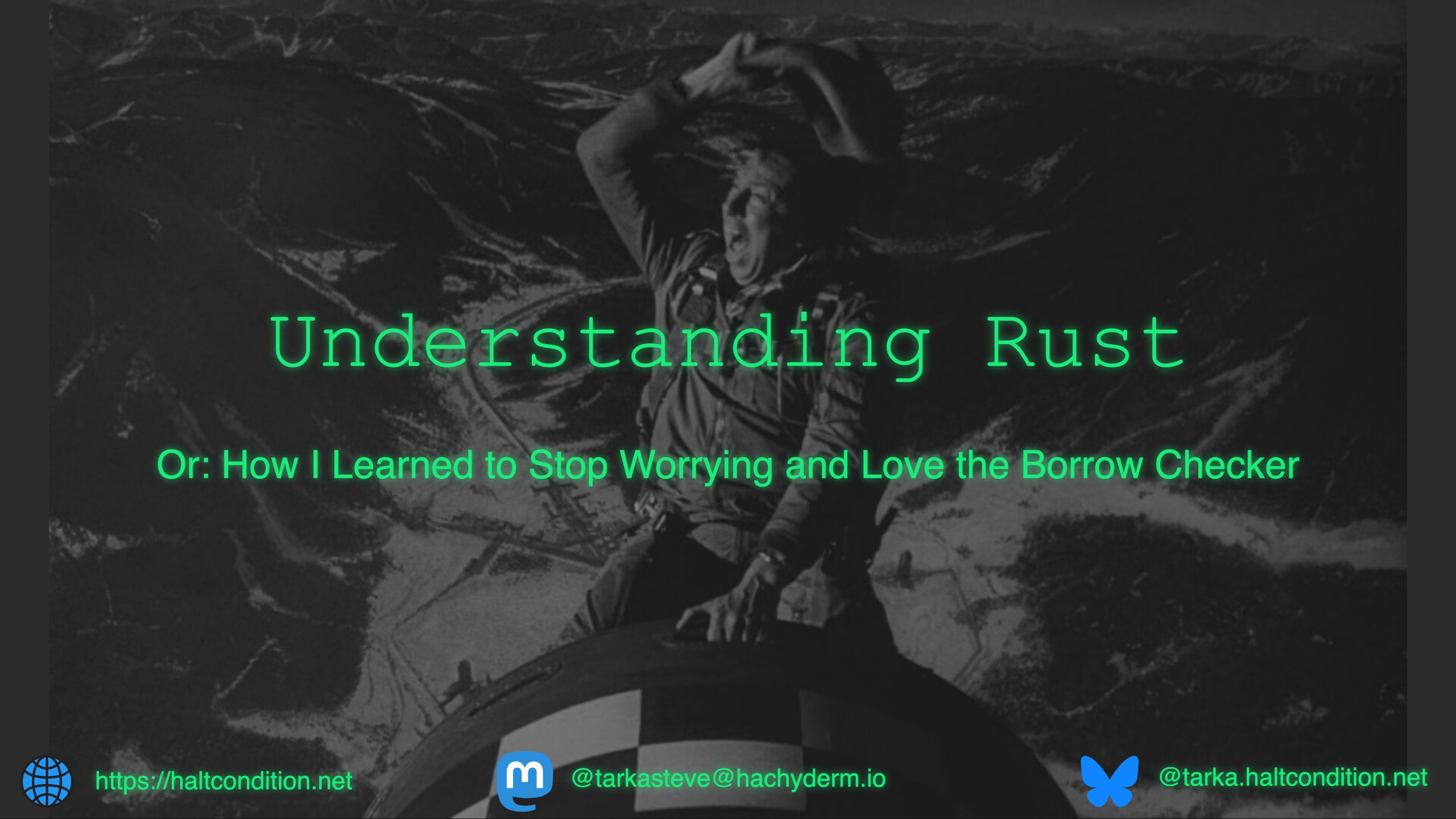


Understanding Rust

Or: How I Learned to Stop Worrying and Love the Borrow Checker

https://haltcondition.net

@tarkasteve@hachyderm.io

@tarka.haltcondition.net

Speaker notes

Hi, I'm Steve Smith, and I'm here to talk about Rust.

Now, this isn't a Rust sales pitch per-se, I assume you've already been sold on the fundamental promise of Rust.
I assume you get the basic moves and borrows, and the elegant error-handling system.
I won't cover those here. However I will advocate for persevering with Rust even when you hit the now-infamous Rust learning curve, but mostly I'm here to help you a bit further up that curve.

I can't cover every aspect of Rust; there's a lot of things like advanced lifetimes and async that I'll only touch on briefly.
My aim here is instead to explain why some of Rust's annoyances are fundamental to what makes it Rust and help you understand them.
I can't give you all the answers here, but I at least can help you ask the right questions.

[The subtitle is really a chapter heading. There are many more chapters to understanding Rust but hopefully this one will bring you a step closer.]

Who Am I?

Getting out of trouble by Understanding Git

Continuous Deployment for a Billion Dollar Order System

```rust
/// Search the file for the next non-sparse file section. Returns the
/// start and end of the data segment.
// FIXME: Should work on *BSD too?
pub fn next_sparse_segments(infd: &File, outfd: &File, pos: u64) -> Result<(u64, u64)> {
    let next_data = match lseek(infd, SeekFrom::Data(pos as i64))? {
        SeekOff::Offset(off) => off,
        SeekOff::EOF => infd.metadata()?.len(),
    };
    let next_hole = match lseek(infd, SeekFrom::Hole(next_data as i64))? {
        SeekOff::Offset(off) => off,
        SeekOff::EOF => infd.metadata()?.len(),
    };

    lseek(infd, SeekFrom::Start(next_data))?; // FIXME: EOF (but shouldn't happen)
    lseek(outfd, SeekFrom::Start(next_data))?;

    Ok((next_data, next_hole))
}
```

xcp: experimental accelerated & pluggable unix copy

https://haltcondition.net

@tarkasteve@hachyderm.io

@tarka.haltcondition.net

Speaker notes

But before I start it's traditional to say who I am. You probably don't care, but tradition must be followed.
I'm Steve Smith, I've been working as a developer and sysadmin since I was 16, most recently at Atlassian, where I spent a lot of my time overlapping development and operations before devops was a thing, and building tools to help teams deploy Atlassian technologies to the cloud.
As part of this I spent a lot of time giving talks to help other developers adopt emerging technologies like continuous deployment and git.

I'm also the author of xcp, an experimental unix cp clone that leverages Rust and modern hardware.

How I Learned to Stop Worrying and Love the Borrow Checker

https://haltcondition.net
@tarkasteve@hachyderm.io
@tarka.haltcondition.net

Speaker notes

This is the fundamental driver for this talk; when we start learning and using Rust we immediately start running into the borrow-checker and become frustrated.
However what most people come to realise is that Rust is merely saving you from yourself; forcing you to abide by the best practices we all claim we stick to in other languages but somehow keep falling through the cracks.

It’s worth noting here that I’m using “Borrow Checker” as a catch-all for Rust’s seemingly annoying rules. Most are the borrow-checker in some way, but this can also include some type-based rules, which are closely related.

Recap: Mutability Rules

```
1 fn mult_mut_example() {
2     let mut x = 1;
3     let mx1 = &mut x;
4     let mx2 = &mut x;
5
6     *mx1 = 2;
7     println!("X = {}", mx2);
8 }
```

```
1 |  let mx1 = &mut x;
2 |            ------ first mutable borrow occurs here
3 |  let mx2 = &mut x;
4 |            ^^^^^^ second mutable borrow occurs here
```

Speaker notes

By default all variables in rust are immutable.
However Rust does allow you to add mutability with the 'mut' keyword.
But there are strict rules around this.
There can only be one mutable reference.

