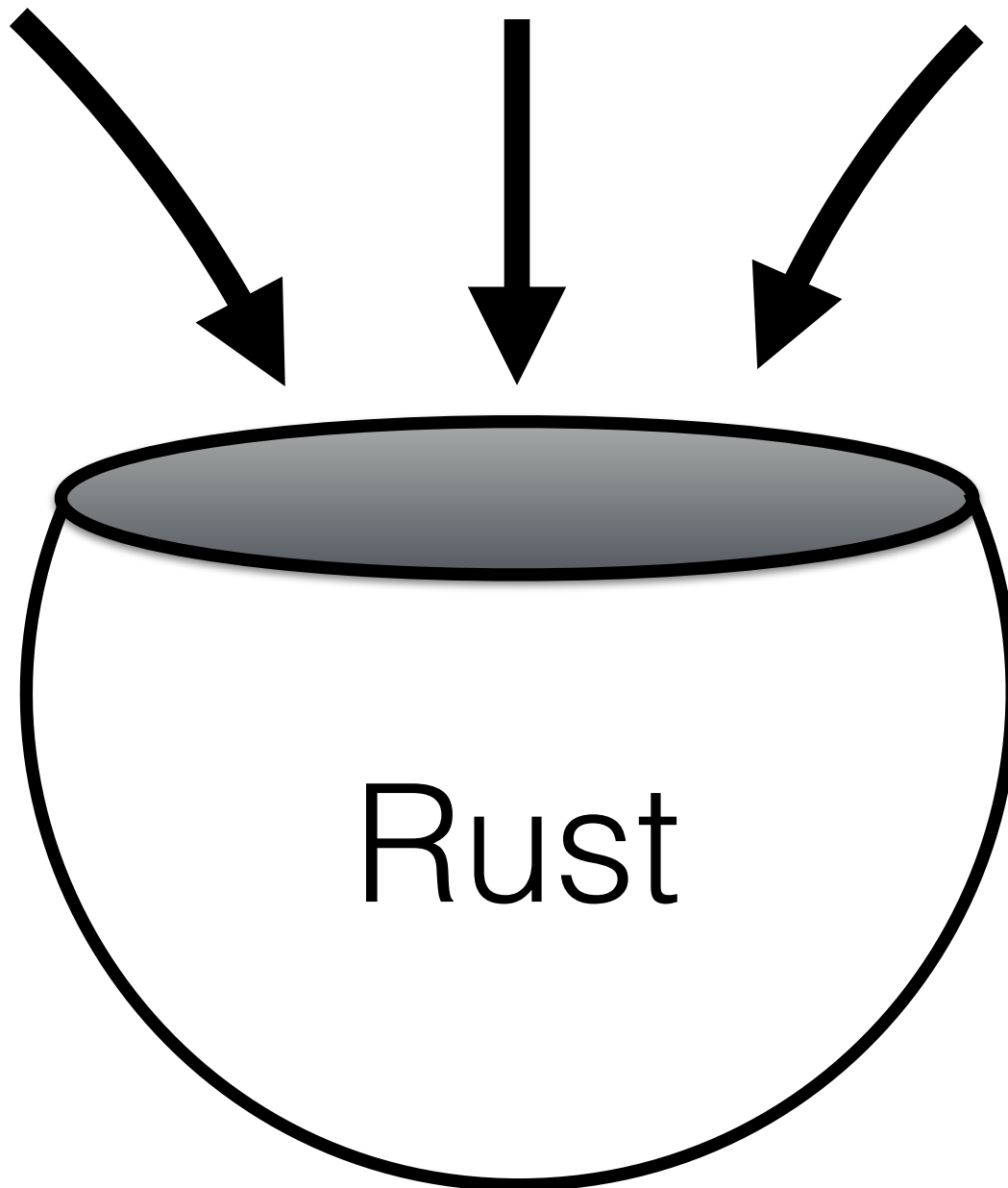


Aaron Turon
Mozilla Research

C/C++

...

ML/Haskell



Safe systems programming

Why Mozilla?

Browsers need **control**.

Browsers need **safety**.

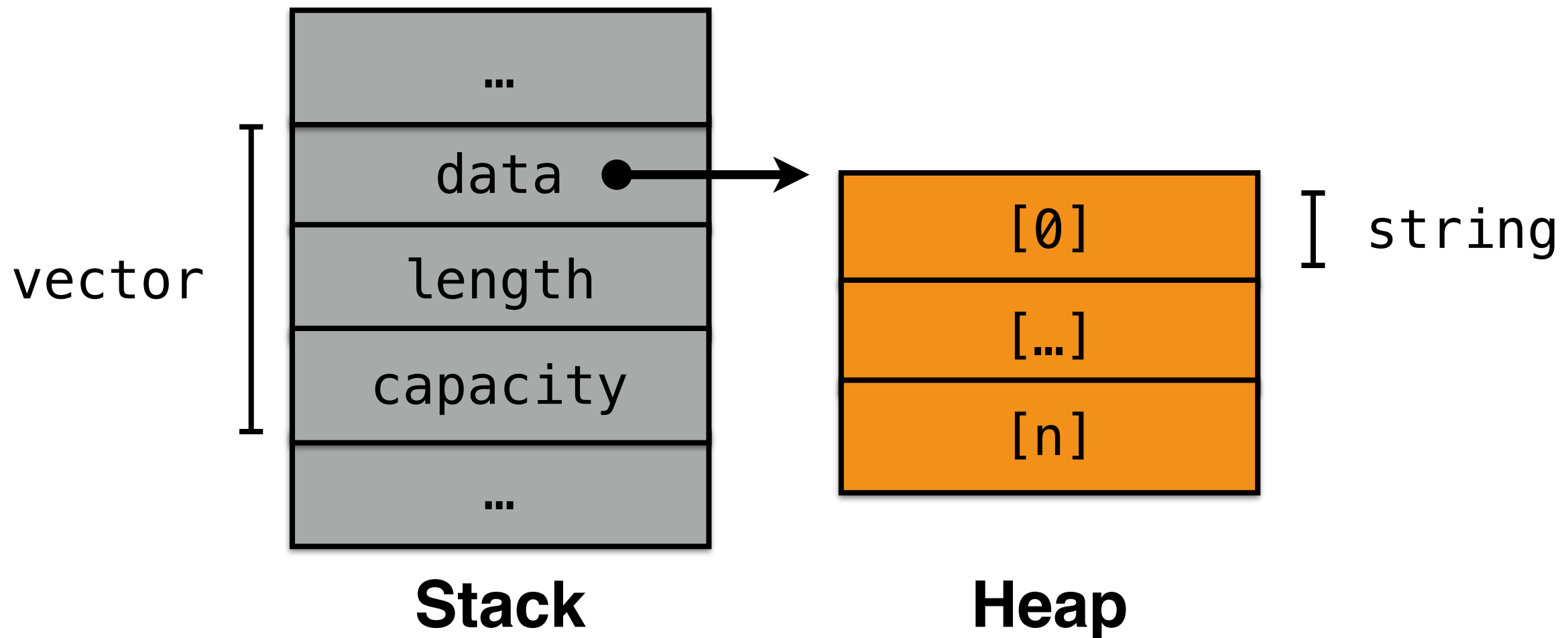
Servo: Next-generation
browser built in Rust.

What is **control**?

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

What is **control**?

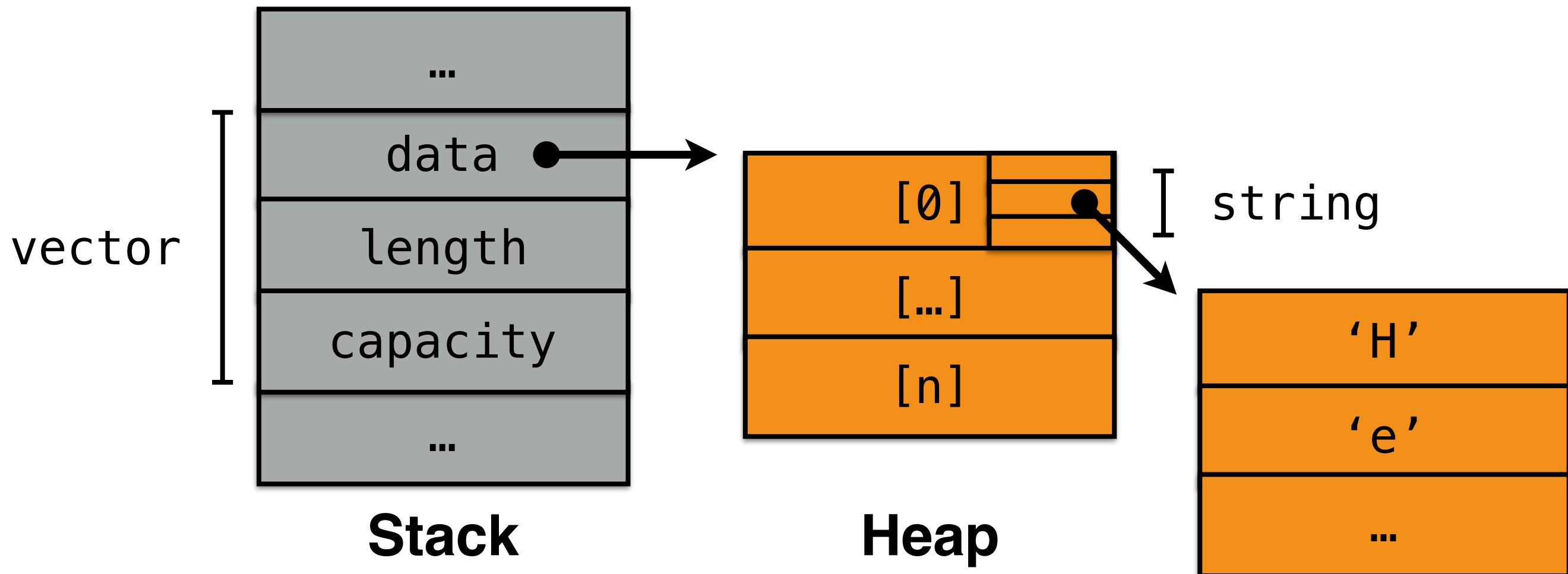
```
void example() {  
    vector<string> vector;    ← Stack and inline layout.  
    ...  
    auto& elem = vector[0];  
    ...  
}
```



What is **control**?

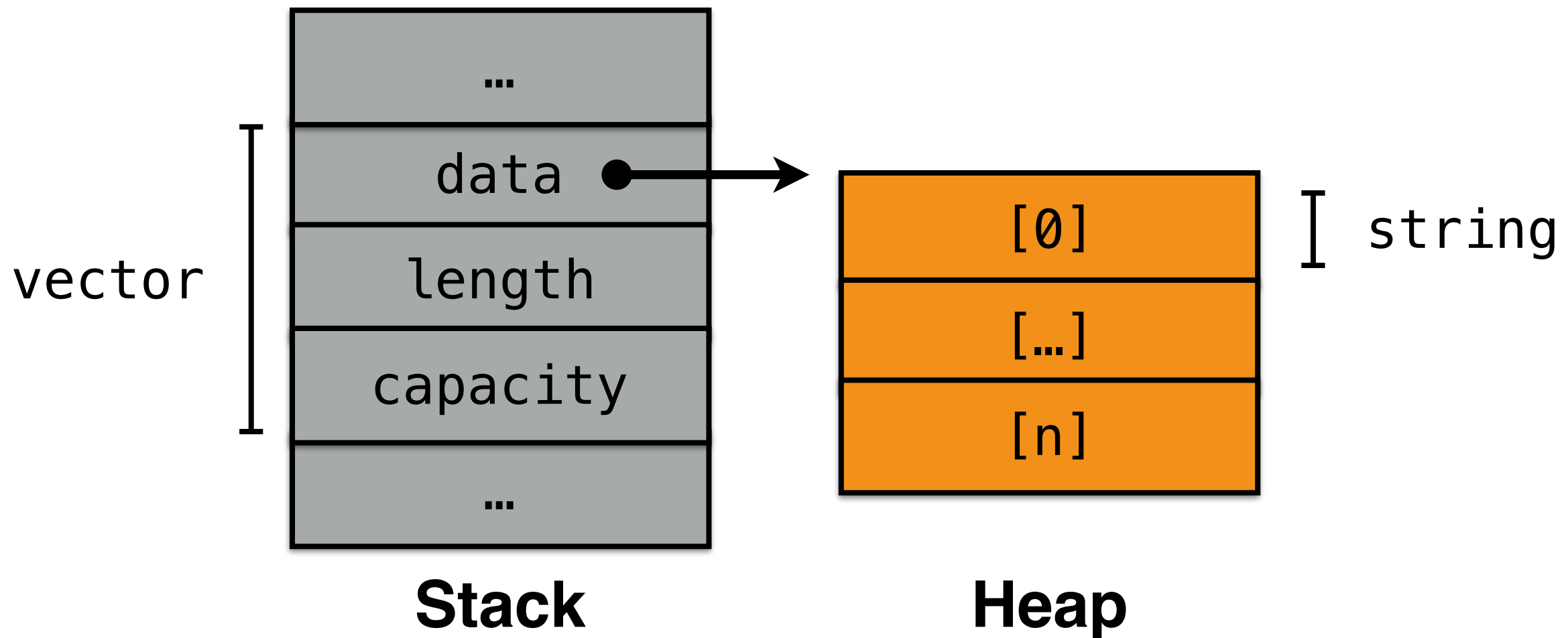
```
void example() {  
    vector<string> vector;  
    ...  
    auto& elem = vector[0];  
    ...  
}
```

← Stack and inline layout.



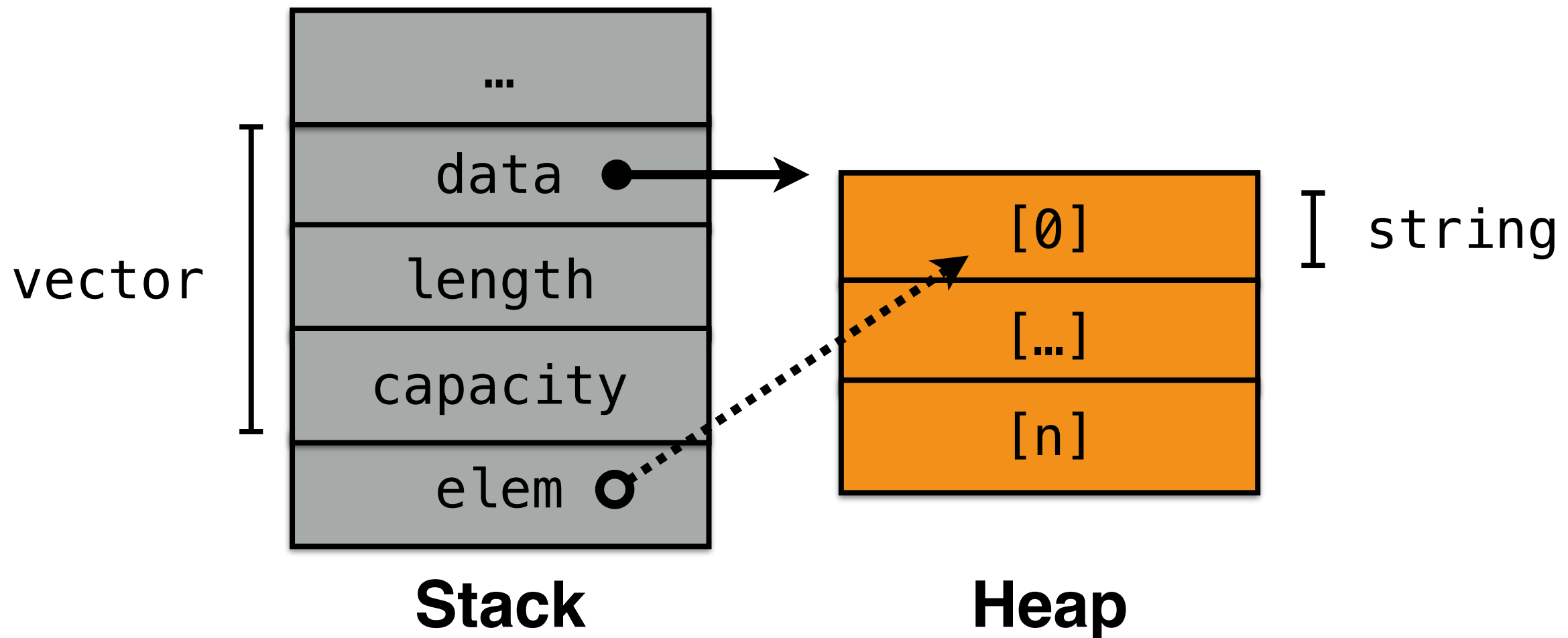
What is **control**?

```
void example() {  
    vector<string> vector;    ← Stack and inline layout.  
    ...  
    auto& elem = vector[0];  
    ...  
}
```



What is **control**?

```
void example() {  
    vector<string> vector;    ← Stack and inline layout.  
    ...  
    auto& elem = vector[0]; ← Lightweight references  
    ...  
}
```

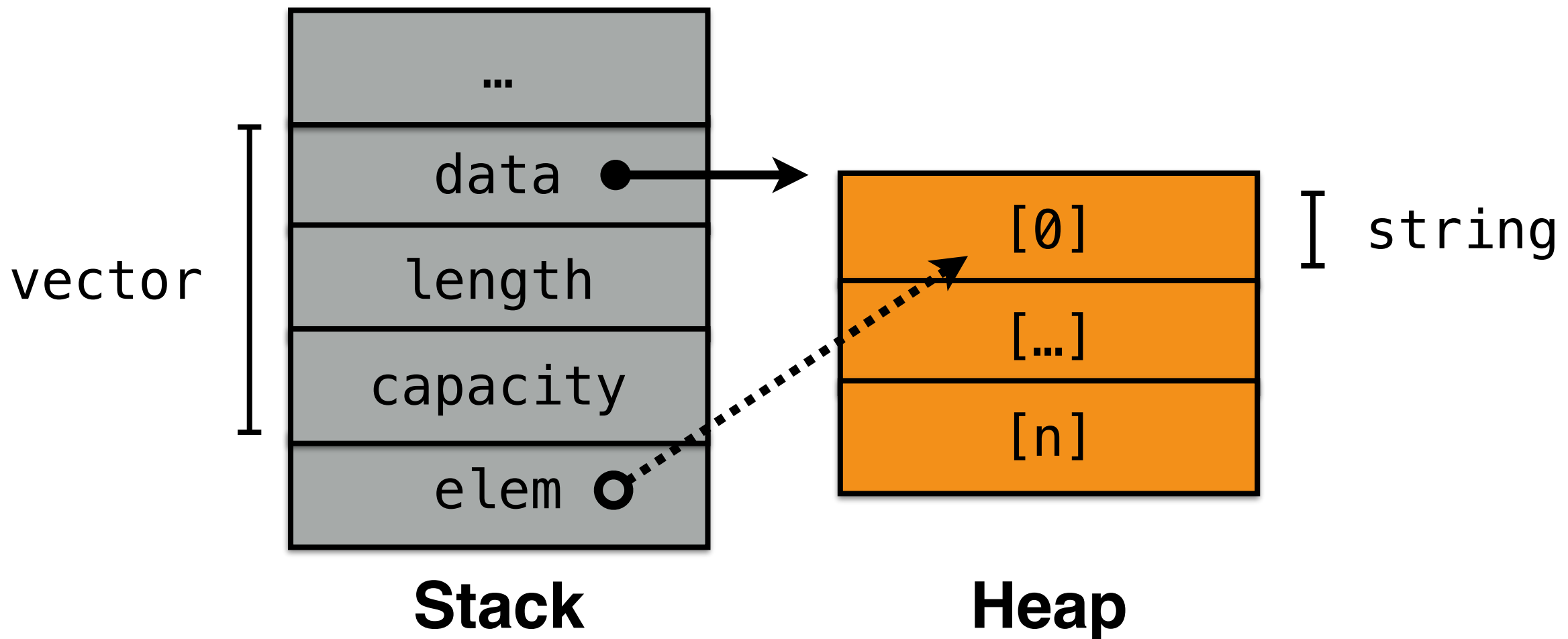


What is **control**?

```

void example() {
    vector<string> vector;    ← Stack and inline layout.
    ...
    auto& elem = vector[0];  ← Lightweight references
    ...
}                             ← Deterministic destruction

```

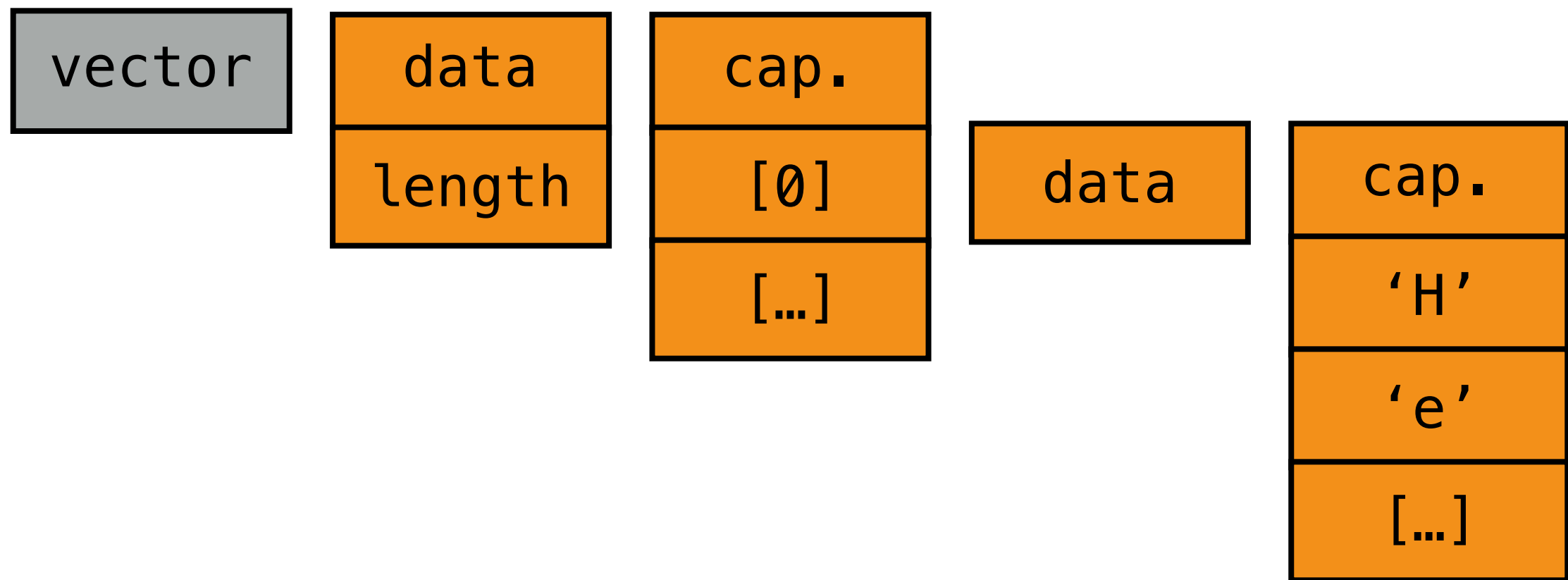


Zero-cost abstraction

Ability to define **abstractions** that
optimize away to nothing.

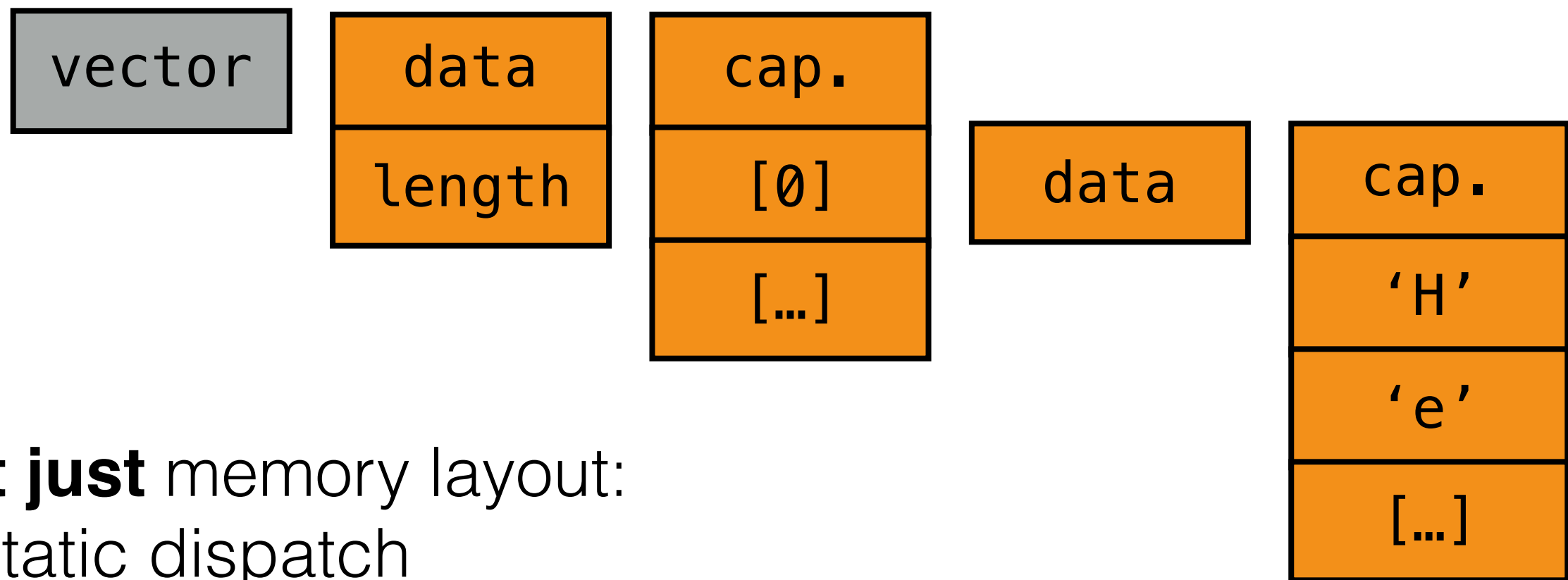
Zero-cost abstraction

Ability to define **abstractions** that
optimize away to nothing.



Zero-cost abstraction

Ability to define **abstractions** that **optimize away to nothing**.



Not just memory layout:

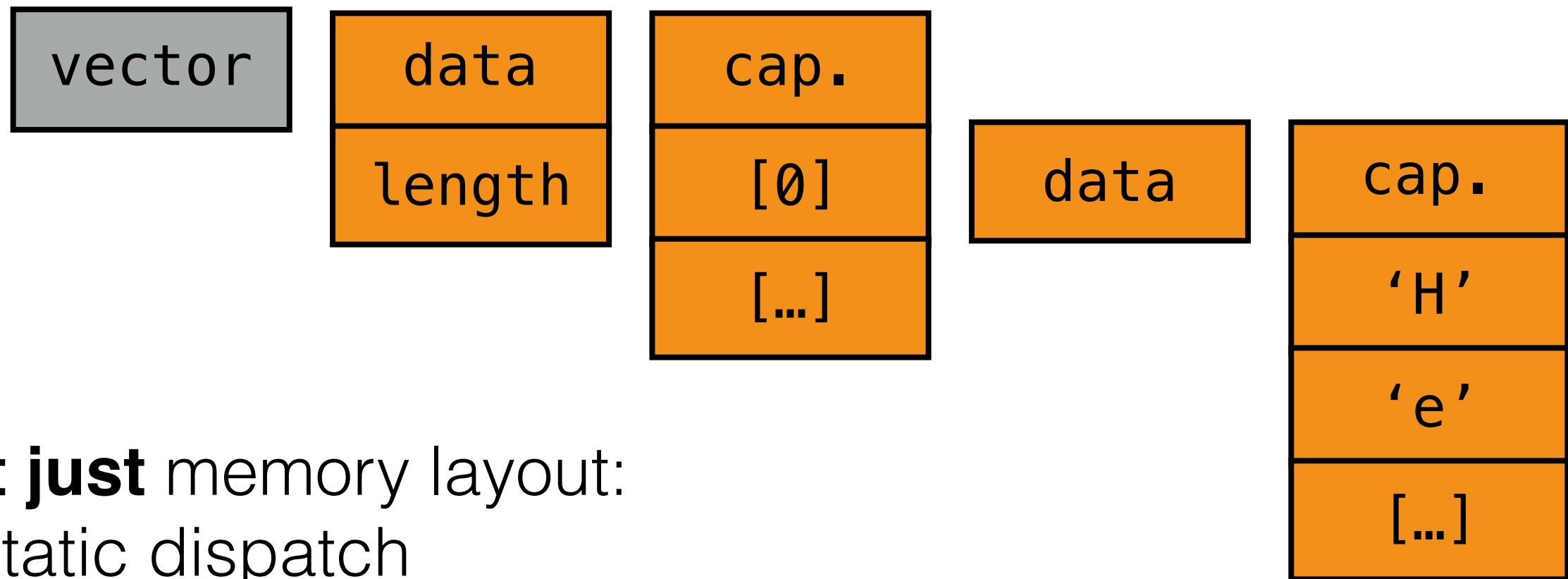
- Static dispatch
- Template expansion
- ...

Zero-cost abstraction



safe

Ability to define **abstractions** that
optimize away to nothing.



Not just memory layout:

- Static dispatch
- Template expansion
- ...

Java

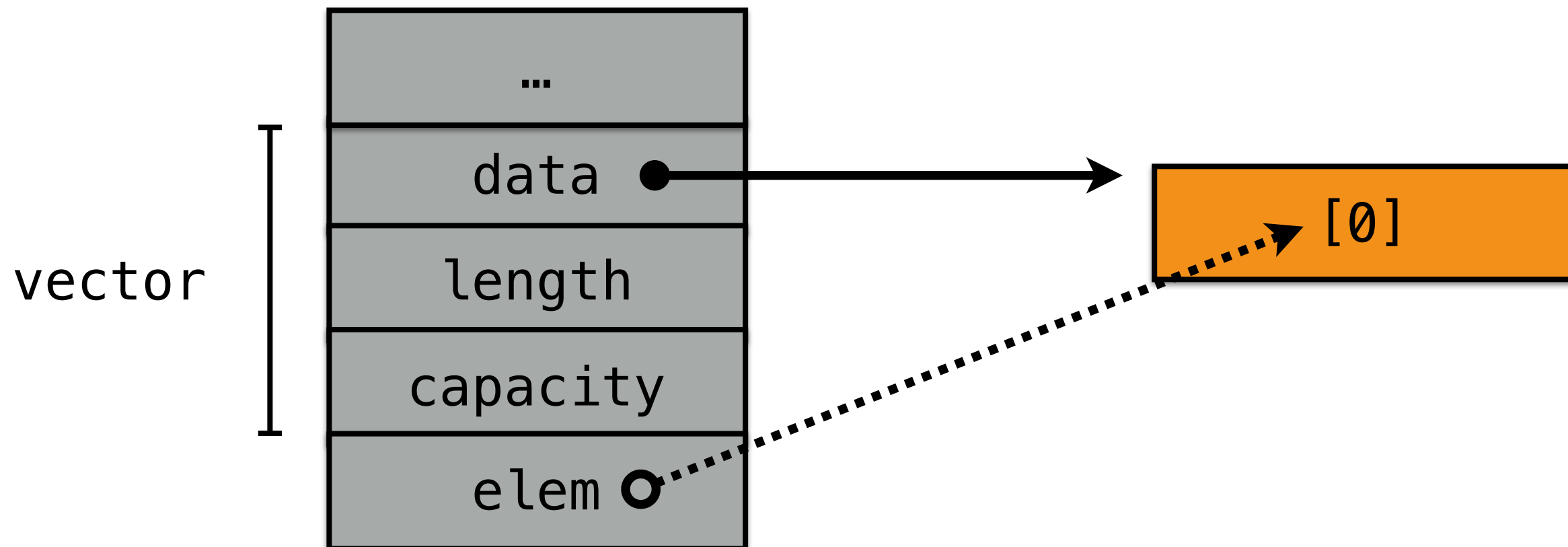
What is **safety**?

```
void example() {  
    vector<string> vector;
```

```
    ...
```

➔

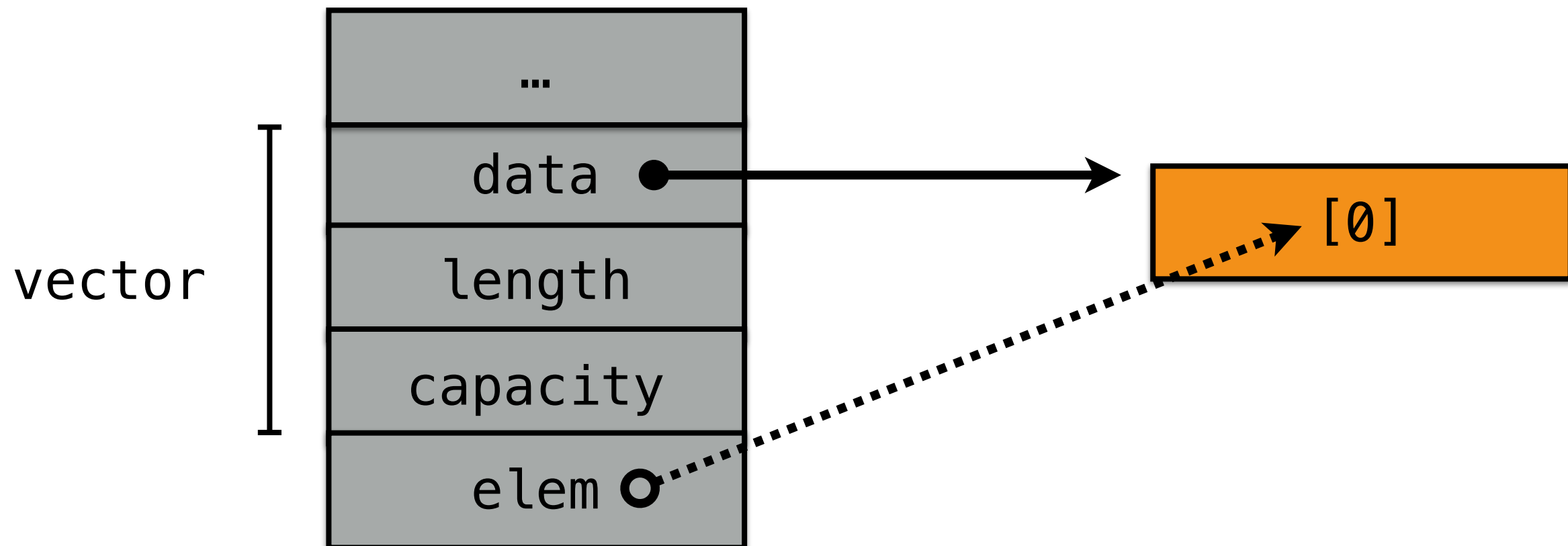
```
    auto& elem = vector[0];  
    vector.push_back(some_string);  
    cout << elem;  
}
```



C++

What is **safety**?

