Interior mutability & Reference counting

Chapters 4 & 9

Daniël de Kok

Motivation interior mutability

Feature $\longrightarrow$ number mapping

Consider a small data structure to turn features into integers:

```
use std::collections::HashMap;
use std::hash::Hash;

pub struct Numberer<T>(HashMap<T, usize>);
```

Feature → number mapping

Consider a small data structure to turn features into integers:

```
use std::collections::HashMap;
use std::hash::Hash;

pub struct Numberer<T>(HashMap<T, usize>);

impl<T> Default for Numberer<T>
where
    T: Eq + Hash,
{
    fn default() -> Self {
        Numberer(HashMap::new())
    }
}
```

Feature → number mapping

```rust
impl<T> Numberer<T>
where
    T: Eq + Hash,
{
    pub fn get(&mut self, val: T) -> usize {
        let next_idx = self.0.len();
        *self.0.entry(val).or_insert(next_idx)
    }
}
```

Feature → number mapping

```rust
impl<T> Numberer<T>
where
    T: Eq + Hash,
{
    pub fn get(&mut self, val: T) -> usize {
        let next_idx = self.0.len();
        *self.0.entry(val).or_insert(next_idx)
    }
}

let mut numberer: Numberer<&'static str> = Numberer::default();
assert_eq!(numberer.get("hello"), 0);
assert_eq!(numberer.get("Rust"), 1);
assert_eq!(numberer.get("hello"), 0);
assert_eq!(numberer.get("again"), 2);
```

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably.

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably. Unsatisfying, because:

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably. Unsatisfying, because:

- `Numberer` can be seen as a total function $T \to \mathbb{N}$.

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably. Unsatisfying, because:

- `Numberer` can be seen as a total function $T \to \mathbb{N}$.
  - By-need index generation is an implementation detail.
  - Alternative implementation: feature hashing.

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably. Unsatisfying, because:

- `Numberer` can be seen as a total function $T \to \mathbb{N}$.
  - By-need index generation is an implementation detail.
  - Alternative implementation: feature hashing.
- The mutable binding trickles up the call chain.

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably. Unsatisfying, because:

- `Numberer` can be seen as a total function $T \to \mathbb{N}$.
  - By-need index generation is an implementation detail.
  - Alternative implementation: feature hashing.
- The mutable binding trickles up the call chain.
  - Suppose that `Classifier` has `Numberer` as a field.
  - Methods that use `Numberer::get` also need to take `&mut self`.

Mutability

```
pub fn get(&mut self, val: T) -> usize
```

`Numberer::get` borrows `self` mutably. Unsatisfying, because:

- `Numberer` can be seen as a total function $T \to \mathbb{N}$.
  - By-need index generation is an implementation detail.
  - Alternative implementation: feature hashing.
- The mutable binding trickles up the call chain.
  - Suppose that `Classifier` has `Numberer` as a field.
  - Methods that use `Numberer::get` also need to take `&mut self`.

We need an escape hatch!

Interior mutability

- **Interior mutability** is our escape hatch.

Interior mutability

- **Interior mutability** is our escape hatch.
- Allows you to mutate members without an `&mut` binding.

Interior mutability

- **Interior mutability** is our escape hatch.
- Allows you to mutate members without an `&mut` binding.
- Borrowing rules still apply:
  - Multiple immutable borrows; xor
  - a single mutable borrow.

Interior mutability

- **Interior mutability** is our escape hatch.
- Allows you to mutate members without an `&mut` binding.
- Borrowing rules still apply:
  - Multiple immutable borrows; xor
  - a single mutable borrow.
- However: **enforced at run-time** rather than compile-time.

Interior mutability

- **Interior mutability** is our escape hatch.
- Allows you to mutate members without an `&mut` binding.
- Borrowing rules still apply:
  - Multiple immutable borrows; xor
  - a single mutable borrow.
- However: **enforced at run-time** rather than compile-time.
- Be judicious with interior mutability: compile-time errors are nicer.

Two types of interior mutability

Rust offers two data types for interior mutability:

1. `Cell`: works with values
2. `RefCell`: works with references

Two types of interior mutability

Rust offers two data types for interior mutability:

1. `Cell`: works with values
2. `RefCell`: works with references

We will first explore `RefCell`, because it fits most naturally with our motivating example.