Recap: Mutability Rules

```
1  fn mut_immut_example() {
2      let mut x = 1;
3      let rox = &x;
4      let mx1 = &mut x;
5
6      *mx1 = 2;
7      println!("X = {}", rox);
8  }
```

```
1  | let rox = &x;
2  |           ^^ immutable borrow occurs here
3  | let mx1 = &mut x;
4  |           ------ mutable borrow occurs here
5  |
6  | *mx1 = 2;
7  | -------- mutable borrow later used here
```

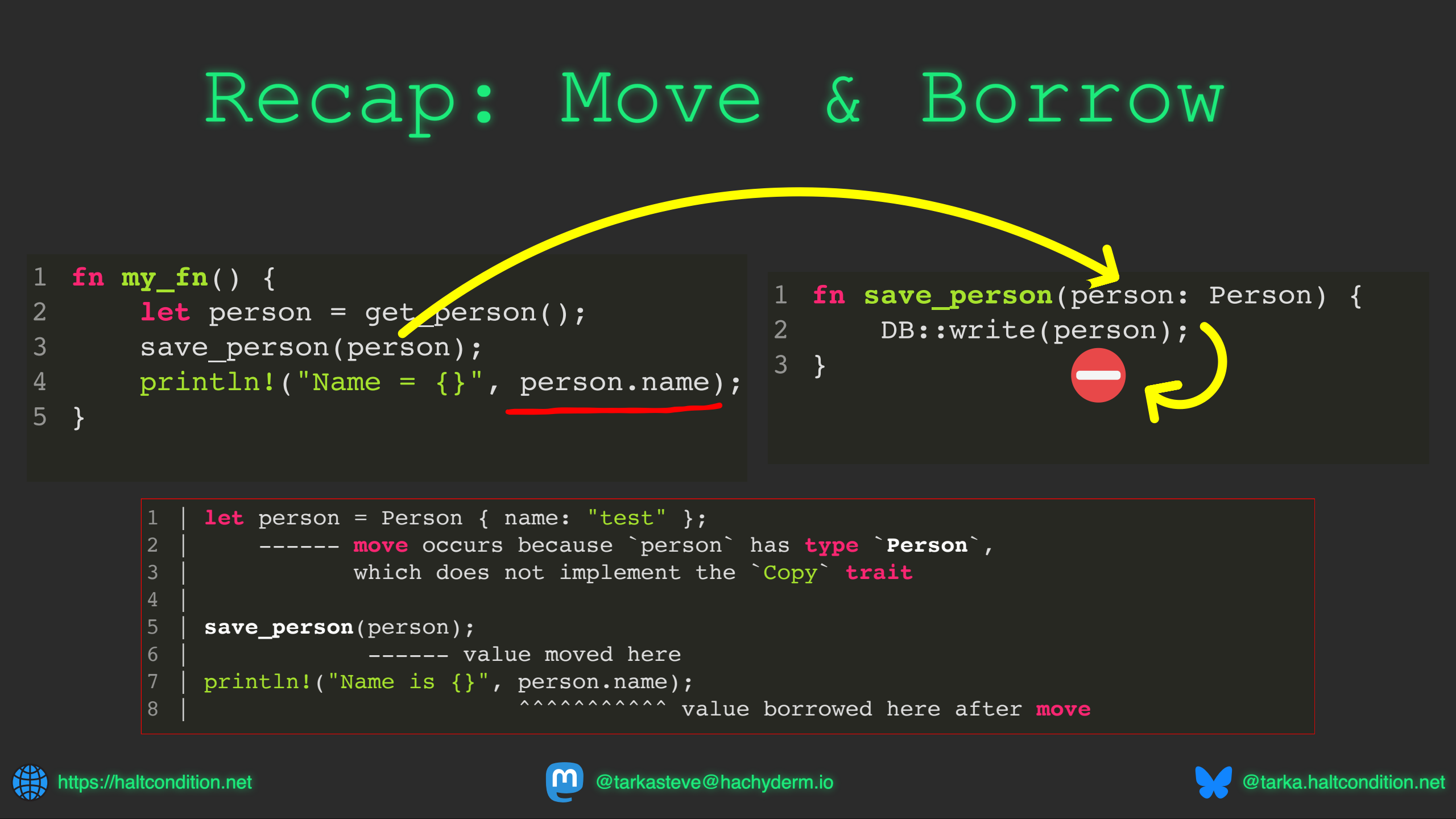
Speaker notes

However, once you take a mutable reference you can't take any non-mutable ones either.
Why? Race conditions.
Modern CPUs cache aggressively, and you can't rely on changes to variables.
If a variable is mutable only one 'view' of it is allowed.

Recap: Move & Borrow

```rust
fn my_fn() {
    let person = get_person();
    save_person(person);
    println!("Name = {}", person.name);
}
```

```rust
fn save_person(person: Person) {
    DB::write(person);
}
```



```
| let person = Person { name: "test" };
|     ------ move occurs because `person` has type `Person`,
|            which does not implement the `Copy` trait
|
| save_person(person);
|             ------ value moved here
| println!("Name is {}", person.name);
|                        ^^^^^^^^^^^^ value borrowed here after move
```

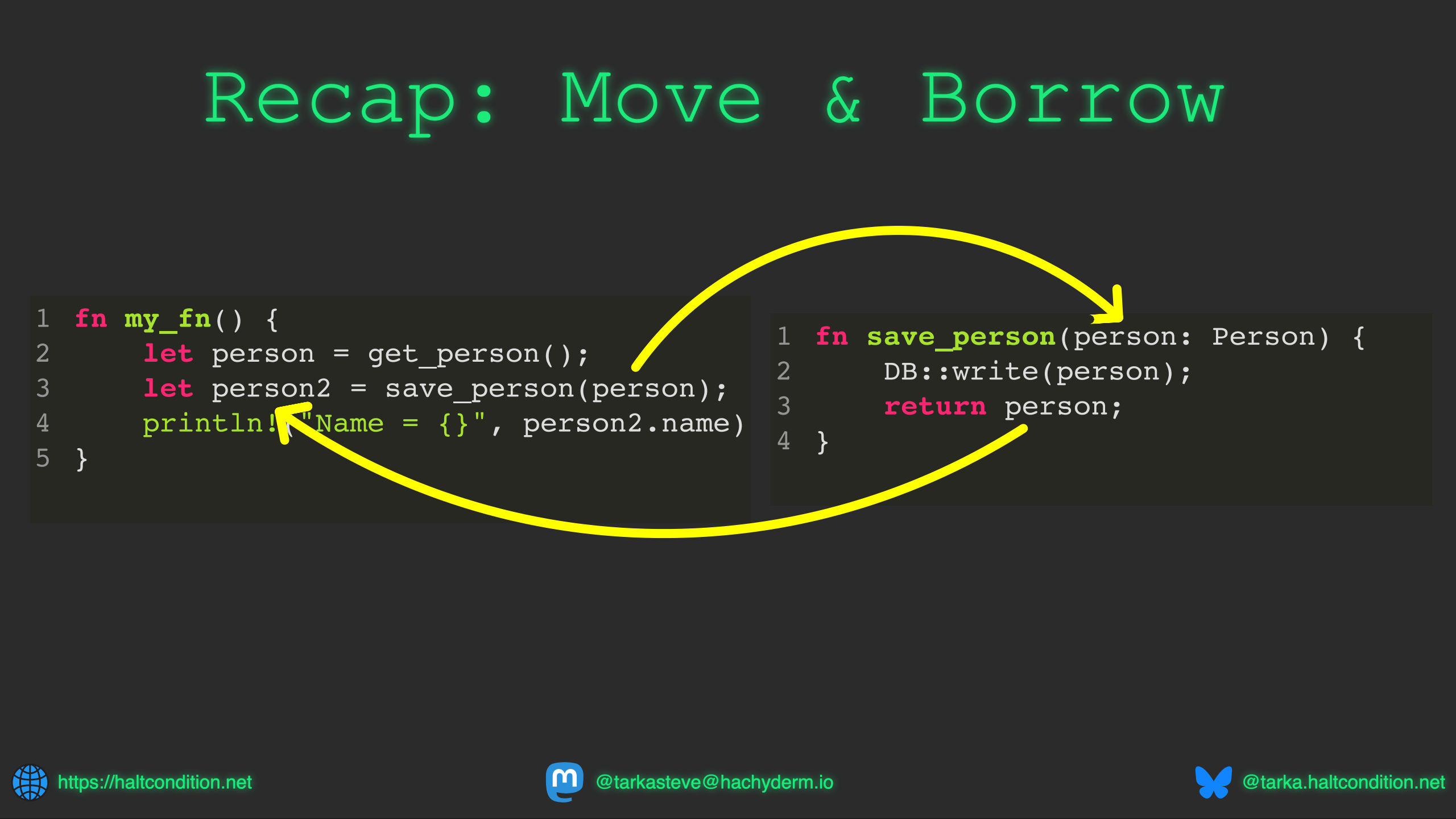
Speaker notes

In Rust everything must have an owner, and by default only one owner, although there are ways around that as we'll see soon.
This rule also works across function boundaries.
When you call a function with a parameter, that function now "owns" that parameter.
At the end of the function, the object goes out of scope. It no longer exists.

