Advanced Rust

Nicholas Matsakis

Themes of the day

1. Lifetimes across functions
2. Lifetimes are part of types
3. Successful borrowing
4. Lifetimes in structs
5. Mutability
6. Open questions

Lifetimes across functions

```rust
pub struct Map<K: Eq, V> {
  elements: Vec<(K, V)>
}

impl<K: Eq, V> Map<K, V> {
  pub fn new() -> Self {
    Map { elements: vec![] }
  }

  pub fn insert(&mut self, key: K, value: V) {
    self.elements.push((key, value));
  }

  pub fn get(&self, key: &K) -> Option<&V> {
    self.elements.iter().rev().find(|pair| pair.0 == *key)
                          .map(|pair| &pair.1)

  }
}
```

Indicates that method will **mutate** the map.

Returns a **reference** to data owned by **self**

```rust
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        map.insert('a', format!("alpha"));
        p = map.get(&'a');
    }
    println!("key for 'a' is {:?}", p);
}
```

reference still valid when map dropped

map dropped here

```
error: `map` does not live long enough
24 |         p = map.get(&'a');
   |             ^^^ does not live long enough
25 |     }
   |     - borrowed value only valid until here
26 |     println!("key for 'a' is {:?}", p);
27 | }
   | - borrowed value must be valid until here
```

https://is.gd/tvOrjf

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        map.insert('a', format!("alpha"));
        p = map.get(&'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

'l

's

Lifetime ('l): span of code where reference is in scope.

must be less than

Scope ('s) of data being borrowed (here, `map`)

How do we **know** that map is borrowed while p is in scope?

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        map.insert('a', format!("alpha"));
        p = map.get(&'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

Take references to
data owned by caller

Returns a reference…
but **to what**?

```
pub fn get(&self, key: &K) -> Option<&V>
```

```
pub fn get<'a>(&'a self, key: &K) -> Option<&'a V>
```

Returns a reference **borrowed from** `self`

Implies:

- as long as the return value is in use,
- `self` is still borrowed.

Lifetime Elision

In the **return type of a function**:

- one argument of reference type? ➡ borrowed from that argument.
- `&self` or `&mut` self method? ➡ borrowed from self.
- otherwise? ➡ error.

```
fn foo(count: usize, data: &[u32]) -> &u32
```

```
    fn foo<'a>(count: usize, data: &'a [u32]) -> &'a u32
```

```
fn bar(&self, count: usize, data: &[u32]) -> &u32
```

```
    fn bar<'a>(&'a self, count: usize, data: &[u32]) -> &'a u32
```

```
fn baz(count: usize, data: &[u32], more: &[u32]) -> &u32
```

```
    error: missing lifetime specifier
```

(play)

```rust
impl<K, V> Map<K, V> {

  pub fn get_or(&self, key: &K, value: &V) -> &V {
    match self.get(key) {
      Some(from_map) => from_map,
      None => value,
    }
  }

}
```

```
error[E0495]: cannot infer an appropriate lifetime
   |
19 |         match self.get(key) {
   |         ^
```

https://is.gd/42ZEWL

```rust
pub fn get_or(&self, key: &K, value: &V) -> &V
```

```rust
pub fn get_or<'a>(&'a self, key: &K, value: &V) -> &'a V {
    match self.get(key) {
      Some(from_map) => from_map, ✅
      None => value, 🚫
    }
}
```

Signature declares that it returns a reference **borrowed from** `self`

But does it?

```
impl<K, V> Map<K, V> {

  pub fn get_or<'a>(&'a self, key: &K, value: &'a V) -> &'a V {
    self.get(key).unwrap_or(value)
  }

}
```

Returns a reference borrowed **either** from `self` **or** from `value`

Implies:

- as long as the return value is in use,
- `self` **and** `value` are still borrowed.

https://is.gd/I2yBr4

```
fn main() {
    let mut map = Map::new();
    map.insert('a', format!("alpha"));
    let v = format!(“fallback”);
    let p = map.get_or('a', &v);
    …
}
```

https://is.gd/qI7c5i

```
fn main() {
    let mut map = Map::new();
    map.insert('a', format!("alpha"));
    let p;
    let v = format!("fallback");
    p = map.get_or('a', &v);
    …
}
```

'l

's

```
error: `v` does not live long enough
30 |     p = map.get_or(&'a', &v);
   |                           ^ does not live long enough
31 |     println!("p={:?}", p);
32 | }
   | - borrowed value dropped before borrower
```

https://is.gd/WfIgHv

Key concept: **Modularity**

```
impl<K, V> Map<K, V> {
  pub fn get_or<'a>(&'a self, key: &K, value: &V) -> &'a V {
    panic!(“signature writing cheques body can't cash”) 😿
  }
}

fn main() {
    let mut map = Map::new();
    map.insert('a', format!(“alpha"));
    let p;
    let v = format!(“fallback”);
    p = map.get_or('a', &v);
    …
}
```

💯

Exercise: **named lifetimes**

http://rust-tutorials.com/exercises/

Cheat sheet:

```
fn foo<'a>(…)  // declare a named lifetime parameter
&'a i32        // reference with lifetime 'a
```

http://doc.rust-lang.org/std

Lifetimes are part of types

```rust
pub fn get(&self, key: &K) -> Option<&V>
```

```rust
pub fn get<'a>(&'a self, key: &K) -> Option<&'a V>
```

```rust
pub fn get<'a,'b>(&'a self, key: &'b K) -> Option<&'a V>
```

Every reference type `&T` is short for `&'lt T` for some lifetime.

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        map.insert('a', format!("alpha"));
        p = map.get(&'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

reference still valid when map dropped

map dropped

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        map.insert('a', format!("alpha"));
        p = map.get(&'a');
    }
    println!("key for 'a' is {:?}", p);
}
```

What is lifetime of this type?

Expression has type `&char`. What lifetime?

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        Map::insert(&mut map, 'a', format!(“alpha"));
        p = Map::get(&map, &'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

Method calls are actually “syntactic sugar” for **function calls**.

Every method can be named via a **fully qualified path**.

The `.` operator also adds **implicit borrows**.

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        Map::insert(&mut map, 'a', format!(“alpha"));
        p = Map::get(&map, &'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

All of these lifetimes must be inferred.

http://www.rockyhorror.com/participation/timewarp.php

$$X + Y = 22$$
$$X - Y = 10$$

$$X = 16$$
$$Y = 6$$

$$X \geq Y$$
$$Y \geq 10$$

$$X = 10$$
$$Y = 10$$

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        Map::insert(&mut map, 'a', format!(“alpha"));
        p = Map::get(&map, &'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

1. Assign each lifetime a variable.
2. Determine **constraints** on those variables.
3. **Solve** the constraints — or try, at least.