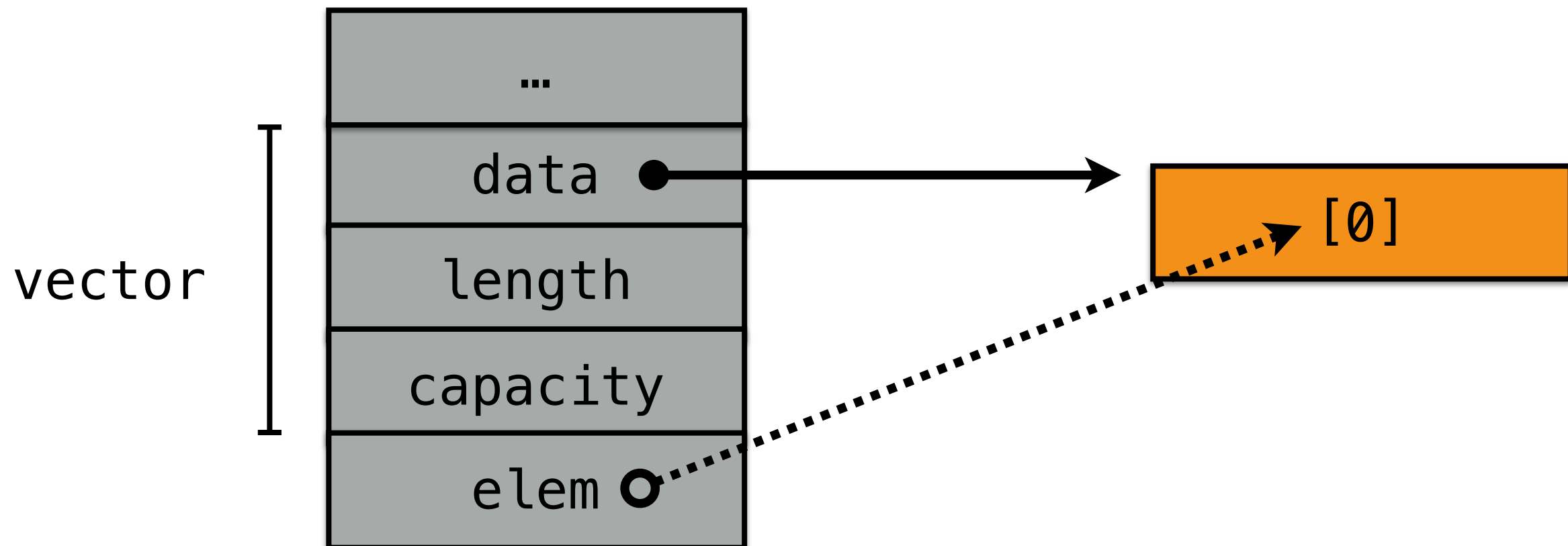
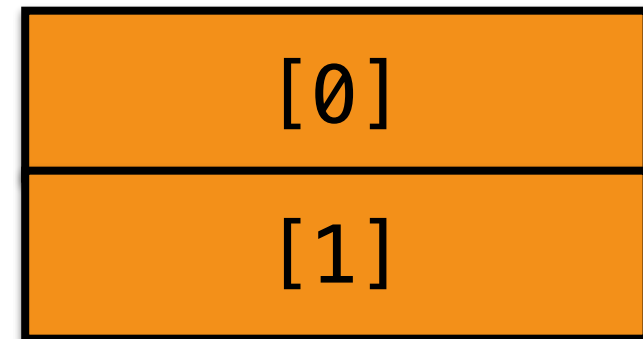
```
void example() {  
    vector<string> vector;  
  
    ...  
    auto& elem = vector[0];  
    vector.push_back(some_string);  
    cout << elem;  
}
```



C++

What is **safety**?

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



C++

What is **safety**?

```
void example() {  
    vector<string> vector;
```

...

```
    auto& elem = vector[0];
```

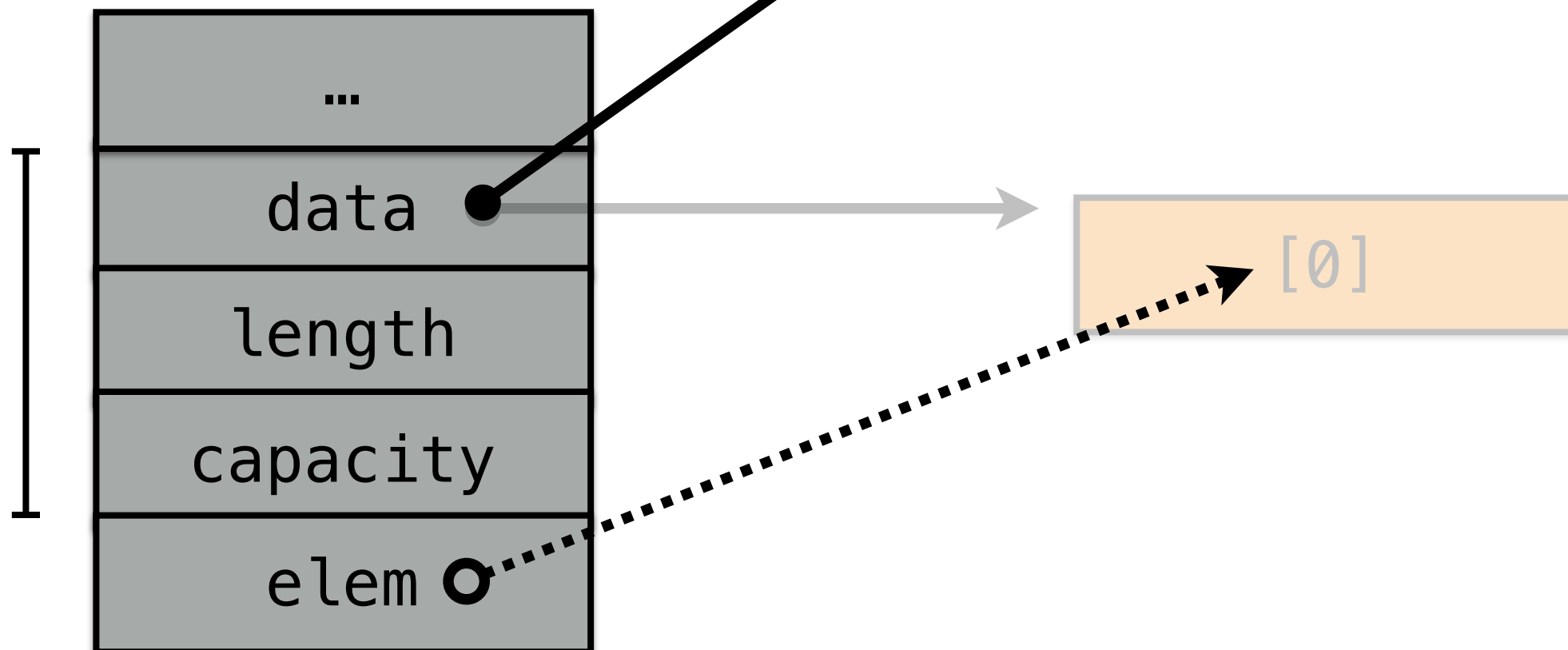
```
    vector.push_back(some_string);
```

```
    cout << elem;
```

```
}
```



vector



C++

What is **safety**?

```
void example() {  
    vector<string> vector;
```

...

```
    auto& elem = vector[0];
```

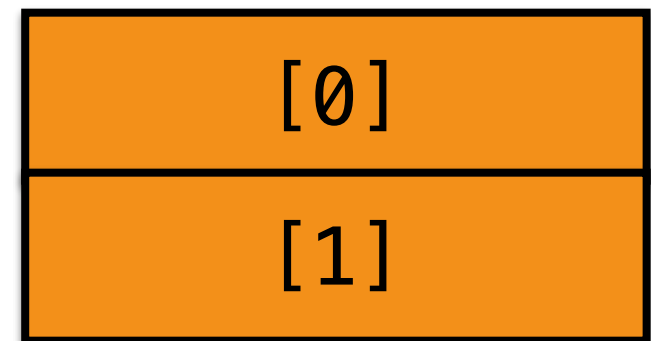
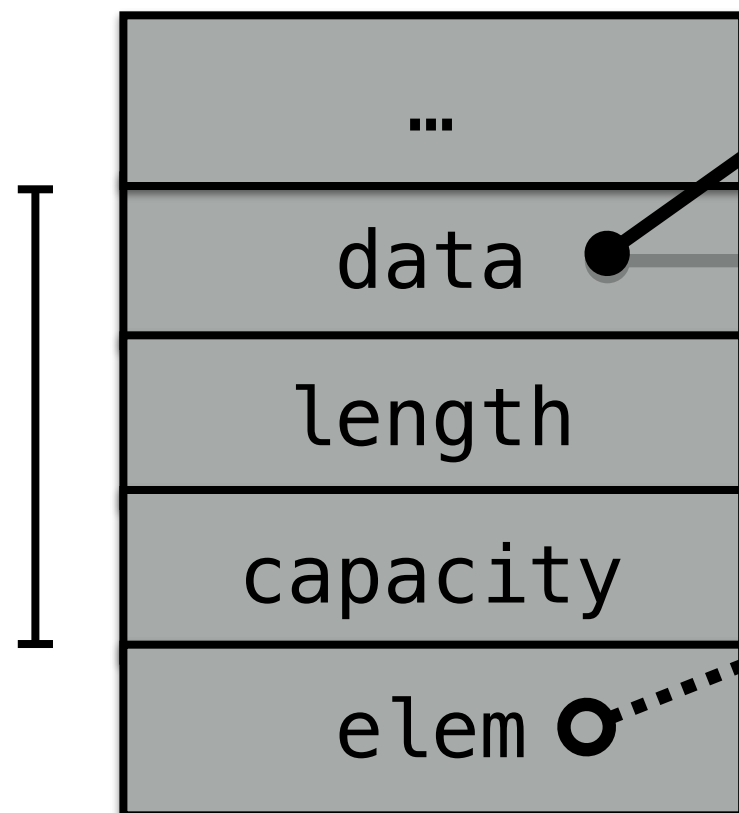
```
    vector.push_back(some_string);
```

```
    cout << elem;
```

```
}
```



vector

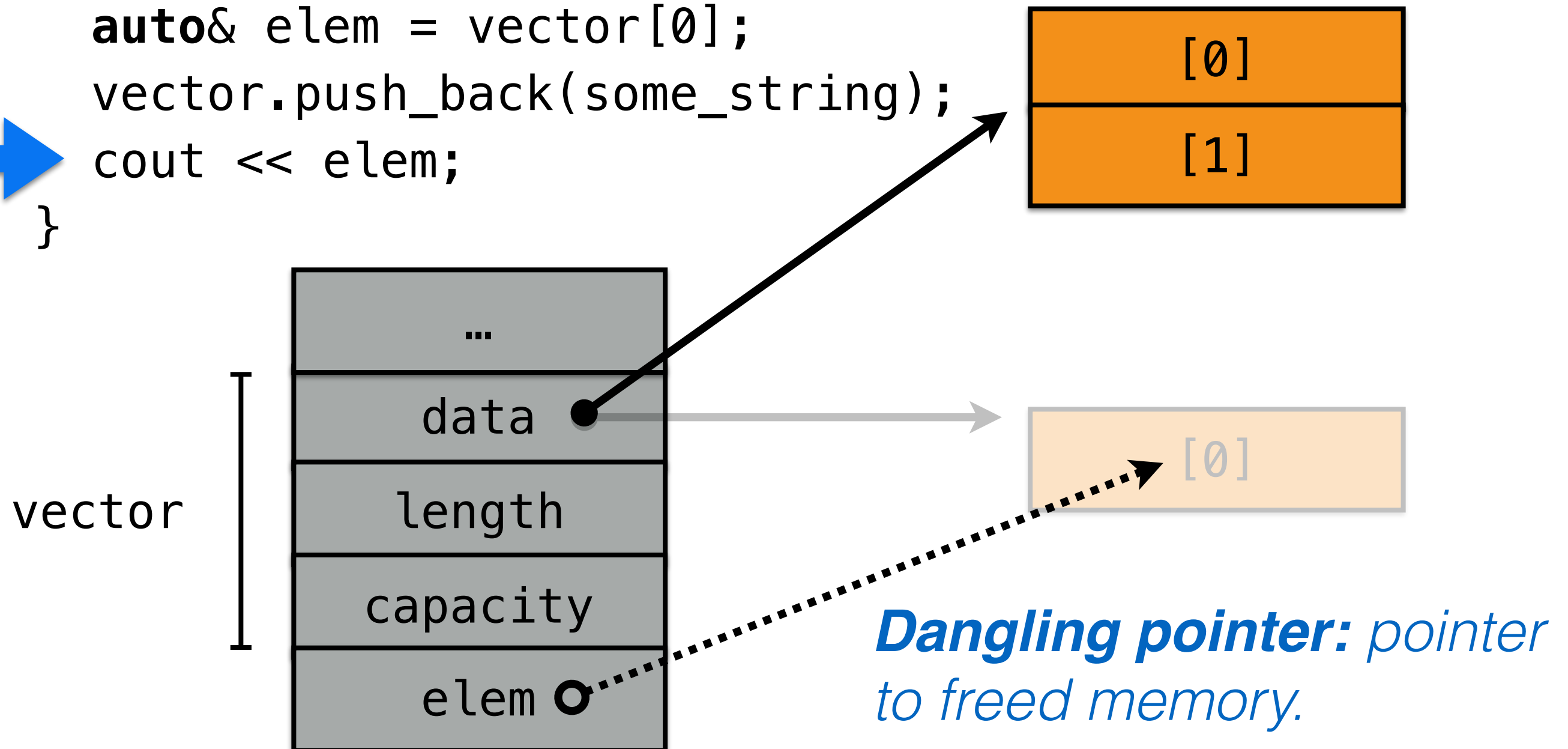


Dangling pointer: pointer
to freed memory.

C++

What is **safety**?

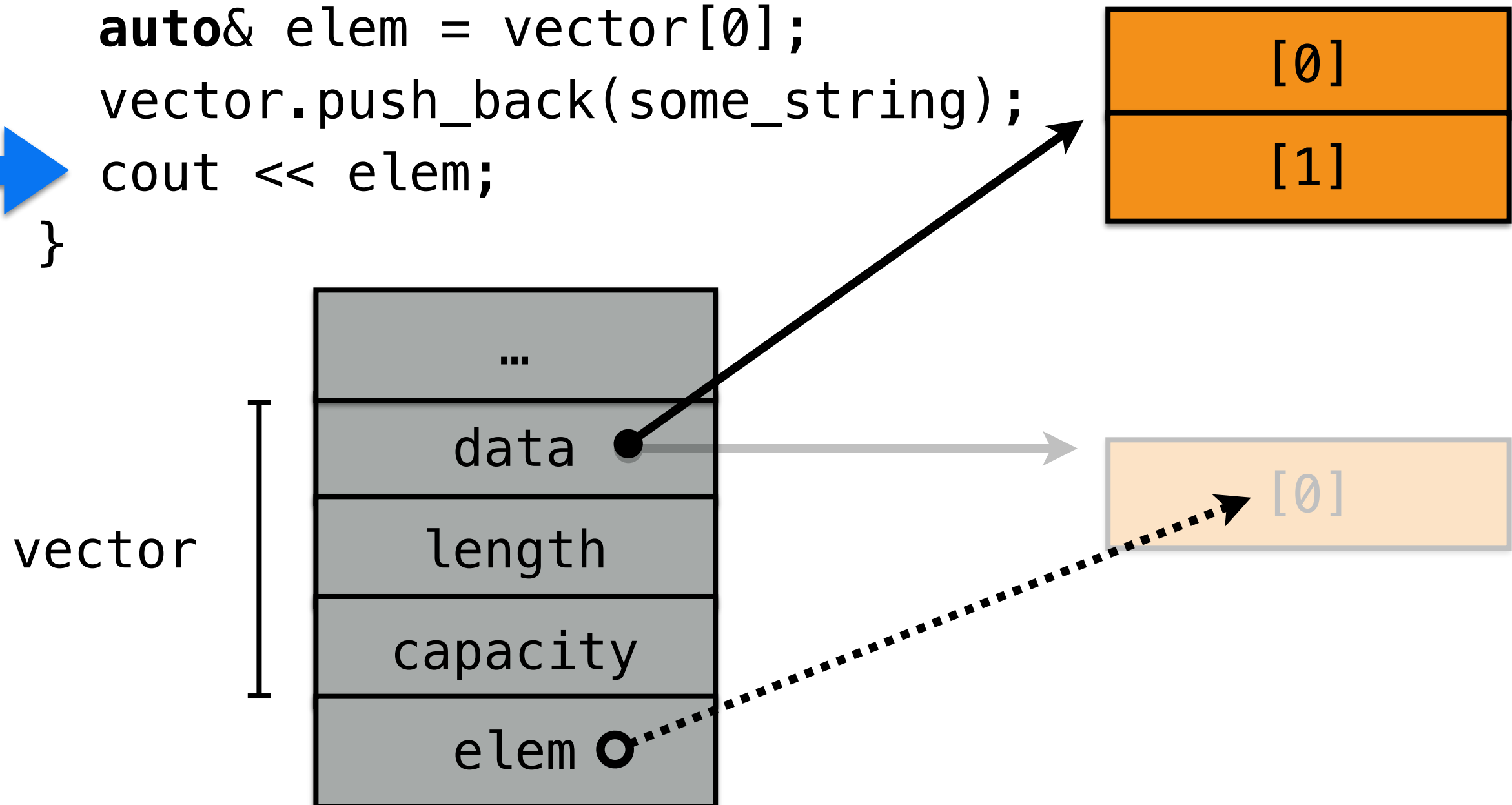
```
void example() {  
    vector<string> vector;  
  
    ...  
    auto& elem = vector[0];  
    vector.push_back(some_string);  
    cout << elem;  
}
```



C++

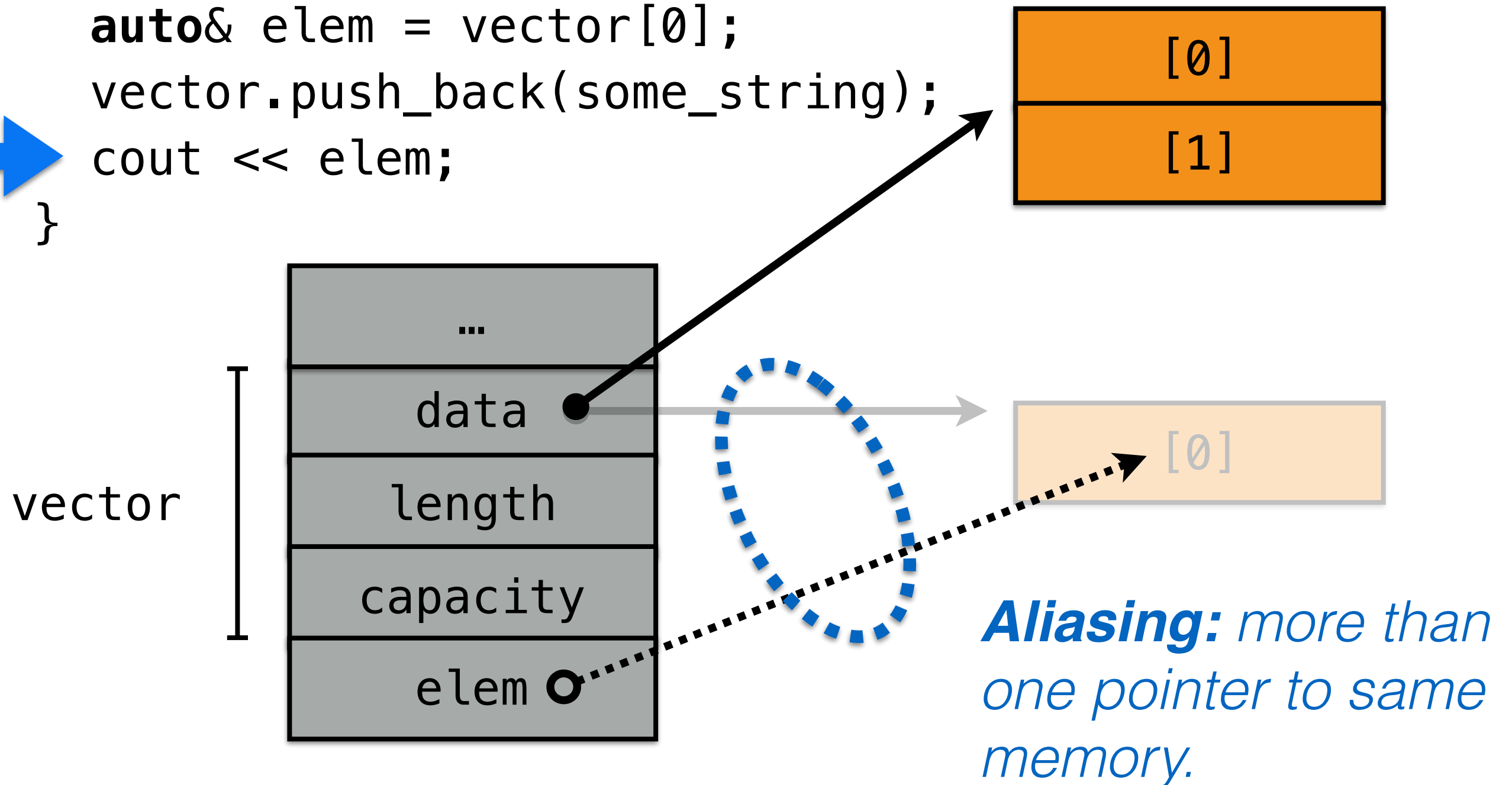
What is **safety**?

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



What is **safety**?

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



What is **safety**?

```
void example() {  
    vector<string> vector;
```

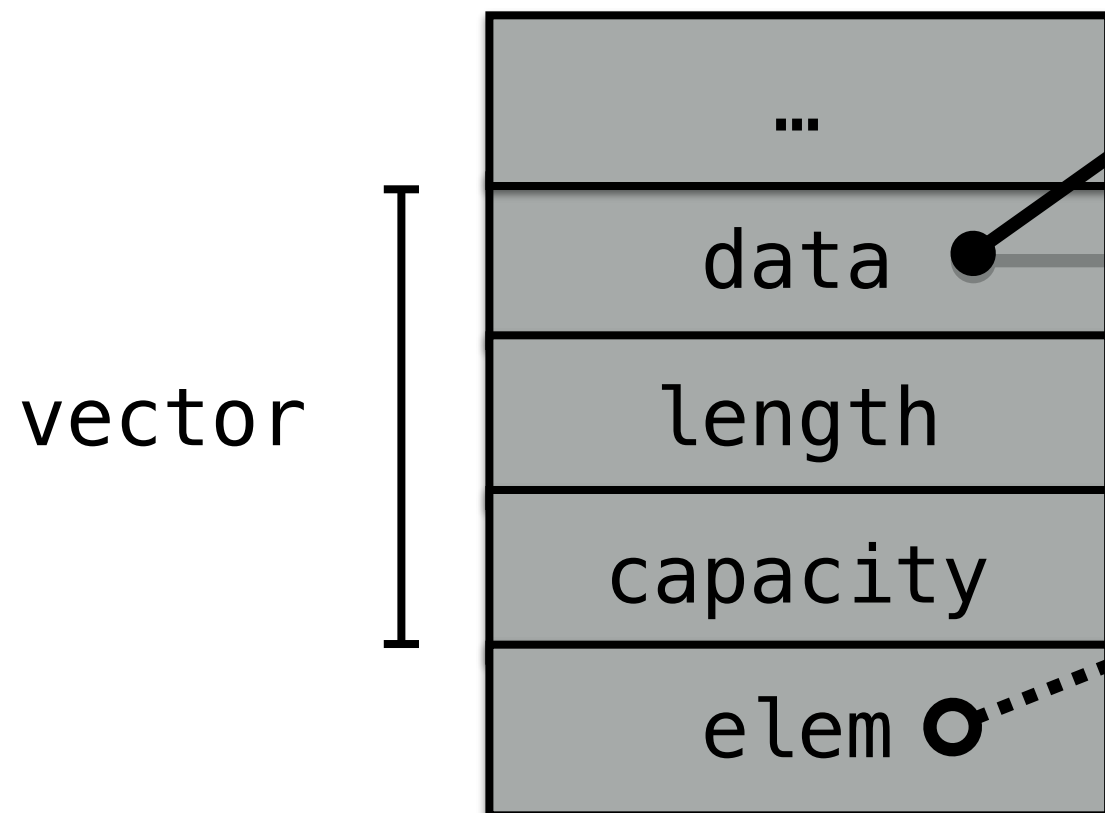
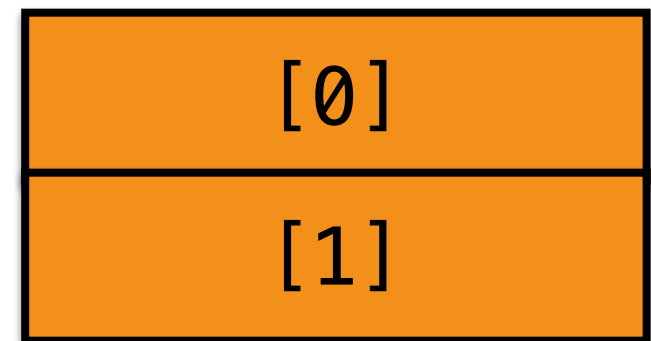
```
    ...
```

```
    auto& elem = vector[0];  
    vector.push_back(some_string);
```

```
    cout << elem;
```

```
}
```

Mutating the vector
freed old contents.



Aliasing: more than
one pointer to same
memory.

What about **GC**?

What about **GC**?

No **control**.

Requires a **runtime**.

What about **GC**?

No **control**.

Requires a **runtime**.

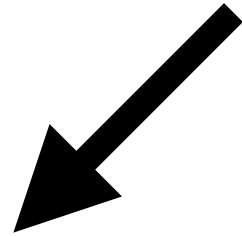
Insufficient to prevent related problems:
iterator invalidation, data races.

Rust's Solution

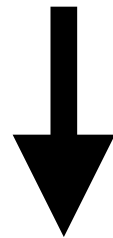
Type system enforces **ownership and borrowing**:

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.

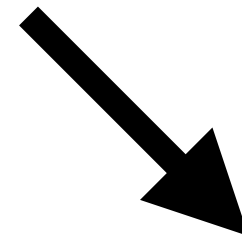
Ownership/Borrowing



No need for
a runtime

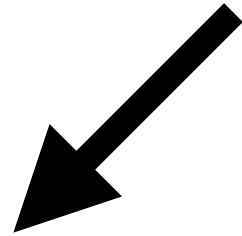


Memory
safety

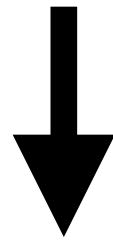


Data-race
freedom

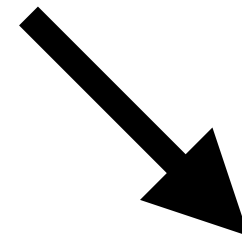
Ownership/Borrowing



No need for
a runtime



Memory
safety

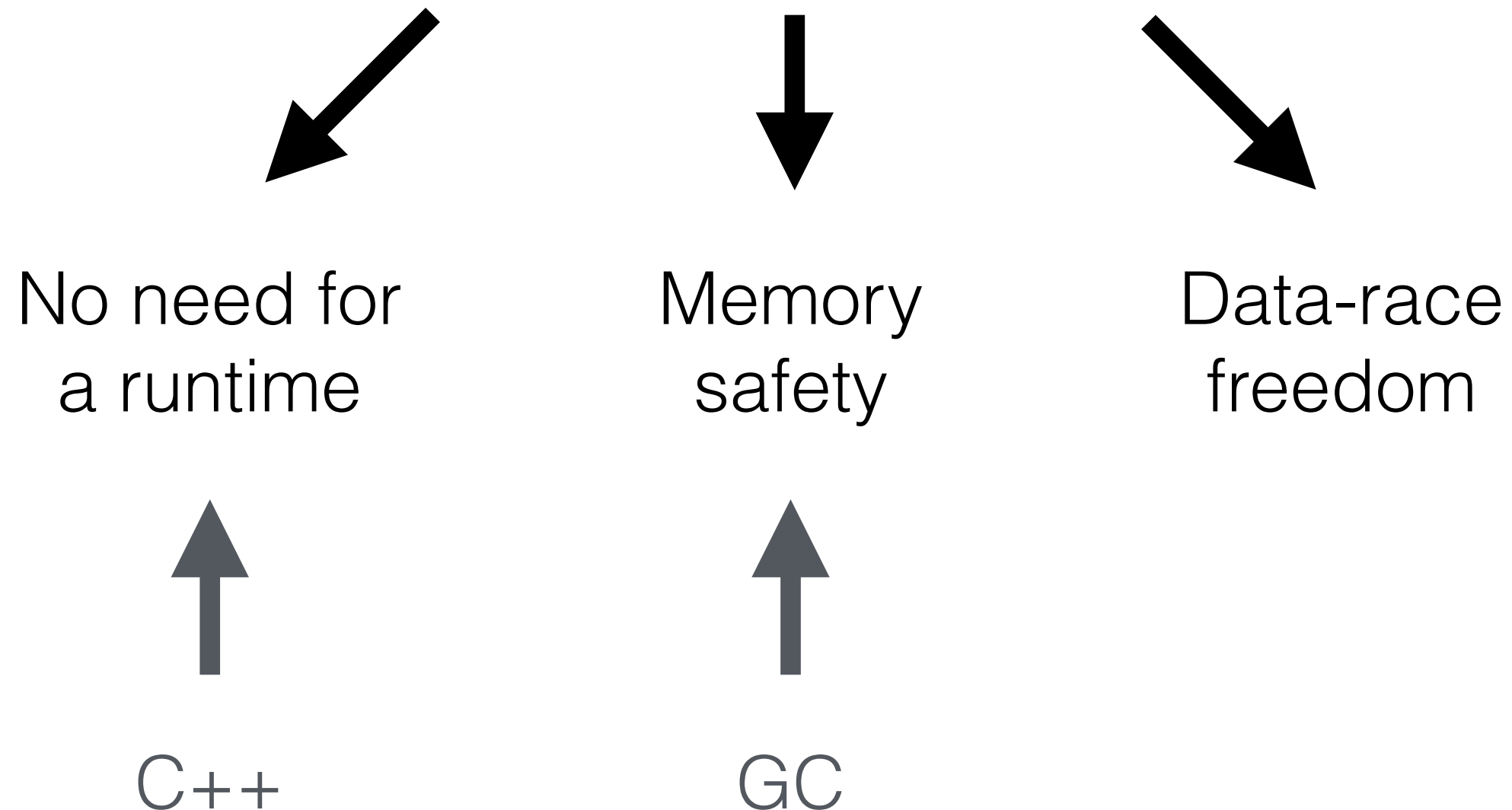


Data-race
freedom



C++

Ownership/Borrowing



More to Rust than Safety

Explicit control is often correlated with **painfully verbose**.

Rust offers **lots of goodies** traditionally associated with garbage-collected runtimes:

- Closures
- Pattern matching
- Type inference
- Traits

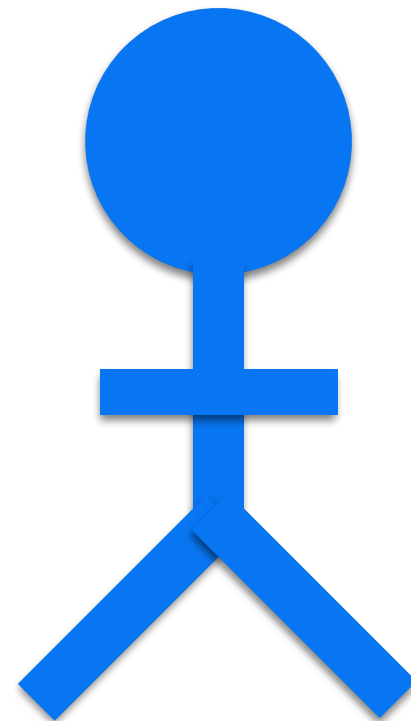
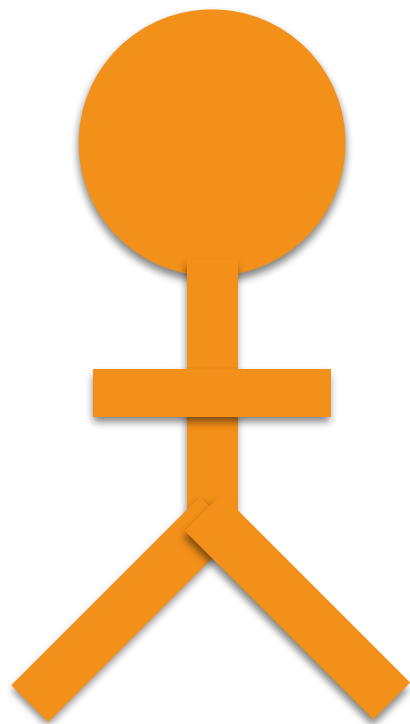
Credit where it is due

Rust has an **active, amazing community**.

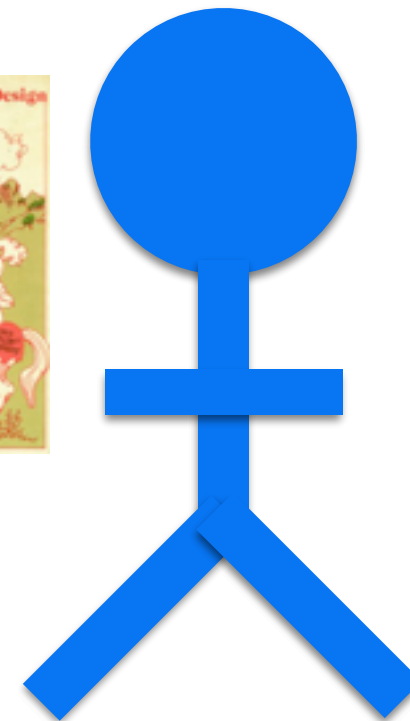
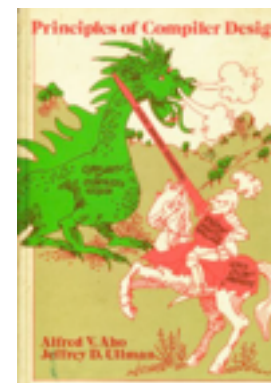


Ownership

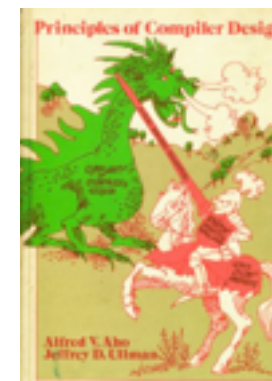
n. The act, state, or right of possessing something.