Recap: Move & Borrow

```rust
fn my_fn() {
    let person = get_person();
    let person2 = save_person(person);
    println!("Name = {}", person2.name)
}
```

```rust
fn save_person(person: Person) {
    DB::write(person);
    return person;
}
```



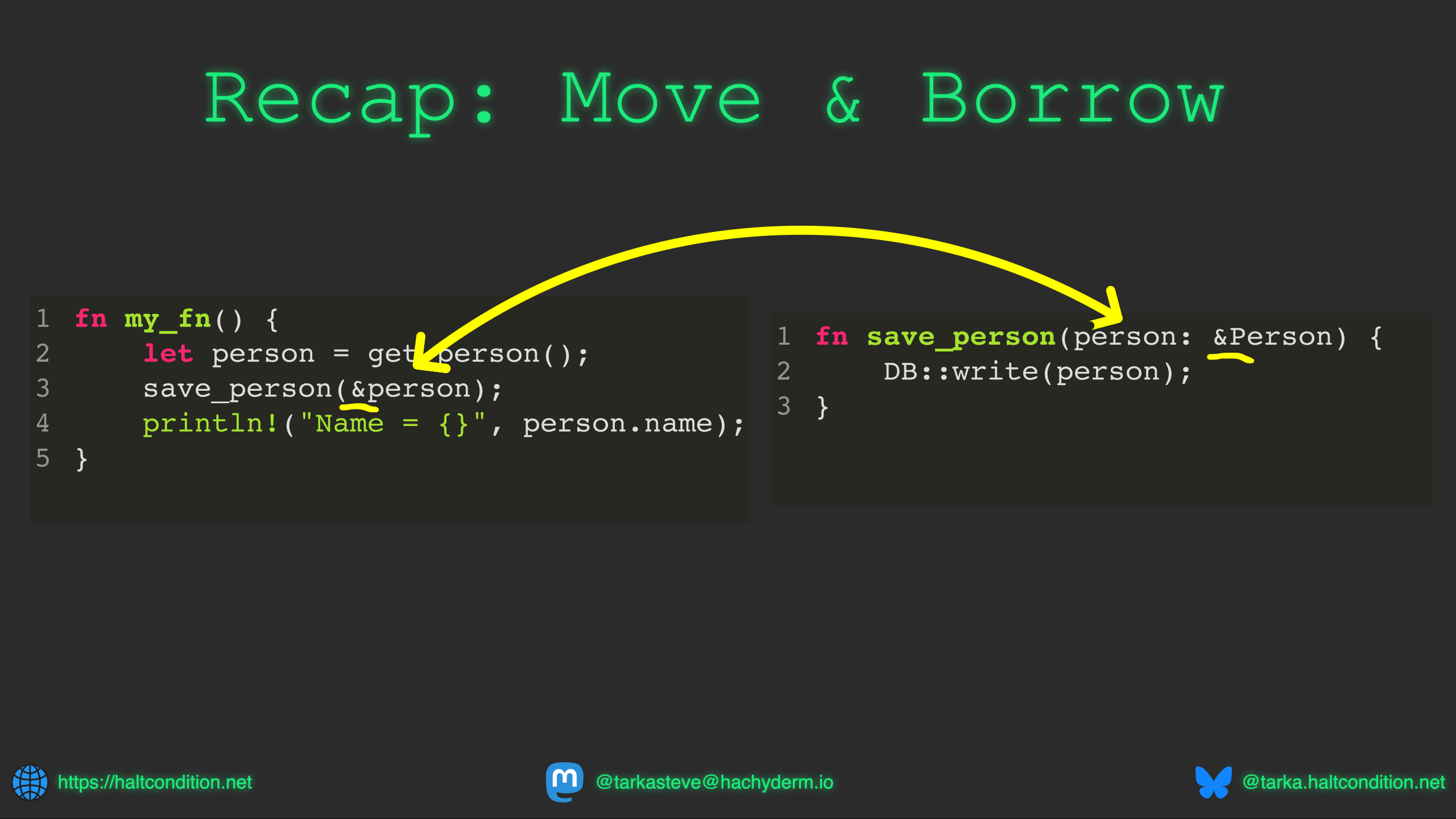
Speaker notes

But what if we want to keep using that object after the call.
One option would be to return the object, moving the ownership back to the caller.
But that's messy and inelegant. So there's a better way.

Recap: Move & Borrow

```
1 fn my_fn() {
2     let person = get_person();
3     save_person(&person);
4     println!("Name = {}", person.name);
5 }
```

```
1 fn save_person(person: &Person) {
2     DB::write(person);
3 }
```



Speaker notes

As we saw with mutability, we can 'borrow' a value temporarily.
In this scenario the 'move' is only temporary and ownership is returned to the caller.

Garbage Collection

Speaker notes

So I want to take a bit of a left-turn at this point and talk about Garbage Collection. Obviously we don’t normally think of Rust as a garbage-collected language. In fact, it’s somewhat notable for having removed GC earlier in its development.

All languages are Garbage Collected

Speaker notes

However, one of the epiphanies that Rust gave me is that not only is it garbage-collected, all languages are garbage-collected, it's just that they are enforced in different ways.

To give the most extreme example of this, take C. C is manually garbage-collected, free(). However if you don't call free() at the appropriate times, sooner or later you'll run into the OS, which will free your memory whether you like it or not. And if the OS doesn't do that you'll eventually invoke the mother of all memory-management techniques, the reset switch.

So there is a spectrum of garbage-collection systems, from manual, through simple reference counting up to sophisticated runtime systems such as the JVM…

All languages are Garbage Collected

https://haltcondition.net
@tarkasteve@hachyderm.io
@tarka.haltcondition.net

Speaker notes

To give the most extreme example of this, take C. C is manually garbage-collected, free(). However if you don’t call free() at the appropriate times, sooner or later you’ll run into the OS, which will free your memory whether you like it or not. And if the OS doesn’t do that you’ll eventually invoke the mother of all memory-management techniques, the reset switch.

So there is a spectrum of garbage-collection systems, from manual, through simple reference counting up to sophisticated runtime systems such as the JVM…

All languages are Garbage Collected

```c
void func() {
    int myNumbers[] =
        {1, 2, 3, ...};
    int num = 512;

    int *ptr1 =
        (int*)malloc(n * sizeof(int));

    int *ptr2 =
        (int*)malloc(n / 2 * sizeof(int));

    // Some stuff with ptrs

    free(ptr1);
}
```

The Borrow Checker is GC

```rust
fn my_fn() {
    let person = get_person();
    save_person(person);
    println!("Name = {}", person.name);
}
```

```rust
fn save_person(person: Person) {
    DB::write(person);
    free(person);
```

The *Compiler* is GC

Speaker notes

However Rust adds a new level; compile-time garbage collection. One of the side-effects of the borrow-checker is that every piece of data's lifecycle is closely tracked. This allows it to know when a variable or data goes out of scope and inject the appropriate cleanup there.

The Compiler is Resource Management

```
1 pub trait Drop {
2     fn drop(&mut self);
3 }
```

Speaker notes

This allows Rust to take a different paradigm towards all resource management, not just memory. There is a special trait called Drop that can be implemented for any type, including user defined types.

(If you're not familiar with traits they're sort of like interfaces in Java, but there are some special ones that hook into the compiler. We'll discuss that in a bit.)

If you implement Drop for your type, the compiler guarantees that the drop() method will be called when your data reaches the end of it's life.

Now, it's tempting to say that other languages offer this, such as destructors, finally blocks, or Go's defer feature. However Rust is interesting in that this is injected at compile time, and cannot be forgotten, and is also deterministic,. This is important; for instance, while Java allows you to define a finalize() method there is no guarantee that it will be called at all.

The Compiler is Resource Management

```
1  struct File {
2      fd: u64;
3  }
4
5  impl File {
6      fn open(path: String) -> File {
7          let fd = libc::open(path);
8          File { fd }
9      }
10 }
```

```
1  pub trait Drop {
2      fn drop(&mut self);
3  }
```

```
1  impl Drop for File {
2      fn drop(&self) {
3          libc::close(self.fd)
4      }
5  }
```

Speaker notes

This guarantee can be leveraged in interesting ways. Imagine if you want to define a simple wrapper around a unix file handle, but must ensure that no handles leak. This is can be done with a struct and implementing the Drop trait.