```
fn main() {
    let p: Option<&'W String>;
    {
        let mut map = Map::new();
        Map::insert(&'X mut map, 'a', format!(“alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

{call to Map::insert}

{use of format!}

{scope of `p`}

1. Assign each lifetime a variable.

```
fn main() {
    let p: Option<&'W String>;
    {
        let mut map = Map::new();
        Map::insert(&'X mut map, 'a', format!(“alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

Type of variable must outlive its scope:

```
Option<&'W String>: {scope of `p`}
&'W String: {scope of `p`}
'W: {scope of `p`}
```

2. Determine **constraints** on those variables.

```
fn main() {
    let p: Option<&'W String>;
    {
        let mut map = Map::new();
        Map::insert(&'X mut map, 'a', format!("alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!("key for 'a' is {:?}", p);
}
```

Types of arguments must outlive a call:

```
&'X mut Map<char, String>: {call of Map::insert}
'X: {call of Map::insert}
…
'Y: {call of Map::get}
…
```

2. Determine **constraints** on those variables.

```
fn main() {
    let p: Option<&'W String>;
    {
        let mut map = Map::new();
        Map::insert(&'X mut map, 'a', format!(“alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

Linking of lifetime from argument to return value:

'Y: 'W

2. Determine **constraints** on those variables.

```
fn main() {
    let p: Option<&'W String>;
    {
        let mut map = Map::new();
        Map::insert(&'X mut map, 'a', format!("alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!("key for 'a' is {:?}", p);
}
```

```
'W: {scope of `p`}
'X: {call of Map::insert}
'Y: 'W
'Z: {call of Map::get}
…
```

```
'W = {scope of `p`}
'X = {call of Map::insert}
'Y = {scope of `p`}
'Z = {call of Map::get}
```

3. **Solve** the constraints — or try, at least.

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        Map::insert(&mut map, 'a', format!("alpha"));
        p = Map::get(&{scope of `p`} map, &'a');
    }
    println!("key for 'a' is {:?}", p);
}
```

{scope of `map`}

{scope of `p`}

'Y = {scope of `p`}

```
fn main() {
    let p: Option<&'W String>;
    {
        let mut map = Map::new();
        Map::insert(&'X mut map, 'a', format!(“alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

Linking of lifetime from argument to return value:

```
'Y: 'W
```

How did we come by this again?

2. Determine **constraints** on those variables.

```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        Map::insert(&mut map, 'a', format!("alpha"));
        p = Map::get(&'Y map, &'Z 'a');
    }
    println!("key for 'a' is {:?}", p);
}
```

```
impl<K,V> Map {
  pub fn get<'a,'b>(&'a self, key: &'b K) -> Option<&'a V>
}
```

```
p = Map::<char, String>::get::<'U, 'V>(&'Y map, &'a');
```

```
p = Map::<char, String>::get::<'U, 'V>(&'Y map, &'a');
```

Expected type of this argument: `&'U Map<char, String>`

Actual type of this argument: `&'Y Map<char, String>`

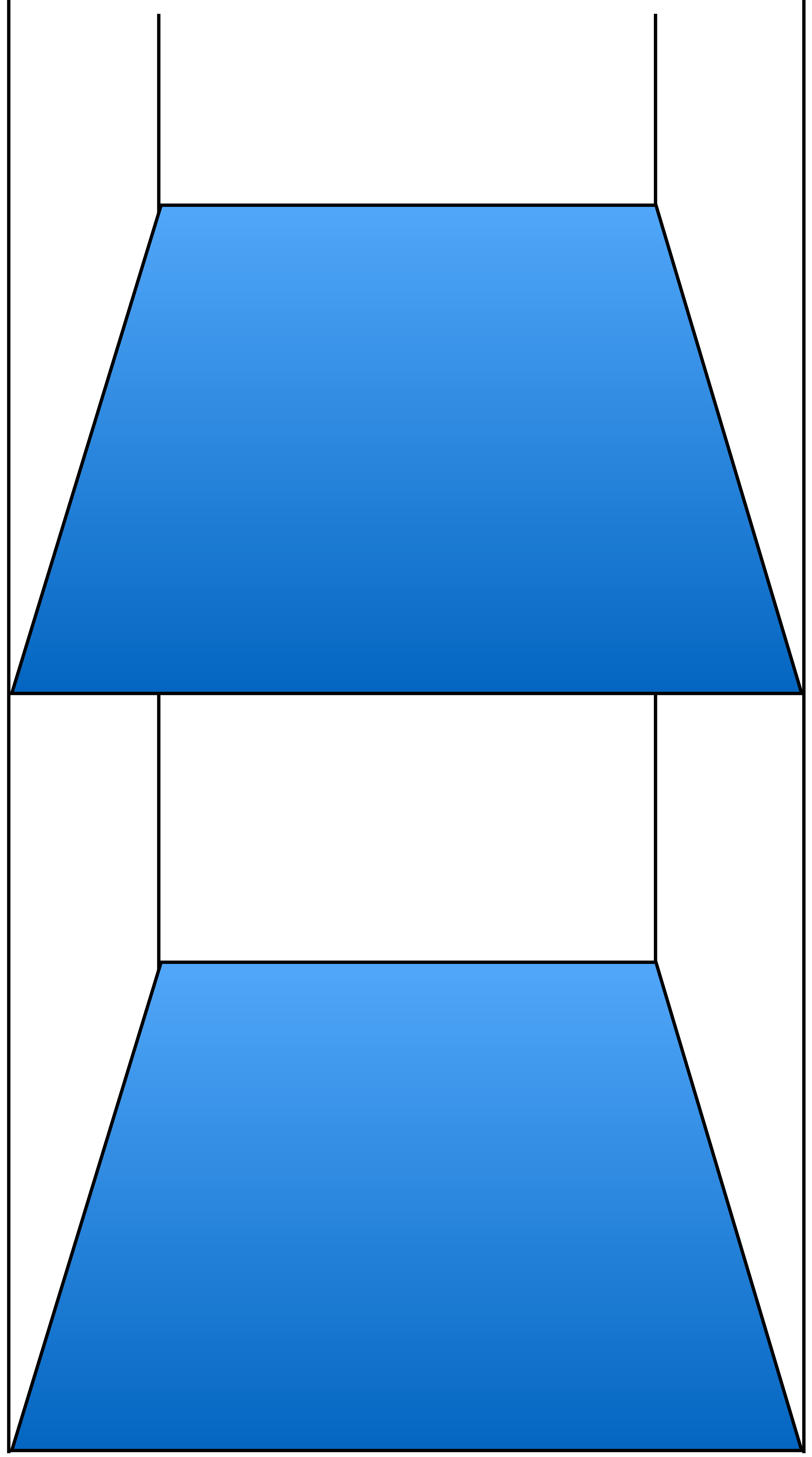
Resulting constraint: `'Y: 'U`

Return type of `Map::get`: `Option<&'U String>`

Type of `p`: `Option<&'W String>`

Resulting constraint: `'U: 'W`

`'Y: 'W`



```
fn main() {                    {scope of `p`}
    let p: Option<&String>;
    {
        …
        p = Map::get(&map, &'a');
    }
    …
}

impl<K,V> Map<K,V> {
  pub fn get<'a, 'b>(&'a self, key: &'b K)
                           -> Option<&'a V> {…}
}
```

```
pub fn get_or<'a>(&'a self, key: &K, value: &’a V) -> &'a V {
    match self.get(key) {
      Some(from_map) => from_map,
      None => value,
    }
}

pub fn get_or<‘a, ‘b>(&'a self, key: &K, value: &’b V) -> &'a V
  where ‘b: ‘a
{
    match self.get(key) {
      Some(from_map) => from_map,
      None => value,
    }
}
```

Overkill in this scenario.
Sometimes useful.