RefCell

RefCell: construction

```
// Create a `RefCell` that owns a `String`.
let cell = RefCell::new("hello RefCell".to_string());
```

RefCell: construction

```
// Create a `RefCell` that owns a `String`.
let cell = RefCell::new("hello RefCell".to_string());

assert_eq!(
  // Replace the owned `String` by another owned `String`,
  // the original owned data is returned.
  cell.replace("goodbye RefCell".to_string()),
  "hello RefCell");
```

RefCell: construction

```
// Create a `RefCell` that owns a `String`.
let cell = RefCell::new("hello RefCell".to_string());

assert_eq!(
  // Replace the owned `String` by another owned `String`,
  // the original owned data is returned.
  cell.replace("goodbye RefCell".to_string()),
  "hello RefCell");

assert_eq!(
  // Move the owned `String` out of the `RefCell`. The
  // `RefCell` is consumed after this.
  cell.into_inner(),
  "goodbye RefCell");
```

RefCell: borrowing

RefCell provides the borrow method to borrow the wrapped value:

```
pub fn borrow(&self) -> Ref<T>
```

RefCell: borrowing

RefCell provides the borrow method to borrow the wrapped value:

```
pub fn borrow(&self) -> Ref<T>
```

- borrow does not simply return &T.

RefCell: borrowing

RefCell provides the borrow method to borrow the wrapped value:

```
pub fn borrow(&self) -> Ref<T>
```

- borrow does not simply return &T.
- It needs a data structure with an associated Drop implementation to keep track of the number of borrows. Why?

RefCell: borrowing

RefCell provides the borrow method to borrow the wrapped value:

```
pub fn borrow(&self) -> Ref<T>
```

- borrow does not simply return &T.
- It needs a data structure with an associated Drop implementation to keep track of the number of borrows. Why?
- To enforce borrowing rules.

RefCell: borrowing

RefCell provides the borrow method to borrow the wrapped value:

```
pub fn borrow(&self) -> Ref<T>
```

- borrow does not simply return &T.
- It needs a data structure with an associated Drop implementation to keep track of the number of borrows. Why?
- To enforce borrowing rules.
- Ref implements the Deref trait.

RefCell: borrowing

```
let cell = RefCell::new("hello RefCell".to_string());

let borrow1 = cell.borrow();
let borrow2 = cell.borrow();
```

RefCell: borrowing

```
let cell = RefCell::new("hello RefCell".to_string());

let borrow1 = cell.borrow();
let borrow2 = cell.borrow();

assert_eq!(borrow1.len(), 13);
assert_eq!(*borrow2, "hello RefCell");
```

RefCell: borrowing mutably

```
let cell = RefCell::new("hello RefCell".to_string());

{
    let mut b = cell.borrow_mut();
    b.push('!');
}

assert_eq!(cell.into_inner(), "hello RefCell!");
```

`RefCell`: borrowing mutably (2)

How does `RefCell` bring interior mutability?

RefCell: borrowing mutably (2)

How does RefCell bring interior mutability? borrow_mut does not borrow self mutably:

```
pub fn borrow_mut(&self) -> RefMut<T>
```

RefCell: borrowing mutably (2)

How does RefCell bring interior mutability? borrow_mut does not borrow self mutably:

```
pub fn borrow_mut(&self) -> RefMut<T>
```

The compile-time borrowing rules are circumvented using **unsafe Rust**.

Enforcement of the borrow rules

The borrow rules are enforced at runtime:

```
let cell = RefCell::new("hello RefCell".to_string());

let immutable = cell.borrow();

// Compiles, but panics at runtime with:
// 'already borrowed: BorrowMutError'
let mut mutable = cell.borrow_mut();
```

Feature mapping (updated)

```rust
use std::cell::RefCell;
use std::collections::HashMap;
use std::hash::Hash;

pub struct Numberer<T>(RefCell<HashMap<T, usize>>);
```