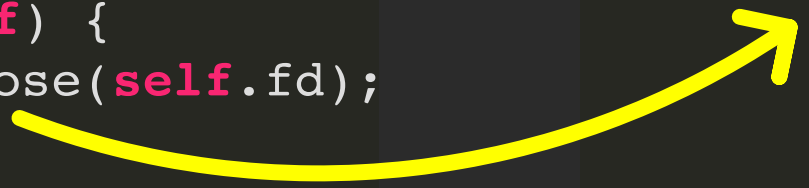
The Compiler is Resource Management

```rust
pub trait Drop {
    fn drop(&mut self);
}
```

```rust
impl Drop for File {
    fn drop(&self) {
        libc::close(self.fd);
    }
}
```

```rust
fn sum_file(path: String) -> u64 {
    let file = File::open(path);

    let sum = file.iter().sum();
    sum
} 🚫 libc::close(self.fd);
```



Speaker notes

Notice that there is no close() method. This is not necessary, as we know that the OS-level close will be called when the File object goes out of scope.

In fact, this is exactly how the File type in the Rust standard library is defined.

The Compiler is Resource Management

```
1  fn do_lots_of_stuff() -> bool {
2
3      let data = generate_data();
4
5  'file {
6          let file = File::open(path);
7          file.write(data.raw());
8      } // 🚫 File closed here
9
10     println!("SHA256 of data is {}", data.sha256());
11 }
```

Speaker notes

This leads to an odd Rust-specific idiom that feels wrong when you first use it; temporary scope blocks.

The Compiler is Resource Management

```
1 fn do_lots_of_stuff() -> bool {
2
3     let data = {
4         let file = File::open(path);
5         file.read_all()
6     }; // 🚫 File closed here
7
8     println!("SHA256 of data is {}", data.sha256());
9 }
```

Speaker notes

Blocks can also return data, so you can create small self-contained areas that clean up after themselves.

The Compiler is Resource Management

```
1  fn do_lots_of_stuff() -> bool {
2
3      let data = {
4          let file = File::open(path);
5          file.read_all()
6      }; // 🚫 File closed here
7
8      println!("SHA256 of data is {}",
9               data.sha256());
10 }
```

'file

```
1  fn do_lots_of_stuff() -> bool {
2      let file = File::open(path);
3      let data = file.read_all();
4
5
6      drop(file); // 🚫 File closed
7
8      println!("SHA256 of data is {}",
9               data.sha256());
10 }
```

'file

Speaker notes

You can also just call drop(obj). This is not generally as elegant as using scopes, but I mention it here as I'll be using it a bit later to illustrate some other concepts.

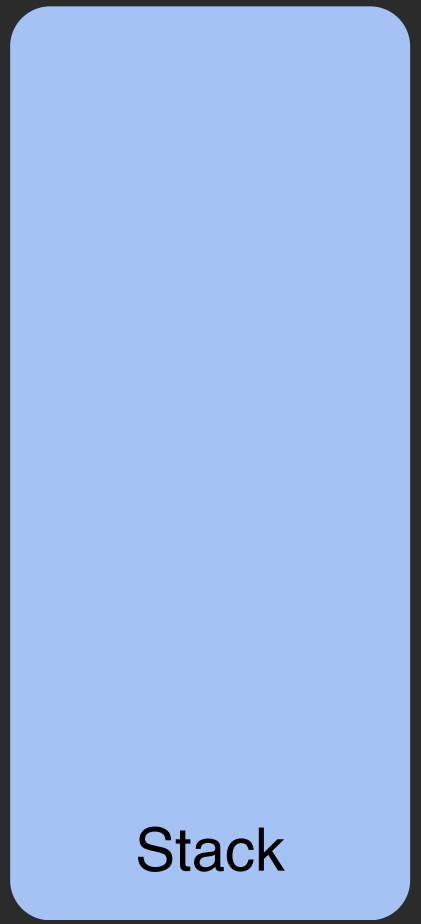
Stack & Heap

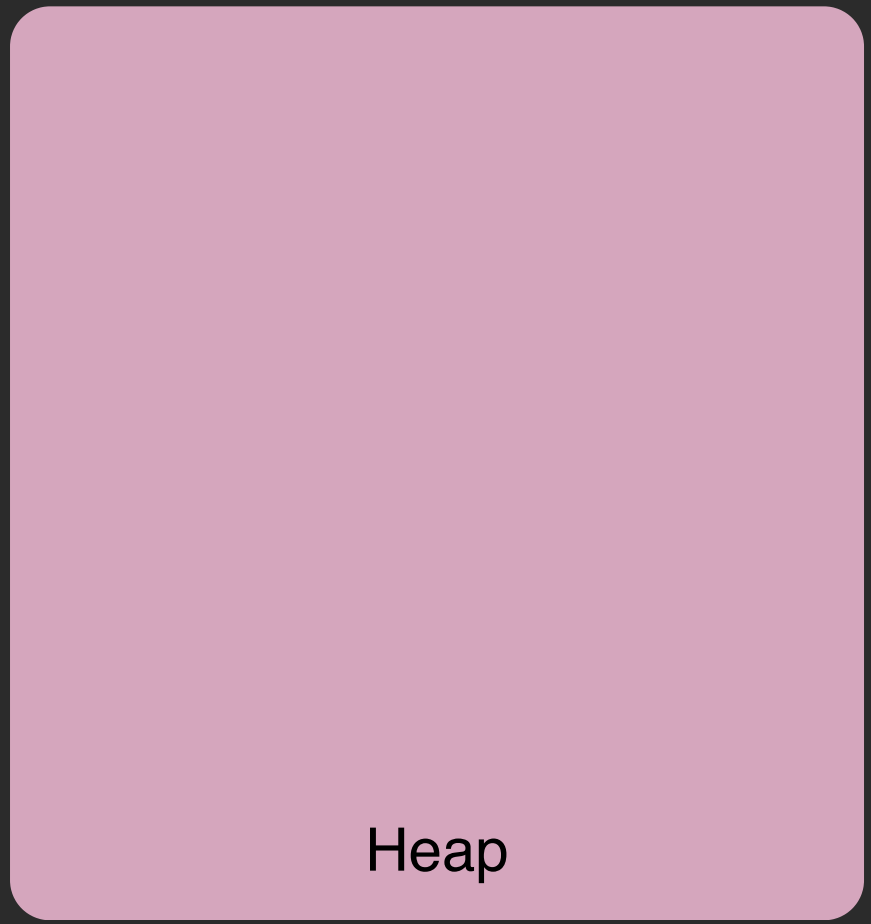
Speaker notes

OK, so the compiler injects cleanup code for us, and we can hook into that to support cleanup of other types. However sometimes we need data that have a longer lifetime than the scope of our function. This is where heap allocation comes in, although it is often hidden in Rust.

Stack & Heap

```
1
2  void func() {
3      int myNumbers[] =
4          {1, 2, 3, ...};
5
6
7
8      int num = 512;
9
10
11     int *ptr1 =
12         (int*)malloc(n * sizeof(int));
13
14     int *ptr2 =
15         (int*)malloc(n / 2 * sizeof(int));
16
17     // Some stuff with ptrs
18
19     free(ptr1);
20 }
```