Exercise: **lifetimes as part of types**

http://rust-tutorials.com/exercises/

Cheat sheet:

```
fn foo<‘a, ‘b>(…)  // declare a named lifetime parameter
  where ‘b: ‘a     // ‘b outlives ‘a
```

http://doc.rust-lang.org/std

Successful borrowing

reference still valid when map dropped

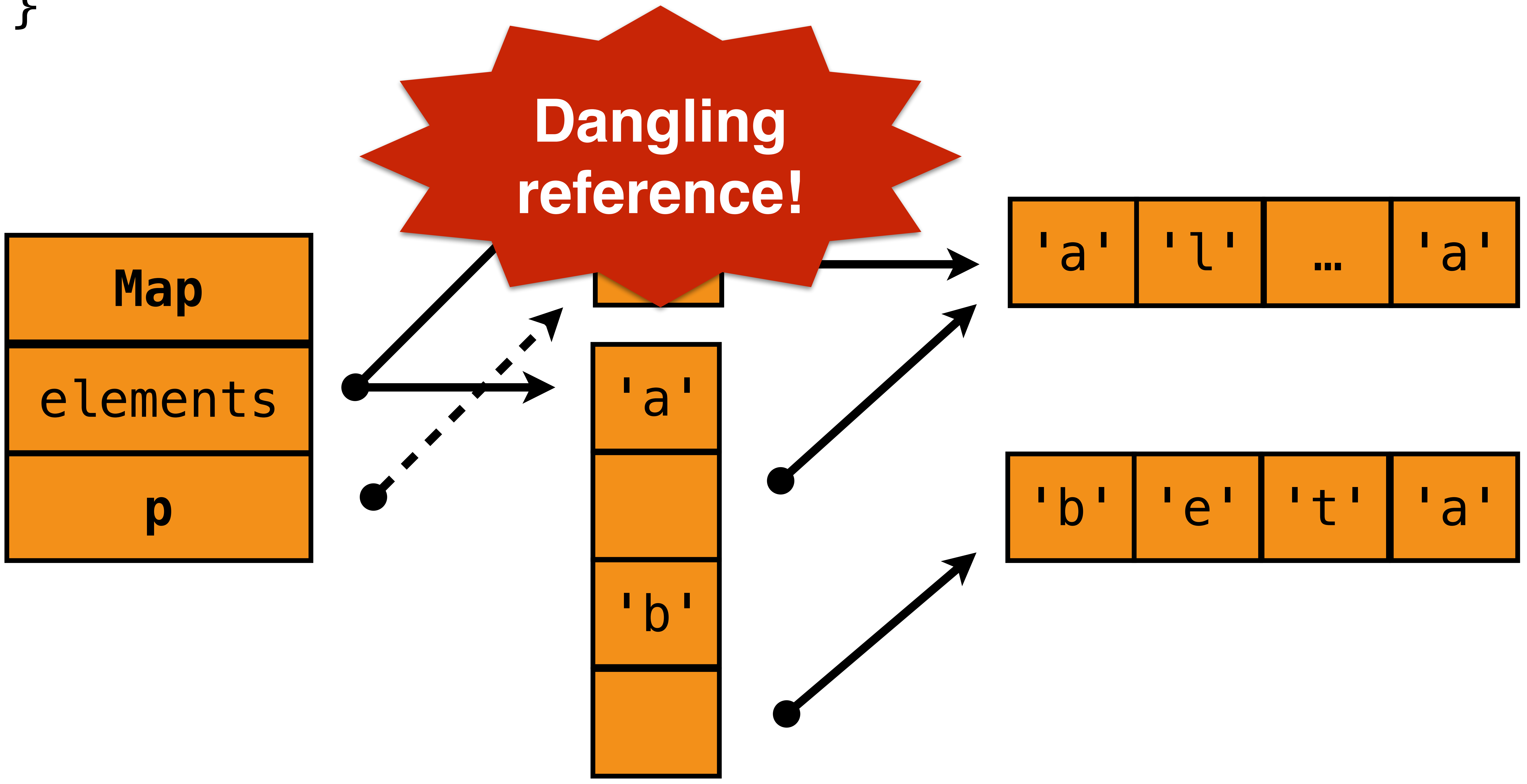
```
fn main() {
    let p: Option<&String>;
    {
        let mut map = Map::new();
        map.insert('a', format!("alpha"));
        p = map.get(&'a');
    }
    println!(“key for 'a' is {:?}”, p);
}
```

map dropped

Tracking lifetimes ensures that values are not dropped while there is a live reference.

But mutation can cause memory to be freed early.

```rust
fn main() {
    let mut map = Map::new();
    map.insert('a', format!("alpha"));
    let p = map.get(&'a');
    map.insert('b', format!("beta"));
}
```



Rust solution

Compile-time read-write-lock:

Shared borrow of X **“read locks”** X.
- Other readers OK.
- No writers.
- Lock lasts for lifetime of borrow.

Mutable borrow of X **“writes locks”** X.
- No other readers or writers.
- Lock lasts for lifetime of borrow.