Feature mapping (updated)

```rust
use std::cell::RefCell;
use std::collections::HashMap;
use std::hash::Hash;

pub struct Numberer<T>(RefCell<HashMap<T, usize>>);

impl<T> Default for Numberer<T>
    where
    T: Eq + Hash,
{
    fn default() -> Self {
        Numberer(RefCell::new(HashMap::new()))
        // Or: Numberer(Default::default())
    }
}
```

Feature mapping (updated)

```
impl<T> Numberer<T>
where
    T: Eq + Hash,
{
    pub fn get(&self, val: T) -> usize {
        let next_idx = self.0.borrow().len();
        *self.0.borrow_mut().entry(val).or_insert(next_idx)
    }
}
```

Feature mapping (updated)

```rust
impl<T> Numberer<T>
where
    T: Eq + Hash,
{
    pub fn get(&self, val: T) -> usize {
        let next_idx = self.0.borrow().len();
        *self.0.borrow_mut().entry(val).or_insert(next_idx)
    }
}

#[test]
fn numberer_test() {
    let numberer: Numberer<&'static str> = Numberer::default();
    assert_eq!(numberer.get("hello"), 0);
    assert_eq!(numberer.get("Rust"), 1);
    assert_eq!(numberer.get("hello"), 0);
    assert_eq!(numberer.get("again"), 2);
}
```

Cell

Introduction

- `Cell` is value-oriented.

Introduction

- `Cell` is value-oriented.
- Does not need/implement run-time borrows checking.

`Cell`: construction

```
// Create a `Cell` that owns a `String`.
let cell = Cell::new("Rustic".to_string());
```

`Cell`: construction

```rust
// Create a `Cell` that owns a `String`.
let cell = Cell::new("Rustic".to_string());

assert_eq!(
  // Replace the owned `String` by another owned `String`,
  // the original owned data is returned.
  cell.replace("cells".to_string()),
  "Rustic");
```

`Cell`: construction

```rust
// Create a `Cell` that owns a `String`.
let cell = Cell::new("Rustic".to_string());

assert_eq!(
  // Replace the owned `String` by another owned `String`,
  // the original owned data is returned.
  cell.replace("cells".to_string()),
  "Rustic");

// Set the value. Drops the owned value.
cell.set("are mutable".to_string());
```

`Cell`: construction

```
assert_eq!(
  // Move the owned `String` out of the `Cell`. The
  // `Cell` is consumed after this.
  cell.into_inner(),
  "cells");
```

`Cell`: Copy types

`Cell` implements a `get` method for copy types, that returns a copy of the current value:

```
let cell = Cell::new(5);
assert_eq!(cell.get(), 5);
cell.set(6);
assert_eq!(cell.get(), 6);
```

`Cell`: `Default` types

`Cell` implements a `take` method for `Default` types. It is equivalent to replacing the value by `Default::default()`:

```
let cell = Cell::new(vec![1, 2, 3]);

assert_eq!(
  // Move the vector [1, 2, 3] out of the cell,
  // replace it by an empty `Vec`.
  cell.take(),
  vec![1, 2, 3]);
```

`Cell`: `Default` types

`Cell` implements a `take` method for `Default` types. It is equivalent to replacing the value by `Default::default()`:

```
let cell = Cell::new(vec![1, 2, 3]);

assert_eq!(
  // Move the vector [1, 2, 3] out of the cell,
  // replace it by an empty `Vec`.
  cell.take(),
  vec![1, 2, 3]);

assert_eq!(
  // Unwrap the inner vector [].
  cell.into_inner(),
  vec![]);
```

In-class assignment

Implement `Numberer` with `Cell` interior mutability.

`Cell`: how does it work?

- Core primitive for interior mutability: `UnsafeCell`:

`Cell`: how does it work?

- Core primitive for interior mutability: `UnsafeCell`:
  - Wraps a value of type T.

`Cell`: how does it work?

- Core primitive for interior mutability: `UnsafeCell`:
  - Wraps a value of type `T`.
  - Provides a `get` method that returns `*mut T`.

`Cell`: how does it work?

- Core primitive for interior mutability: `UnsafeCell`:
  - Wraps a value of type `T`.
  - Provides a `get` method that returns `*mut T`.
  - Users of `UnsafeCell` should enforce the borrowing rules.

`Cell`: how does it work?