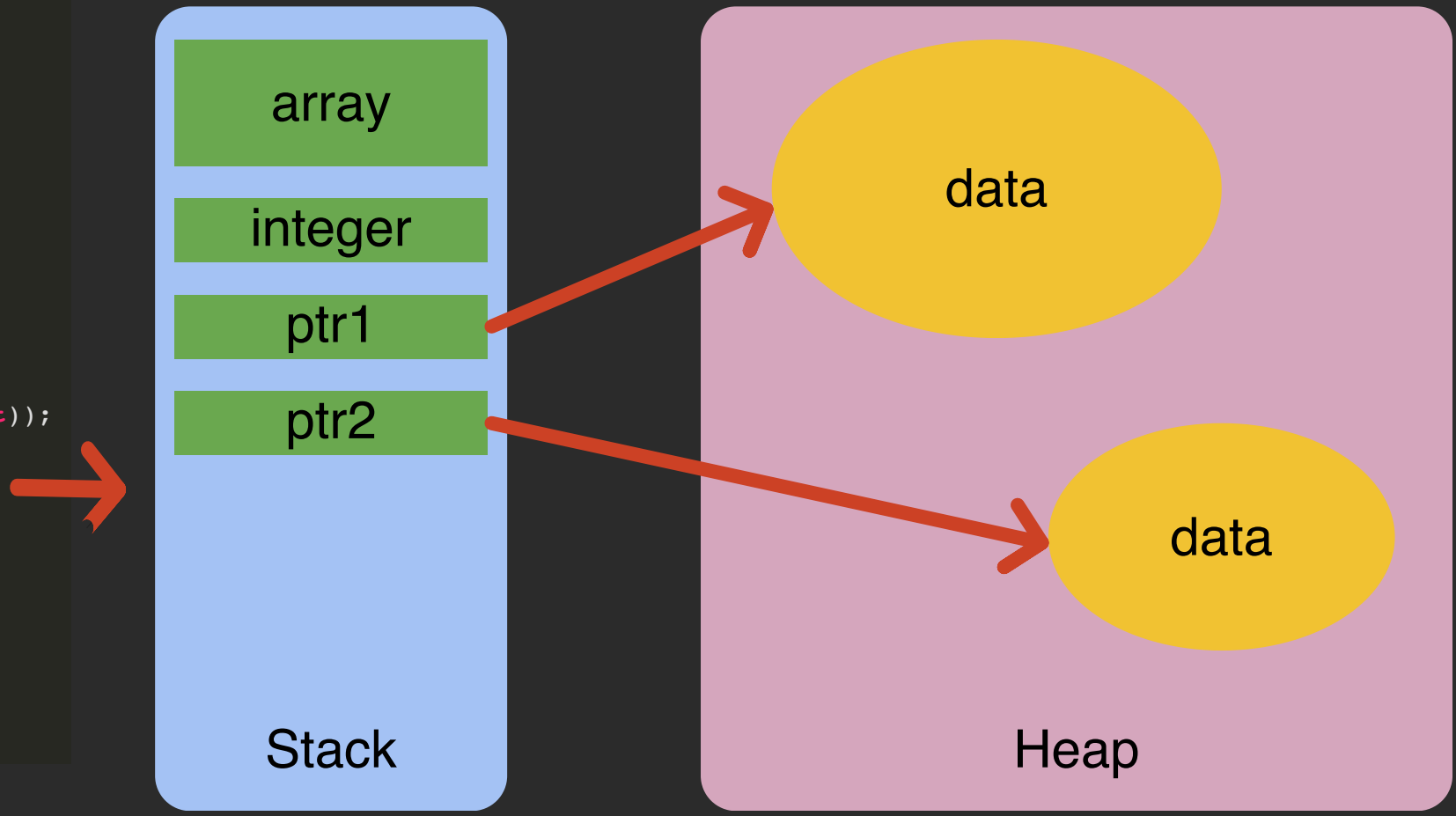


Speaker notes

So a brief, simplified and rather inaccurate recap of stack vs heap. Data allocated on the stack is local to the function being called, and is automatically reclaimed at the end of that function. However we can hook into a longer-lived pool of memory called the heap. Once allocated by the OS these chunks of heap remain live until they are explicitly released with free(). Failing to do this causes memory leaks, doing it at the wrong point causes crashes and security vulnerabilities.

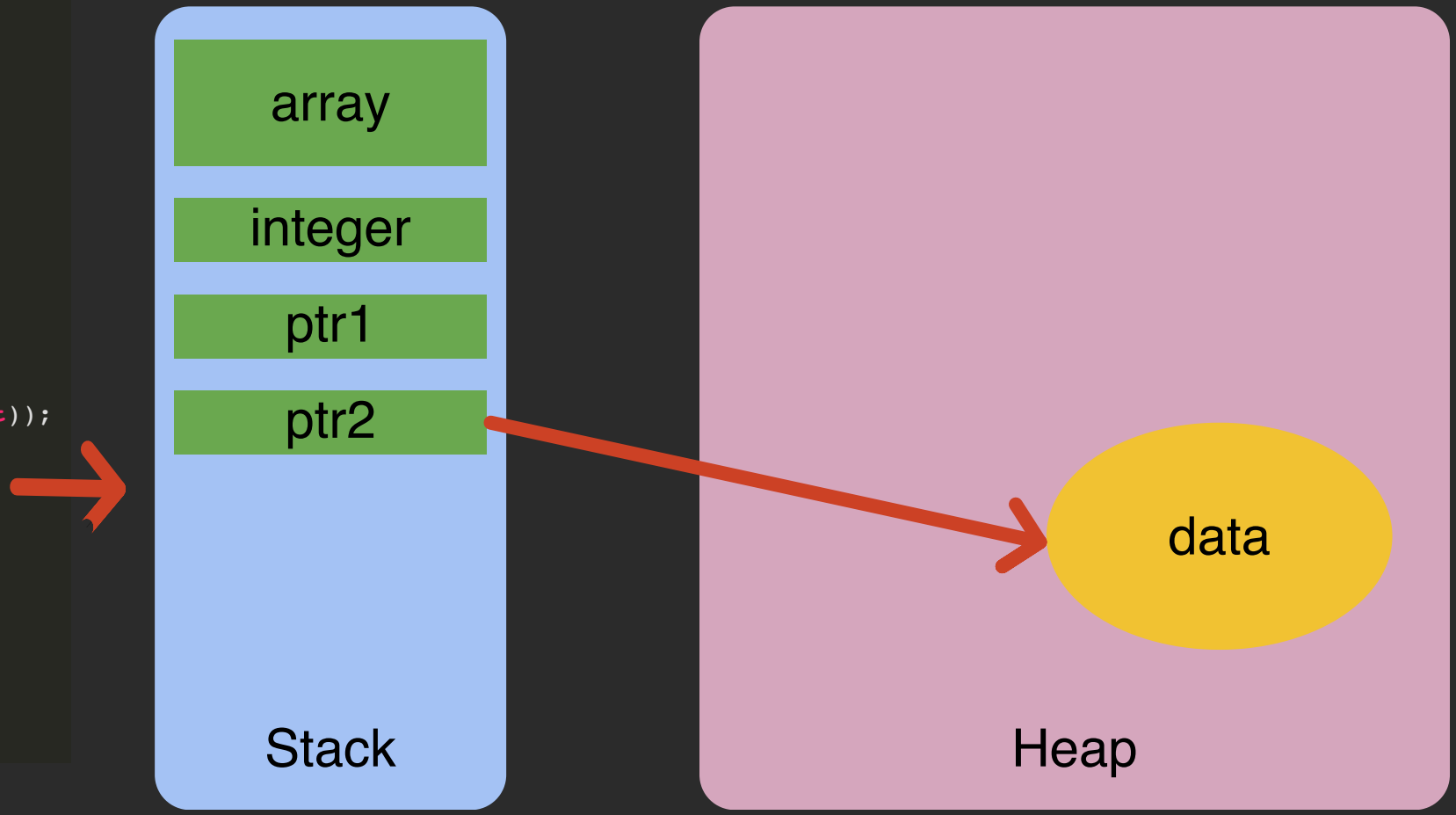
Stack & Heap

```
1
2  void func() {
3      int myNumbers[] =
4          {1, 2, 3, ...};
5
6
7
8      int num = 512;
9
10
11     int *ptr1 =
12         (int*)malloc(n * sizeof(int));
13
14     int *ptr2 =
15         (int*)malloc(n / 2 * sizeof(int));
16
17     // Some stuff with ptrs
18
19     free(ptr1);
20 }
```