Never have a reader/writer at same time.

```
fn main() {
    let mut map = Map::new();
    Map::insert(&mut map, 'a', format!("alpha"));
    let p = Map::get(&map, &'a');
    Map::insert(&mut map, 'b', format!("beta"));
}
```

Shared borrow of map

Lifetime of shared borrow

Mutable borrow of map

```
error[E0502]: cannot borrow `map` as mutable because
              it is also borrowed as immutable
22 |     let p = map.get(&"a");
   |             --- immutable borrow occurs here
23 |     map.insert("a", format!("alpha"));
   |     ^^^ mutable borrow occurs here
24 | }
   | - immutable borrow ends here
```

https://is.gd/lyi2ru

```
pub fn remove(&mut self, key: &K) {
  for (index, pair) in self.elements.iter().enumerate() {
    if pair.0 == *key {
      self.elements.remove(index);
      return;
    }
  }
}
```

Lifetime of shared borrow

Mutable borrow of map

Shared borrow of map

https://is.gd/AxhTUo

```rust
pub fn remove(&mut self, key: &K) {
  let mut found = None;
  for (index, pair) in self.elements.iter().enumerate() {
    if pair.0 == *key {
      found = Some(index);
      break;
    }
  }

  if let Some(index) = found {
    self.elements.remove(index);
  }
}
```

Lifetime of shared borrow

Mutable borrow

```rust
fn index_of(&self, key: &K) -> Option<usize> {
  self.elements.iter().position(|pair| pair.0 == *key)
}

pub fn remove(&mut self, key: &K) {
  match self.index_of(key) {
    Some(index) => self.elements.remove(index),
    None => ()
  }
}
```

Takeaway:

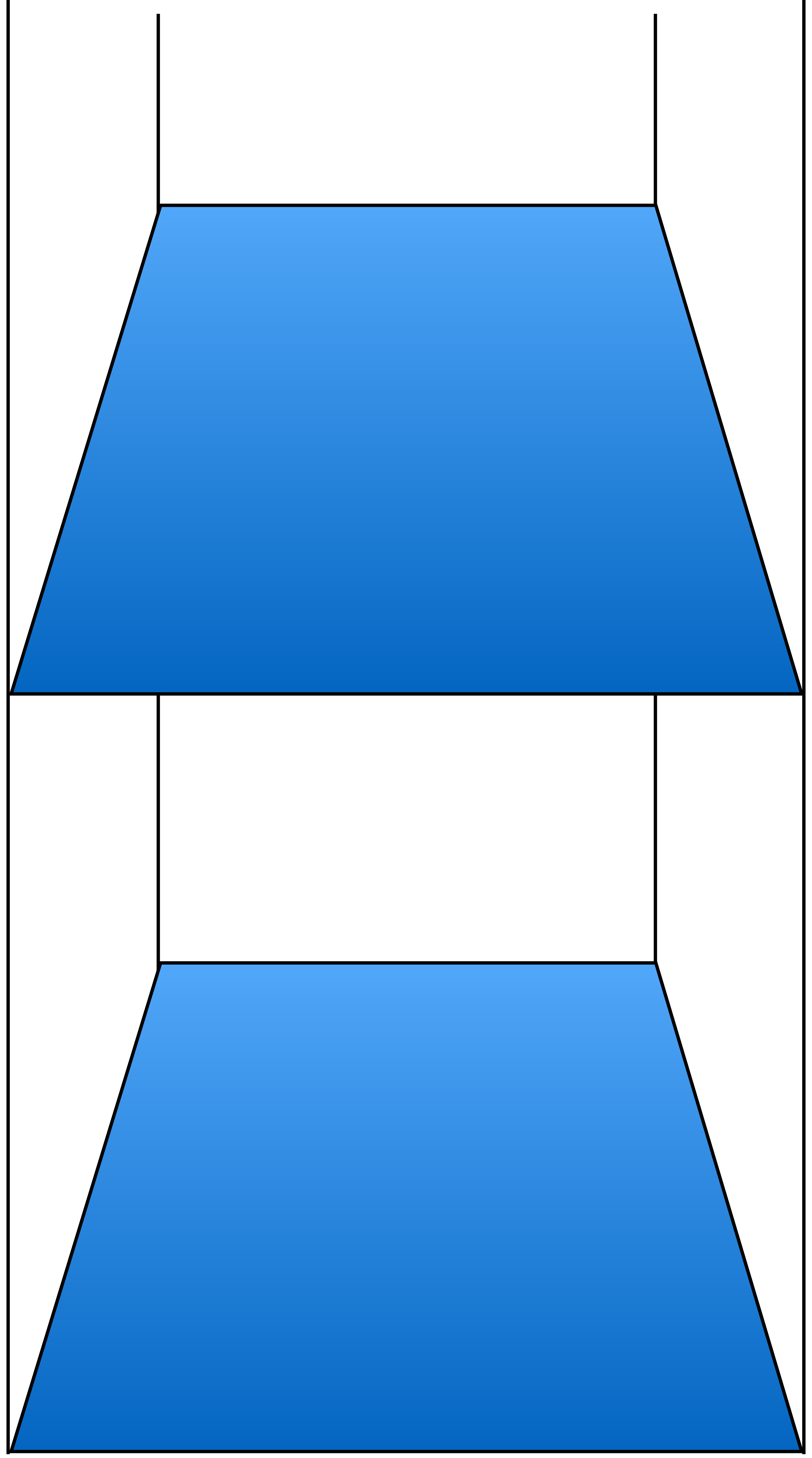
Separate **queries**, which read state, from **actions**.

Create **re-usable helpers** for queries.