- Core primitive for interior mutability: `UnsafeCell`:
  - Wraps a value of type `T`.
  - Provides a `get` method that returns `*mut T`.
  - Users of `UnsafeCell` should enforce the borrowing rules.
- `Cell` wraps `UnsafeCell`:

`Cell`: how does it work?

- Core primitive for interior mutability: `UnsafeCell`:
  - Wraps a value of type `T`.
  - Provides a `get` method that returns `*mut T`.
  - Users of `UnsafeCell` should enforce the borrowing rules.
- `Cell` wraps `UnsafeCell`:
  - Provides safety by not providing references to the wrapped data.

`RefCell`: how does it work?

- The wrapped value is stored in an `UnsafeCell`.

`RefCell`: how does it work?

- The wrapped value is stored in an `UnsafeCell`.
- A cell is used to keep track of borrows:

`RefCell`: how does it work?

- The wrapped value is stored in an `UnsafeCell`.
- A cell is used to keep track of borrows:

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

`RefCell`: how does it work?

- The wrapped value is stored in an `UnsafeCell`.
- A cell is used to keep track of borrows:

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

- BorrowFlag is a `usize` with one of the following values:
  - `0`: no borrows
  - `!0`: a mutable borrow
  - `[1, MAX-1]`: N immutable borrows

`RefCell`: how does it work?

- The wrapped value is stored in an `UnsafeCell`.
- A cell is used to keep track of borrows:

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

- BorrowFlag is a `usize` with one of the following values:
  - `0`: no borrows
  - `!0`: a mutable borrow
  - `[1, MAX-1]`: N immutable borrows

RefCell: how does it work?

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

- `borrow()` permitted when `borrow != !0`
  - sets `borrow` to `borrow + 1`
- `borrow_mut()` permitted when `borrow` is 0
  - sets `borrow` to `!0`
- When a `borrow()`/`borrow_mut()` is not permitted → panic.

Reference counting

Introduction

- For some data, there is no clear single owner.

Introduction

- For some data, there is no clear single owner.
- Examples:
  - A model that is used by multiple views in a GUI application.
  - Immutable data structures with sharing.
  - Graphs (but watch out for cycles!).

Introduction

- For some data, there is no clear single owner.
- Examples:
  - A model that is used by multiple views in a GUI application.
  - Immutable data structures with sharing.
  - Graphs (but watch out for cycles!).
- Rust provides shared ownership with reference counting throug Rc.

Reference counting

- Reference counting is a form of **garbage collection**.

Reference counting

- Reference counting is a form of **garbage collection**.
- Data is stored with a counter:

Reference counting

- Reference counting is a form of **garbage collection**.
- Data is stored with a counter:
  - Creating a new reference increments the counter.

Reference counting

- Reference counting is a form of **garbage collection**.
- Data is stored with a counter:
  - Creating a new reference increments the counter.
  - Dropping a reference decrements the counter.

Reference counting

- Reference counting is a form of **garbage collection**.
- Data is stored with a counter:
  - Creating a new reference increments the counter.
  - Dropping a reference decrements the counter.
  - The data is dropped when the counter reaches 0.