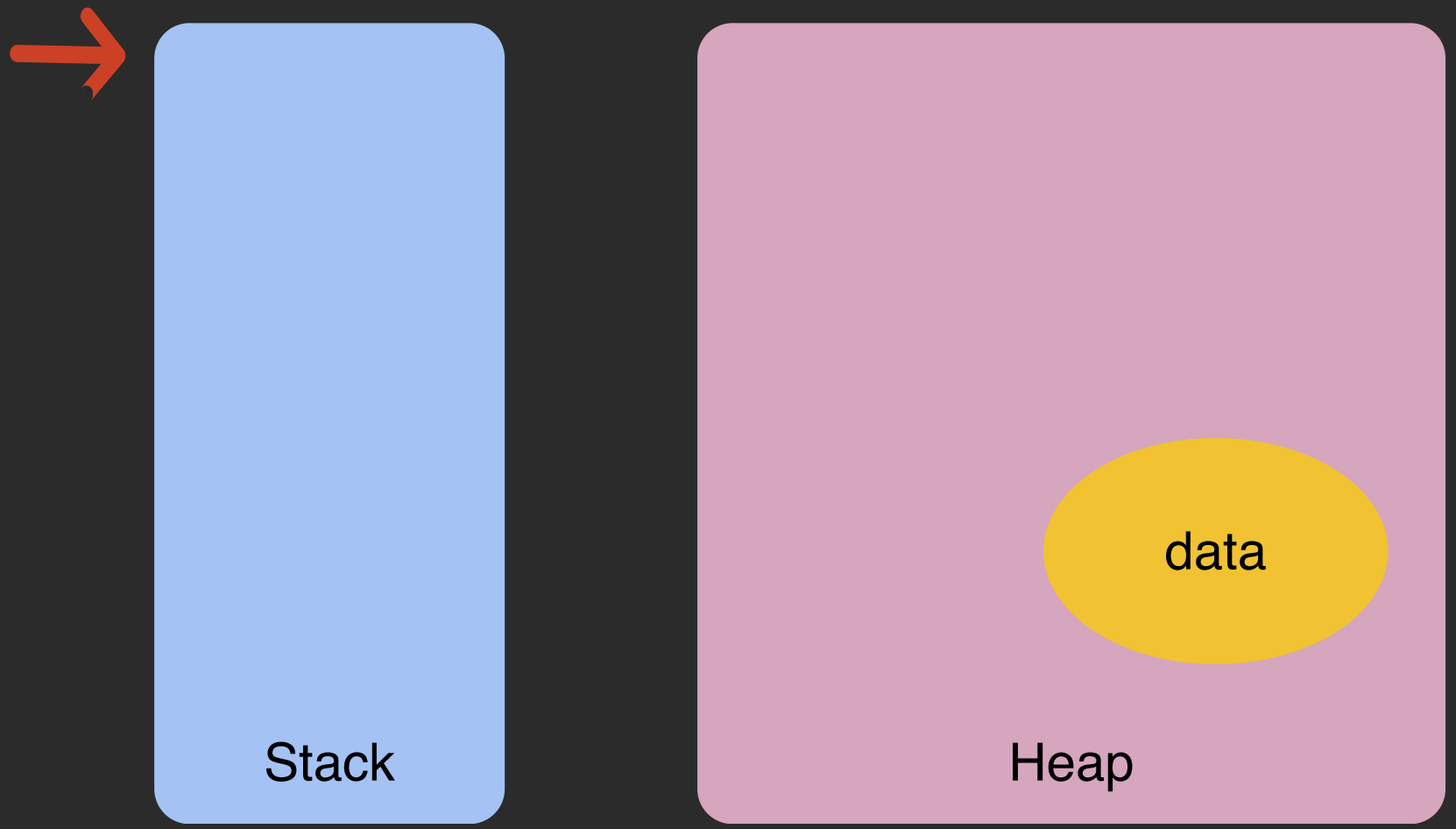


Stack & Heap

```
1
2  void func() {
3      int myNumbers[] =
4          {1, 2, 3, ...};
5
6
7
8      int num = 512;
9
10
11     int *ptr1 =
12         (int*)malloc(n * sizeof(int));
13
14     int *ptr2 =
15         (int*)malloc(n / 2 * sizeof(int));
16
17     // Some stuff with ptrs
18
19     free(ptr1);
20 }
```

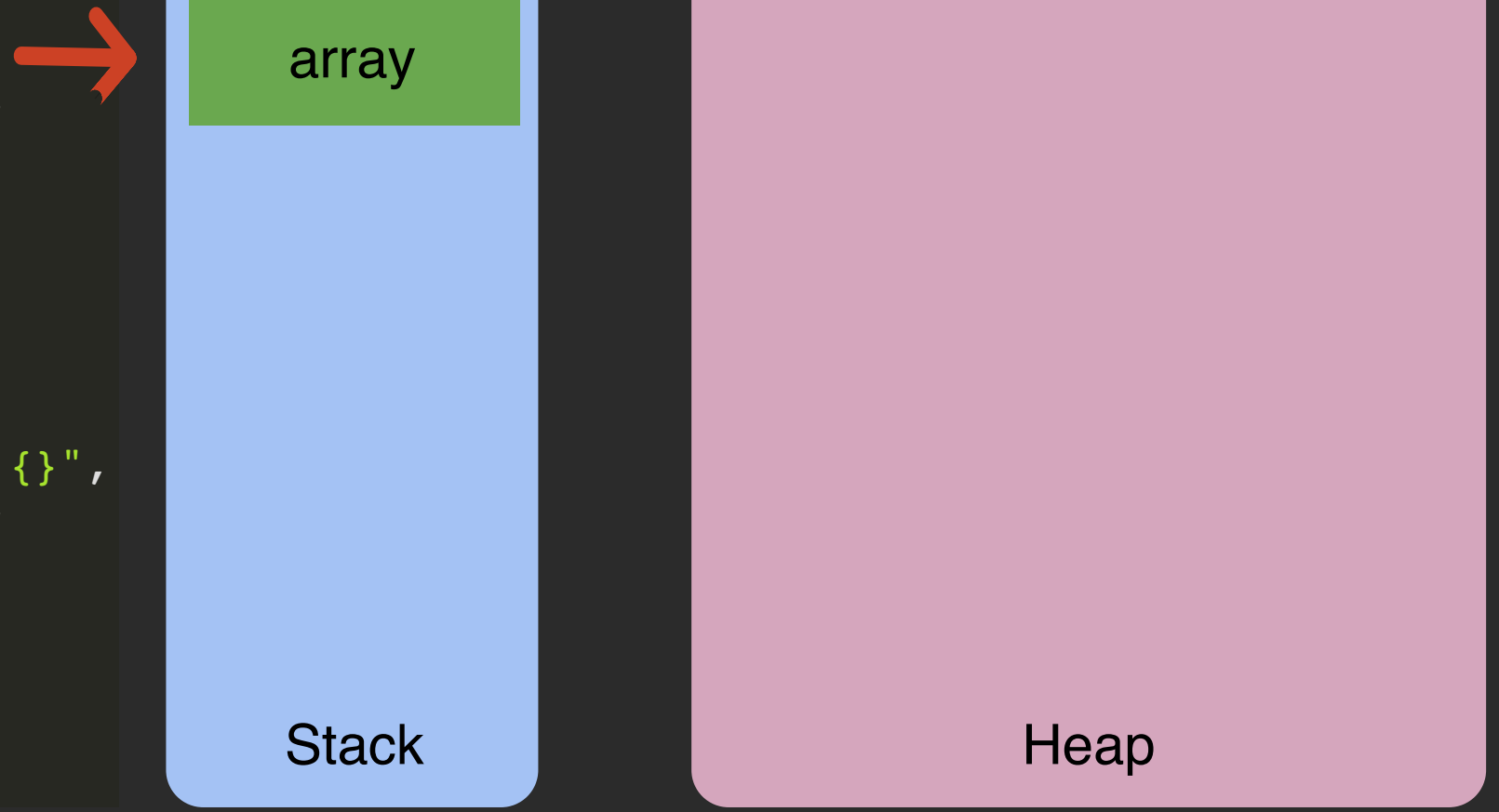


Stack & Heap



Heap in Rust

```
1  fn do_boxed() {
2      let array =
3          [1, 2, 3, 4, ...];
4
5
6      let boxed =
7          Box::new(array);
8
9
10     println!("Box size is {}",
11              boxed.len());
12
13 }
```