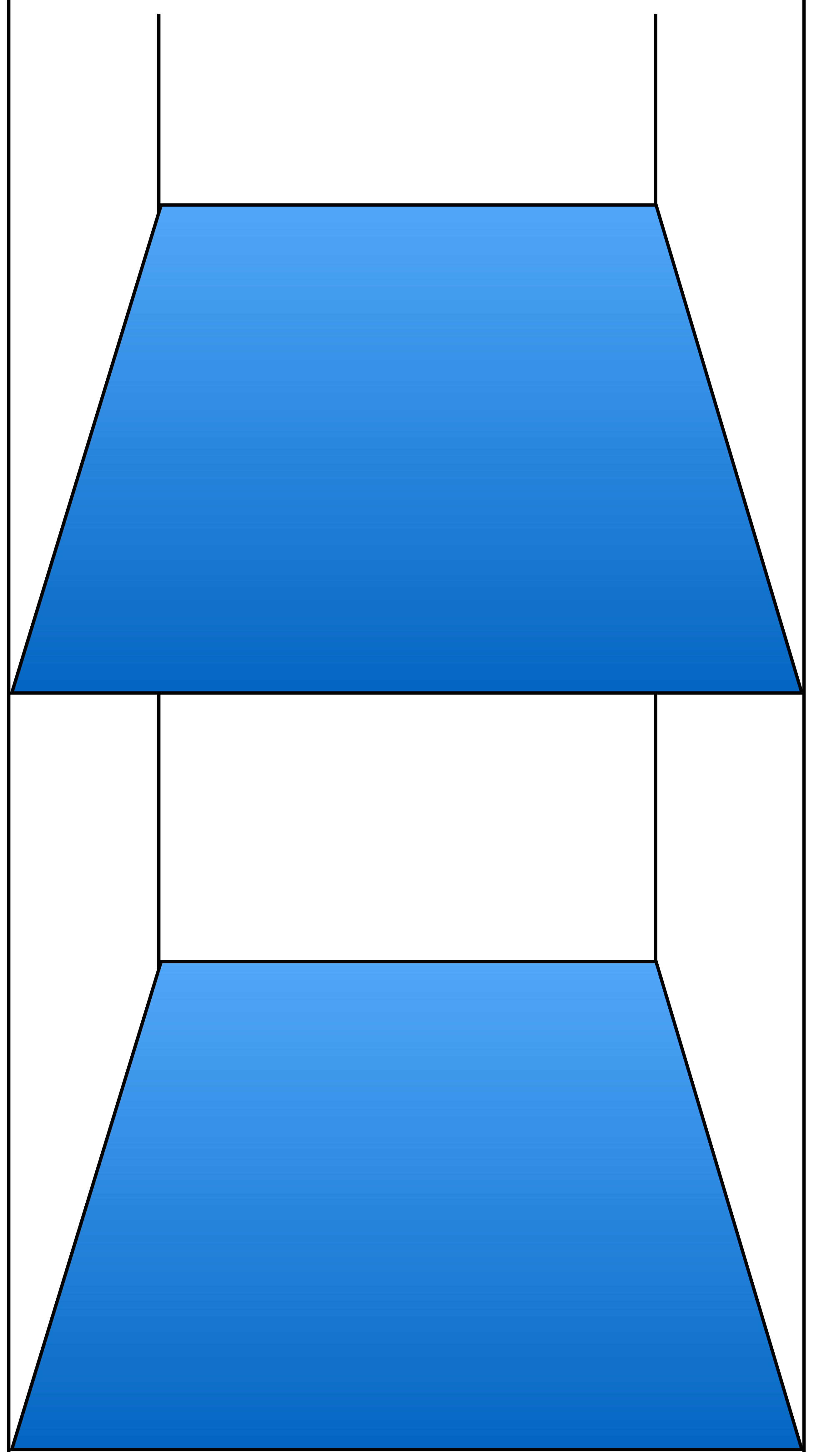
```rust
pub fn get_or_insert(&mut self, key: K, value: V) -> &V {
  for pair in &self.elements {
    if pair.0 == key {
      return &pair.1;
    }
  }

  self.elements.push((key, value));
  &self.elements.last().unwrap().1
}
```

```rust
pub fn get_or_insert<'a>(&'a mut self, key: K, value: V) -> &'a V {
  for pair in &'a self.elements {
    if pair.0 == key {
      return &'a pair.1;
    }
  }

  self.elements.push((key, value));
  &self.elements.last().unwrap().1
}
```

https://is.gd/fmWUQh



```
fn caller() {
    …
    let p = map.get_or_insert(…);
    …
}
impl<K,V> Map<K,V> {
  pub fn get_or_insert<'a>(…) -> &'a V {
    for pair in &'a self.elements {
      if pair.0 == key {
        return &'a pair.1;
      }
    }

    self.elements.push((key, value));
    &self.elements.last().unwrap().1
  }
}
```



```
fn caller() {
    …
    let p = map.get_or_insert(…);
    …
}

impl<K,V> Map<K,V> {
  pub fn get_or_insert<'a>(…) -> &'a V {
    match self.get(key) {
      Some(value) => return value,
      None => (),
    }

    self.elements.push((key, value));
    &self.elements.last().unwrap().1
  }
}
```

```rust
pub fn get_or_insert<'a>(&'a mut self, key: K, value: V) -> &'a V {
  if let Some(index) = self.index_of(&key) {
    return self.get(&key).unwrap();
  }

  self.elements.push((key, value));
  &self.elements.last().unwrap().1
}
```

Takeaway:

Separate **queries**, which read state, from **actions**.

Create **re-usable helpers** for queries.

Careful across fn boundaries.

```rust
pub struct Categorizer {
  categories: HashMap<String, String>,
  histogram: HashMap<String, usize>,
}

impl Categorizer {
  pub fn category(&self, class: &str) -> &str {
    self.categories.get(class)
  }

  pub fn histogram(&mut self, class: &str) {
    let category = self.category(class);
    *self.histogram.get_mut(category) += 1;
  }
}
```

```rust
pub struct Categorizer {
  categories: HashMap<String, String>,
  histogram: HashMap<String, usize>,
}

impl Categorizer {
 …
 pub fn histogram(&mut self, class: &str) {
    let category = self.categories.get(class);
    *self.histogram.get_mut(category) += 1;
  }
}
```

Takeaway:

Factor distinct state into subtypes.

Exercise: **successful borrowing**

http://rust-tutorials.com/exercises/

Takeaway:

Separate **queries**, which read state, from **actions**.

Create **re-usable helpers** for queries.

Careful across fn boundaries.

http://doc.rust-lang.org/std

Lifetimes in Structs

Previous section we had:

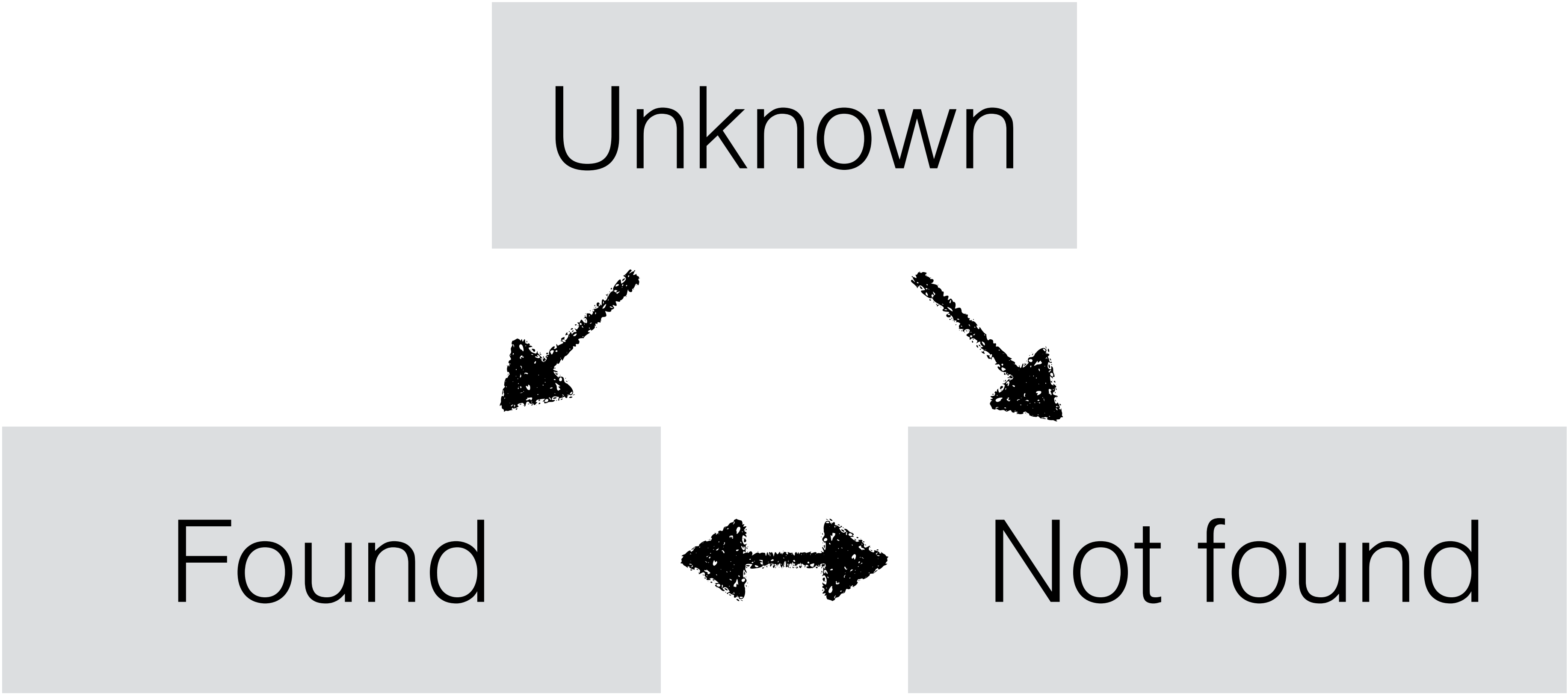
```
*map.get_or_insert(key, 0) += 1
```

But standard library does:

```
*map.entry(key).or_insert(0) += 1
```

Let's do that!

```
pub enum Entry<'map, K, V>
  where K: Eq, K: 'map, V: 'map
{
  Found(FoundEntry<'map, K, V>),
  NotFound(NotFoundEntry<'map, K, V>),
}
```




```
pub struct FoundEntry<'map, K, V>
  where K: Eq, K: 'map, V: 'map
{
  index: usize,
  elements: &'map mut Vec<(K,V)>
}
```

```
pub struct NotFoundEntry<'map, K, V>
  where K: Eq, K: 'map, V: 'map
{
  key: K,
  elements: &'map mut Vec<(K,V)>
}
```

```
enum Entry<'map, K, V>
  where K: Eq, K: 'map, V: 'map
{
  Found(FoundEntry<'map, K, V>),
  NotFound(NotFoundEntry<'map, K, V>),
}
```

Interpretation #1:

Lifetime of the reference to the map (or parts of the map).

Interpretation #2:

Lifetime of the entry itself.

```
fn main() {
    let mut map = Map::new();
    …
    {                                     'X = {scope of `entry`}
      let entry: Entry<'X, _, _> = map.entry(key);
      entry.or_insert(value);
    }
    map.insert(another_key, another_value);
}
                                             {scope of `map`}
```

Observation:

`Entry` has a “write-lock” for the duration of **‘X**.

```rust
enum Entry<'map, K, V>
  where K: Eq, K: 'map, V: 'map
{
  Found(FoundEntry<'map, K, V>),
  NotFound(NotFoundEntry<'map, K, V>),
}

struct FoundEntry<'map, K, V>
  where K: Eq, K: 'map, V: 'map
{
  index: usize,
  elements: &'map mut Vec<(K,V)>
}
```

Safe to borrow K, V for ‘map

```
impl<K, V> Map<K, V>
  where K: Eq
{
  pub fn entry<'map>(&'map mut self, key: K) -> Entry<'map, K, V> {
    …
  }
}
```

Caller gives us unique access to the map for the lifetime 'map.

Lifetime 'map continues as long as entry is in use.

BTW: This style also works. Not recommended.

```
pub fn entry(&mut self, key: K) -> Entry<K, V>
```

```
impl<K, V> Map<K, V>
  where K: Eq
{
  pub fn entry<‘map>(&’map mut self, key: K) -> Entry<‘map, K, V> {
    let pos = self.index_of(&key);
    match pos {
      Some(index) =>
        Entry::Found(FoundEntry {
          index: index,
          elements: &mut self.elements,
        }),

      None => …
    }
  }
}
```

Lifetime will be ‘map,
because of return type.

```rust
impl<K, V> Map<K, V>
  where K: Eq
{
  pub fn entry<'map>(&'map mut self, key: K) -> Entry<'map, K, V> {
    let pos = self.elements.iter().position(|pair| pair.0 == key);
    match pos {
      Some(index) => …,

      None =>
        Entry::NotFound(NotFoundEntry {
          key: key,
          elements: &mut self.elements,
        }),
    }
  }
}
```

```
impl<‘map, K, V> FoundEntry<'map, K, V>
  where K: Eq
{
  fn get(self) -> &’map mut V {
    &mut self.elements[self.index]
  }
}
```

`K: ‘map` implied in impls and fns

How do we know data.index is still valid?

```
impl<‘map, K, V> NotFoundEntry<'map, K, V>
  where K: Eq
{
  fn insert(self, value: V) -> &’map mut V {
    self.elements.push((self.key, value));
    &mut self.elements.last_mut().unwrap().1
  }
}
```

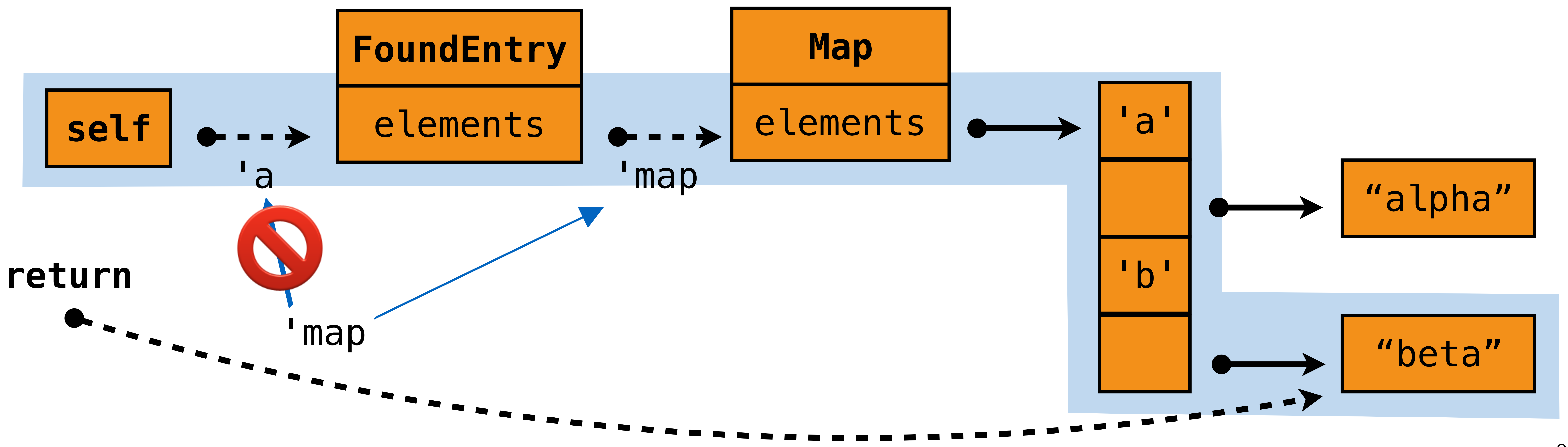
Would `insert()` make sense on `FoundEntry`?

```
impl<‘map, K, V> Entry<'map, K, V>
  where K: Eq
{
  fn or_insert(self, value: V) -> &’map mut V {
    match self {
      Entry::Found(data) => data.get(),
      Entry::NotFound(data) => data.insert(value),
    }
  }
}
```

```
impl<‘map, K, V> FoundEntry<'map, K, V>
  where K: Eq
{
  fn get(&mut self) -> &’map mut V {
    &mut self.elements[self.index]
  }
}
```