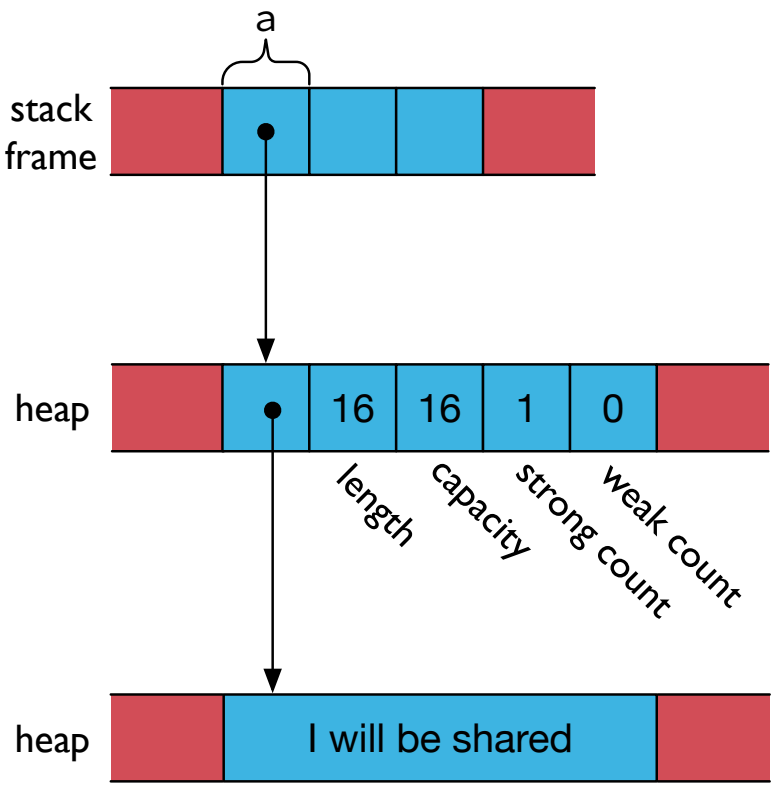
Reference counting

- Reference counting is a form of **garbage collection**.
- Data is stored with a counter:
  - Creating a new reference increments the counter.
  - Dropping a reference decrements the counter.
  - The data is dropped when the counter reaches 0.
- Reference in this context is not to be confused with Rust's references.

Reference counting

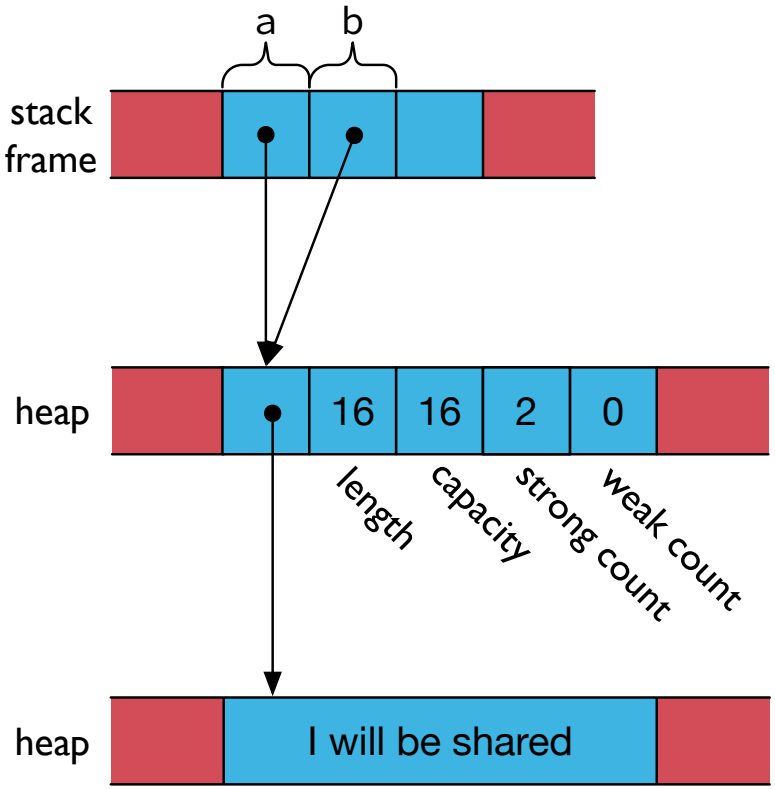
- Reference counting is a form of **garbage collection**.
- Data is stored with a counter:
  - Creating a new reference increments the counter.
  - Dropping a reference decrements the counter.
  - The data is dropped when the counter reaches 0.
- Reference in this context is not to be confused with Rust's references.
- Standard form of garbage collection in e.g. (C)Python.