Speaker notes

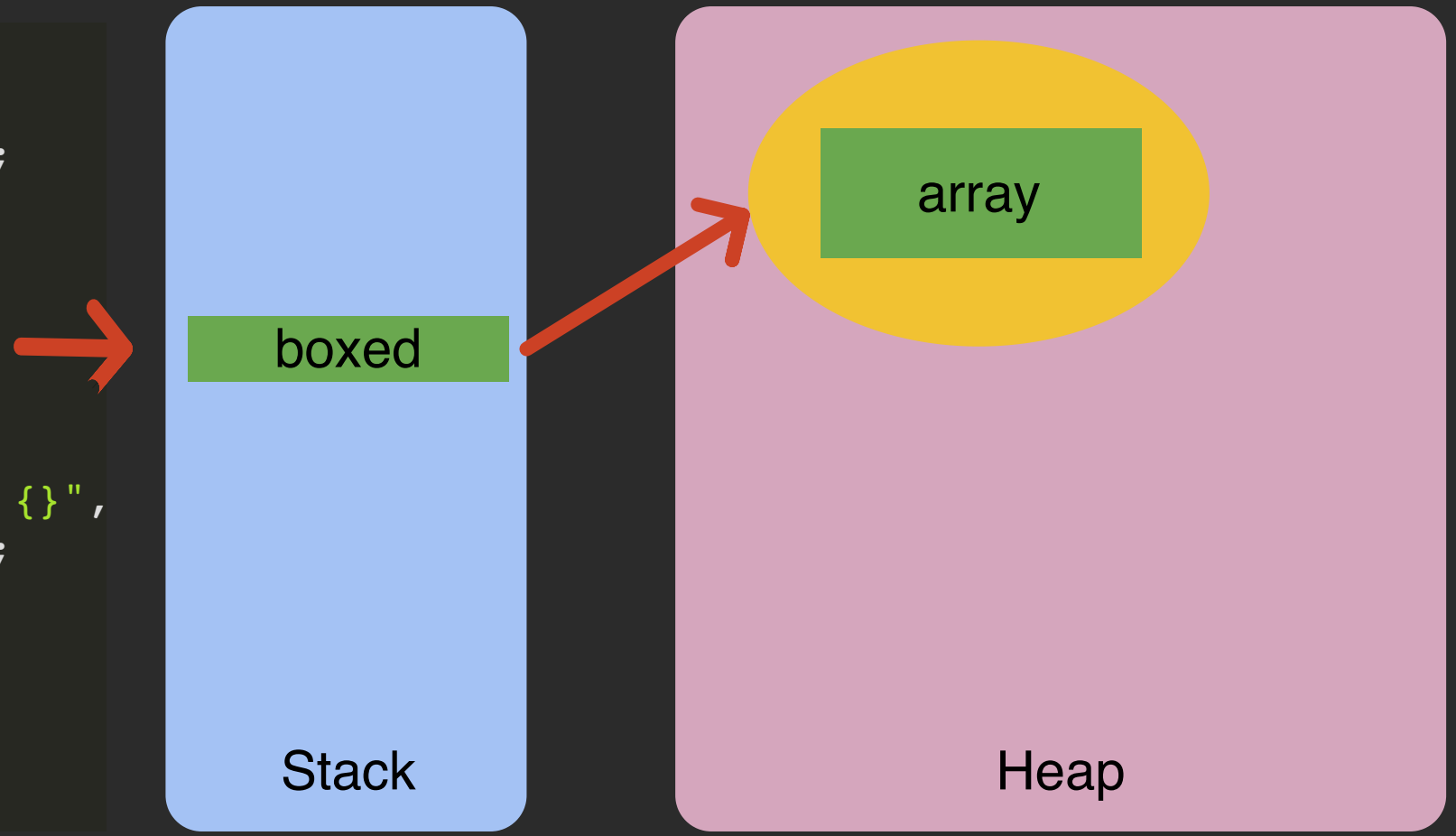
Rust has its own idiom for allocating and freeing help; Box.

Worst name ever.

When you call Box::new() you move an object from the stack to the heap and are assigned a handle.

Heap in Rust

```
1  fn do_boxed() {
2      let array =
3          [1, 2, 3, 4, ...];
4
5
6      let boxed =
7          Box::new(array);
8
9
10     println!("Box size is {}",
11              boxed.len());
12
13 }
```



Speaker notes

Rust has its own idiom for allocating and freeing help; Box.

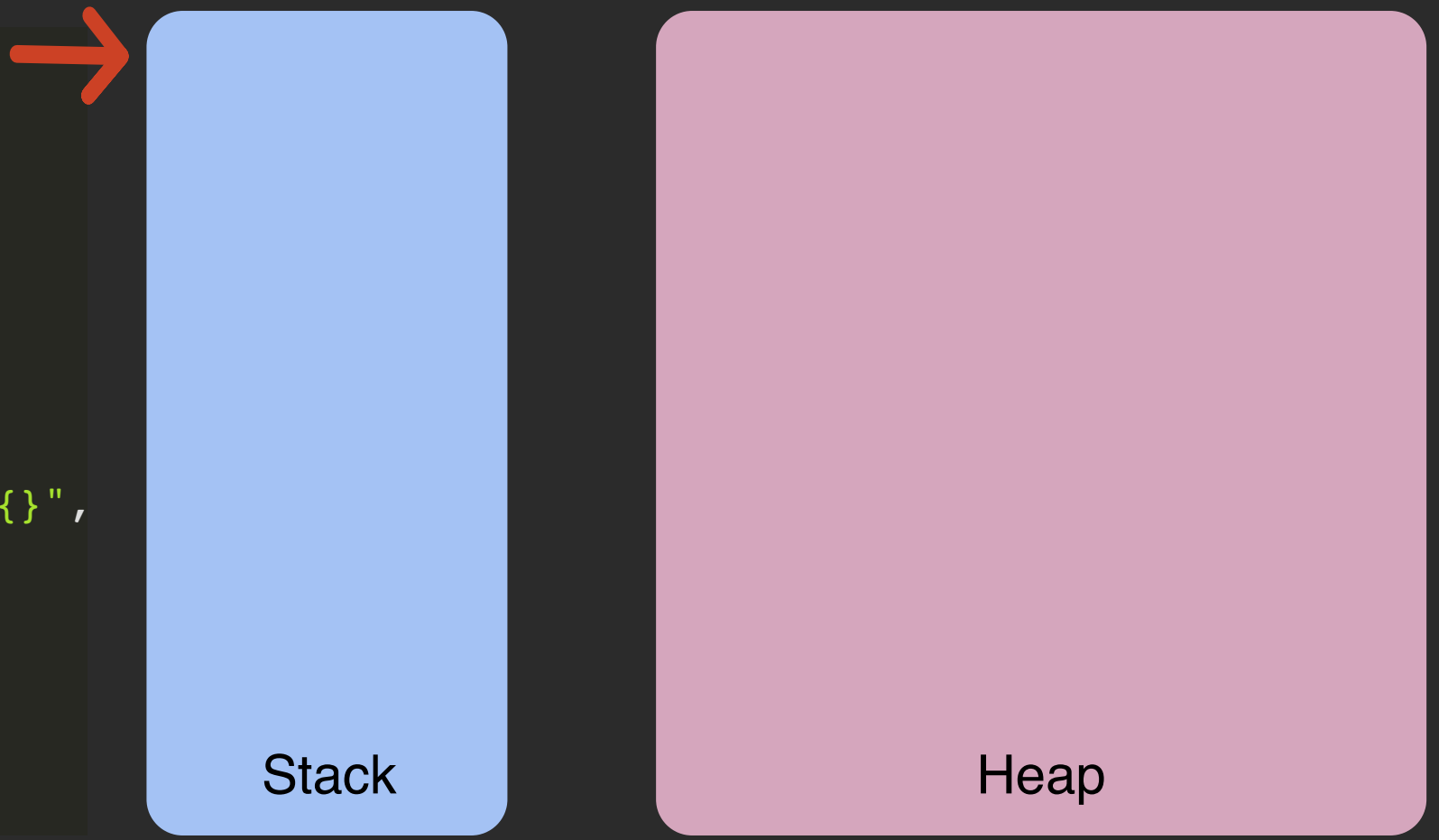
Worst name ever.

When you call Box::new() you move an object from the stack to the heap and are assigned a handle.

Heap in Rust

```
1  fn do_boxed() {
2      let array =
3          [1, 2, 3, 4, ...];
4
5
6      let boxed =
7          Box::new(array);
8
9
10     println!("Box size is {}",
11              boxed.len());
12 }
```

'boxed



Heap GC in Rust

```rust
impl<T> Box<T> {
    /// Allocates memory on the heap
    /// and then places `x` into it.
    pub fn new(x: T) -> Self {
        #[rustc_box]
        // Compiler inserts malloc()
        // magic here.
    }
}

impl Drop for Box<T> {
    fn drop(&mut self) {
        self.ptr.deallocate();
    }
}
```

```rust
fn my_fn() {
    let data = ...;
    let boxed = Box::new(data);

    // Do some stuff

} 🚫 deallocate();
```

Speaker notes

Box is interesting in that it is one of the few places where deep magic happens that isn’t generally accessible. In particular the #[rustc_box] attribute is marker for the compile to inject memory allocation code. However the rest of Box is pretty much what you’d expect, especially the Drop implementation, which calls the deallocator.