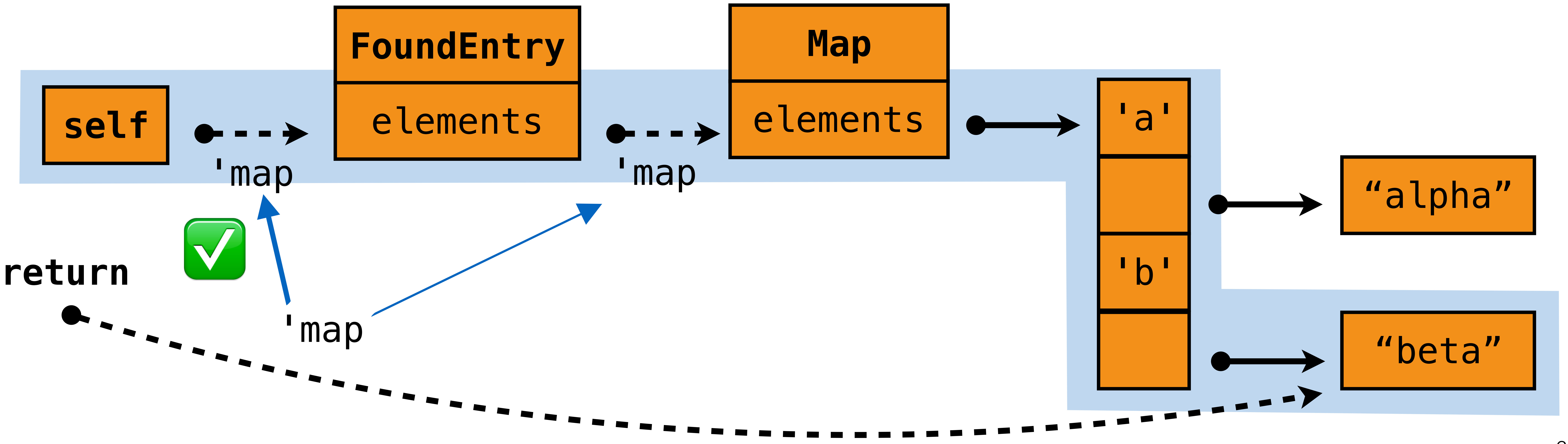
Why not this?

```
impl<‘map, K, V> FoundEntry<'map, K, V>
  where K: Eq
{
  fn get<‘a>(&’a mut self) -> &’map mut V {
    &mut self.elements[self.index]
  }
}
```



```
impl<‘map, K, V> FoundEntry<'map, K, V>
  where K: Eq
{
  fn get(&’map mut self) -> &’map mut V {
    &mut self.elements[self.index]
  }
}
```

OK, why not **this**?



Why not **this**?

```
fn or_insert(&'map mut self) -> &'map mut V
```

```
fn helper<'a>(map: &'a mut Map<K,V>) -> &'a mut V {
  map.entry(some_key()).or_insert(value)
}
```

```
fn helper<'a>(map: &'a mut Map<K,V>) -> &'a mut V {
  let mut entry = map.entry(some_key());
  Entry::or_insert(&mut entry, value)
}
```

What is **lifetime** of this borrow?
What is **scope** of entry?

`'a` 🚫

Takeaway:

- Structs can store references too
- Reference gives a “lock” on borrowed value
- Can encode a state machine
  - type per state, **fn**(self) -> [new state]

Exercise: **lifetimes in structs**

http://rust-tutorials.com/exercises/

Takeaway:

Separate **queries**, which read state, from **actions**.

Create **re-usable helpers** for queries.

Careful across fn boundaries.

http://doc.rust-lang.org/std

Sharing and mutability

Mutable reference in Rust:

really a **unique reference.**

Shared references can permit mutation:

but caution is required.

“Mutation is the root of all evil.”
— Strawman functional programmer

Don’t buy it.

```
let mut counter = 0;
counter += 1;
```

And yet…

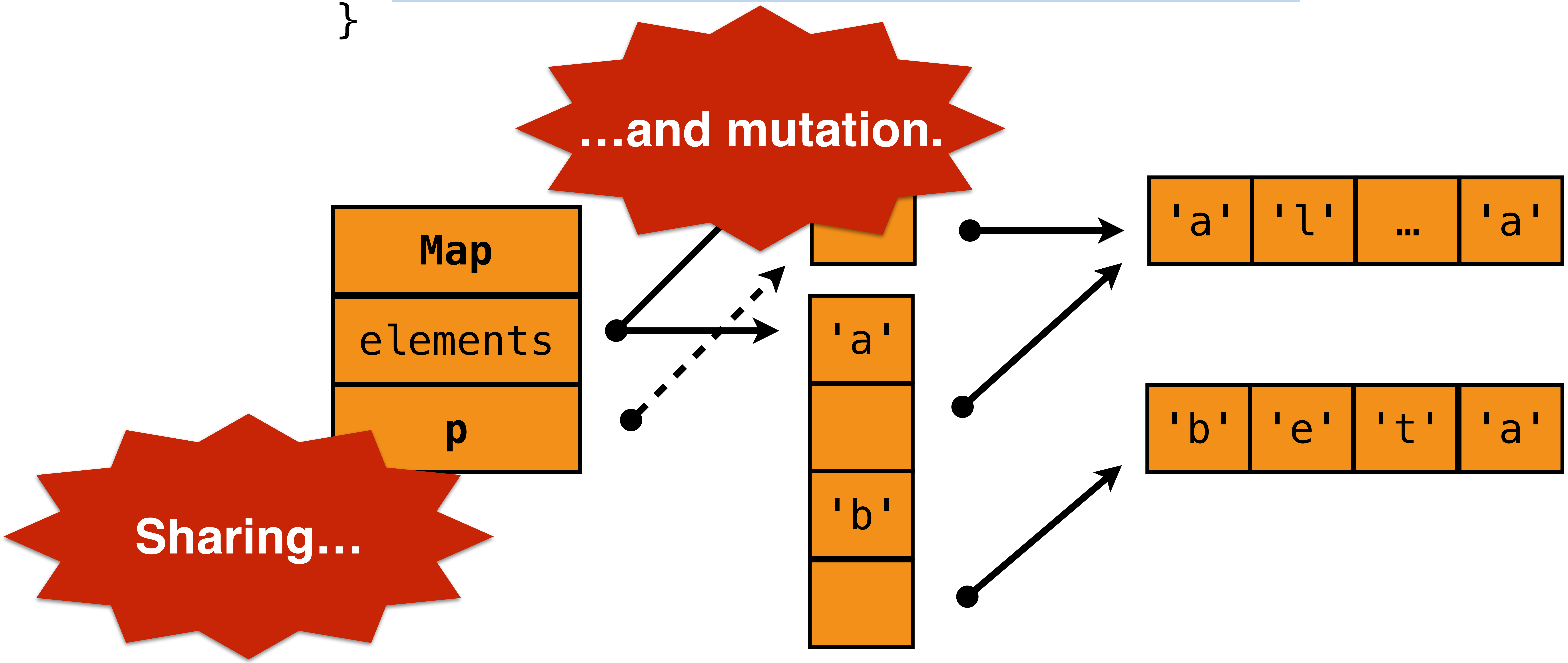
```
mod backend {
  fn search(…) {
    for entry in &context.big_map {
      process(entry);
    }
  }
}
```