Ownership (T)



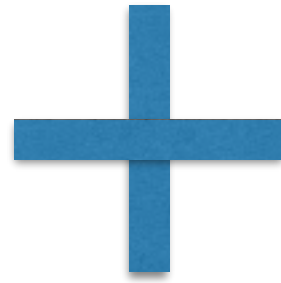
Ownership (T)



Ownership (T)

Ownership (T)

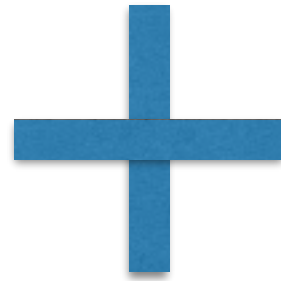
Aliasing



Mutation

Ownership (T)

~~Aliasing~~

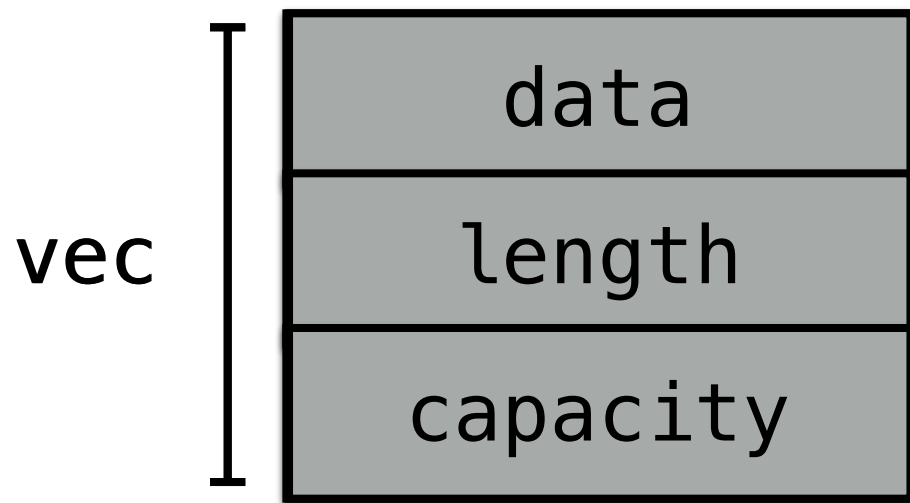


Mutation

Ownership (T)

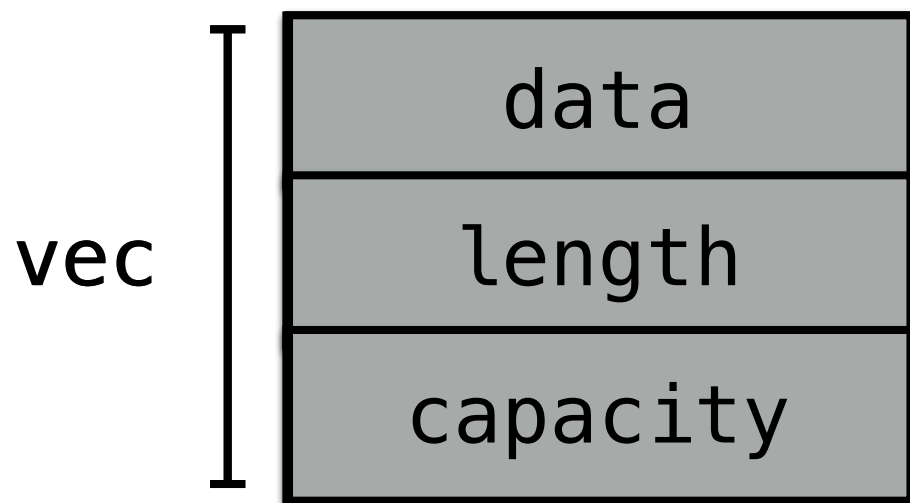
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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



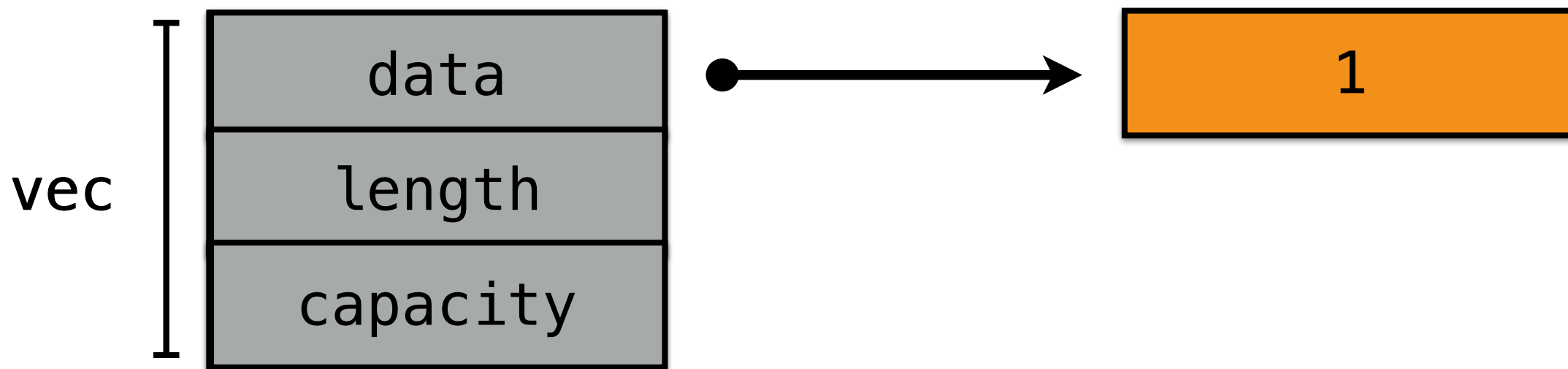
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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



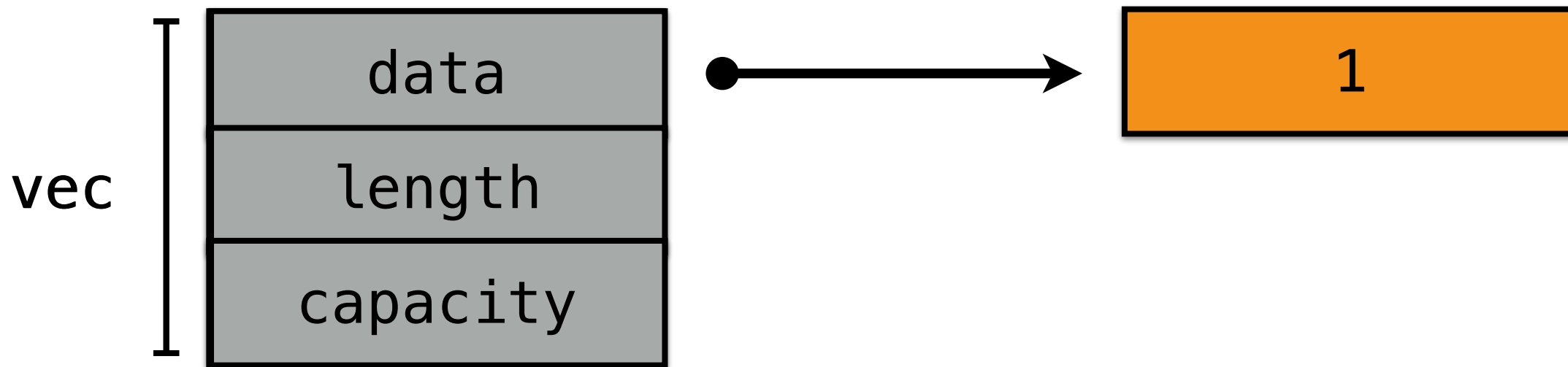
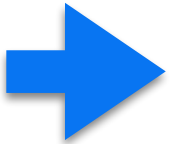
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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



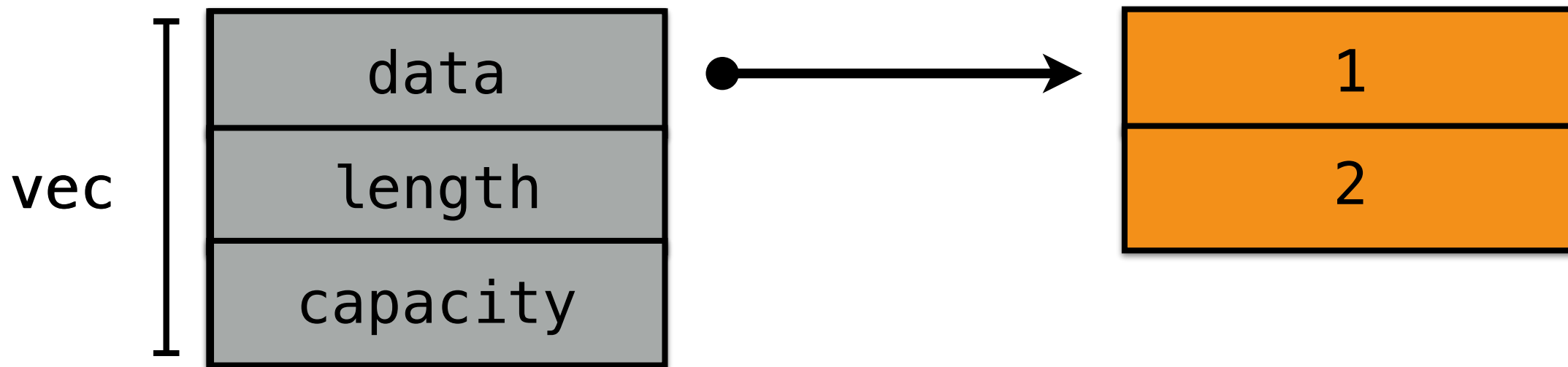
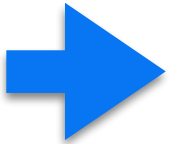
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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



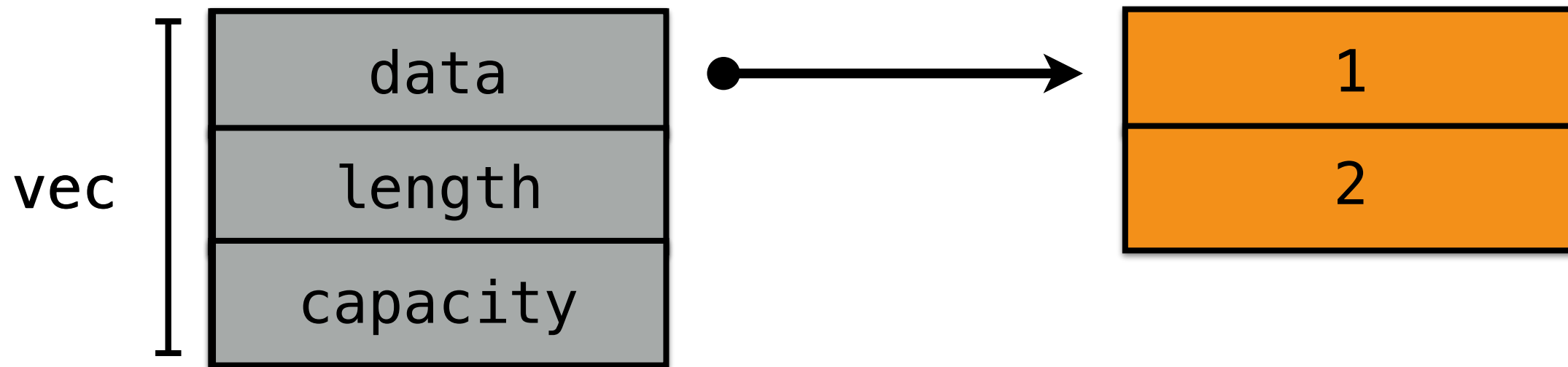
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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



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

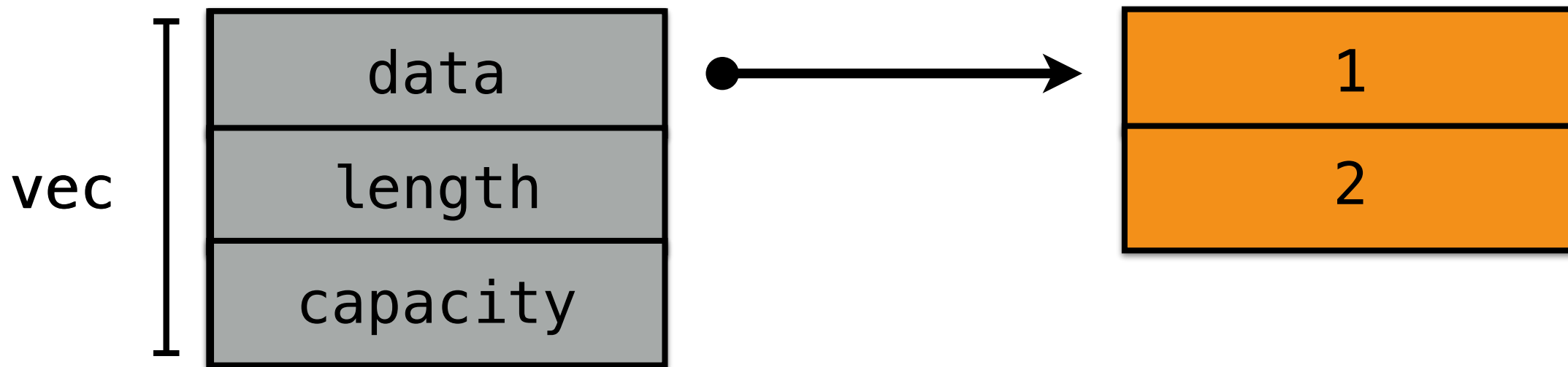
```
fn take(vec: Vec<int>) {  
    // ...  
}
```



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

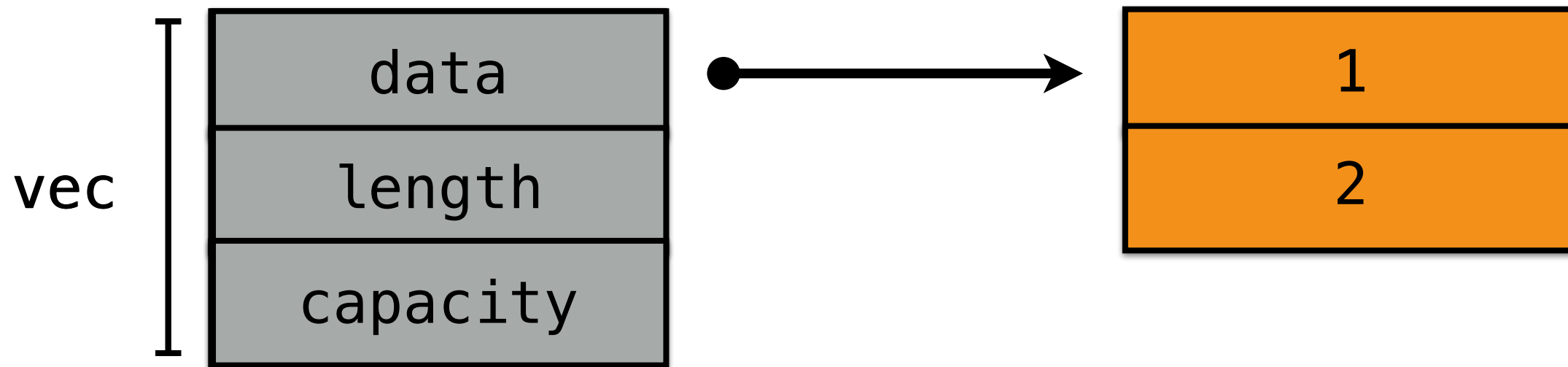
```
fn take(vec: Vec<int>) {  
    // ...  
}
```

Take ownership
of a Vec<int>



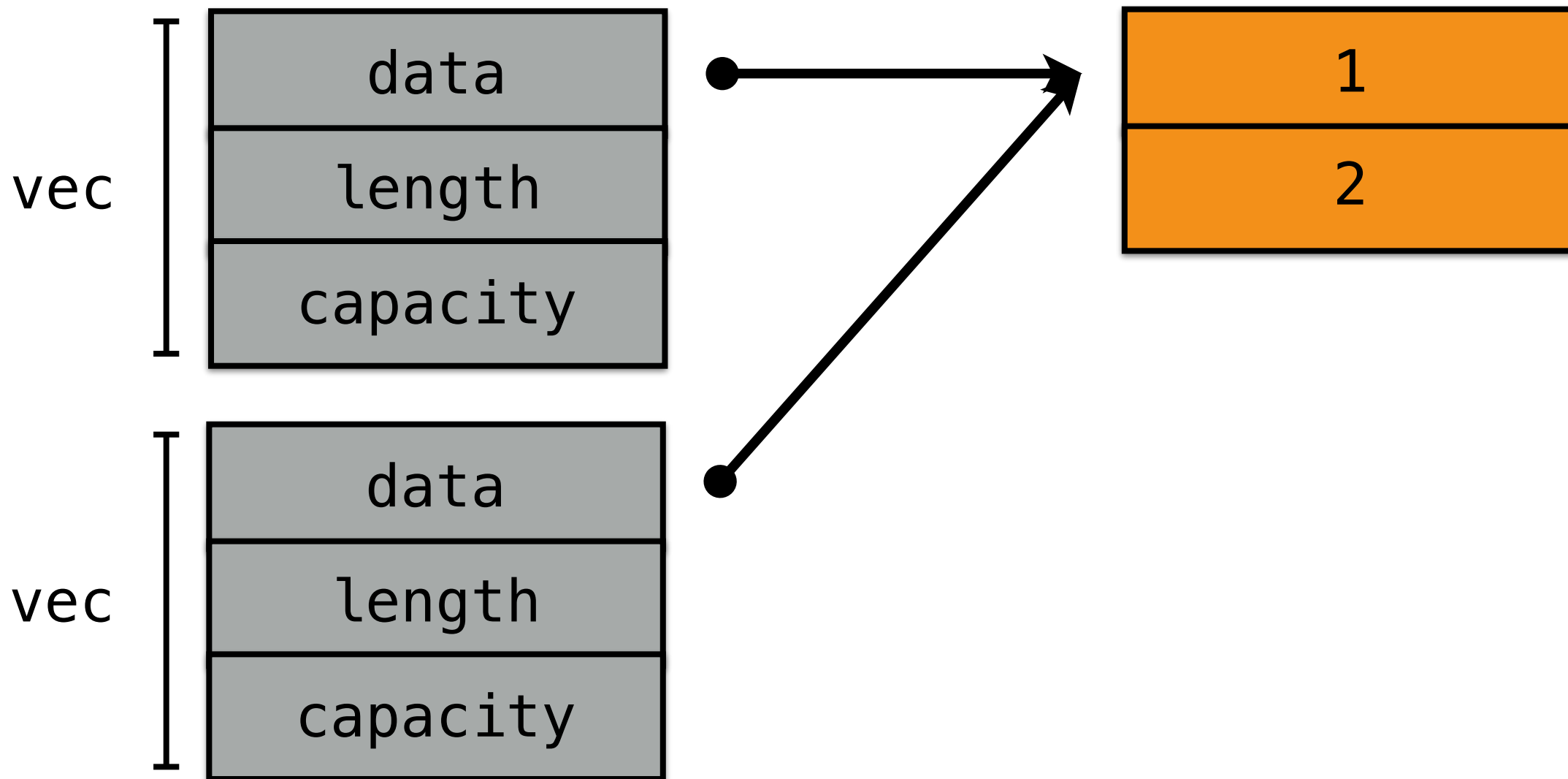
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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



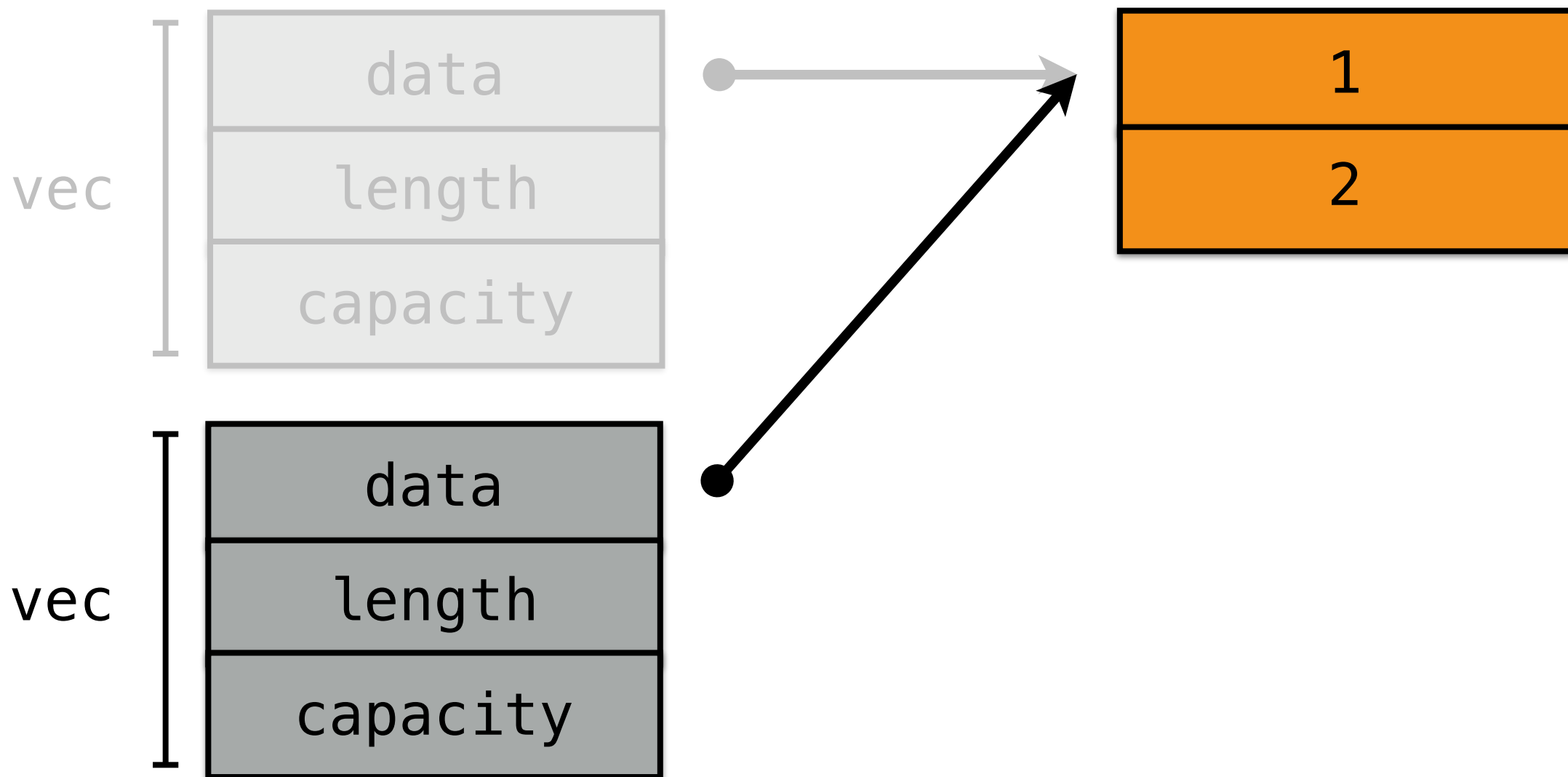
```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

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

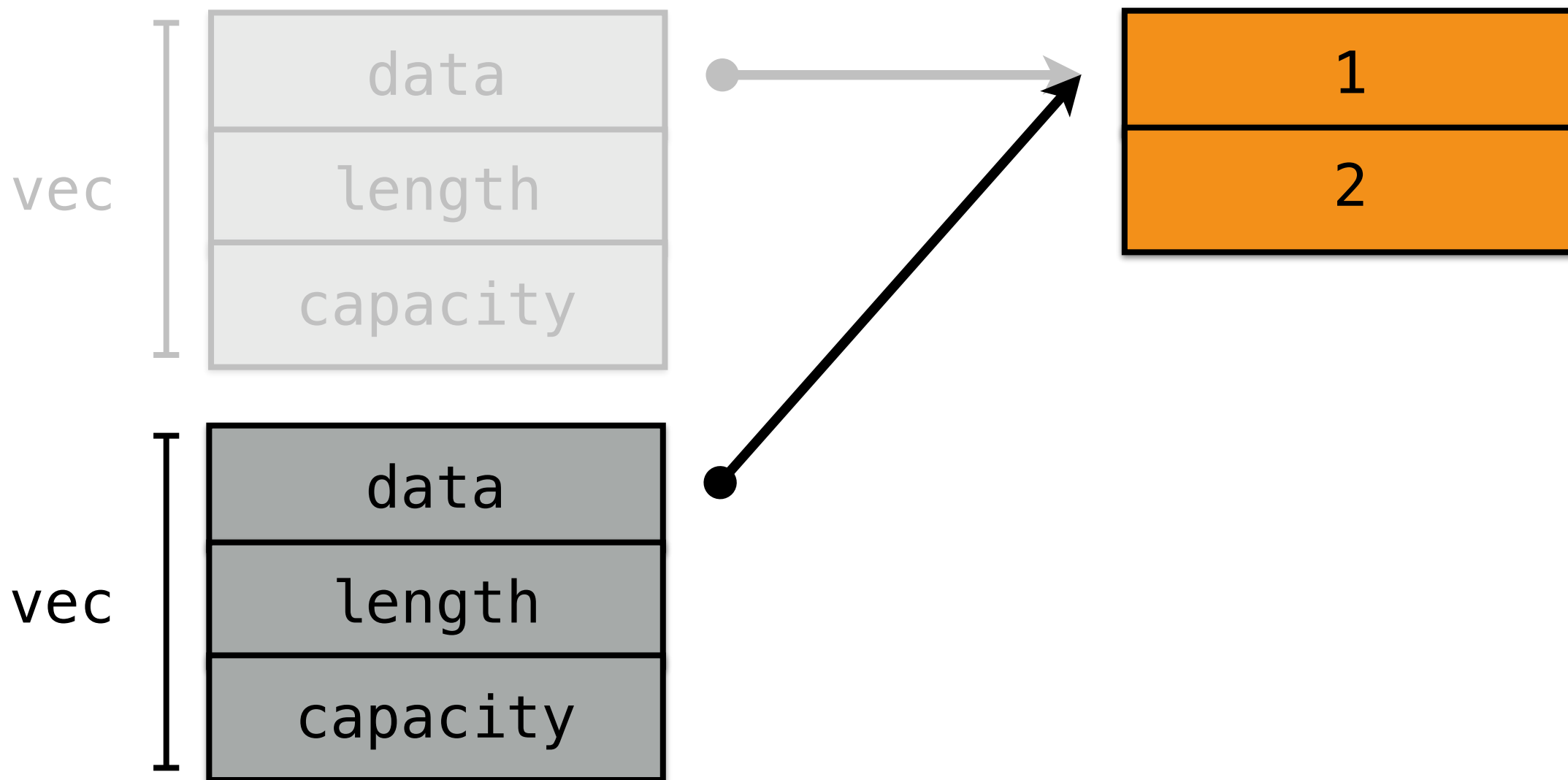


```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}
```

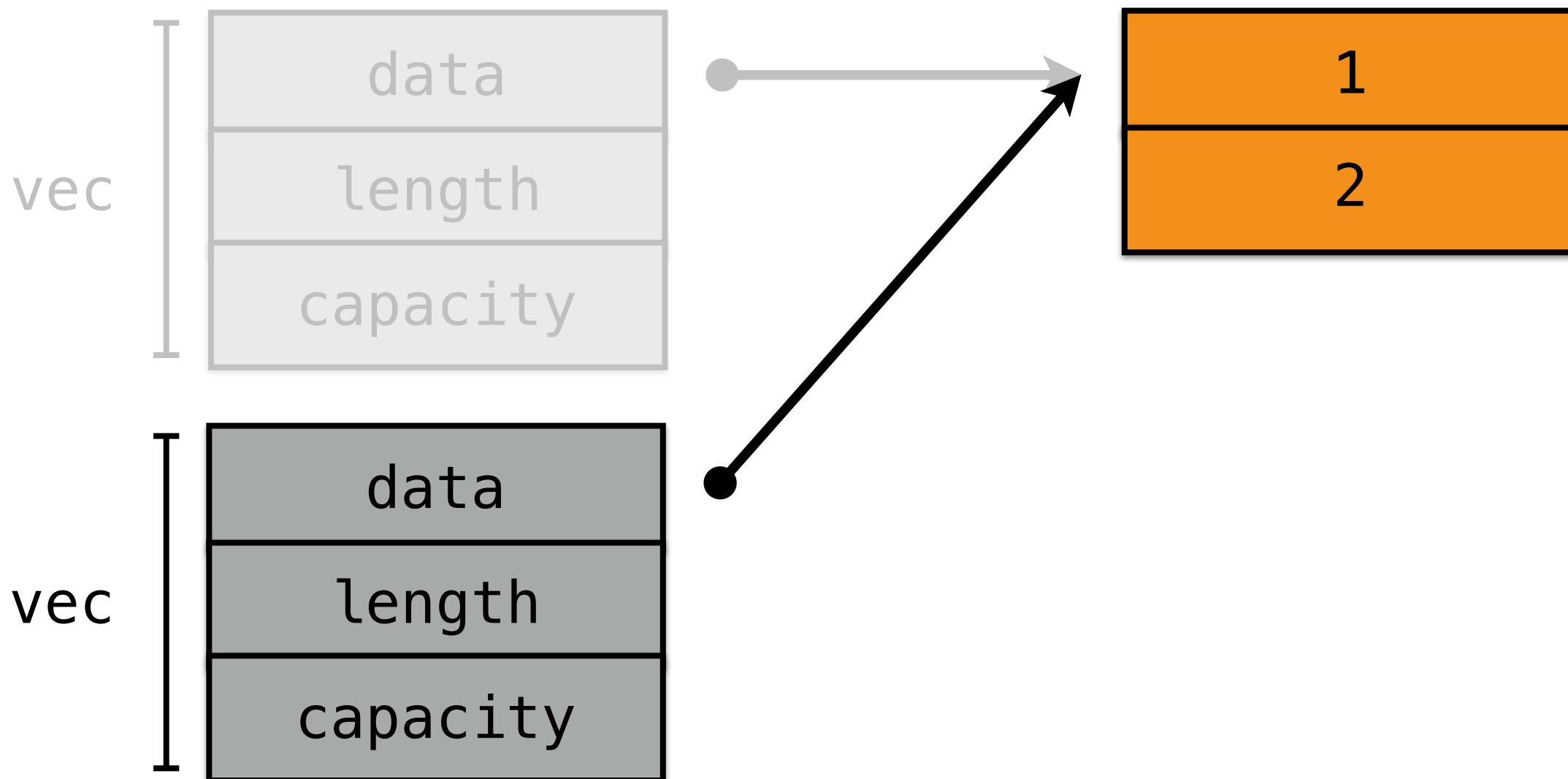
```
fn take(vec: Vec<int>) {  
    // ...  
}
```



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



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

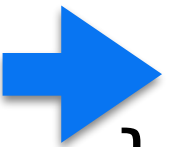


```
fn give() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    take(vec);  
    ...  
}  
fn take(vec: Vec<int>) {  
    // ...  
}
```



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

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



...

}



Compiler **enforces** moves

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

Compiler **enforces** moves

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



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<int>) {  
    // ...  
}
```



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<int>) {  
    // ...  
}
```



← **Error:** vec has been moved

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<int>) {  
    // ...  
}
```



Error: vec has been moved

Compiler **enforces** moves

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

 **Error:** vec has been moved

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

Prevents:

- use after free
- double moves
- ...

Borrow

v. To receive something with the promise of returning it.