Shared ownership through Rc



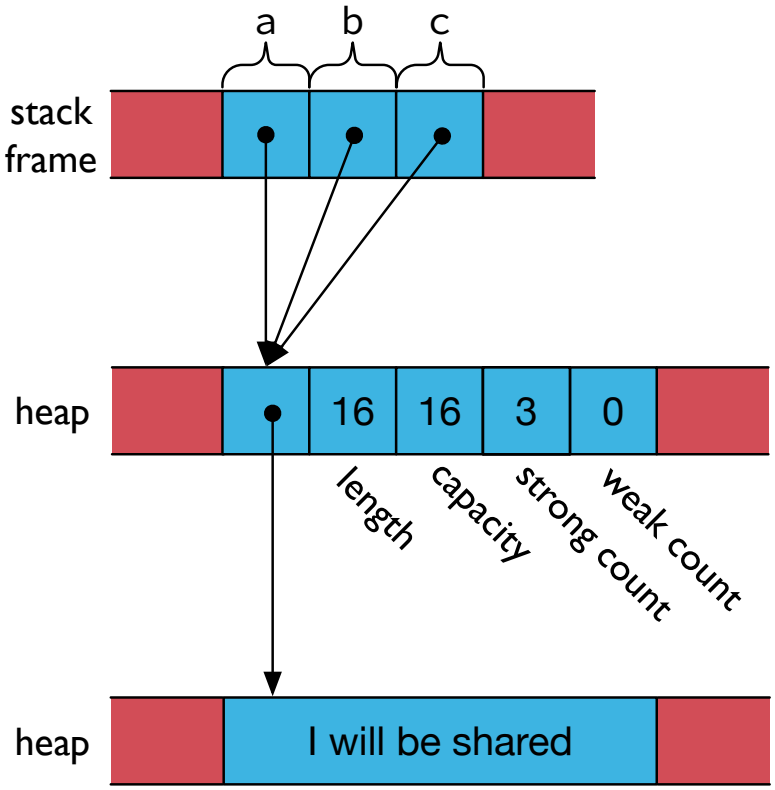

```
let a = Rc::new("I will be shared".to_string());
assert_eq!(Rc::strong_count(&a), 1);
```

Shared ownership through Rc



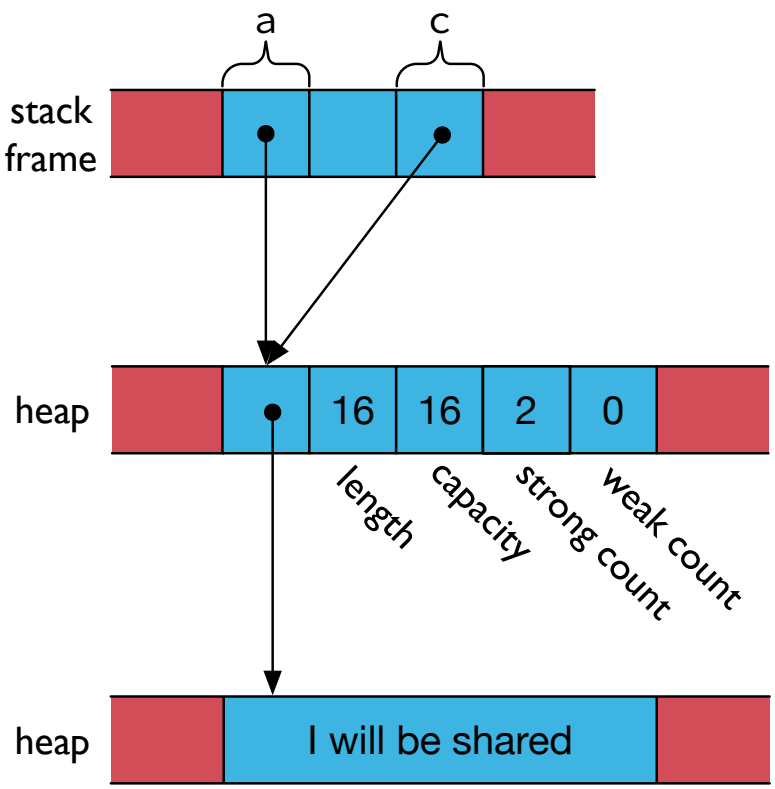

```
let b = a.clone();
assert_eq!(Rc::strong_count(&a), 2);
```