Sharing…

```
mod middle {
  fn lazy_fill(…) {
    if !context.big_map.contains(&key) {
      context.big_map.insert(key, …);
    }
```

…and mutation.

```
fn main() {
    let mut map = Map::new();
    map.insert('a', format!("alpha"));
    let p = map.get(&'a');
    map.insert('b', format!("beta"));
}
```



Shared == Immutable*

```
fn helper(name: &String) {
  println!("{}", name);
}
```

OK. Just reads.

```
fn helper(name: &String) {
  name.push_str("foo");
}
```

Error. Writes.

```
error: cannot borrow immutable borrowed content `*name`
       as mutable
    name.push_str("s");
    ^~~~
```

* **Actually:** mutation only in **controlled circumstances**.

If you have a mutable reference `&mut T`…

- You have **unique access** for the lifetime of that reference.
- No other references to T.

If we have a shared reference `&Foo<T>`…

- The API of `Foo` can enforce that same guarantee!
  - *…and thus we can make mutation safe.*

```rust
struct Cell<T> {…}

impl<T: Copy> Cell<T> {
  fn new(value: T) -> Self {…}

  fn get(&self) -> T {…}

  fn set(&self, value: T) {…}
}
```

```rust
let counter = Cell::new(0);
let value = counter.get();
counter.set(value + 1);
```

```rust
let counter = Rc::new(Cell::new(0));
let counter2 = counter.clone();

let value = counter.get(); // 0
counter.set(value + 1);

let value = counter2.get(); // 1
```

```rust
use std::cell::UnsafeCell;

struct Cell<T> {
  data: UnsafeCell<T>
}

impl<T: Copy> Cell<T> {
  …

  fn set(&self, value: T) {
    unsafe {
      let ptr: *mut T = self.data.get();
      *ptr = value;
    }
  }
}
```

Why is this safe?

- API offers no way to get a reference to T.
- Not safe to pass between threads.
  - (The default for UnsafeCell)

Great for Cell<u32>, but Cell<Vec<u32>>…

```rust
use std::cell::UnsafeCell;

struct Cell<T> {
  data: UnsafeCell<T>
}

impl<T: Clone> Cell<T> {
  …

  fn set(&self, value: T) {
    unsafe {
      let ptr: *mut T = self.data.get();
      *ptr = value.clone();
    }
  }
}
```

```rust
impl Clone for MyType {
  fn clone(&self) {
    // may have access to the cell!
  }
}
```

What could go wrong?

```
let vec = RefCell::new(vec![]);

{
  let mut p = vec.borrow_mut();         ← Acquires “write lock”.
  // let mut q = vec.borrow();          ← Would panic.
  p.push(format!(“data”));              ← Mutation permitted.
}  ← Release “read locks”.

{
  let p = vec.borrow();
  let q = vec.borrow();                 ← Acquires “read lock”.
  assert_eq!(&p[0], &q[0]);
}  ← Release “read locks”.
```

Unlike entry, `borrow` takes `&self`:

- Guarantees `RefCell` will not be moved
- Does **not** guarantee unique access

```
struct RefCell<T> {…}

struct Ref<‘b, T: ‘b> {…}

impl<T> RefCell<T> {
  fn borrow<‘b>(&’b self) -> Ref<‘b, T> {
    // twiddle bits to acquire lock
    Ref { … } // return a Ref that contains `self`
  }
}


impl<‘b, T> Deref for Ref<‘b, T: ‘b> {
  type Target = T;
  …
}
```

```
{
  let vec = RefCell::new(vec![]);

  let mut p = vec.borrow_mut();
  p.push(format!(“data”));

  let p = vec.borrow();
  let q = vec.borrow();
  assert_eq!(&p[0], &q[0]);
}
```

```
{
  let mut vec = vec![];

  let mut p = &mut vec;
  p.push(format!(“data”));

  let p = &vec;
  let q = &vec;
  assert_eq!(&p[0], &q[0]);
}
```

```
mod backend {
  fn search(…) {
    for entry in context.big_map.borrow() {
      process(entry);
    }
  }
}
```

```
mod middle {
  fn lazy_fill(…) {
    …
    context.big_map.borrow_mut()
                   .insert(key, …);
  }
}
```

```rust
pub struct Context {
  pub big_map: RefCell<Map<…>>
}
```

Open-ended access is error-prone.

```rust
pub struct Context {
  big_map: RefCell<Map<…>>
}

impl Context {
  pub fn find_entry(&self, key: &K) -> V {
    self.big_map.borrow().get(key).cloned()
  }

  pub fn add_entry(&self, k: K, v: V) {
    self.big_map.borrow_mut().insert(k, v);
  }
}
```

Controlled accessors can be audited, but less flexible and repetitive.

No best answer yet.

What about threads?

- Cell and RefCell cannot be shared across threads.
- AtomicU32 and Mutex can but offer similar usability tradeoffs.

Some alternatives to explore:

- Avoid using shared/mutability:
  - often you can replace a `&T` with an index into a vector
- Persistent data structures.

Your experiences?

Exercise: **aliasing and mutability**

http://rust-tutorials.com/exercises/

Takeaway:

Separate **queries**, which read state, from **actions**.

Create **re-usable helpers** for queries.

Careful across fn boundaries.

http://doc.rust-lang.org/std

Open questions