Shared borrow (&T)

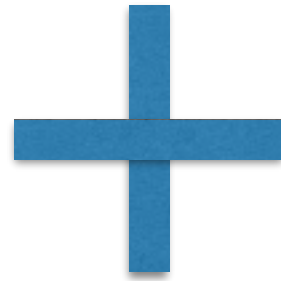


Shared borrow (&T)

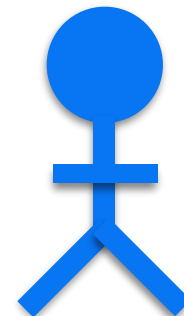
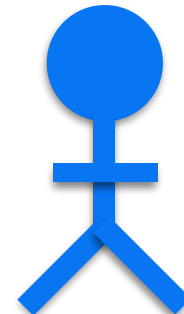
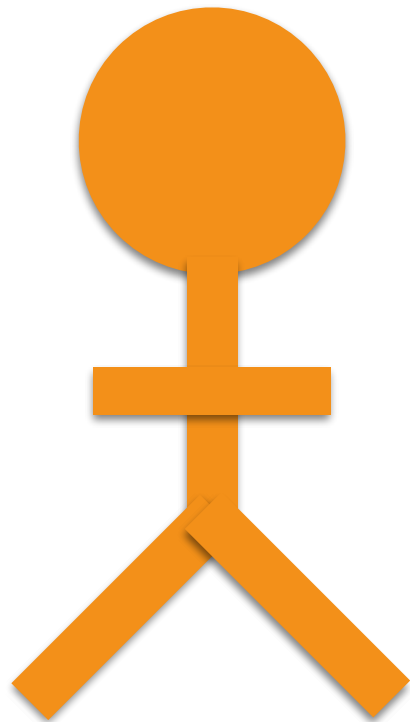


Shared borrow (&T)

Aliasing

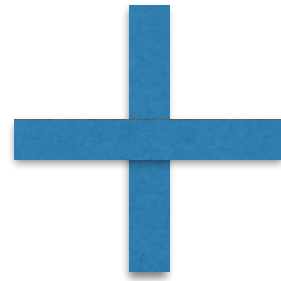


Mutation

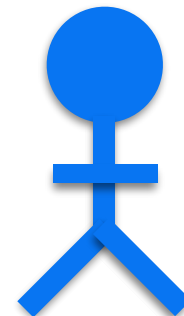
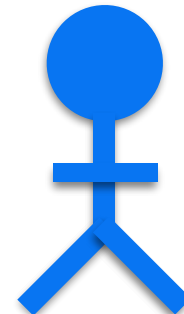
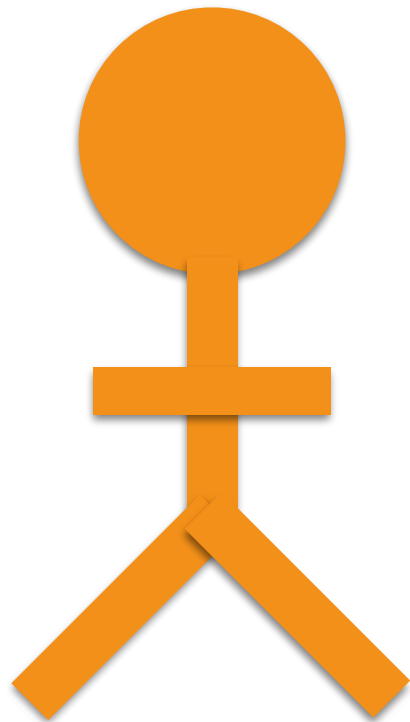


Shared borrow (&T)

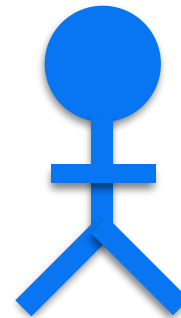
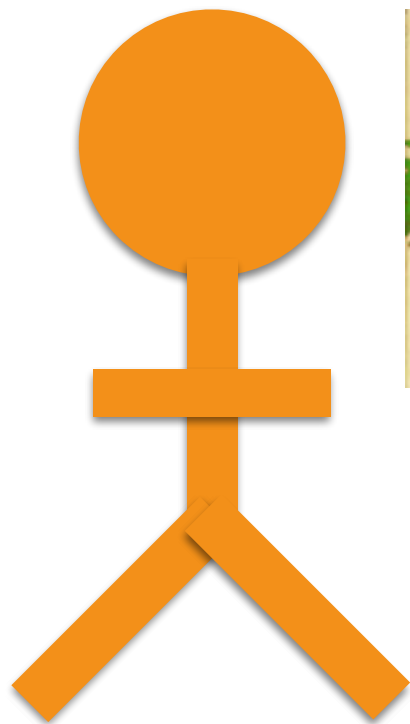
Aliasing



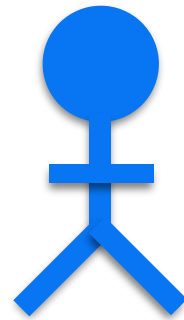
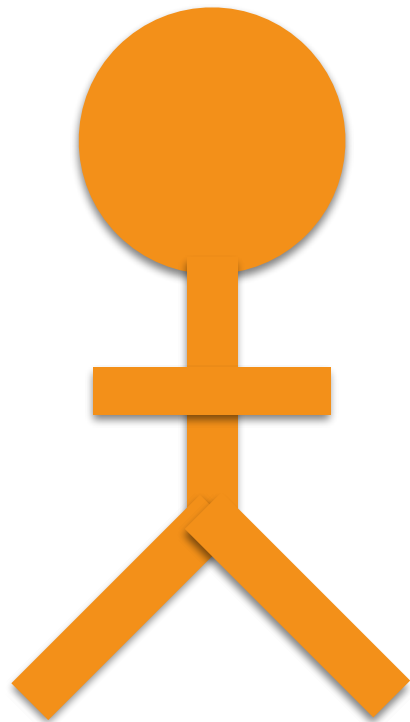
~~Mutation~~



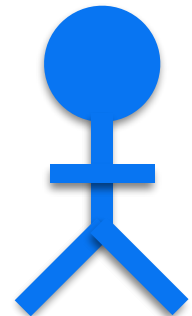
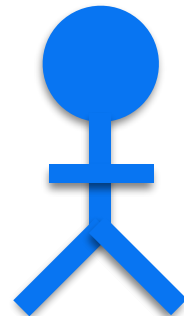
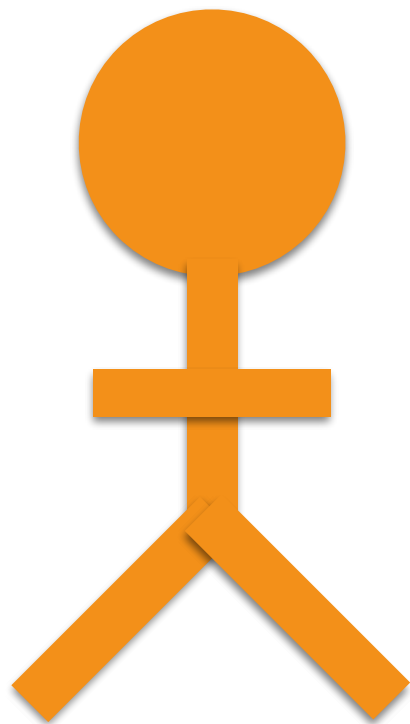
Shared borrow (&T)



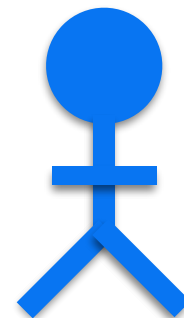
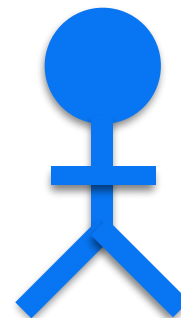
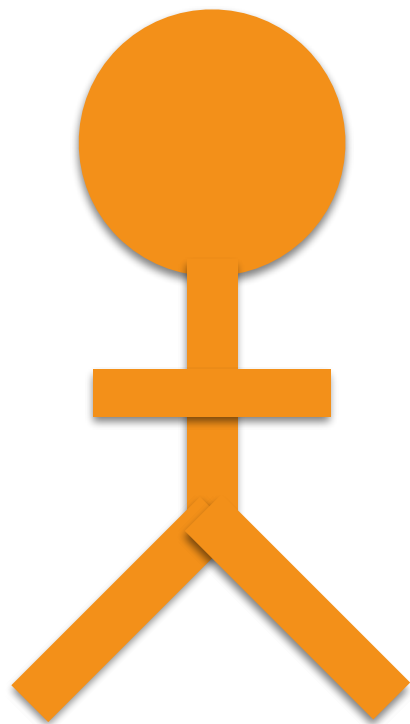
Mutable borrow (&mut T)



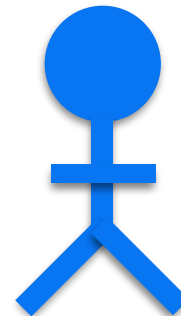
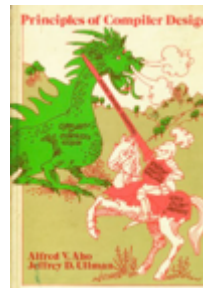
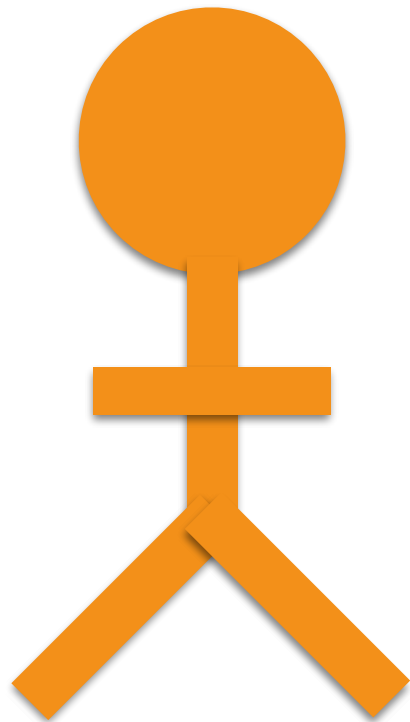
Mutable borrow (&mut T)



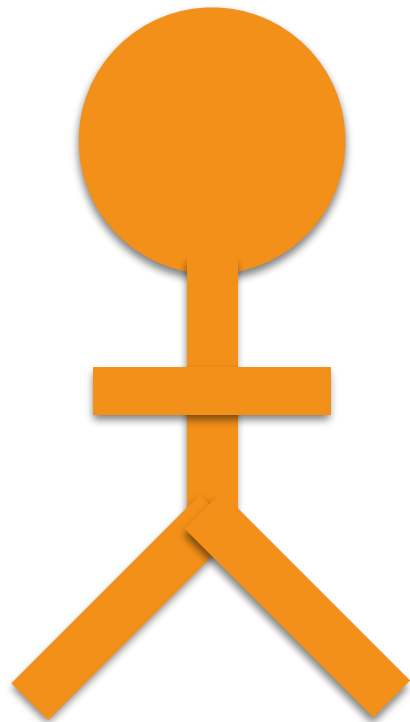
Mutable borrow (&mut T)



Mutable borrow (&mut T)



Mutable borrow (&mut T)

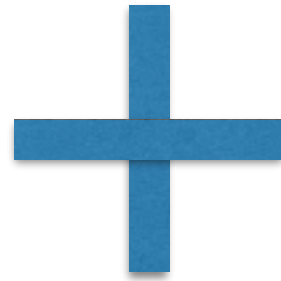


Mutable borrow (&mut T)



Mutable borrow (&mut T)

Aliasing

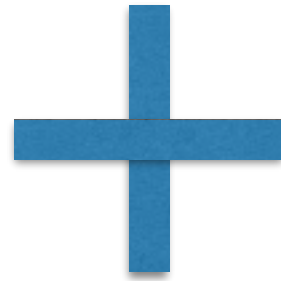


Mutation



Mutable borrow (&mut T)

~~Aliasing~~



Mutation



Mutable borrow (&mut T)

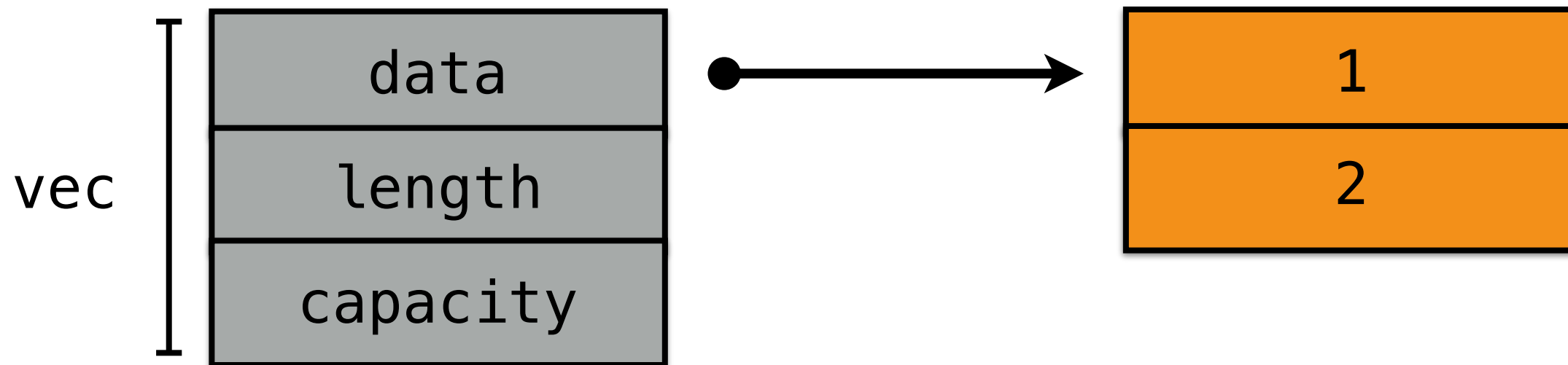


```
fn lender() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    use(&vec);  
  
    ...  
}  
  
fn use(vec: &Vec<int>) {  
    // ...  
}
```



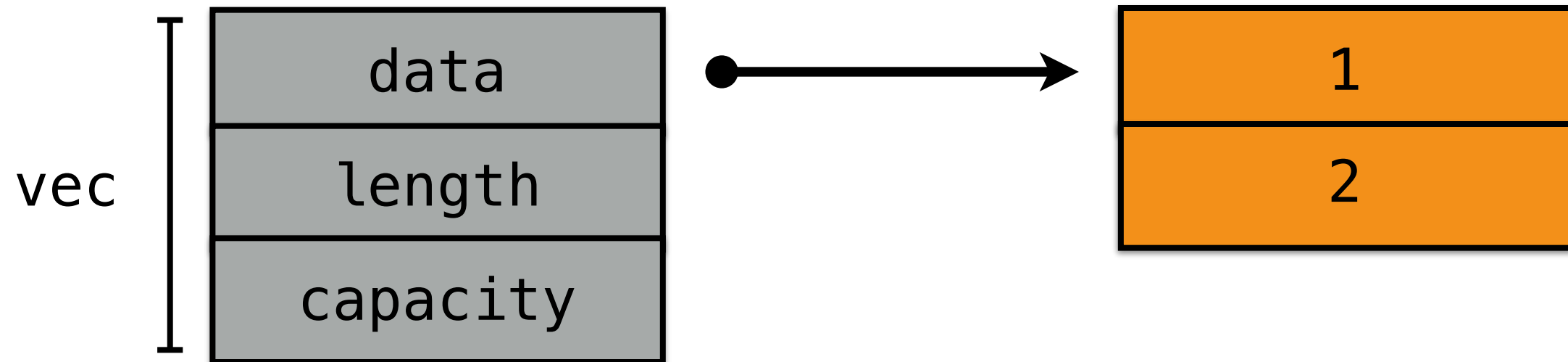
```
fn lender() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    use(&vec);  
  
    ...  
}
```

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



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

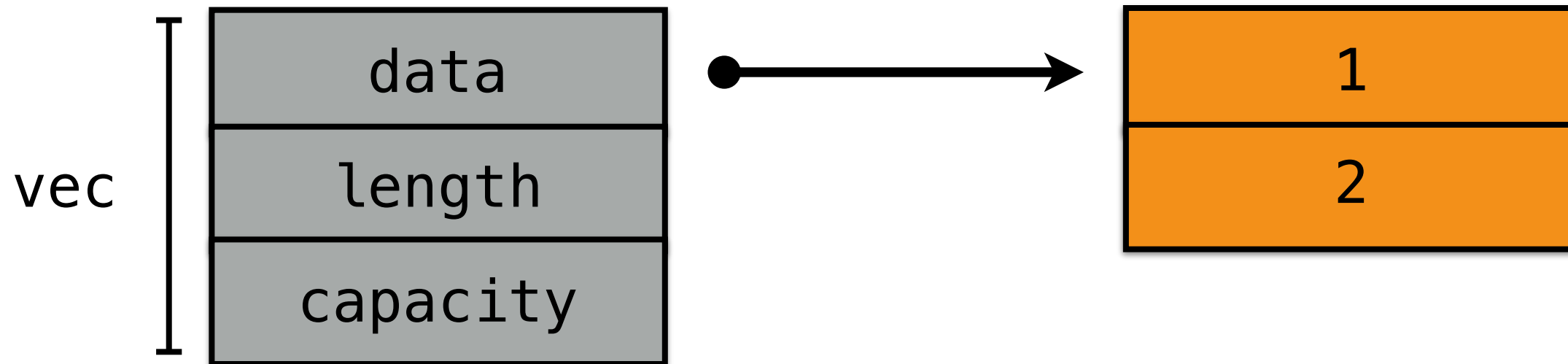
```
fn use(vec: &Vec<int>) {  
    // ...  
}
```



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

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

↑
“Shared reference
to Vec<int>”

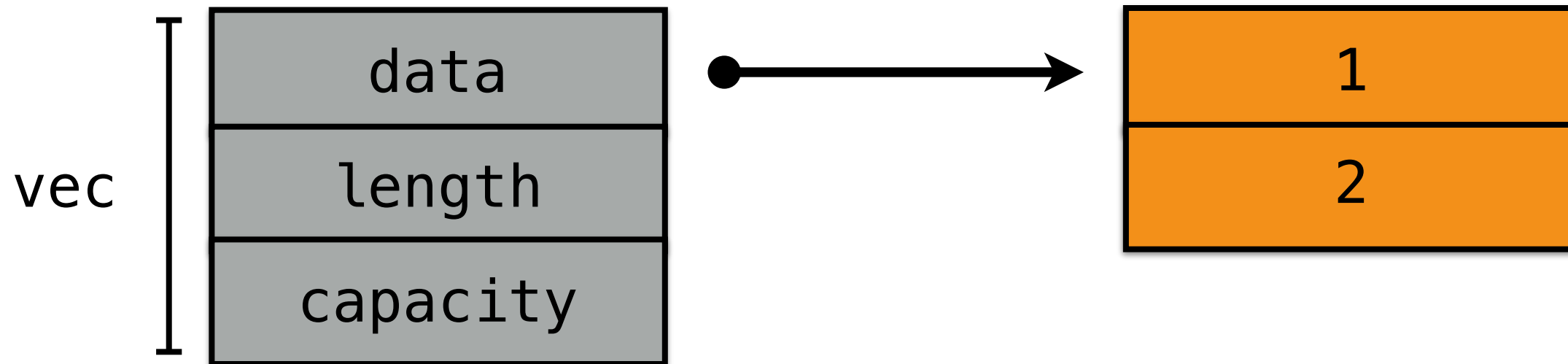


```
fn lender() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    use(&vec);  
    ...  
}
```

Loan out vec

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

“Shared reference
to Vec<int>”



```

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

```

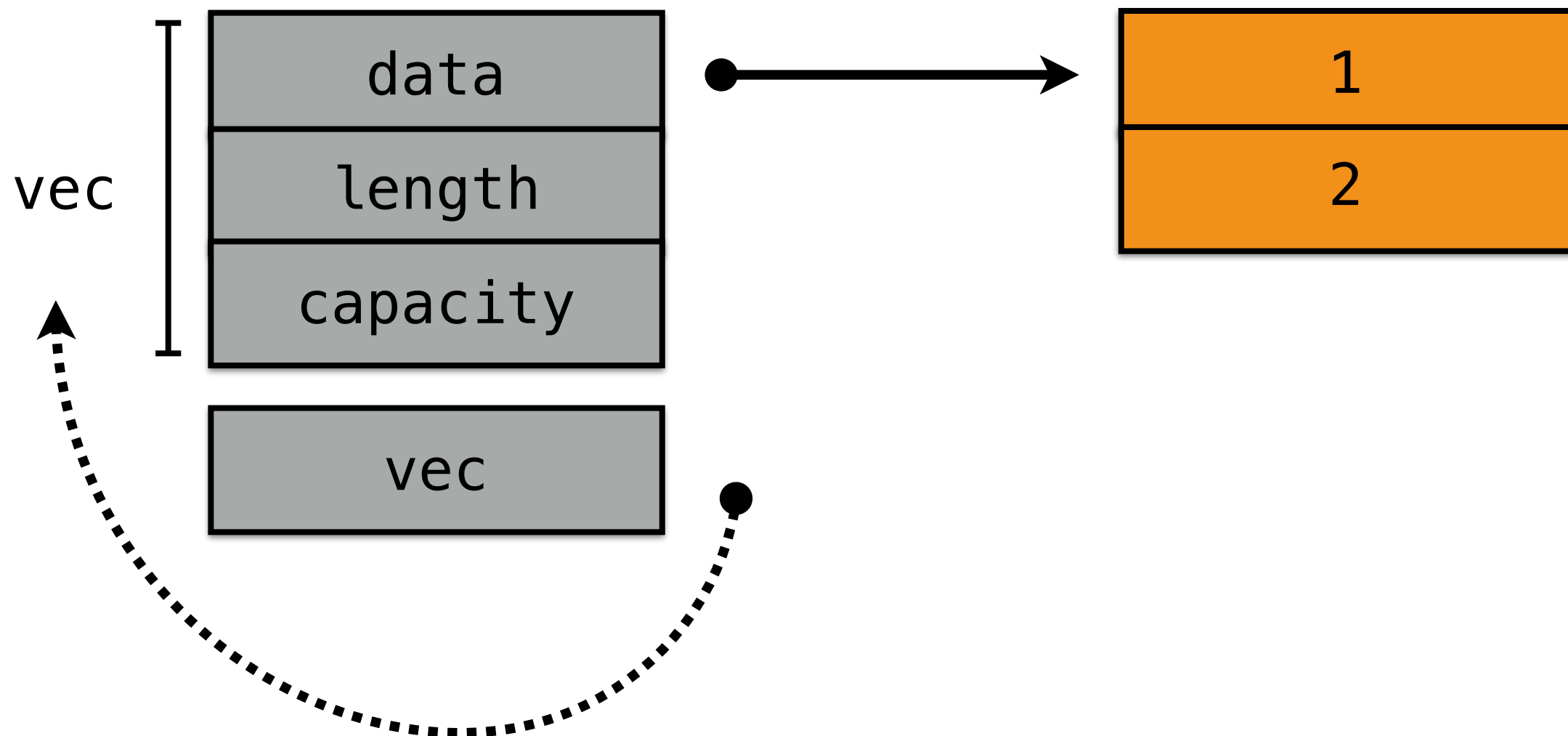
Loan out vec

```

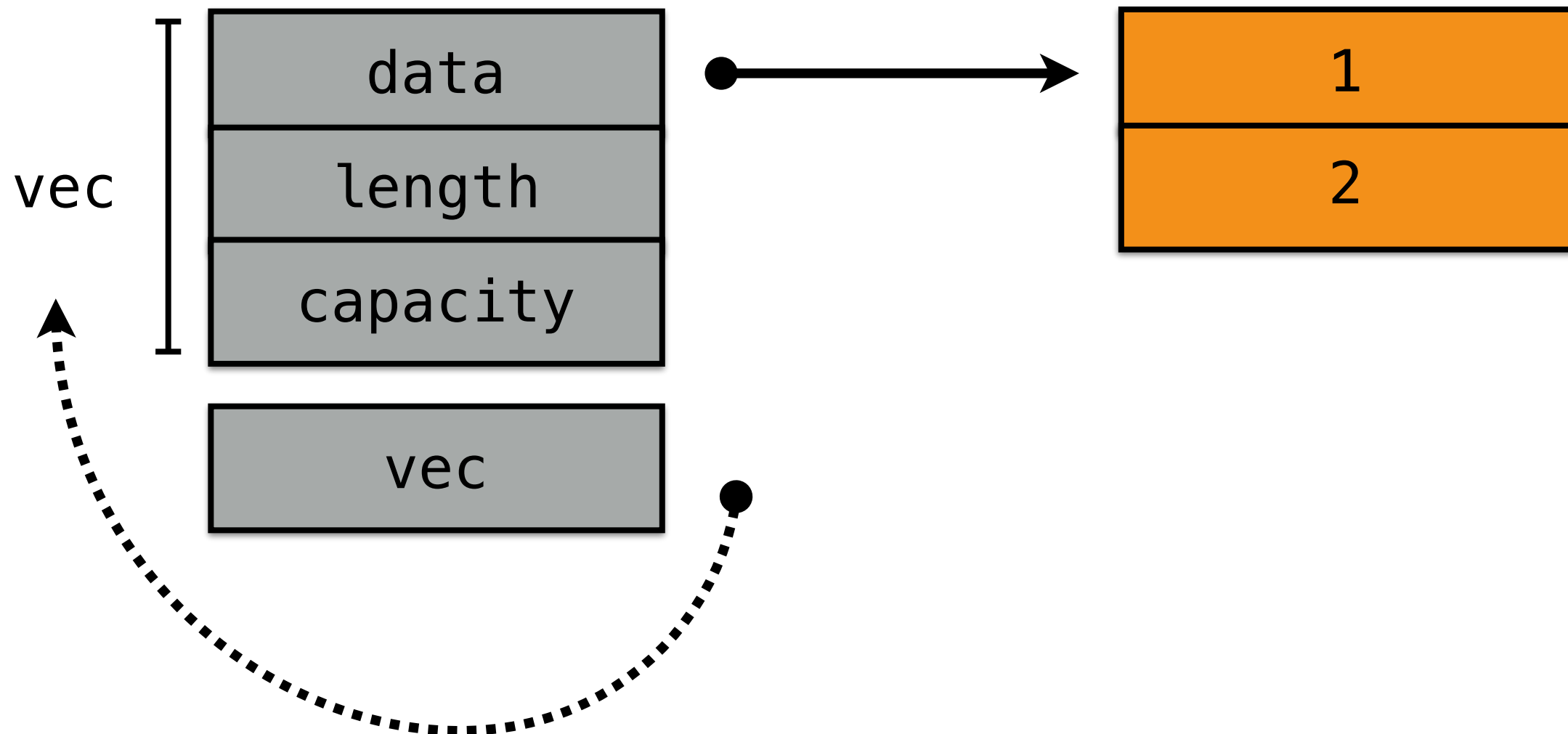
fn use(vec: &Vec<int>) {
  // ...
}

```

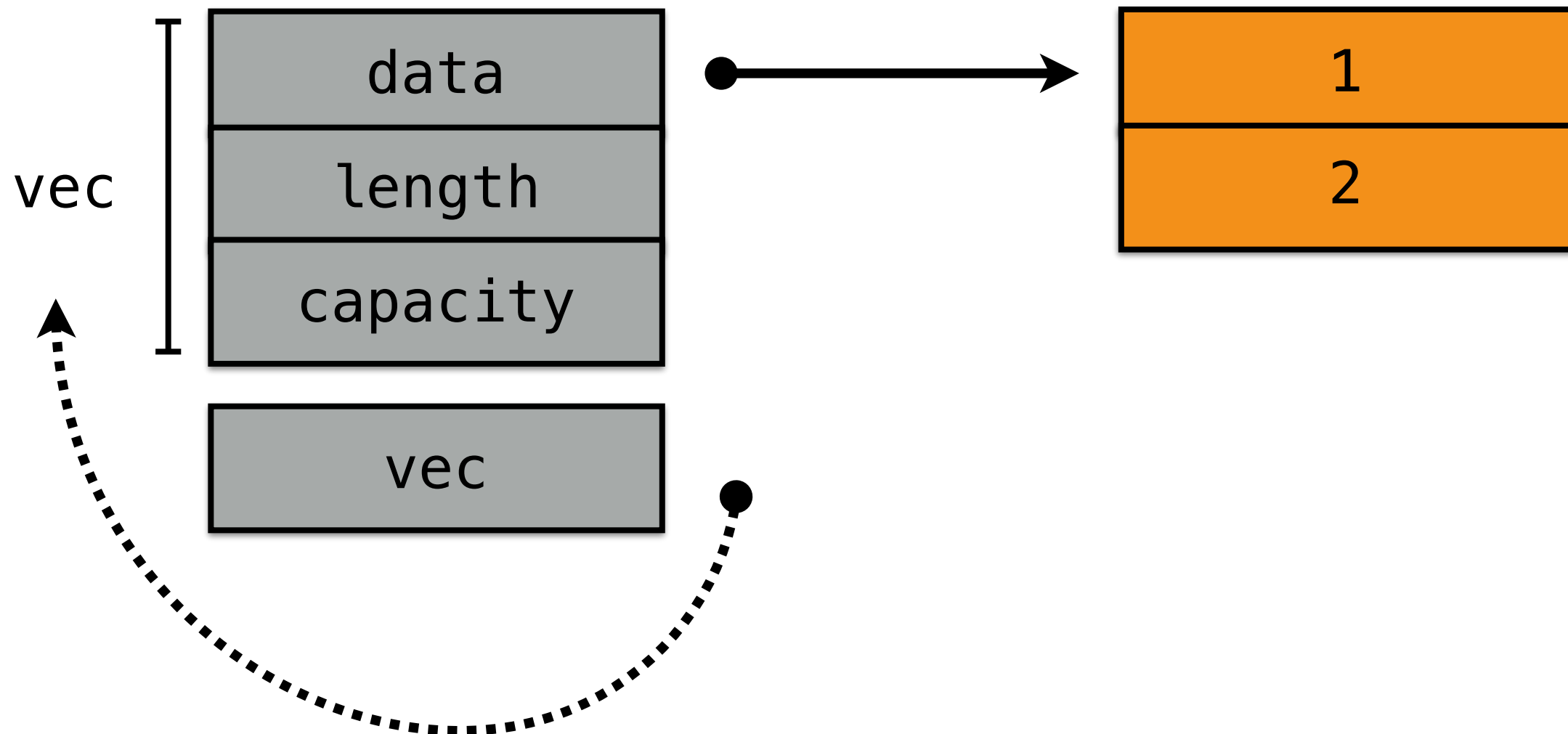
“Shared reference
to Vec<int>”



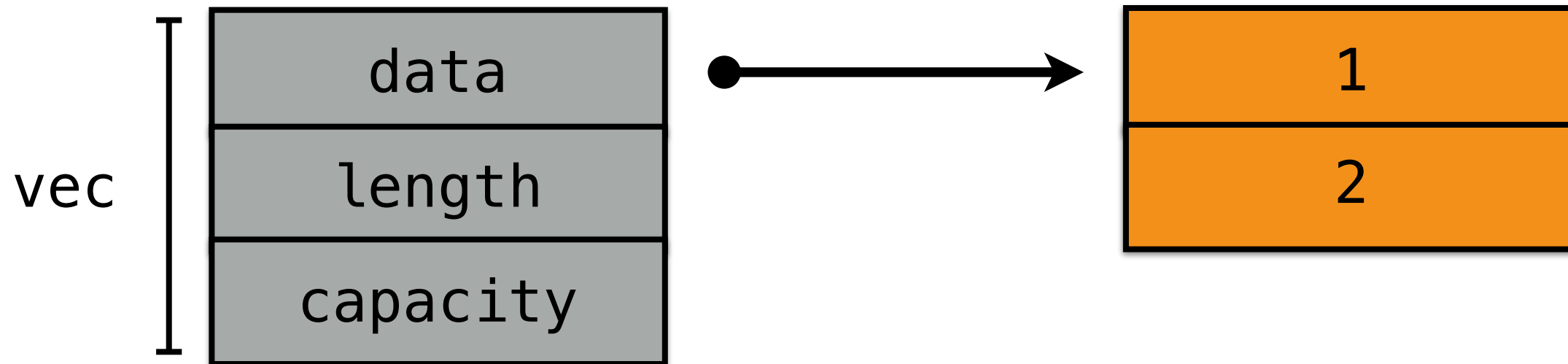
```
fn lender() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    use(&vec);  
    ...  
}  
→ fn use(vec: &Vec<int>) {  
    // ...  
}
```



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

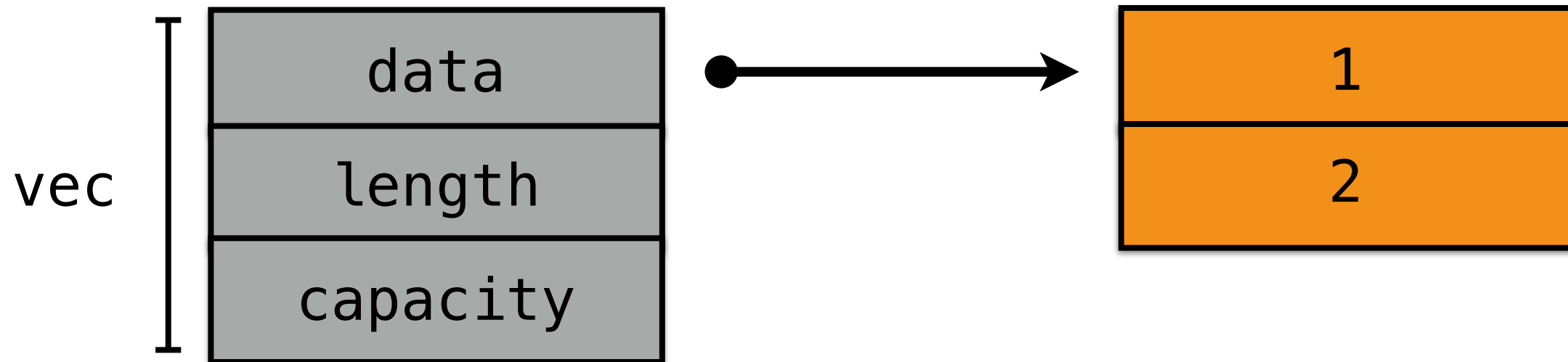
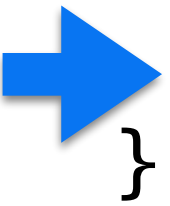


```
fn lender() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    use(&vec);  
    ...  
}  
fn use(vec: &Vec<int>) {  
    // ...  
}
```

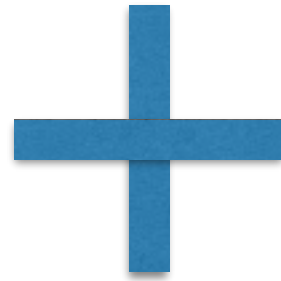


```
fn lender() {  
    let mut vec = Vec::new();  
    vec.push(1);  
    vec.push(2);  
    use(&vec);  
    ...  
}
```

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



Aliasing

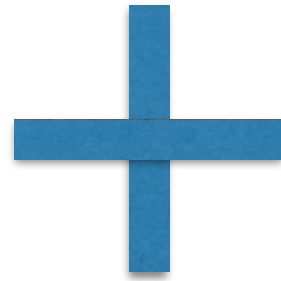


~~Mutation~~

Shared references are **immutable**:

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

Aliasing

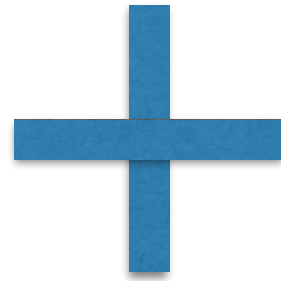


~~Mutation~~

Shared references are **immutable**:

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

Aliasing

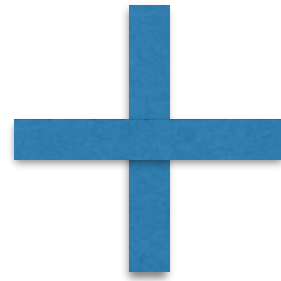


~~Mutation~~

Shared references are **immutable**:

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

Aliasing



~~Mutation~~

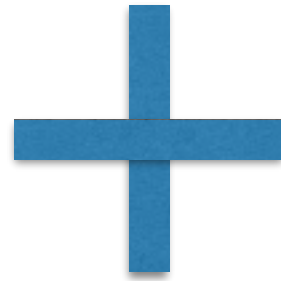
Shared references are **immutable**:

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



Error: cannot mutate shared reference

Aliasing



~~Mutation~~

Shared references are **immutable**:^{*}

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

Error: cannot mutate shared reference

^{*} Actually: mutation only in controlled circumstances

Mutable references

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

Mutable references

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



mutable reference to Vec<int>

Mutable references

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

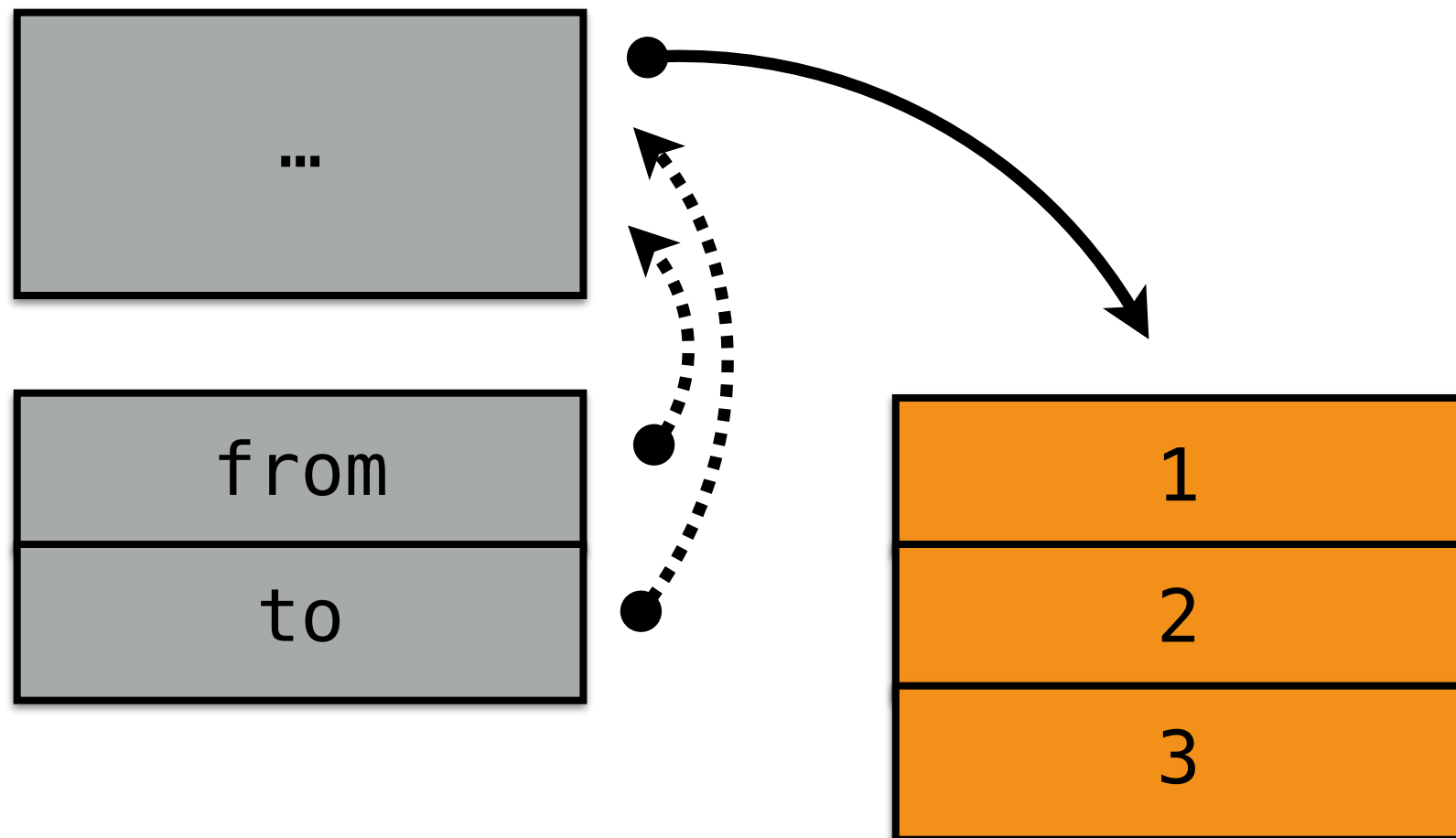
↑
push() is legal

↑
mutable reference to Vec<int>

Efficient Iteration

➔

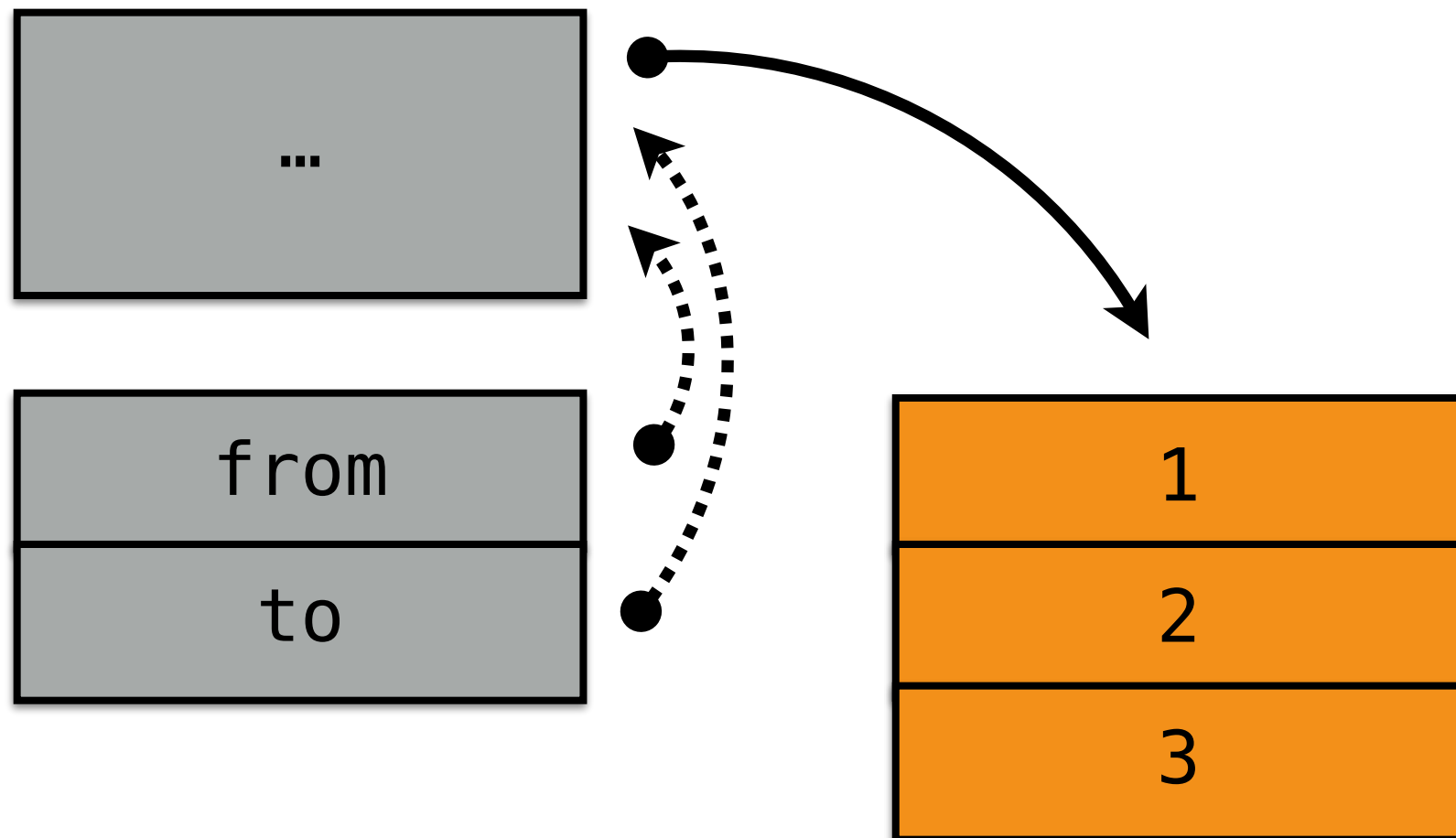
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {  
    for elem in from.iter() {  
        to.push(*elem);  
    }  
}
```



Efficient Iteration

➔

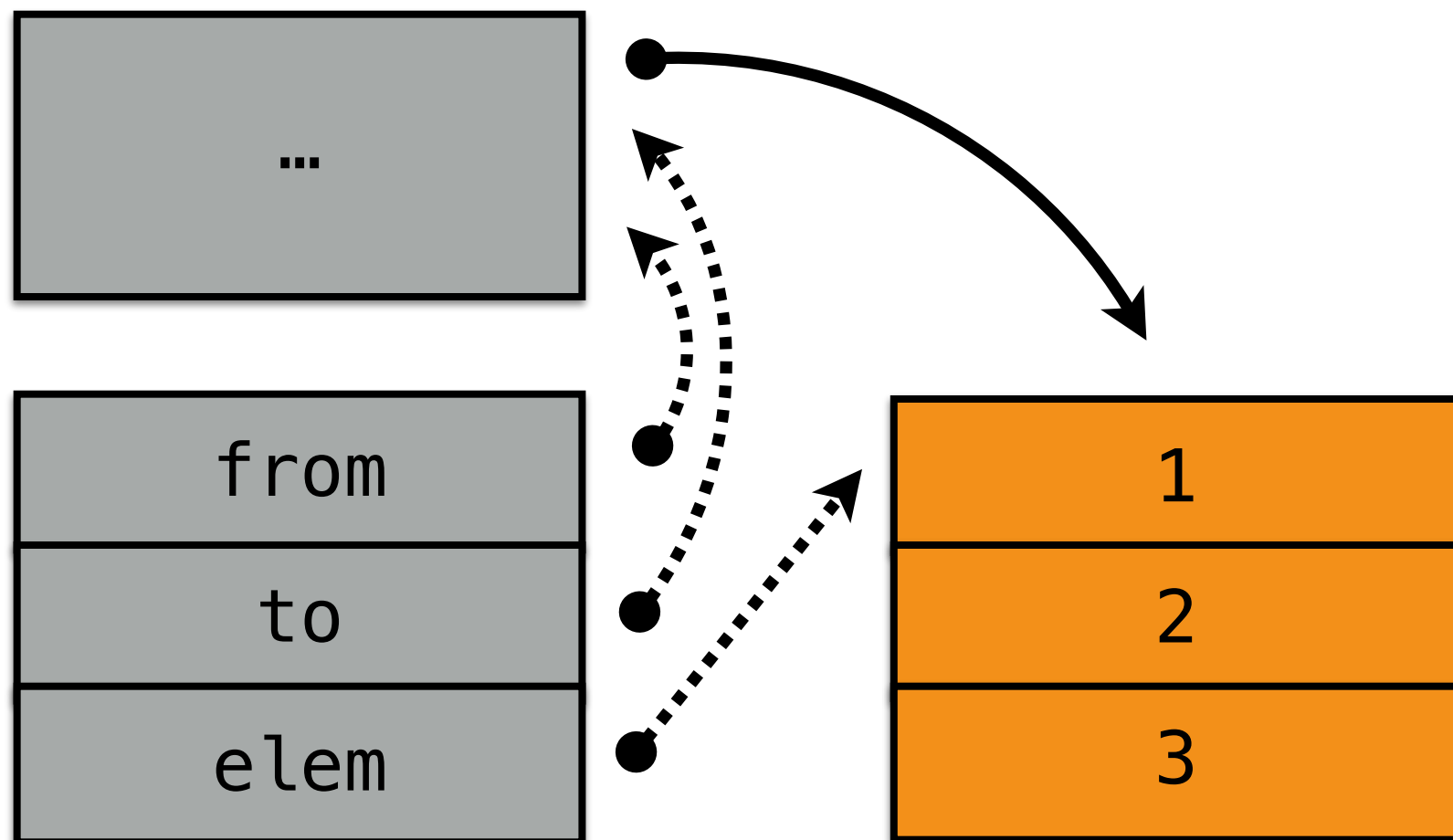
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {  
    for elem in from.iter() {  
        to.push(*elem);  
    }  
}
```



Efficient Iteration

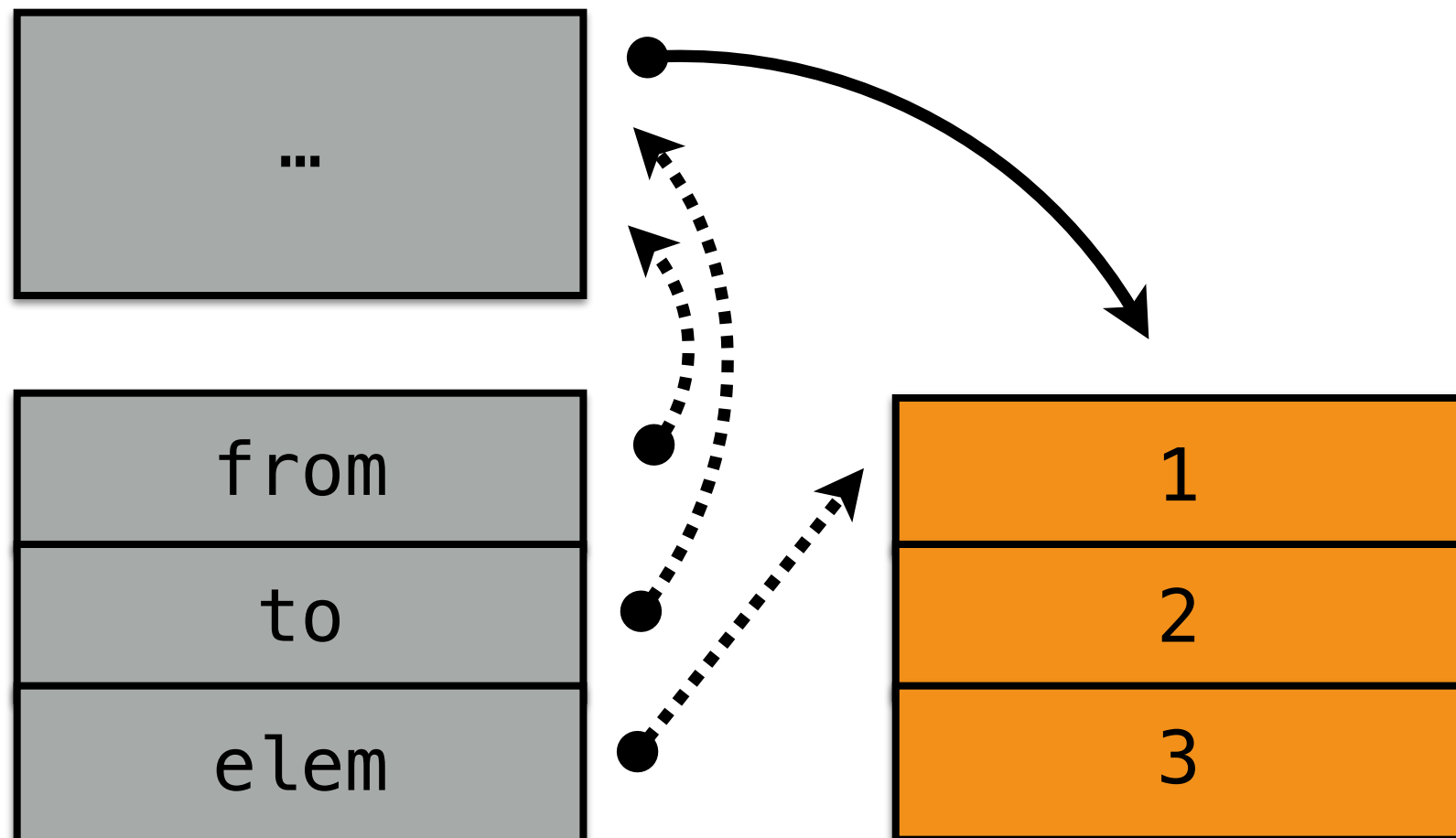
➔

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



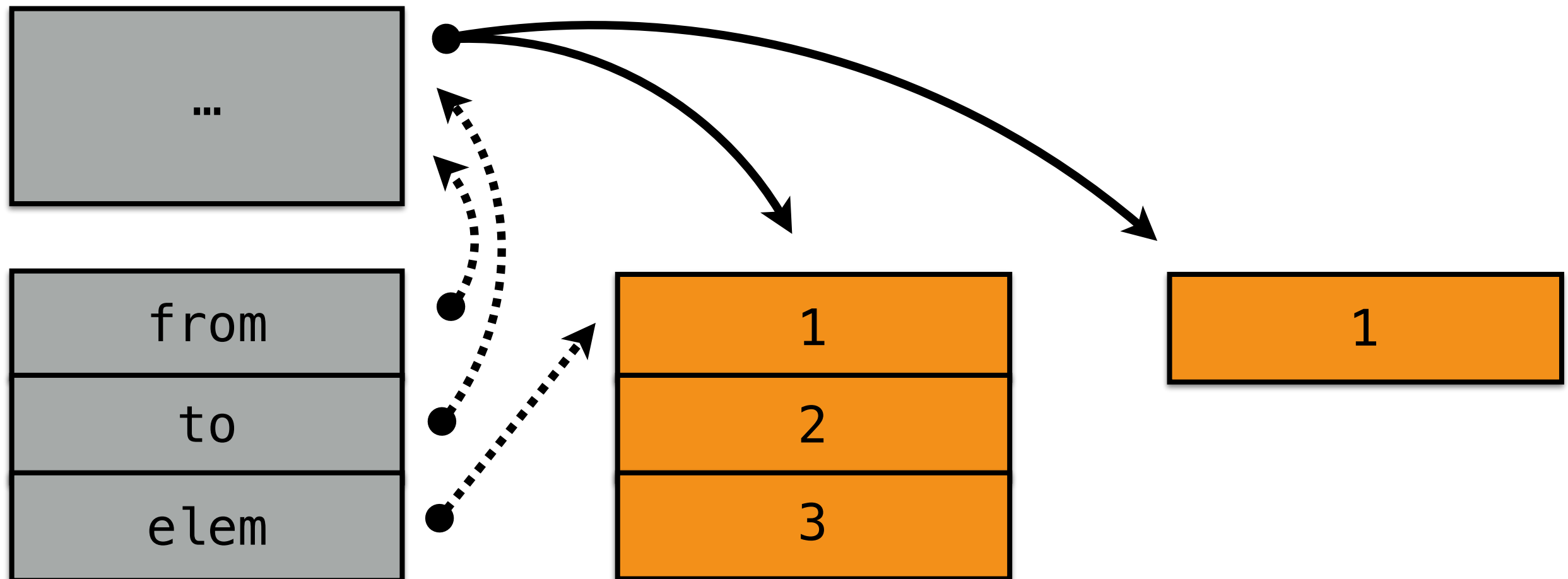
Efficient Iteration

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



Efficient Iteration

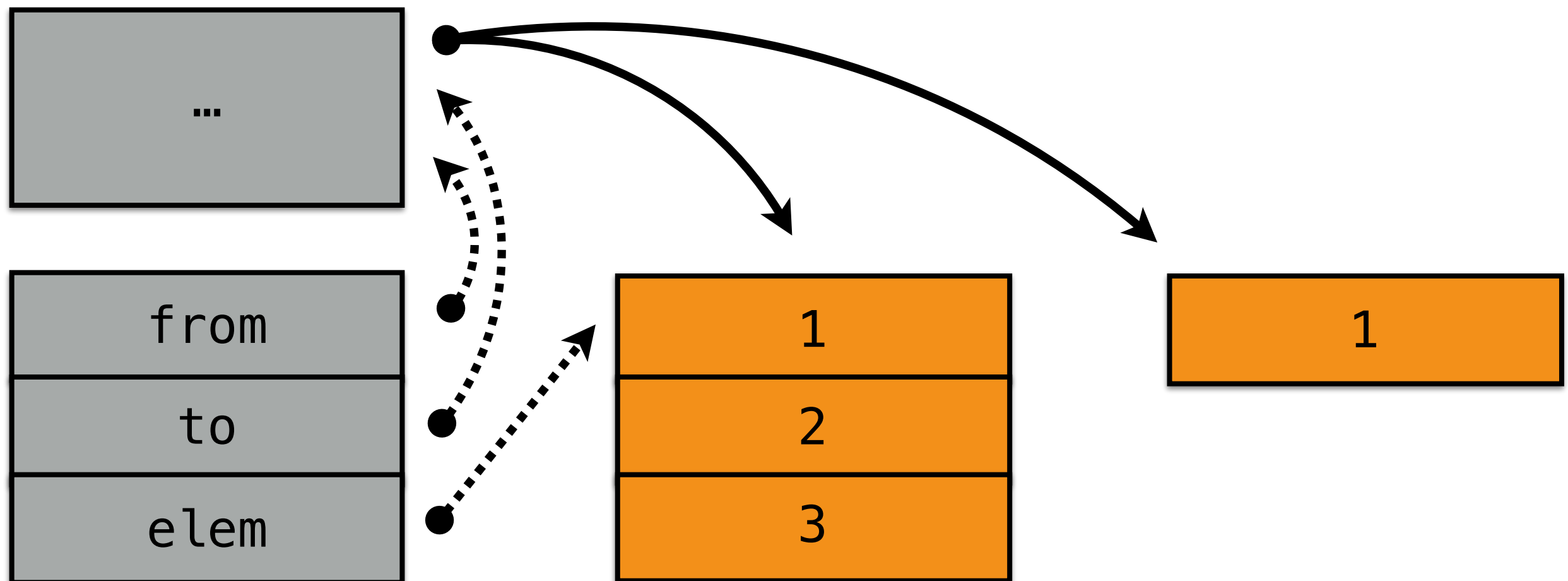
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {  
    for elem in from.iter() {  
        to.push(*elem);  
    }  
}
```



Efficient Iteration



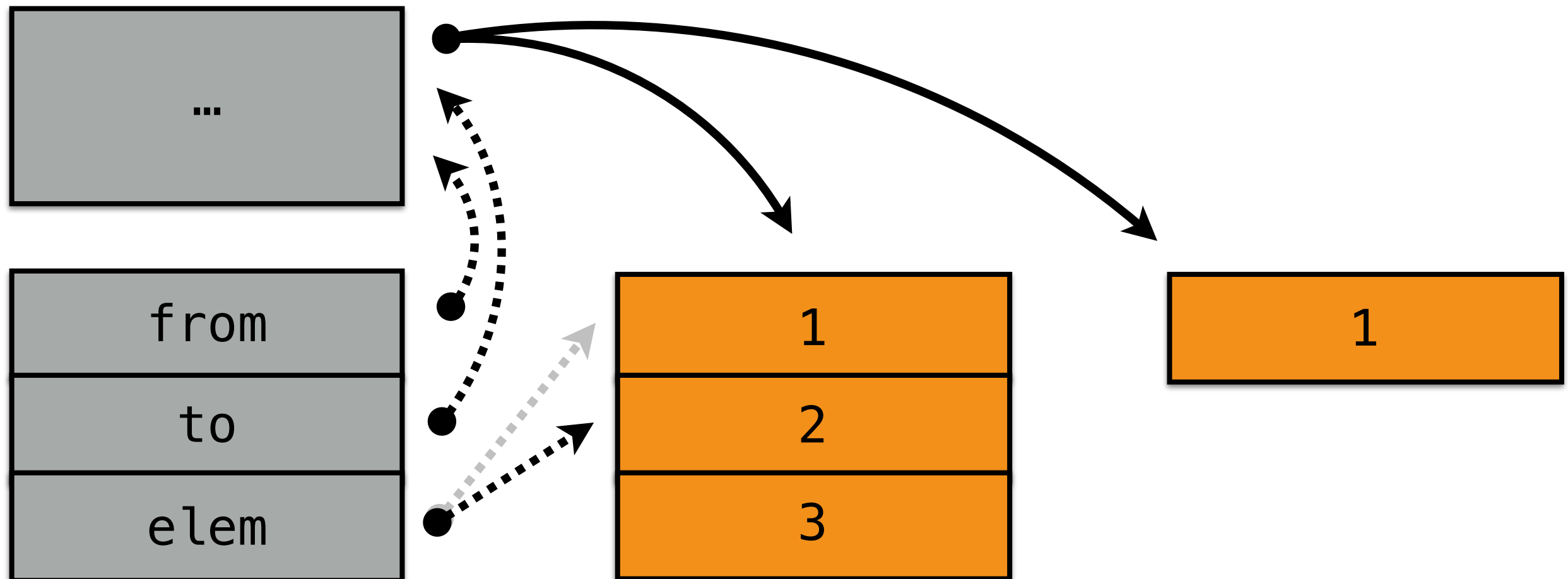
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {  
    for elem in from.iter() {  
        to.push(*elem);  
    }  
}
```



Efficient Iteration

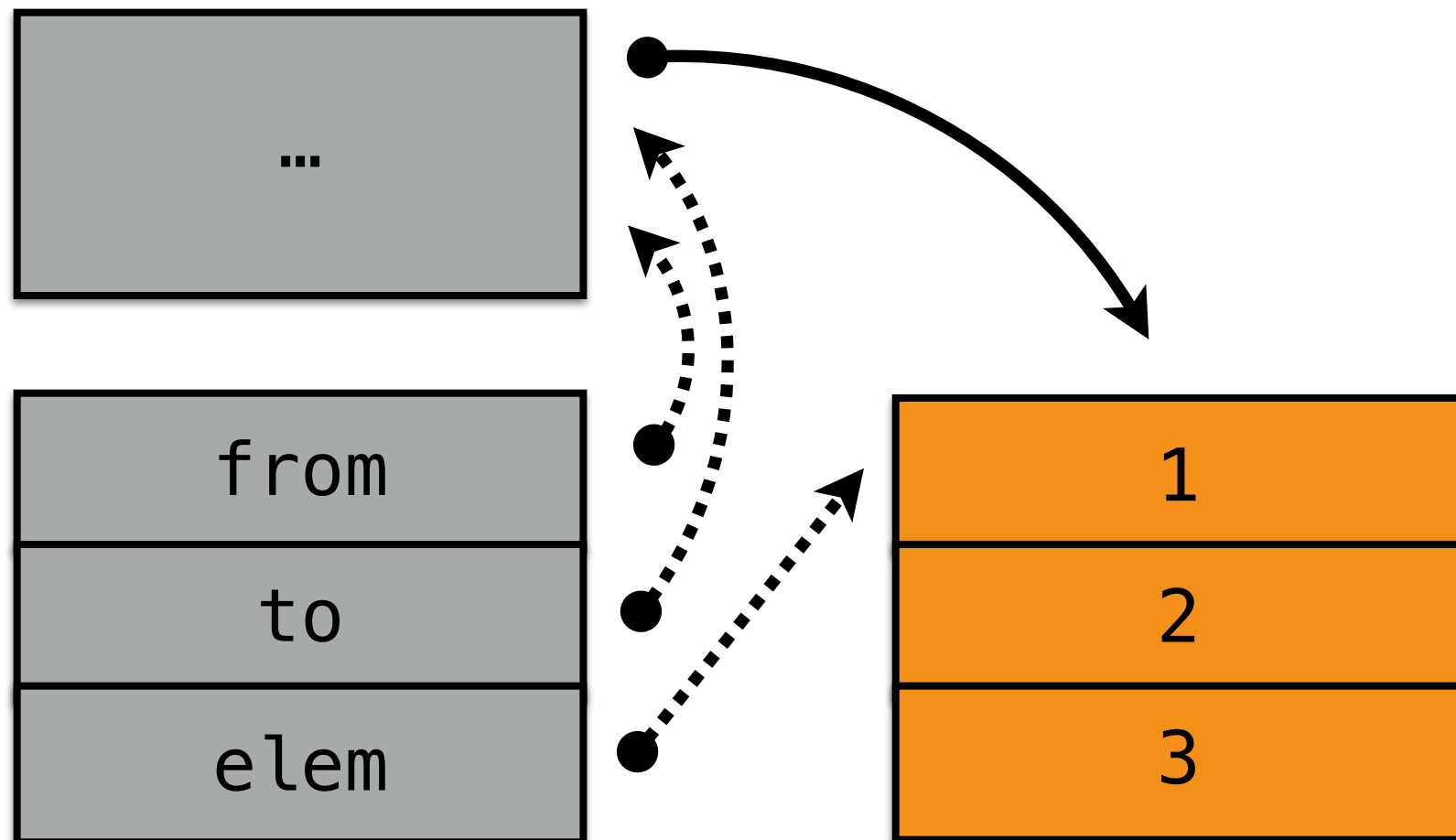


```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {  
    for elem in from.iter() {  
        to.push(*elem);  
    }  
}
```



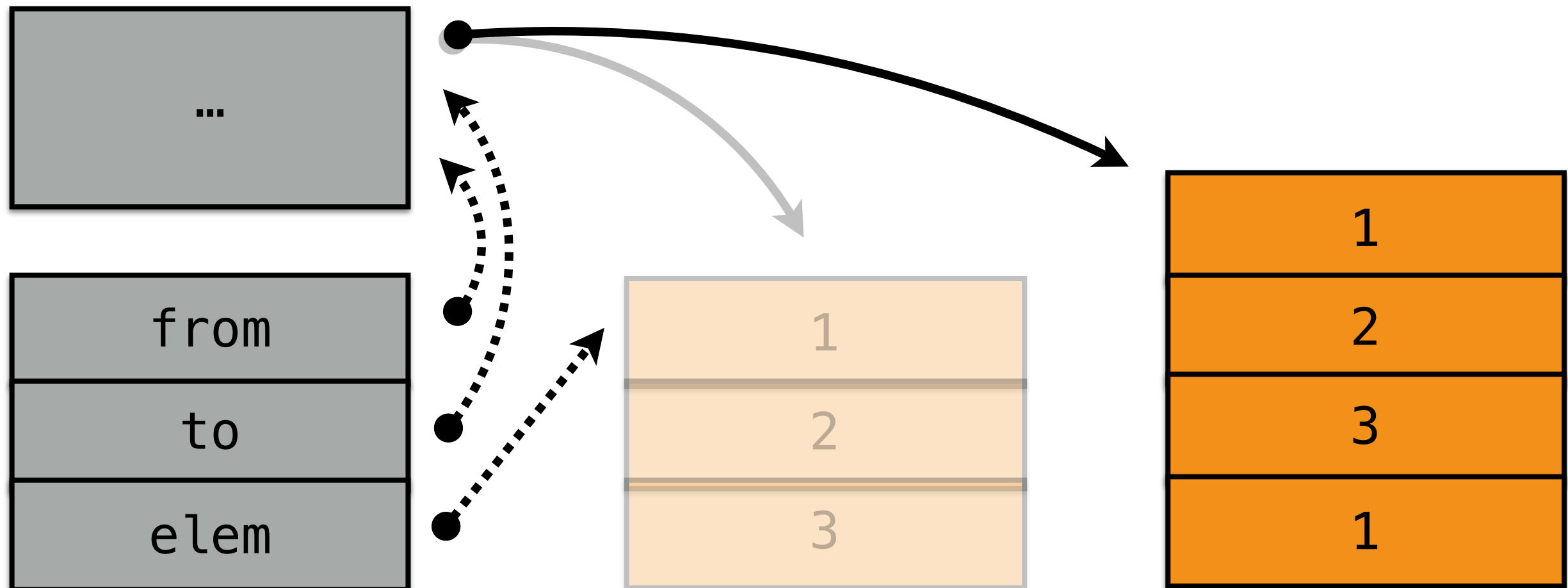
What if **from** and **to** are equal?