Shared ownership through Rc



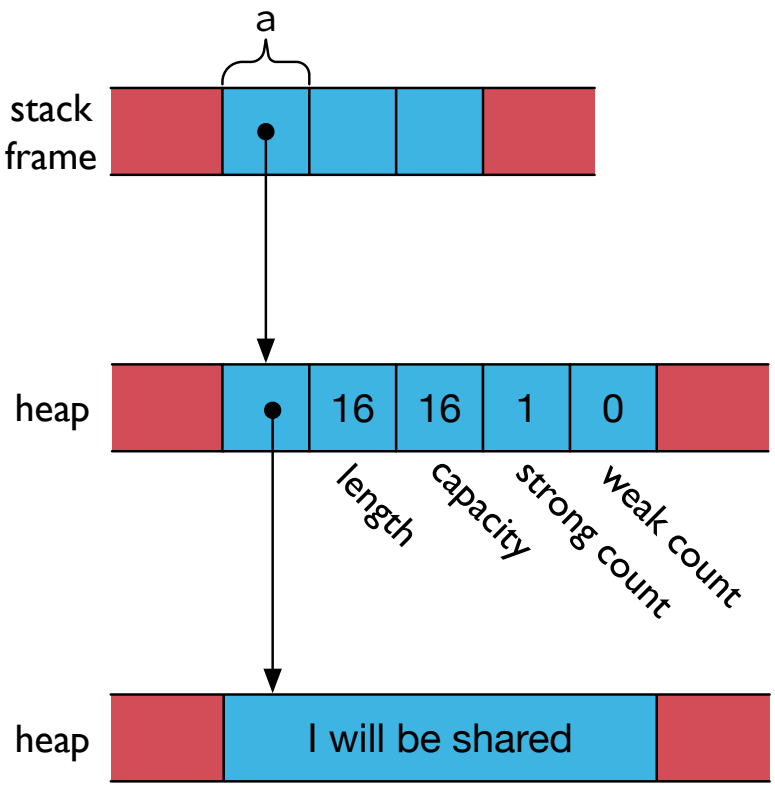

```
let c = a.clone();
assert_eq!(Rc::strong_count(&a), 3);
```

Shared ownership through Rc



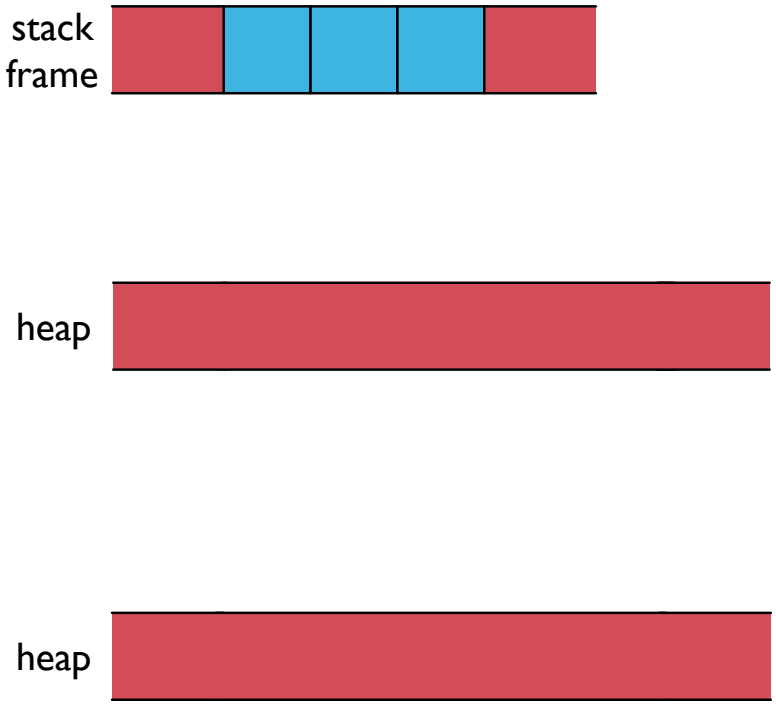

```
drop(b);
assert_eq!(Rc::strong_count(&a), 2);
```

Shared ownership through Rc




```
drop(c);
assert_eq!(Rc::strong_count(&a), 1);
```

Shared ownership through Rc




```
drop(a);
```

Using Rc data

Rc implements Deref:

```
let a = Rc::new("I will be shared".to_string());
let b = a.clone();
assert_eq!(b.len(), 16);
```

Combining `Rc` and `RefCell`

- `Rc<T>` is immutable (unless the reference count is 1).