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



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

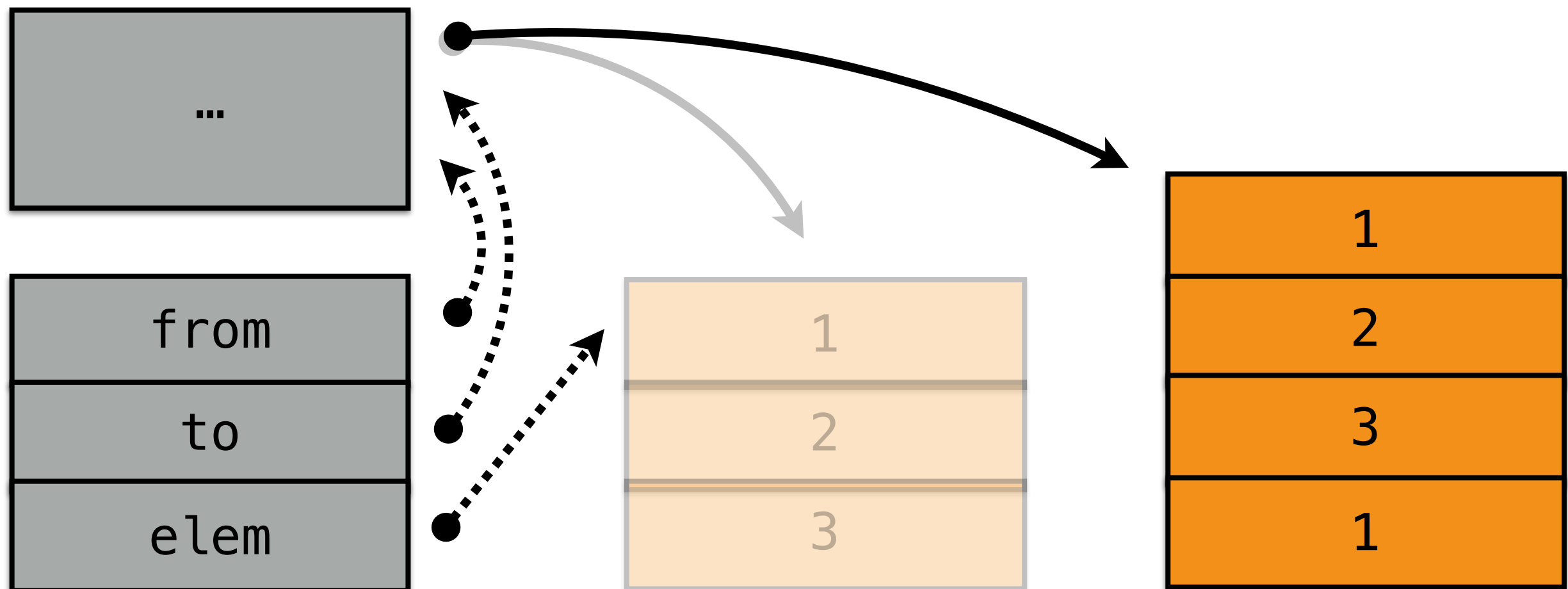
```
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {  
    for elem in from.iter() {  
        to.push(*elem);  
    }  
}
```



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

➡

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

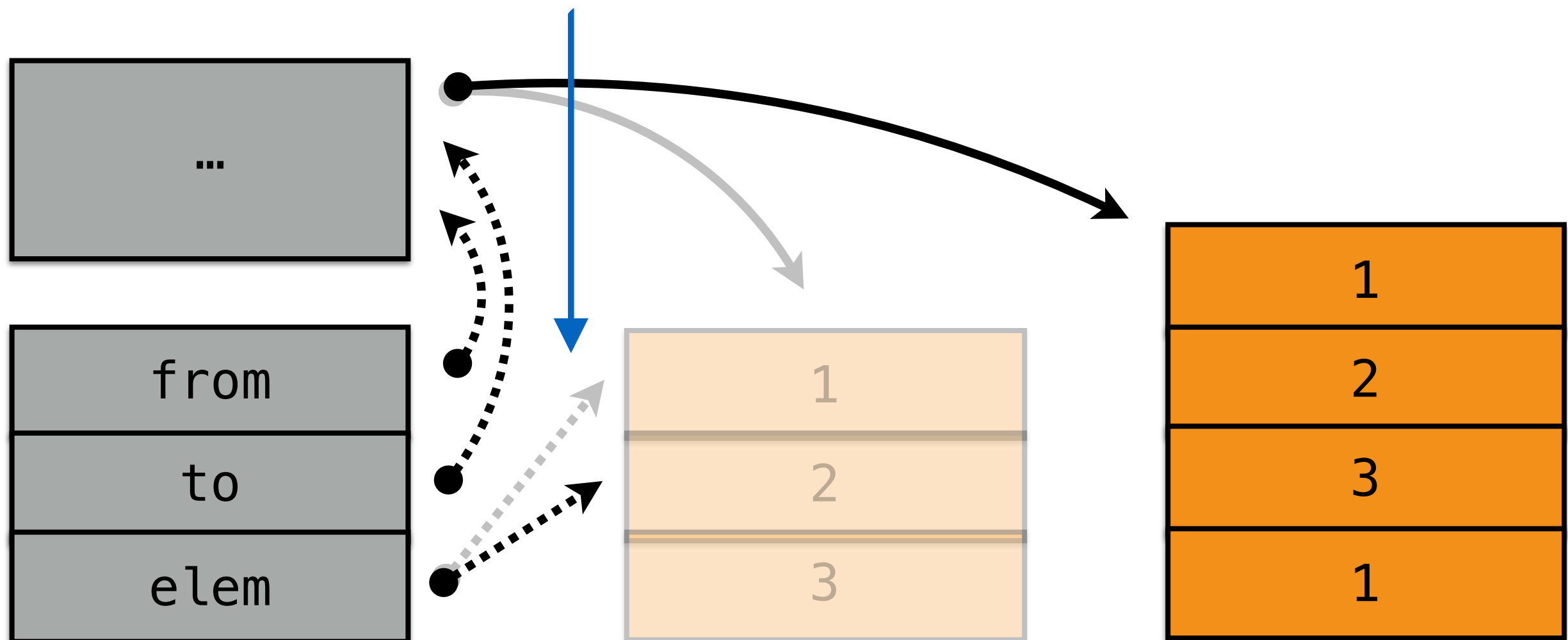


What if **from** and **to** are equal?

➔

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

dangling pointer



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

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

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

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



shared reference

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

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

shared reference

Error: cannot have both shared
and mutable reference at same
time

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

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

shared reference

Error: cannot have both shared
and mutable reference at same
time

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

```
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



```
{  
    let mut vec = Vec::new();  
    ...  
    for i in range(0, vec.len()) {  
        let elem: &int = &vec[i];  
        ...  
        vec.push(...);  
    }  
    ...  
    vec.push(...);  
}
```

Borrows restrict access to the original path for their duration.

&	no writes, no moves
&mut	no access at all

```

{
  let mut vec = Vec::new();
  ...
  for i in range(0, vec.len()) {
    let elem: &int = &vec[i];
    ...
    vec.push(...);
  }
  ...
  vec.push(...);
}

```



Borrows restrict access to the original path for their duration.

&	no writes, no moves
&mut	no access at all

```

{
    let mut vec = Vec::new();
    ...
    for i in range(0, vec.len()) {
        let elem: &int = &vec[i];
        ...
        vec.push(...);
    }
    ...
    vec.push(...);
}

```



Borrows restrict access to the original path for their duration.

&	no writes, no moves
&mut	no access at all

```

{
  let mut vec = Vec::new();
  ...
  for i in range(0, vec.len()) {
    let elem: &int = &vec[i];
    ...
vec.push(...);
  }
  ...
  vec.push(...);
}

```




Error: vec[i] is borrowed,
cannot mutate

Borrows restrict access to the original path for their duration.

&	no writes, no moves
&mut	no access at all

```

{
  let mut vec = Vec::new();
  ...
  for i in range(0, vec.len()) {
    let elem: &int = &vec[i];
    ...
vec.push(...);
  }
  ...
  vec.push(...);
}

```




Error: vec[i] is borrowed,
cannot mutate


OK. loan expired.

Borrows restrict access to the original path for their duration.

&	no writes, no moves
&mut	no access at all

Concurrency

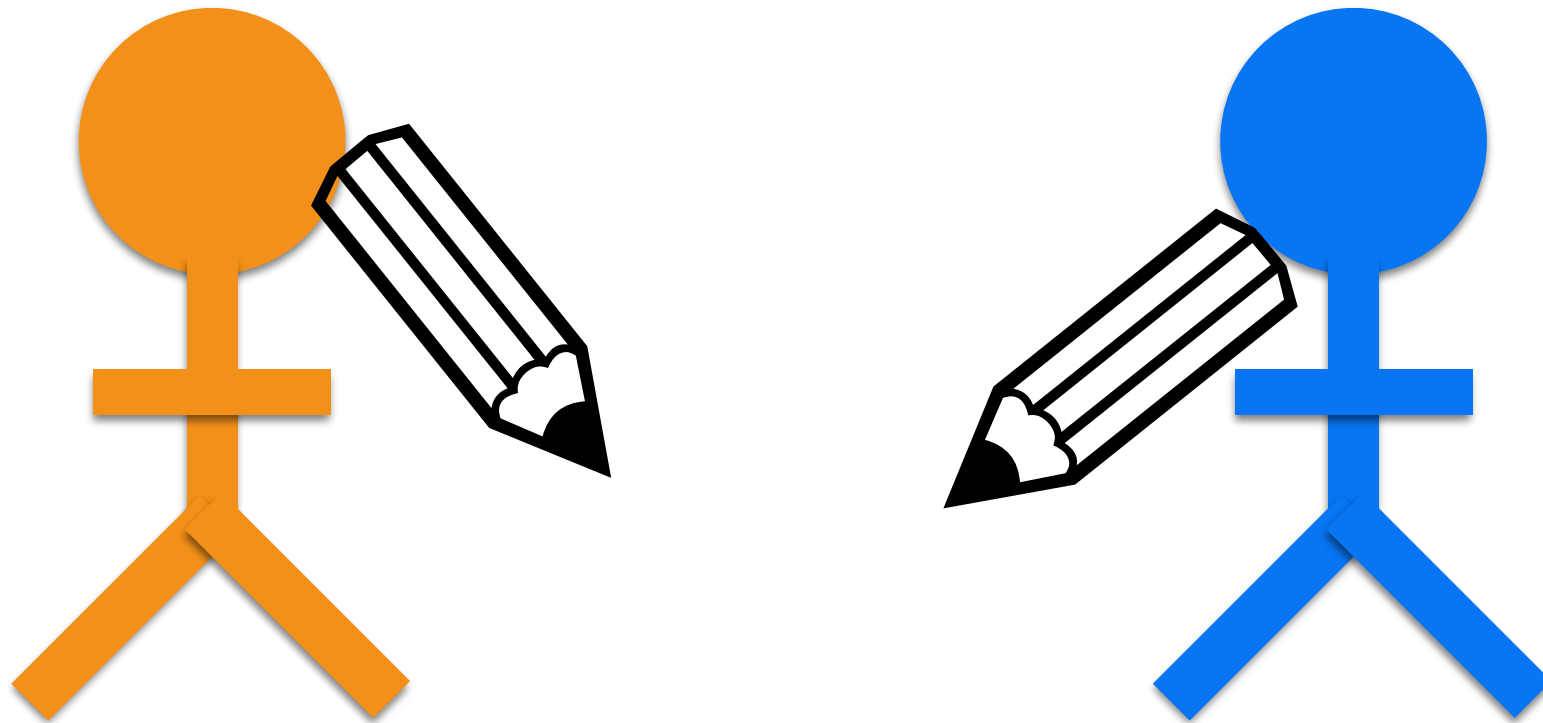
n. several computations executing simultaneously, and potentially interacting with each other

Data race



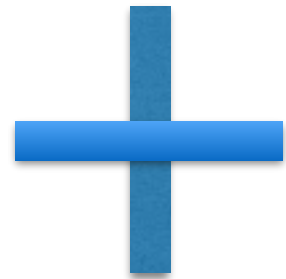
Two **unsynchronized** threads
accessing **same data**
where **at least one writes**.

Data race



Two **unsynchronized** threads
accessing **same data**
where **at least one writes**.

Aliasing



Mutation



No ordering



Data race

Sound familiar?

Aliasing

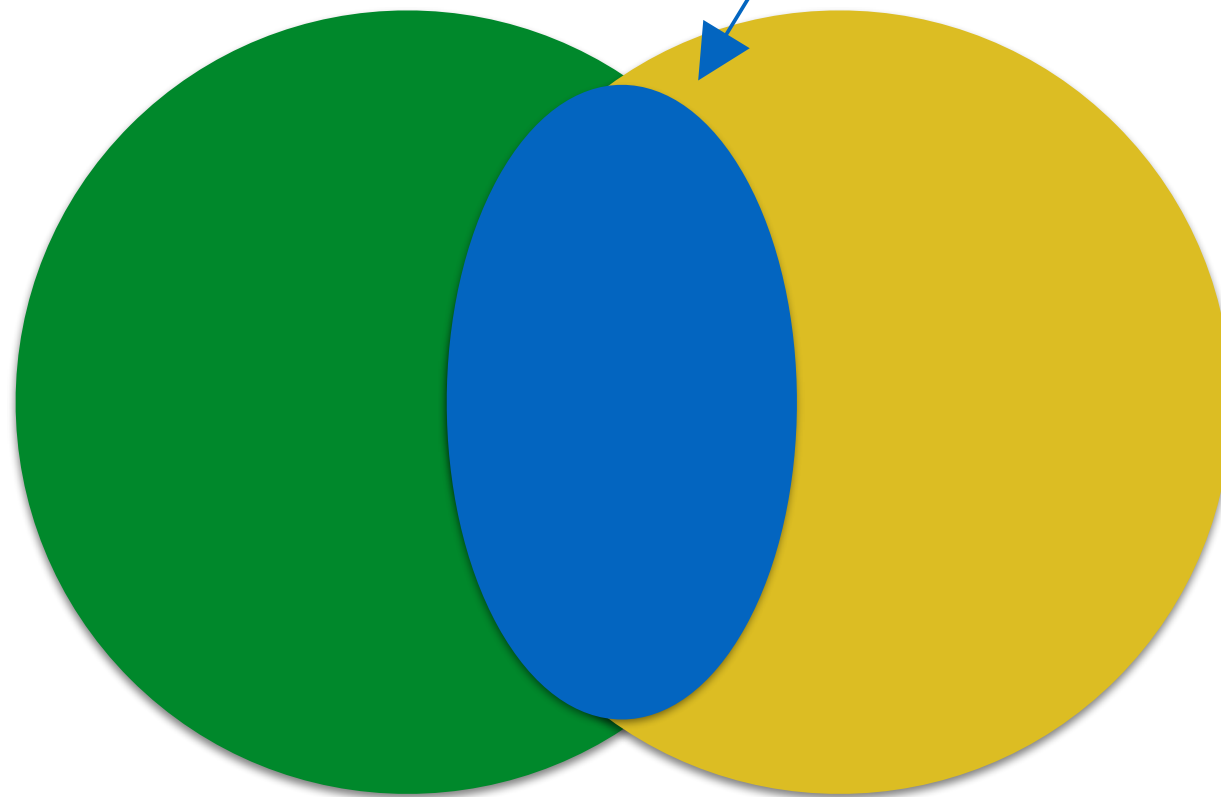
Mutation

No ordering

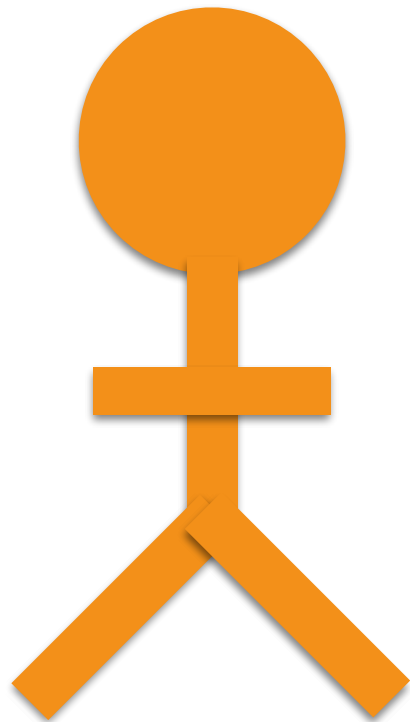
Data race

Actors—forbid **aliasing**

Rust—forbid both from
occurring simultaneously

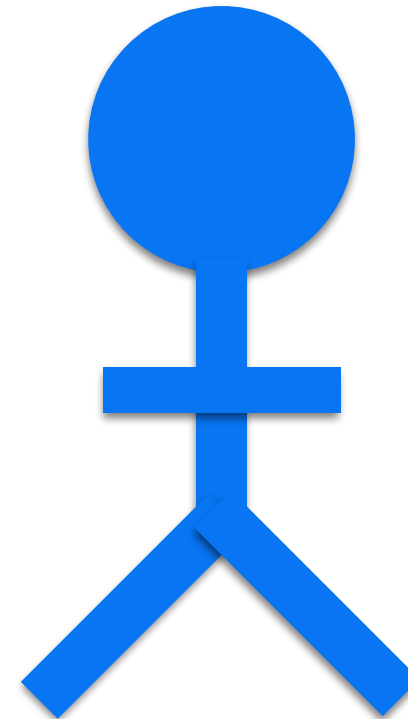
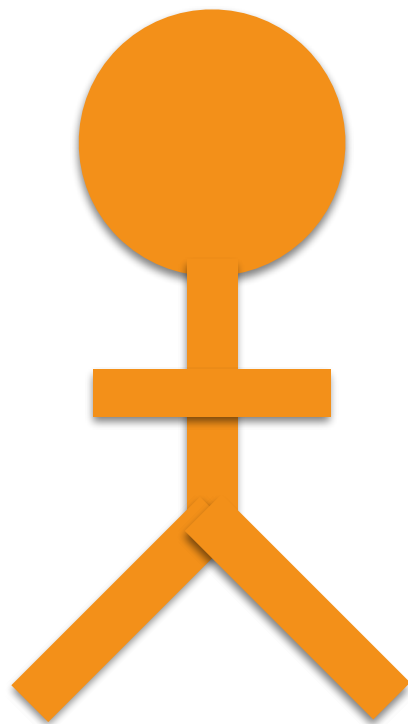


Functional—forbid **mutation**



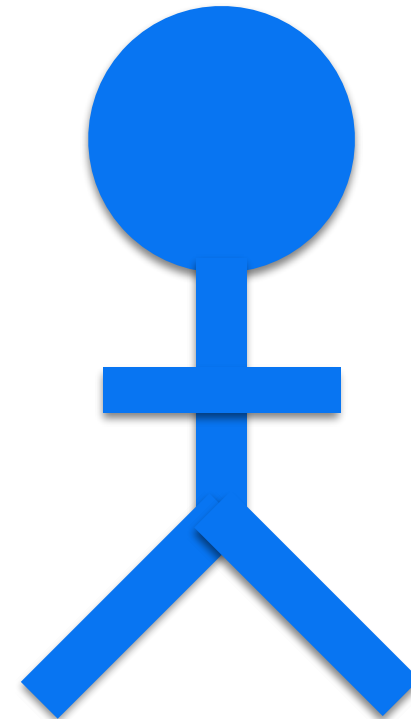
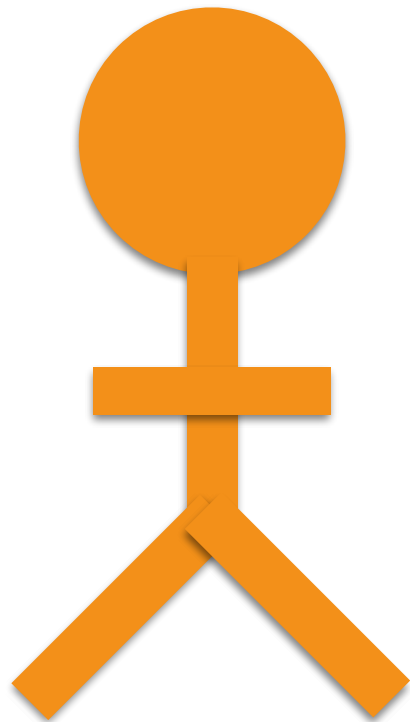
Messaging

(ownership)



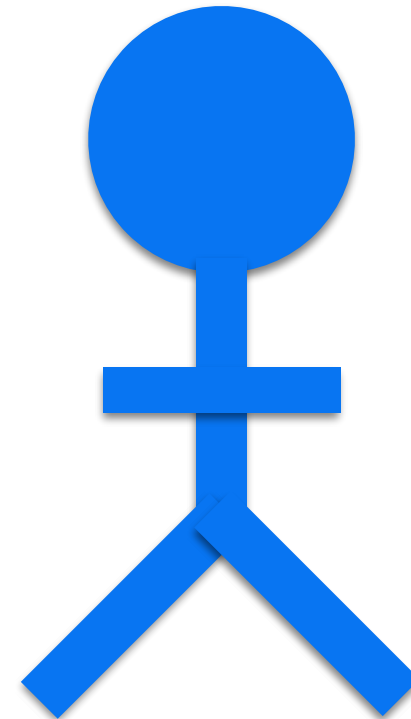
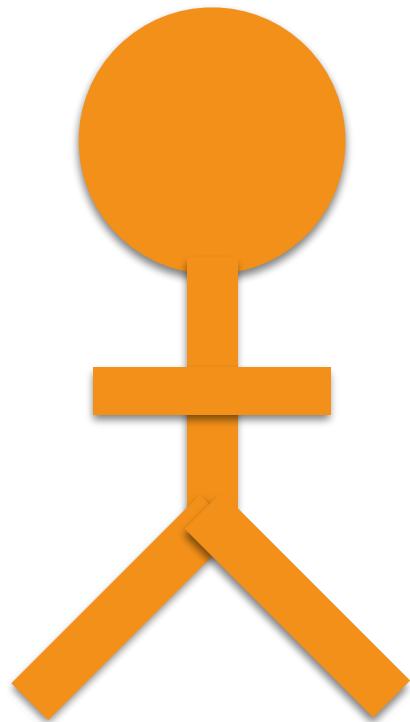
Messaging

(ownership)



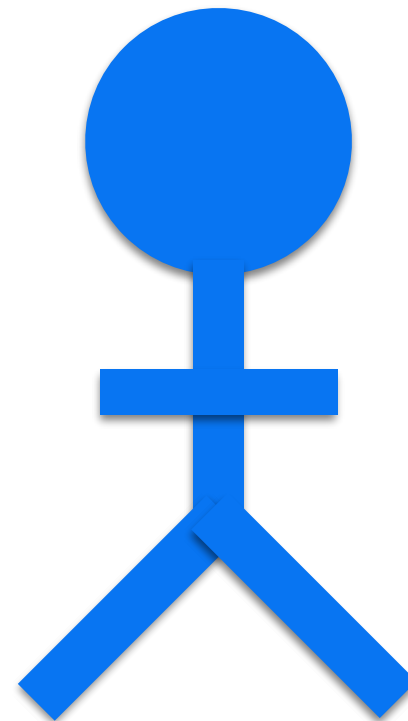
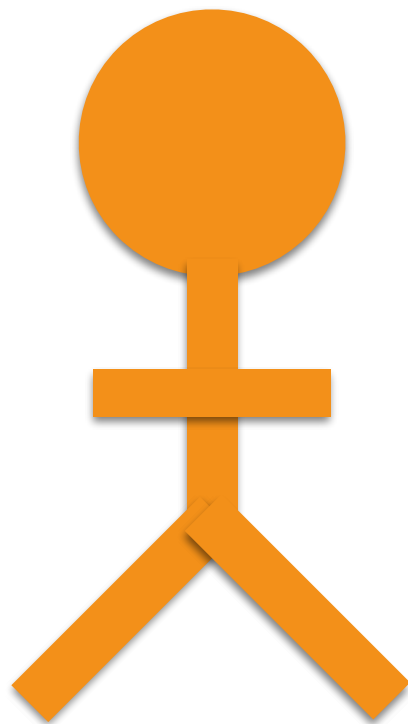
Messaging

(ownership)



Messaging

(ownership)



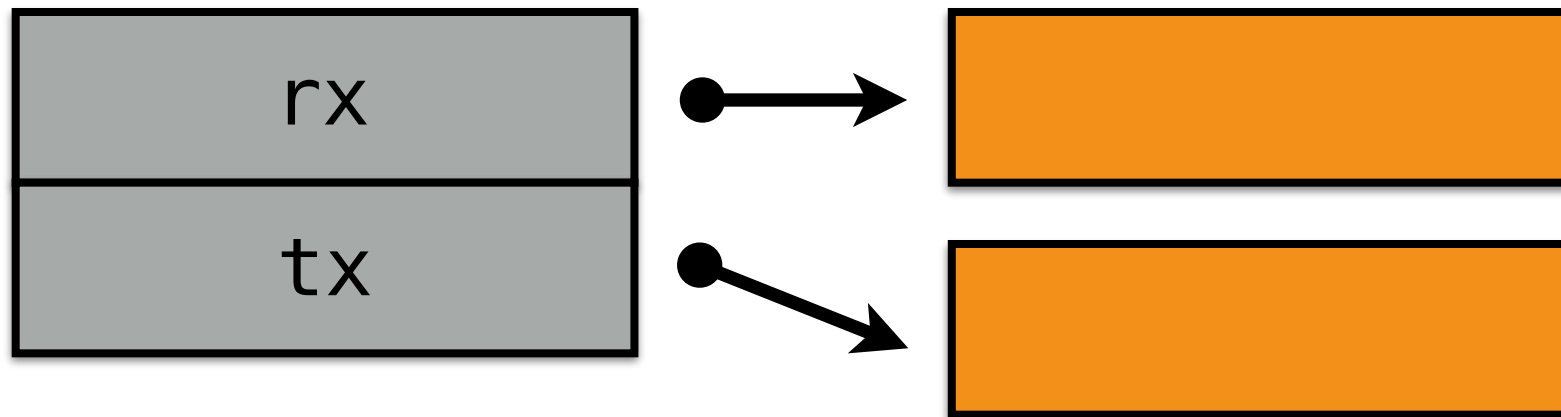
Messaging

(ownership)

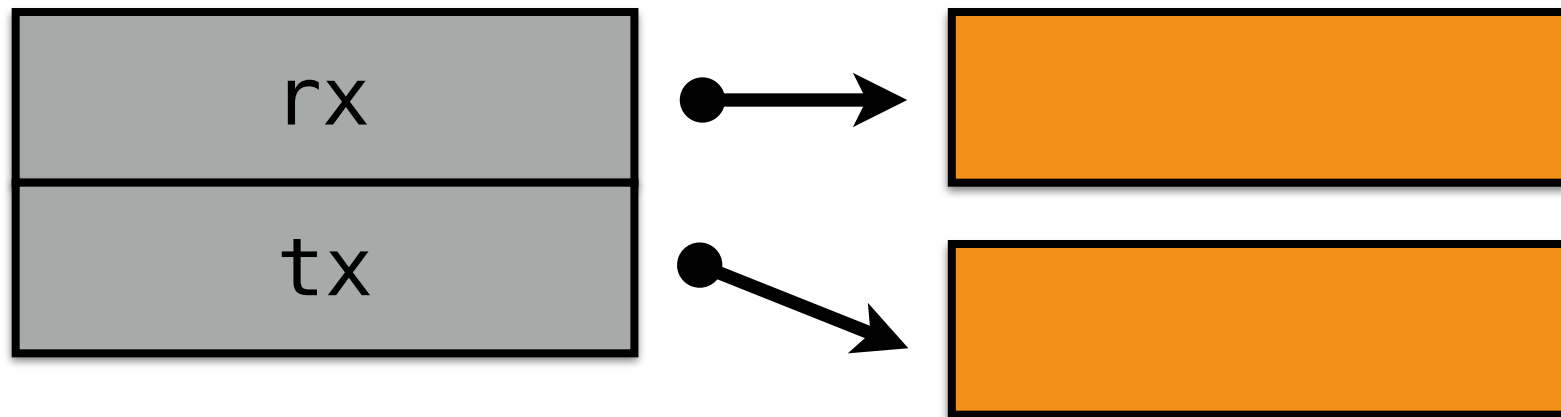


```
fn parent() {  
  let (tx, rx) = channel();  
  spawn(move || {...});  
  let m = rx.recv();  
}
```

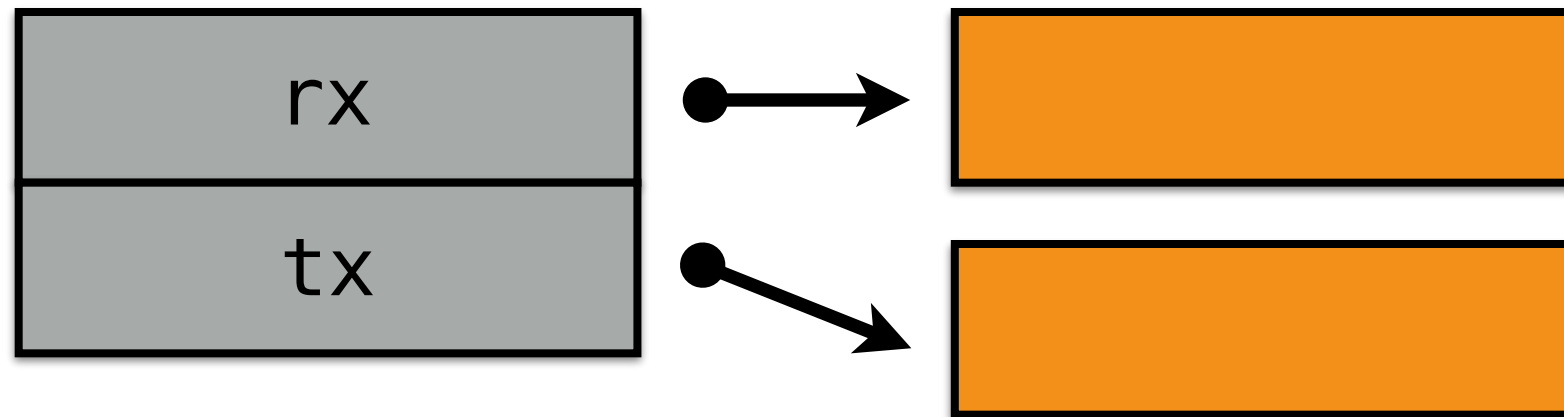
```
fn parent() {  
    let (tx, rx) = channel();  
    spawn(move || {...});  
    let m = rx.recv();  
}
```



```
fn parent() {  
    let (tx, rx) = channel();  
    spawn(move || {...});  
    let m = rx.recv();  
}
```

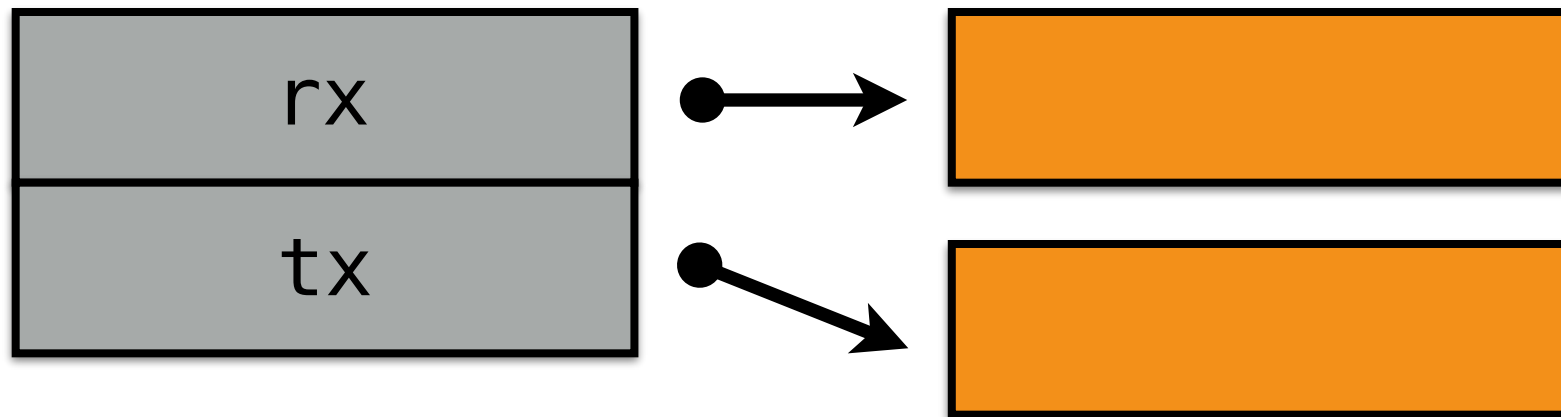


```
fn parent() {  
  let (tx, rx) = channel();  
  spawn(move || {...});  
  let m = rx.recv();  
}
```



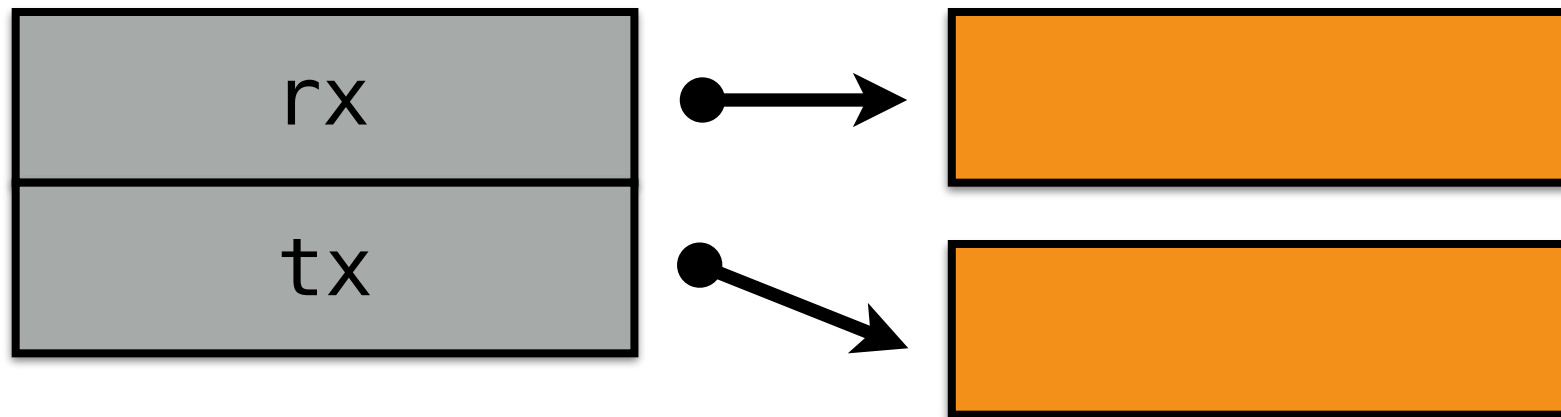
```
fn parent() {  
    let (tx, rx) = channel();  
    spawn(move || {...});  
    let m = rx.recv();  
}
```

```
move || {  
    let m = Vec::new();  
    ...  
    tx.send(m);  
}
```



```
fn parent() {  
    let (tx, rx) = channel();  
    spawn(move || {...});  
    let m = rx.recv();  
}
```

```
move || {  
    let m = Vec::new();  
    ...  
    tx.send(m);  
}
```



```
fn parent() {  
  let (tx, rx) = channel();  
  spawn(move || {...});  
  let m = rx.recv();  
}
```

```
move || {  
  let m = Vec::new();  
  ...  
  tx.send(m);  
}
```



```

fn parent() {
  let (tx, rx) = channel();
  spawn(move || {...});
  let m = rx.recv();
}

```

→

```

move || {
  let m = Vec::new();
  ...
  tx.send(m);
}

```



```

fn parent() {
  let (tx, rx) = channel();
  spawn(move || {...});
  let m = rx.recv();
}

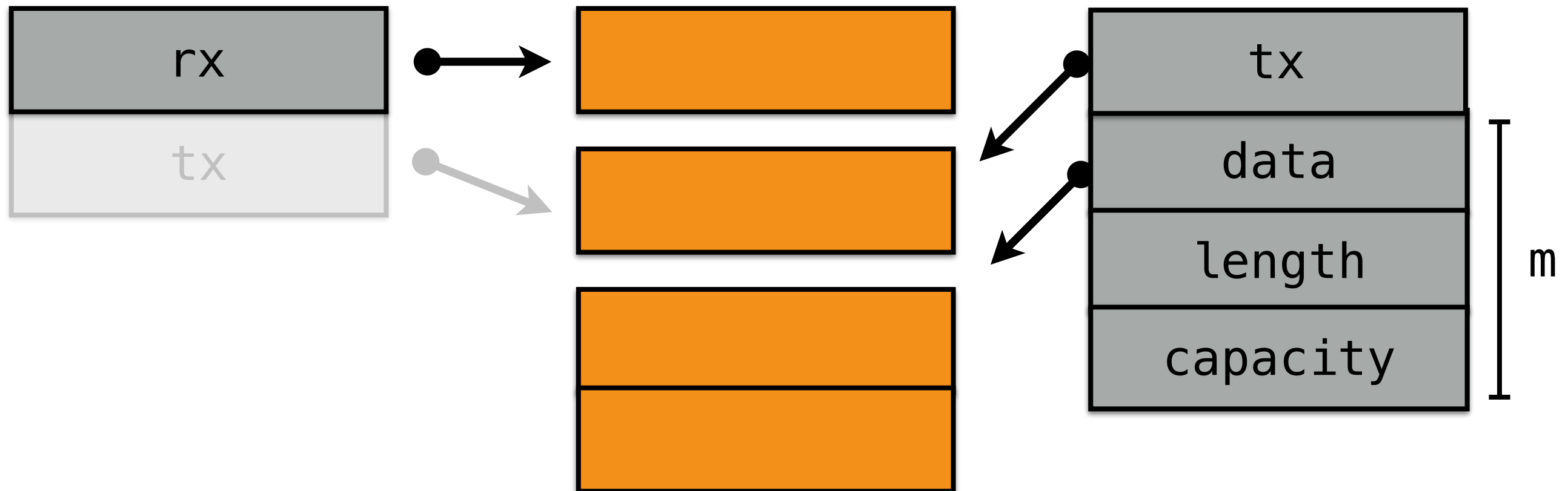
```

➡

```

move || {
  let m = Vec::new();
  ...
  tx.send(m);
}

```



```

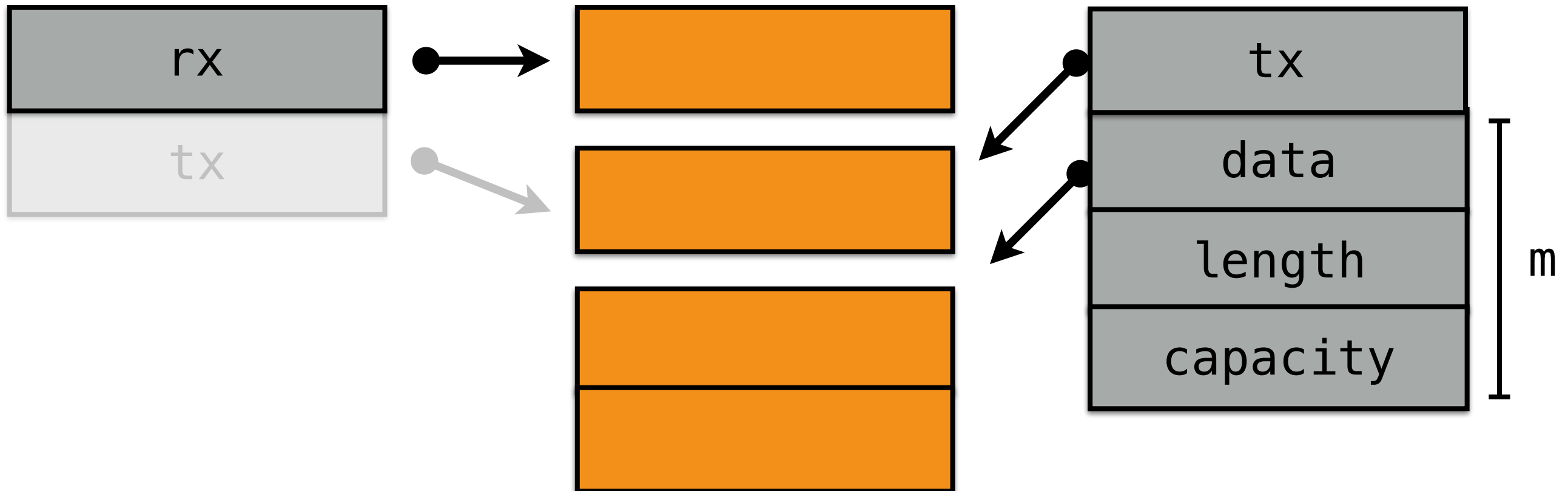
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {...});
  let m = rx.recv();
}

```

```

move || {
  let m = Vec::new();
  ...
  tx.send(m);
}

```



```

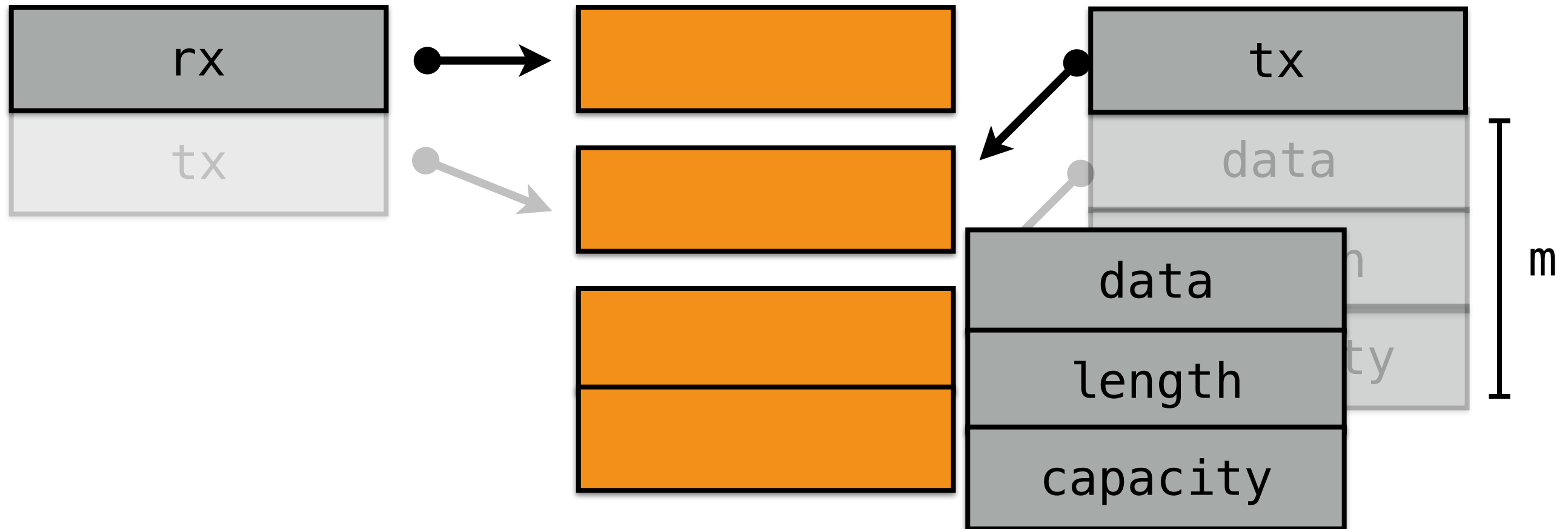
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {...});
  let m = rx.recv();
}

```

```

move || {
  let m = Vec::new();
  ...
  tx.send(m);
}

```



```

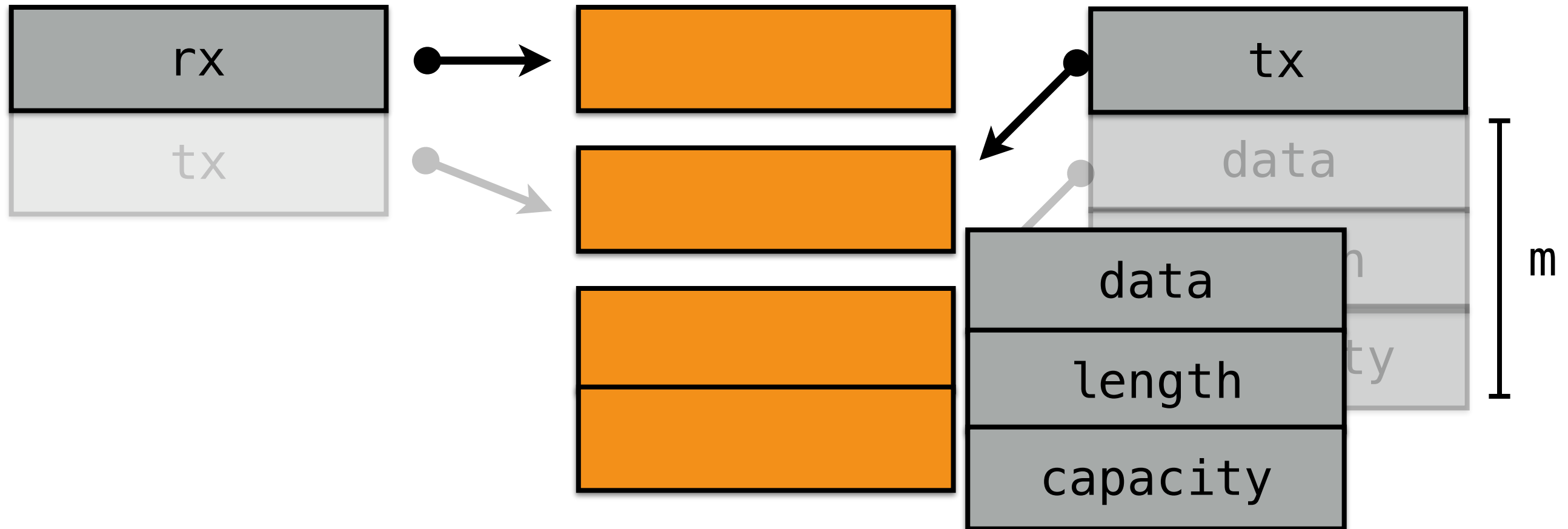
fn parent() {
    let (tx, rx) = channel();
    spawn(move || {...});
    let m = rx.recv();
}

```

```

move || {
    let m = Vec::new();
    ...
    tx.send(m);
}

```



```

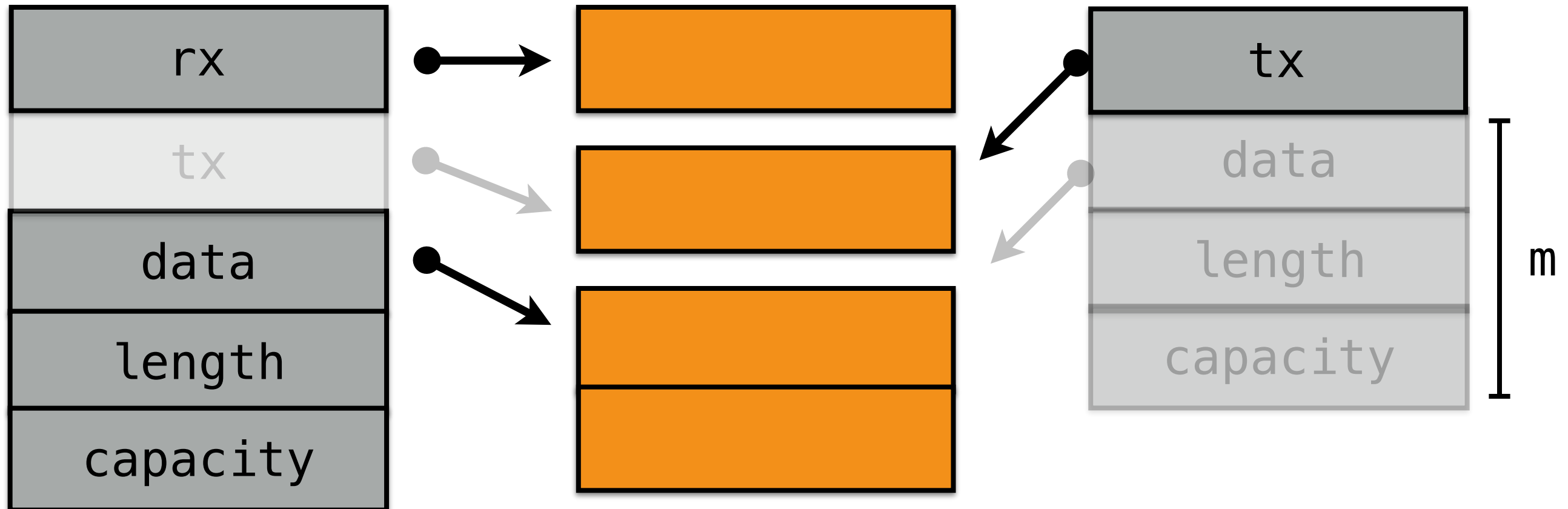
fn parent() {
    let (tx, rx) = channel();
    spawn(move || {...});
    let m = rx.recv();
}

```

```

move || {
    let m = Vec::new();
    ...
    tx.send(m);
}

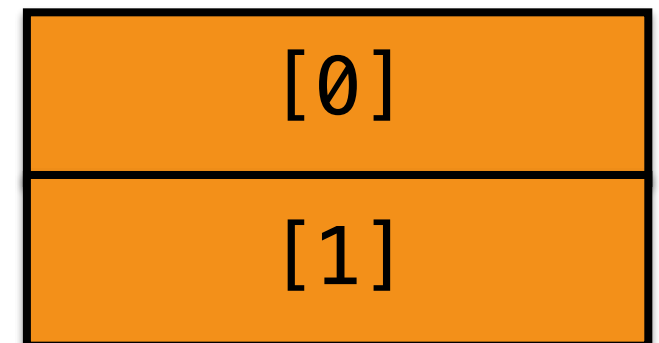
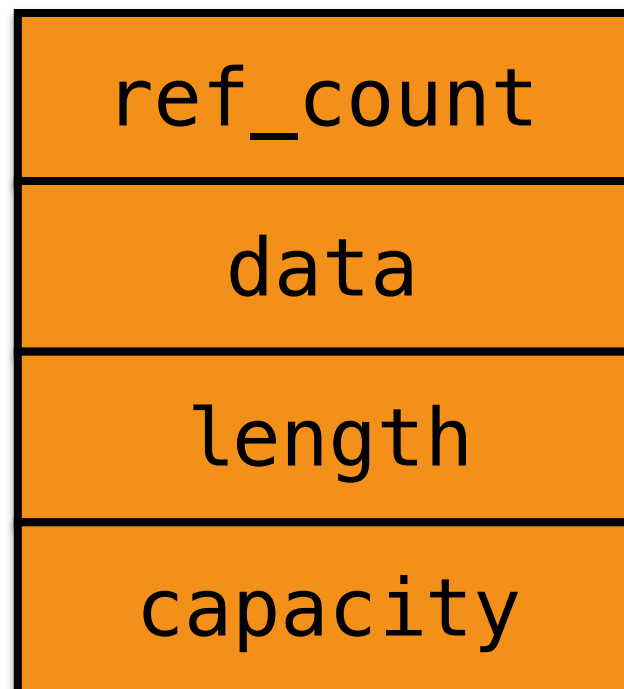
```





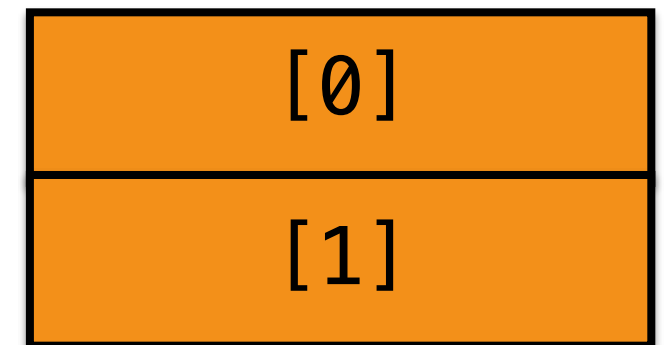
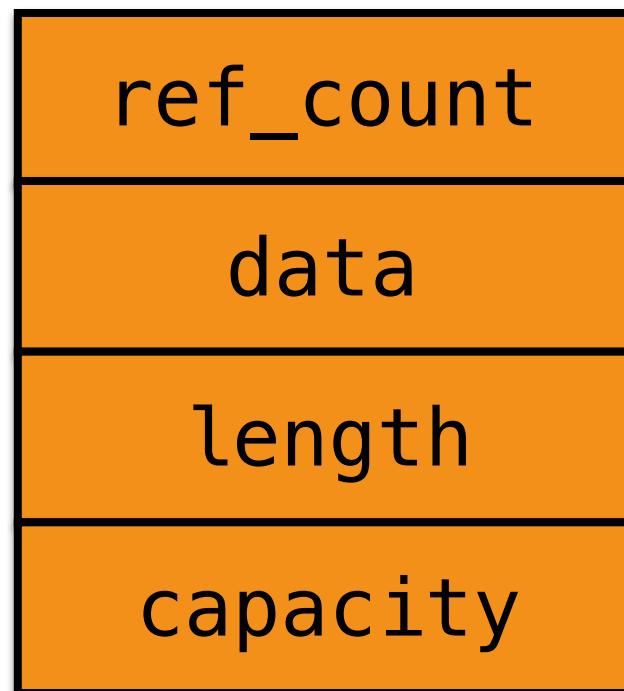
Shared read-only access
(ownership, borrowing)

Arc<Vec<int>>



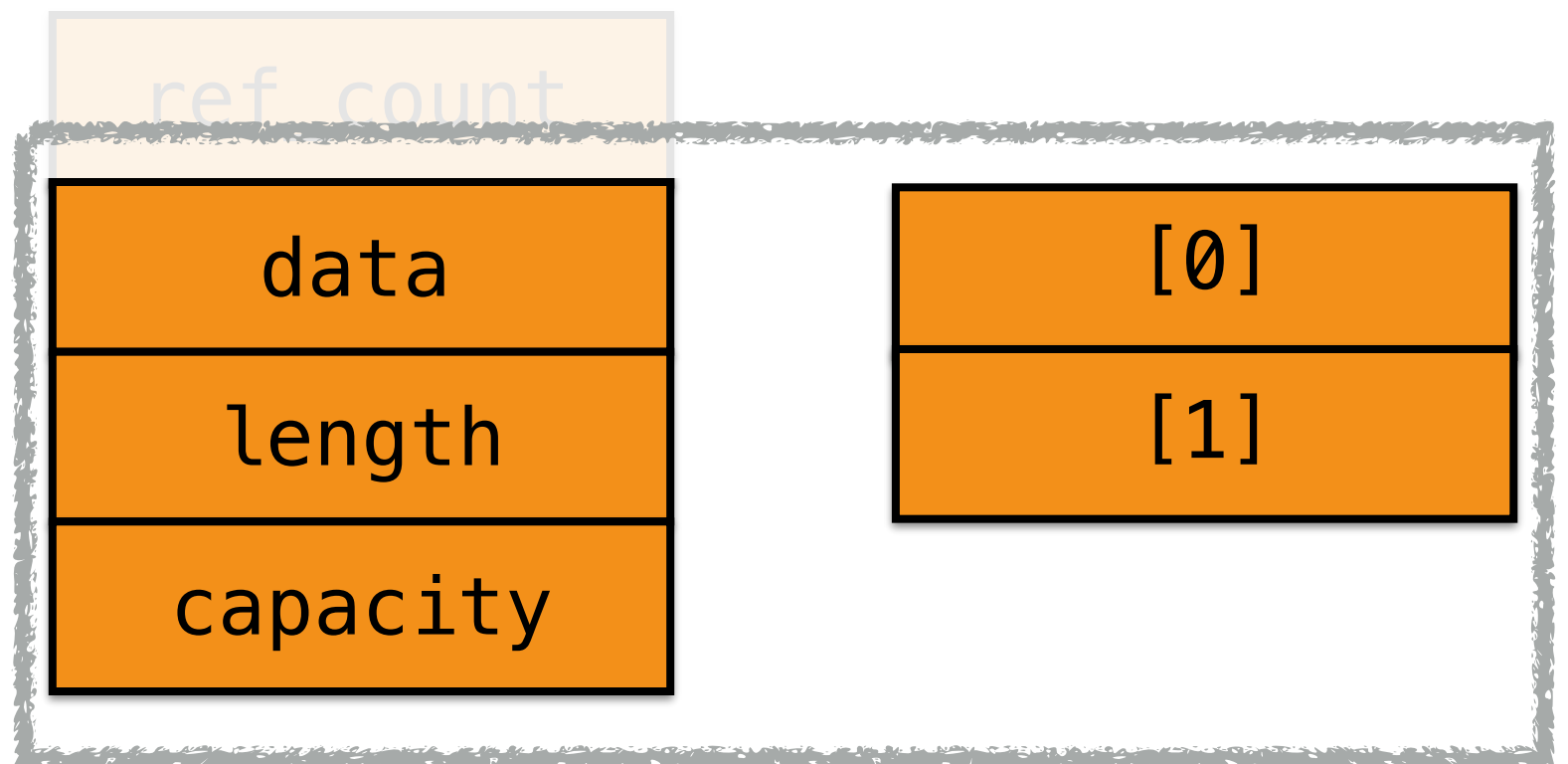
Arc<Vec<int>>

(ARC = Atomic Reference Count)



Arc<Vec<int>>

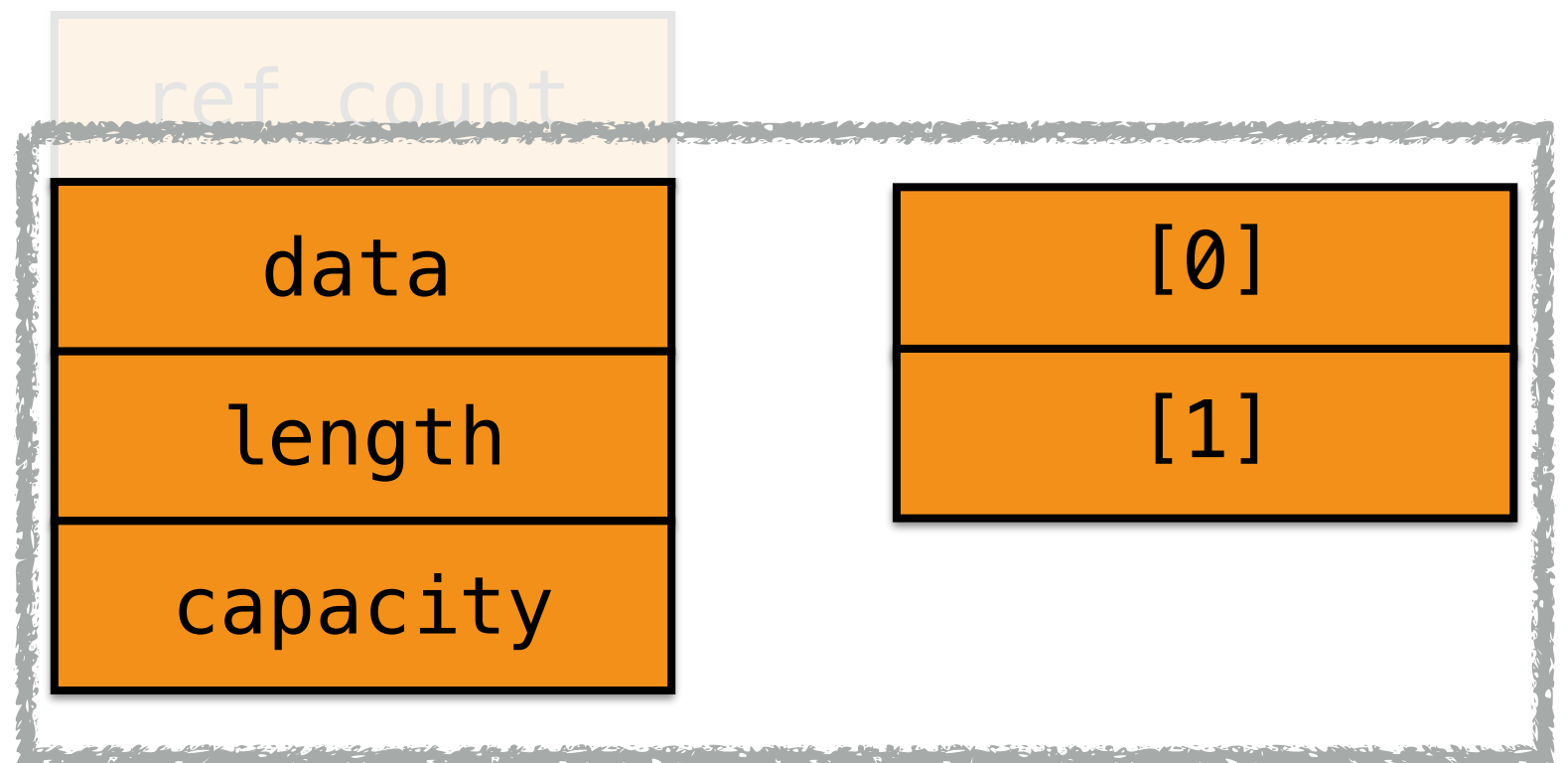
(ARC = Atomic Reference Count)



Vec<int>

Arc<Vec<int>>

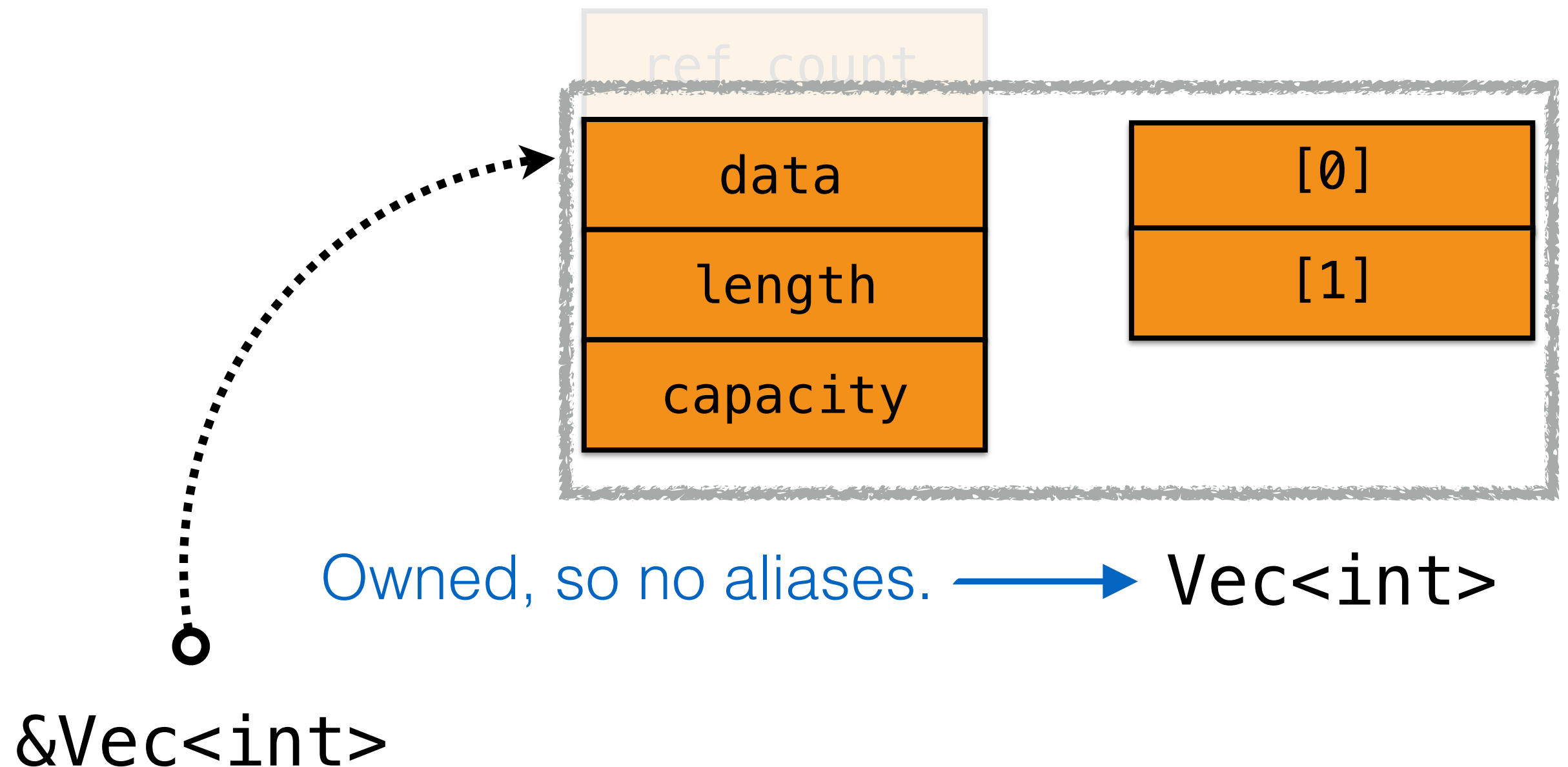
(ARC = Atomic Reference Count)



Owned, so no aliases. $\longrightarrow$ Vec<int>

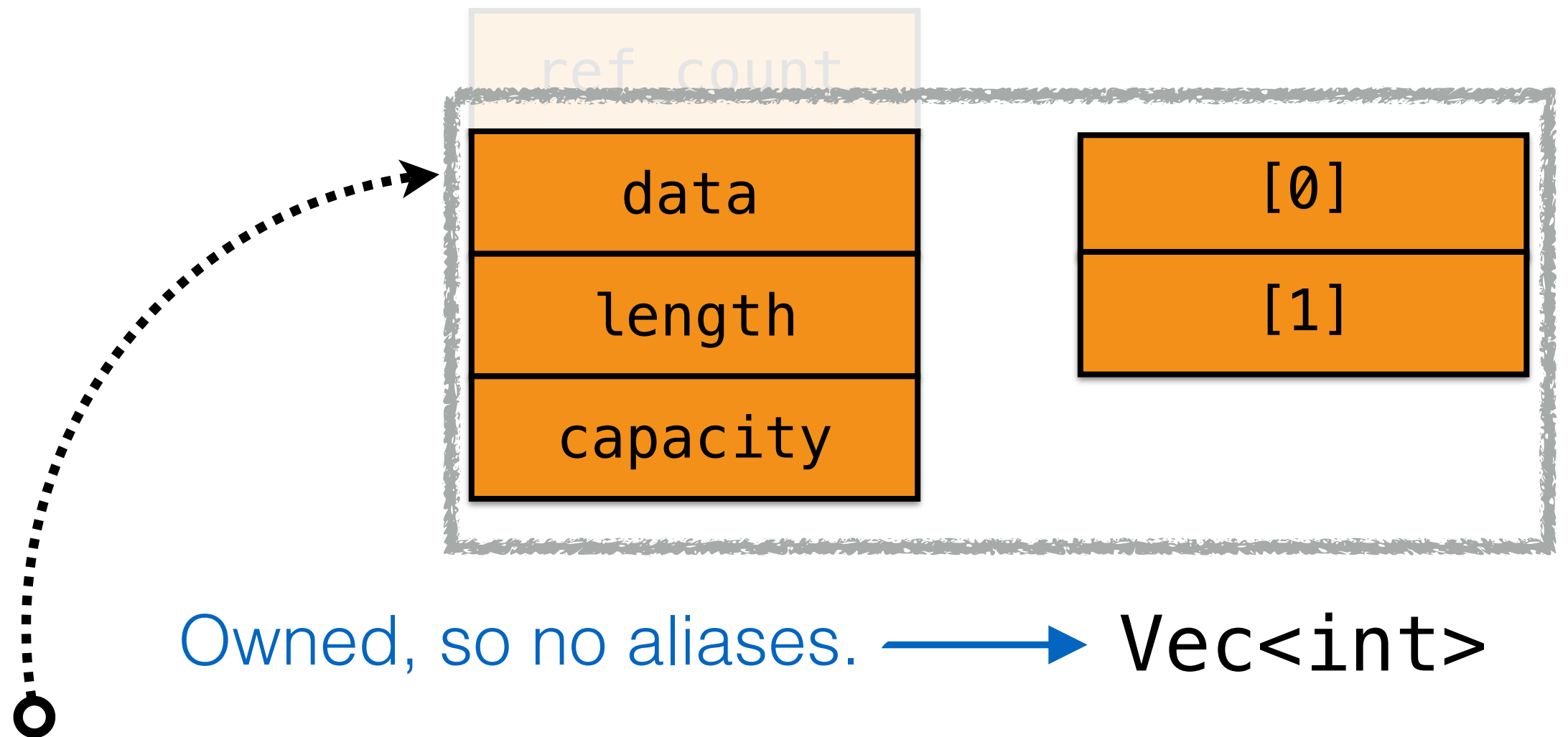
Arc<Vec<int>>

(ARC = Atomic Reference Count)



Arc<Vec<int>>

(ARC = Atomic Reference Count)



&Vec<int>



Shared reference, so
Vec<int> is immutable.

Thread safety checking

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



Only "sendable" types

Thread safety checking

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



Only “sendable” types