Combining Rc and RefCell

- Rc<T> is immutable (unless the reference count is 1).
- Wrapping RefCell<T> in Rc gives us mutable reference-counted memory:

```
let s = "I will be shared".to_string();
let a: Rc<RefCell<String>> = Rc::new(RefCell::new(s));
let b = a.clone();
b.borrow_mut().push_str("... Done!");
assert_eq!(*a.borrow(), "I will be shared... Done!");
```

Combining Rc and RefCell

- Rc<T> is immutable (unless the reference count is 1).
- Wrapping RefCell<T> in Rc gives us mutable reference-counted memory:

```
let s = "I will be shared".to_string();
let a: Rc<RefCell<String>> = Rc::new(RefCell::new(s));
let b = a.clone();
b.borrow_mut().push_str("... Done!");
assert_eq!(*a.borrow(), "I will be shared... Done!");
```

- Similarly wrapping in Rc<T> in RefCell gives us reference counting pointers that can be updated.

Watch out: cycles

Using `Rc` with `RefCell` makes it possible to create cycles in memory.

Watch out: cycles

Using `Rc` with `RefCell` makes it possible to create cycles in memory.

```
#[derive(Debug)]
enum List {
    Cons(usize, RefCell<Rc<List>>),
    Nil,
}
```

Watch out: cycles

Using `Rc` with `RefCell` makes it possible to create cycles in memory.

```
#[derive(Debug)]
enum List {
    Cons(usize, RefCell<Rc<List>>),
    Nil,
}

let a = Rc::new(Cons(1, RefCell::new(Rc::new(Nil))));
let b = Rc::new(Cons(2, RefCell::new(a.clone())));
```

Watch out: cycles

Using `Rc` with `RefCell` makes it possible to create cycles in memory.

```
#[derive(Debug)]
enum List {
    Cons(usize, RefCell<Rc<List>>),
    Nil,
}

let a = Rc::new(Cons(1, RefCell::new(Rc::new(Nil))));
let b = Rc::new(Cons(2, RefCell::new(a.clone())));

if let Cons(_, cell) = &*a {
  *cell.borrow_mut() = b.clone();
}
```

Ramifications

- Cycles are not deallocated!

Ramifications

- Cycles are not deallocated!
- Some functions are not well-behaved on memory with cycles.
  - E.g. the `println` macro will panic with a stack overflow.

Ramifications

- Cycles are not deallocated!
- Some functions are not well-behaved on memory with cycles.
  - E.g. the `println` macro will panic with a stack overflow.
- Cycles can be broken with weak references.

Conclusion

Conclusion

- Use `Cell` or `RefCell` for interior mutability.
- Use `Rc` for shared ownership.

Conclusion

- Use `Cell` or `RefCell` for interior mutability.
- Use `Rc` for shared ownership.
- Be judicious with these three data structures:
  - Guarantees change go from compile-time to run-time.
  - `Rc` + `RefCell` can create cycles.
  - Try with exterior mutability and single owner first.

Conclusion

- Use `Cell` or `RefCell` for interior mutability.
- Use `Rc` for shared ownership.
- Be judicious with these three data structures:
  - Guarantees change go from compile-time to run-time.
  - `Rc` + `RefCell` can create cycles.
  - Try with exterior mutability and single owner first.
- `Cell` and `RefCell` references cannot be shared between threads.

Conclusion

- Use `Cell` or `RefCell` for interior mutability.
- Use `Rc` for shared ownership.
- Be judicious with these three data structures:
  - Guarantees change go from compile-time to run-time.
  - `Rc` + `RefCell` can create cycles.
  - Try with exterior mutability and single owner first.
- `Cell` and `RefCell` references cannot be shared between threads.
- `Rc` cannot be shared nor send between threads.

Conclusion

- Use `Cell` or `RefCell` for interior mutability.
- Use `Rc` for shared ownership.
- Be judicious with these three data structures:
  - Guarantees change go from compile-time to run-time.
  - `Rc` + `RefCell` can create cycles.
  - Try with exterior mutability and single owner first.
- `Cell` and `RefCell` references cannot be shared between threads.
- `Rc` cannot be shared nor send between threads.
- There are thread-safe counterparts, such as `Mutex`, `RwLock`, and `Arc`.