```
Arc<Vec<int>>: Send
```

```
Rc<Vec<int>> : !Send
```



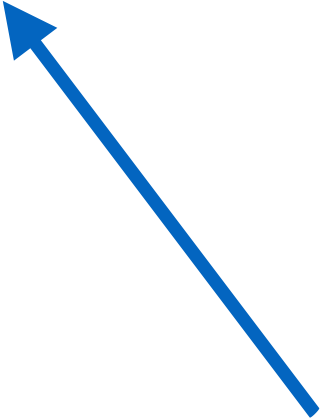
Locked mutable access

(ownership, borrowing)

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

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

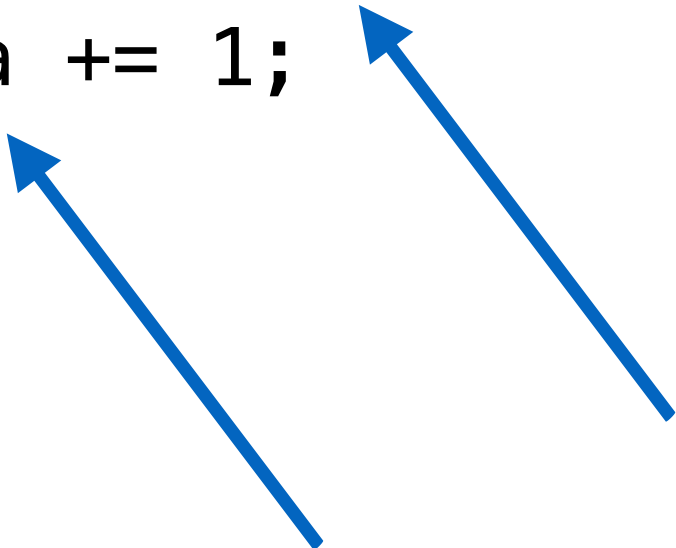
```
fn sync_inc(mutex: &Mutex<int>) {  
    let mut data = mutex.lock();  
    *data += 1;  
}
```



Destructor releases lock

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

Destructor releases lock
Yields a mutable reference to data



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

Destructor releases lock

Yields a mutable reference to data

Destructor runs here, releasing lock

Parallel

adj. occurring or existing at the same time



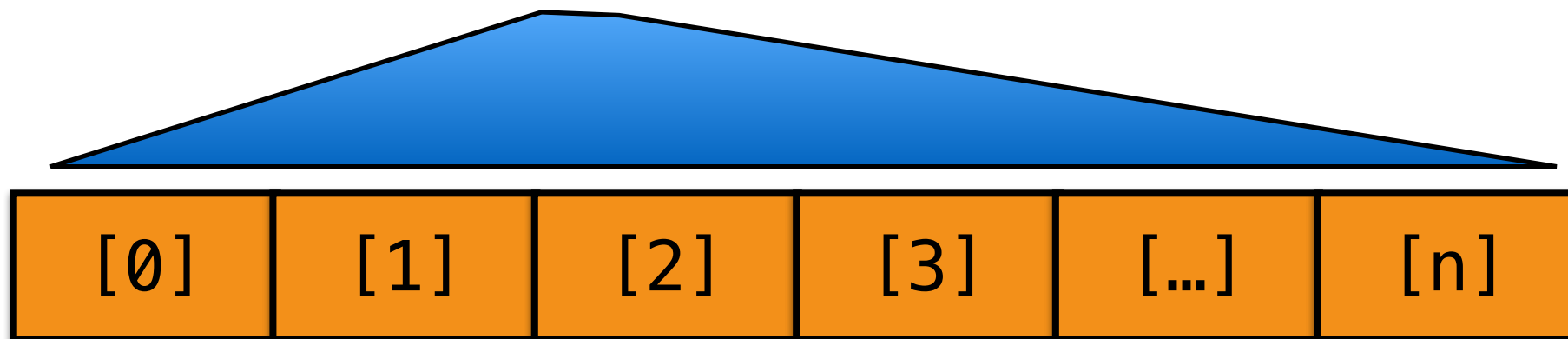
```
fn qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    qsort(less);  
    qsort(greater);  
}
```

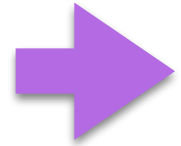


➔

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

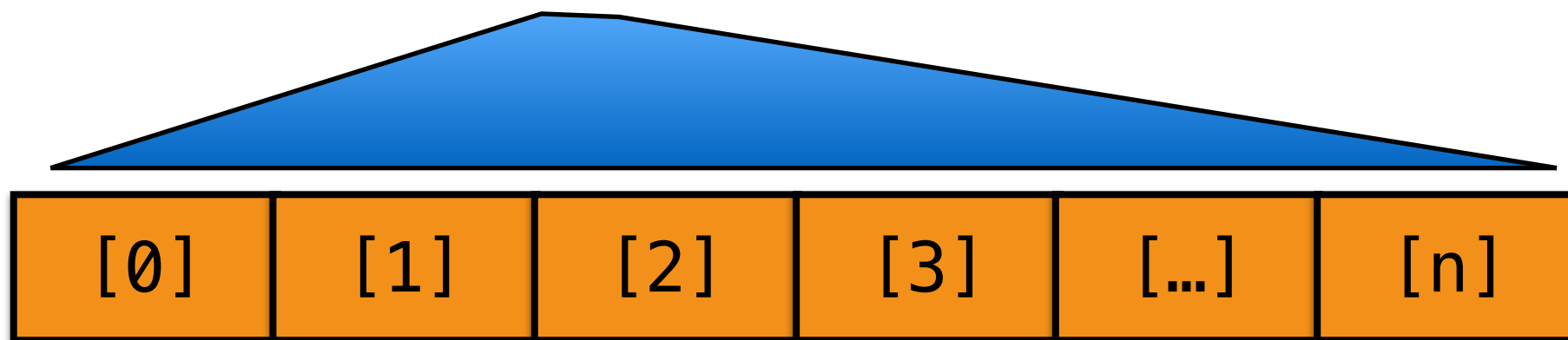
```
let vec: &mut [int] = ...;
```

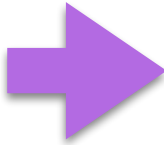




```
fn qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    qsort(less);  
    qsort(greater);  
}
```

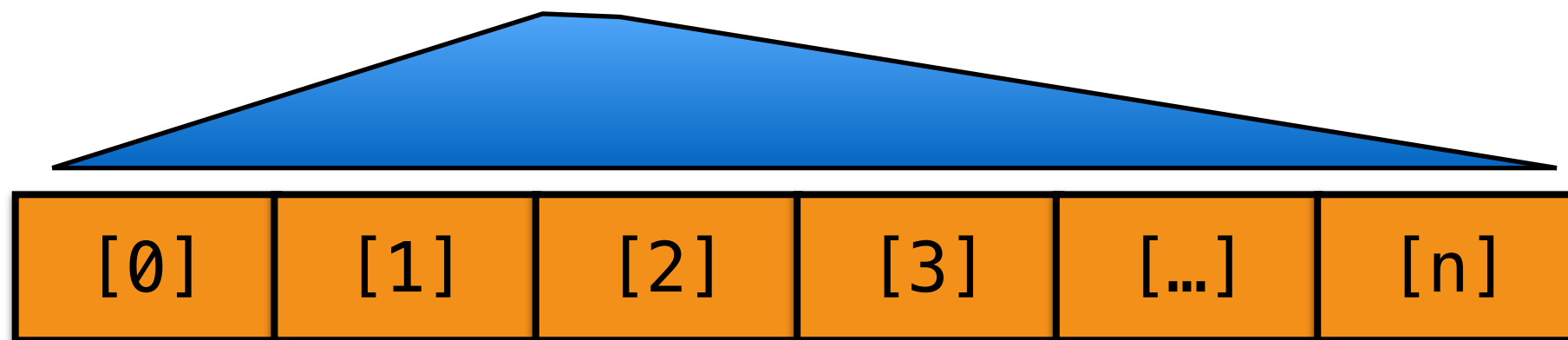
```
let vec: &mut [int] = ...;
```





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

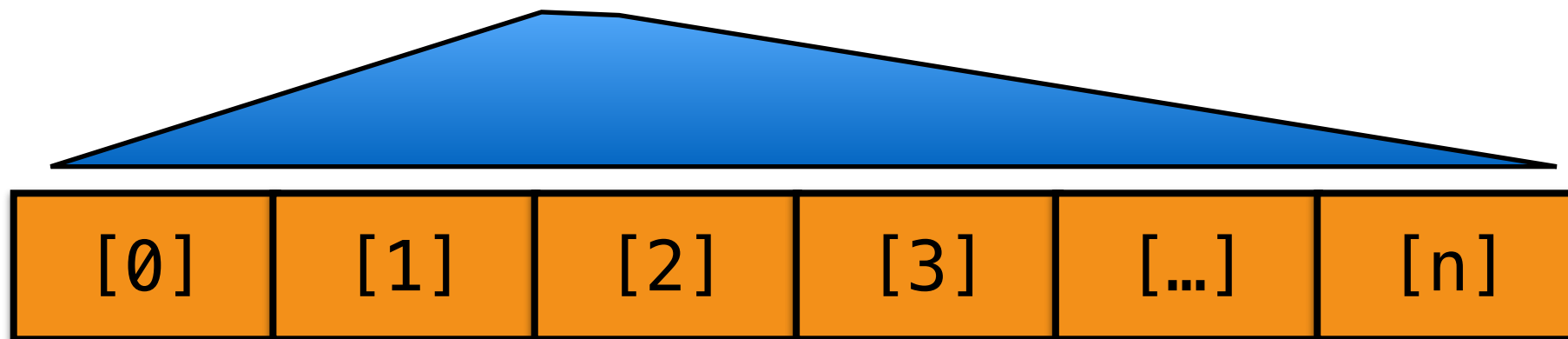
```
let vec: &mut [int] = ...;
```





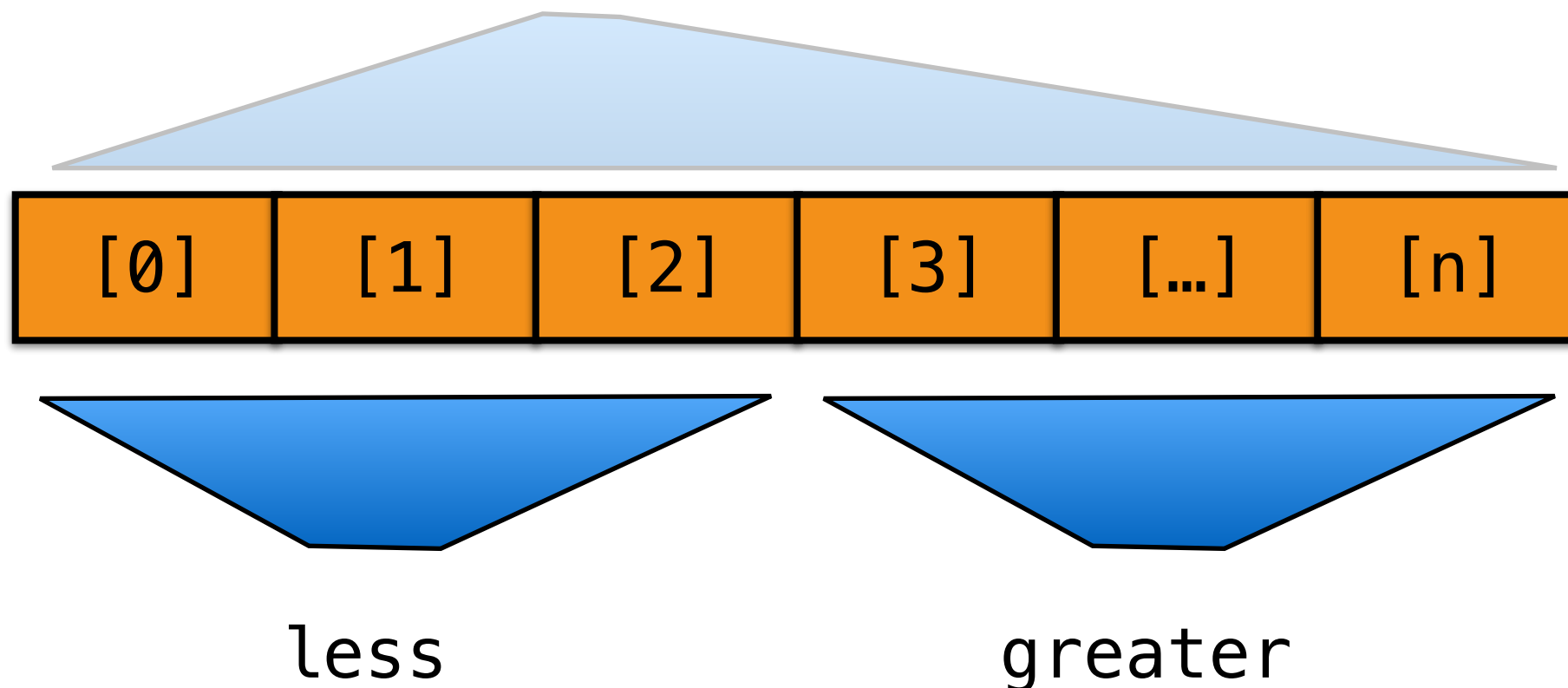
```
fn qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    qsort(less);  
    qsort(greater);  
}
```

```
let vec: &mut [int] = ...;
```



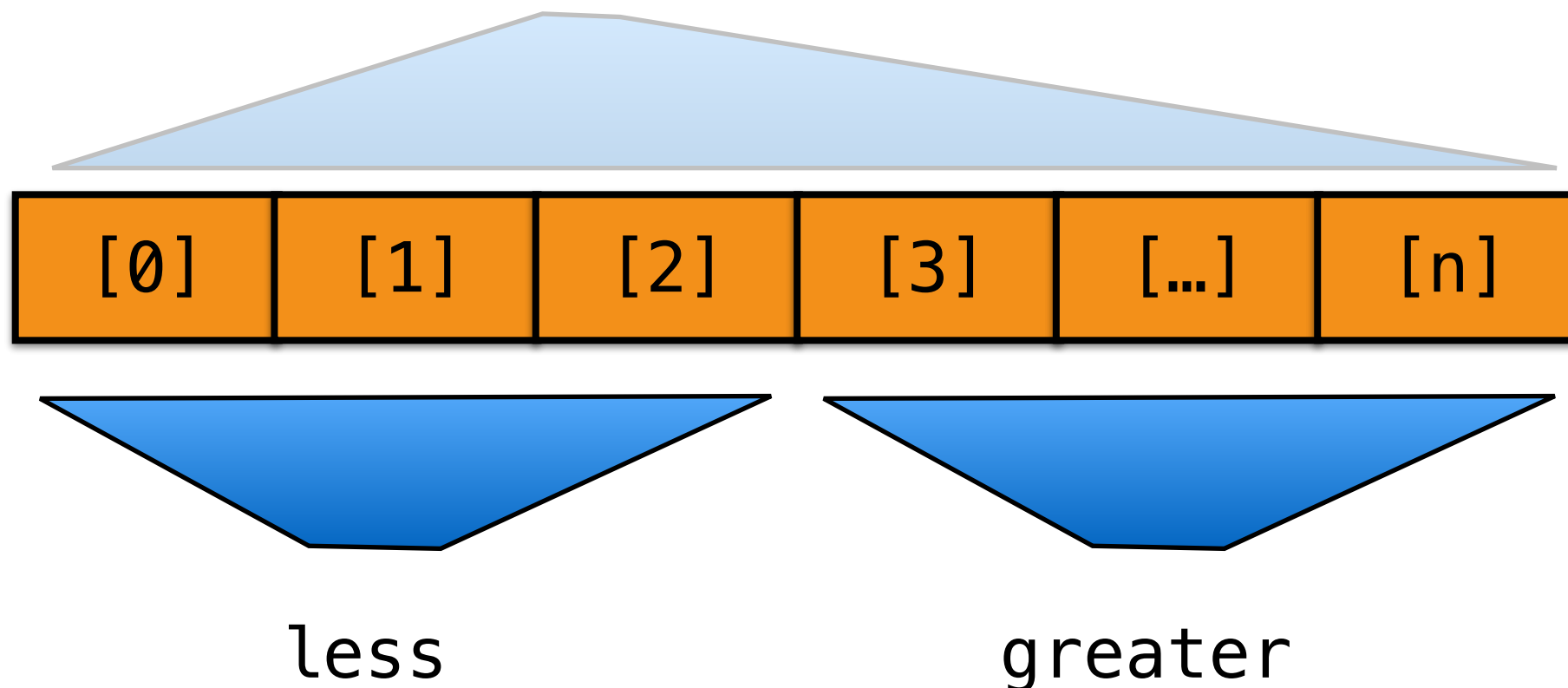
fn qsort(vec: &mut [int]) {
 if vec.len() <= 1 { return; }
 let pivot = vec[random(vec.len())];
 let mid = vec.partition(vec, pivot);
 let (less, greater) = vec.split_at_mut(mid);
 qsort(less);
 qsort(greater);
}

let vec: &mut [int] = ...;

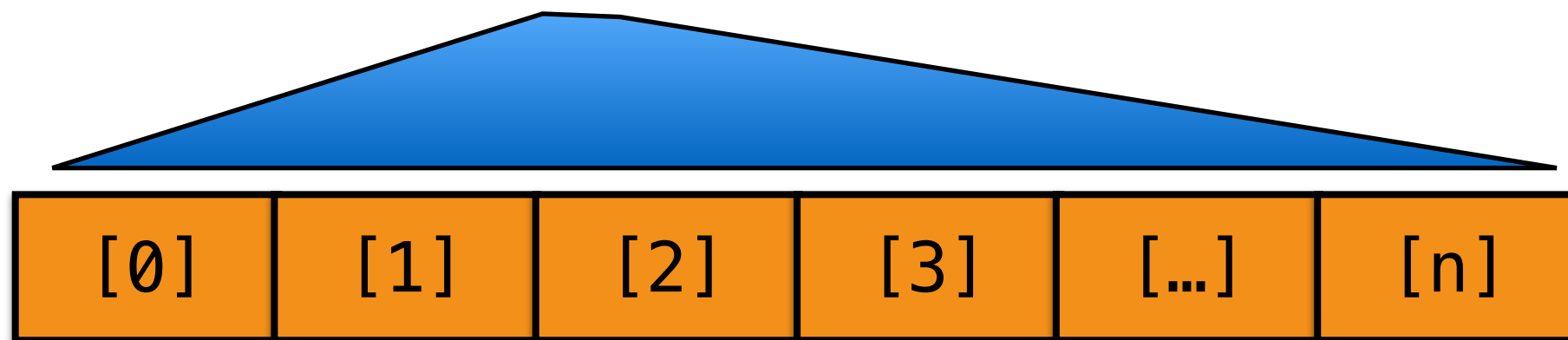


```
fn qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    qsort(less);  
    qsort(greater);  
}
```

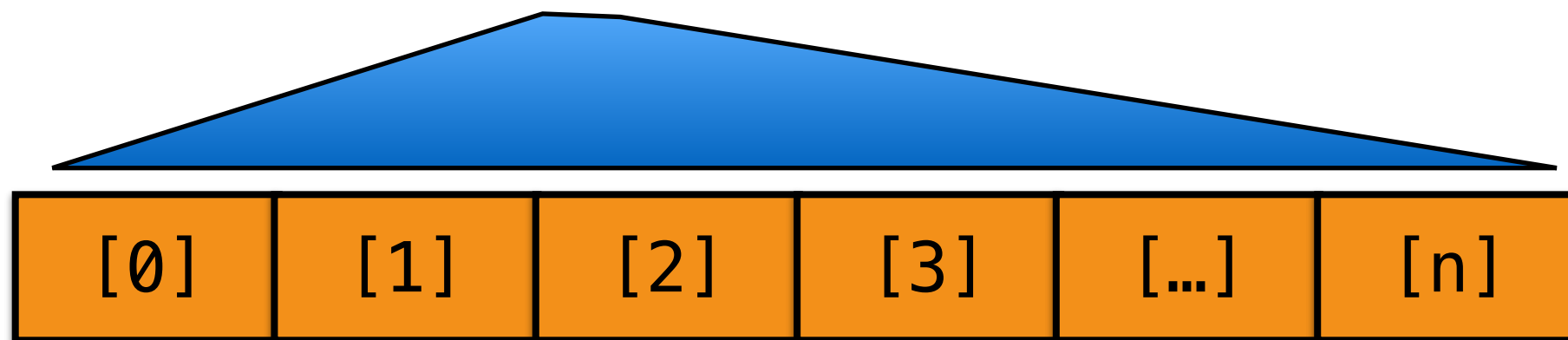
```
let vec: &mut [int] = ...;
```



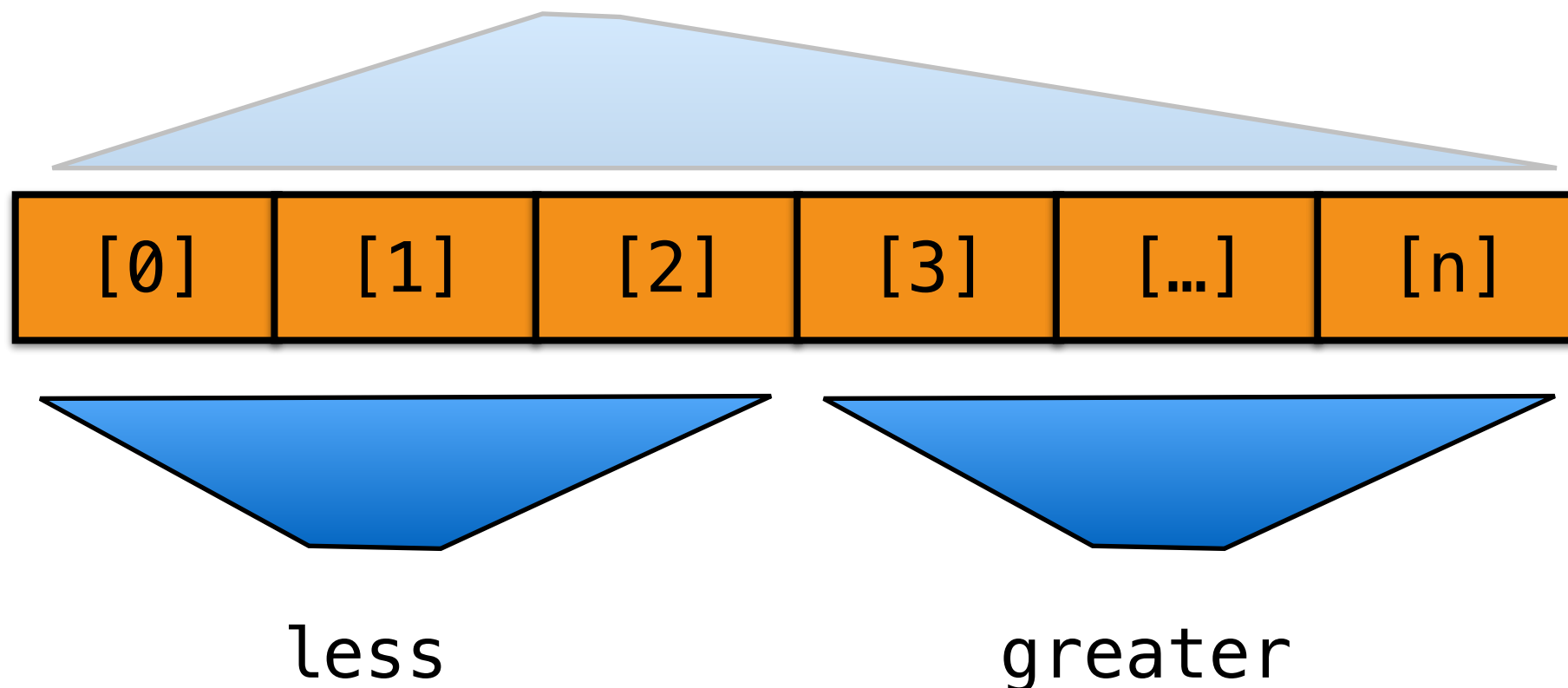
```
fn parallel_qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    parallel::join(  
        || parallel_qsort(less),  
        || parallel_qsort(greater)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



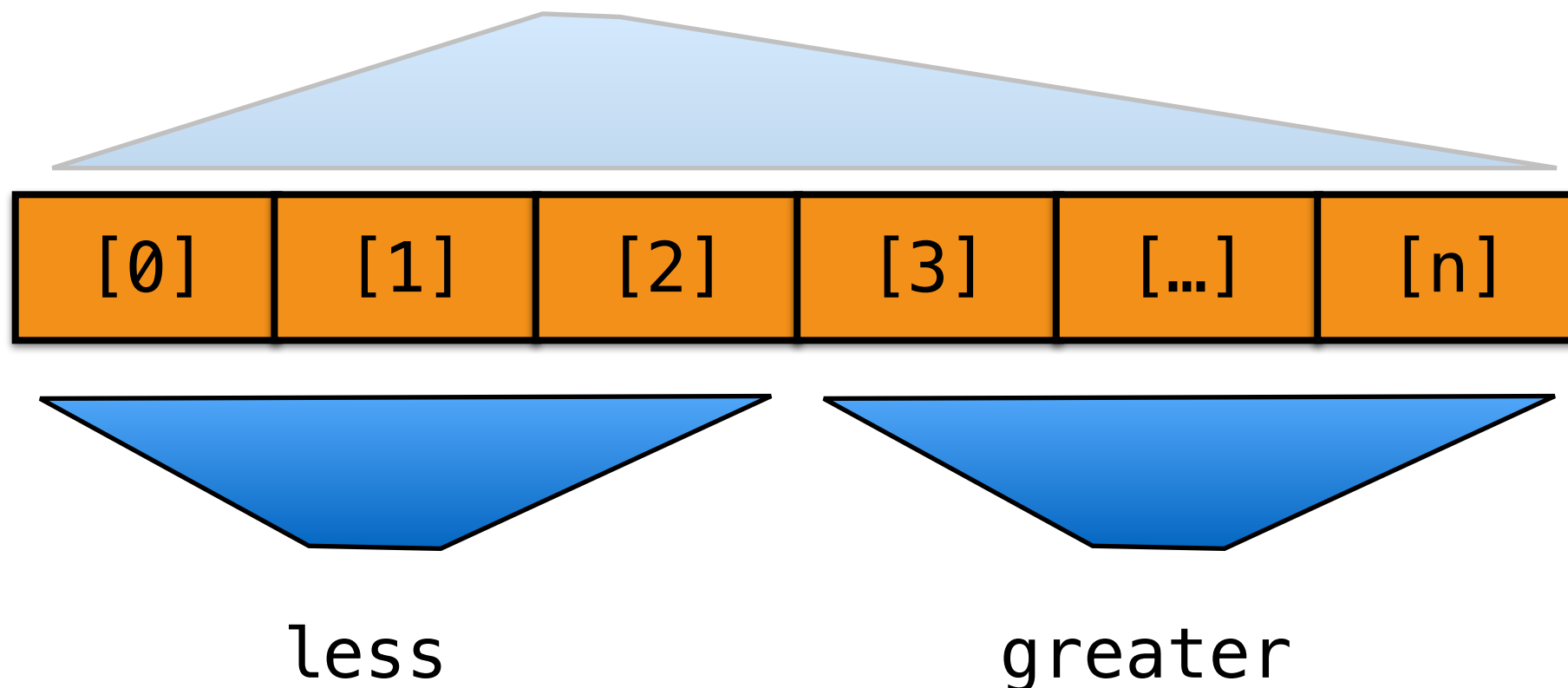
```
fn parallel_qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    parallel::join(  
        || parallel_qsort(less),  
        || parallel_qsort(greater)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



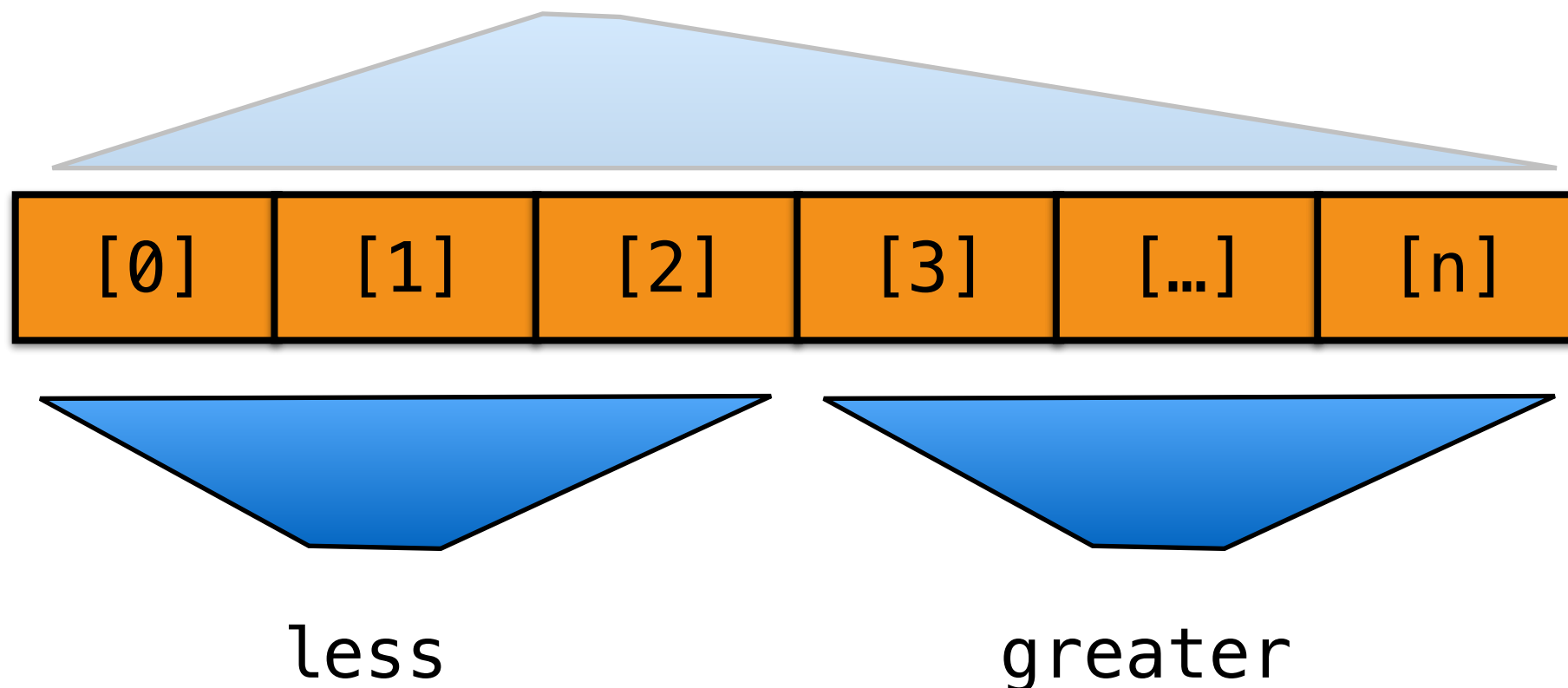
```
fn parallel_qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    parallel::join(  
        || parallel_qsort(less),  
        || parallel_qsort(greater)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



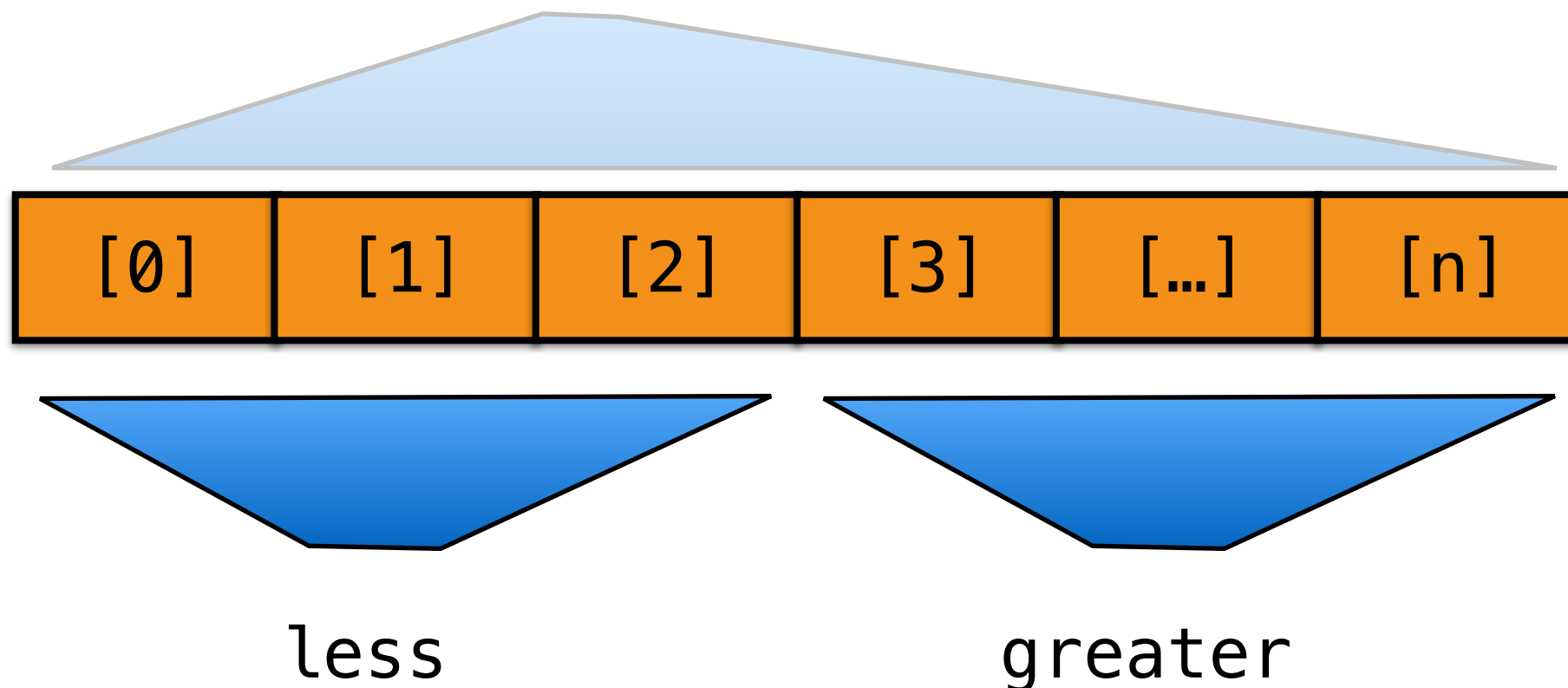
```
fn parallel_qsort(vec: &mut [int]) {  
    if vec.len() <= 1 { return; }  
    let pivot = vec[random(vec.len())];  
    let mid = vec.partition(vec, pivot);  
    let (less, greater) = vec.split_at_mut(mid);  
    parallel::join(  
        || parallel_qsort(less),  
        || parallel_qsort(greater)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



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



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



And beyond...

Concurrency is an area of **active development**.

Either already have or have plans for:

- Atomic primitives
- Non-blocking queues
- Concurrent hashtables
- Lightweight thread pools
- Futures
- CILK-style fork-join concurrency
- etc.

And beyond...

Concurrency is an area of **active development**.

Either already have or have plans for:

- Atomic primitives
- Non-blocking queues
- Concurrent hashtables
- Lightweight thread pools
- Futures
- CILK-style fork-join concurrency
- etc.

Always data-race free

Unsafe

adj. not safe; hazardous

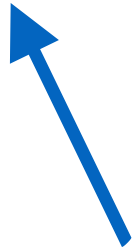
Safe abstractions

```
unsafe {  
    ...  
}
```

- Useful for:
 - Uninitialized memory
 - Interfacing with C code
 - Building parallel abstractions like ARC
- Ownership/borrowing permit creating **safe** abstraction boundaries.

Safe abstractions

unsafe {
...
}

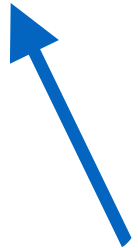


Trust me.

- Useful for:
 - Uninitialized memory
 - Interfacing with C code
 - Building parallel abstractions like ARC
- Ownership/borrowing permit creating **safe** abstraction boundaries.

Safe abstractions

```
fn something_safe(...) {  
    unsafe {  
        ...  
    }  
}
```



Trust me.

- Useful for:
 - Uninitialized memory
 - Interfacing with C code
 - Building parallel abstractions like ARC
- Ownership/borrowing permit creating **safe** abstraction boundaries.

Safe abstractions

```
fn something_safe(...) {
```

```
    unsafe {
```

```
        ...
```

```
    }
```

```
}
```

Validates input, etc.

Trust me.

- Useful for:
 - Uninitialized memory
 - Interfacing with C code
 - Building parallel abstractions like ARC
- Ownership/borrowing permit creating **safe** abstraction boundaries.

Status of Rust

1.0 Beta: March 31st

1.0 Final: May 15th

- Stable syntax, core type system
- Basic standard library
- Blossoming ecosystem on crates.io

Subsequent releases coming
every six weeks.

Conclusions

- Rust combines **high-level features** with **low-level control**.
- Rust gives **strong safety guarantees** beyond what GC can offer:
 - Deterministic destruction
 - Data race freedom
 - Iterator invalidation