Box in std

Vec<> and vec![]

String

Rc<>

Arc<>

Speaker notes

Box is used extensively inside the standard library, not least in Vec and String. Basically if something needs to be dynamically sized then it needs to go on the heap.

Sharing

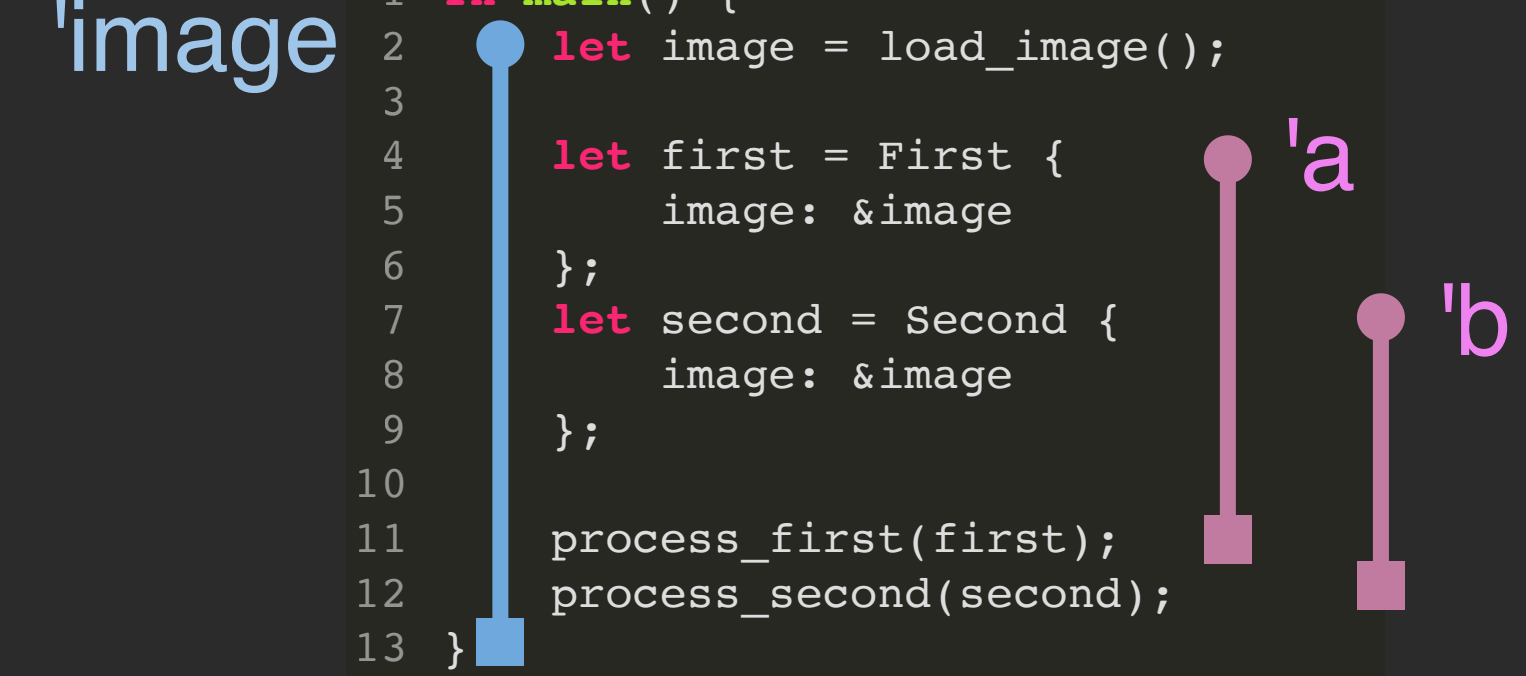
Speaker notes

You may have noticed that just moving data onto the heap doesn’t actually change how long it lives; because it’s owned by Box and will be deallocated when the box goes out of scope. This is still useful, but sometimes we want to access this data from multiple places.

Sharing & Lifetimes

```rust
struct First<'a> {
    image: &'a Vec<u8>,
    // Other stuff
}

struct Second<'b> {
    image: &'b Vec<u8>,
    // Different stuff
}
```



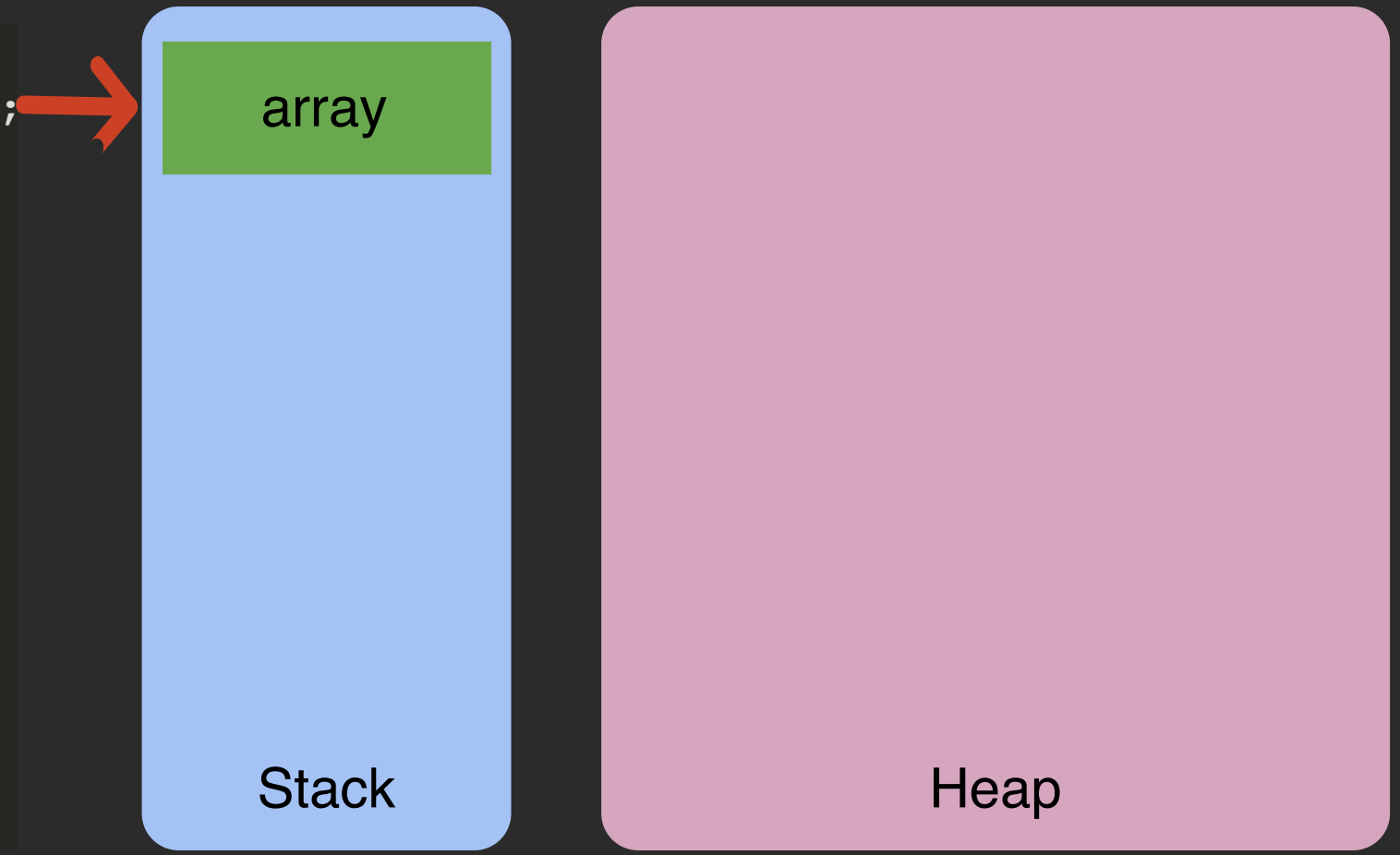
Speaker notes

The most Rust-like method of doing this is using references. We can take multiple immutable references to an object and share them as long as the underlying object remains alive. However, this is one of the few places (in my experience) where you need to explicitly express lifetimes unless you are doing something advanced.

In this example, we need to specify how long the reference will live in the second struct. Basically we're just telling the compiler "I promise to not create a reference to anything that lives shorter than this struct". And no, you can't use this to trick the compiler, it still checks.

Rc<>

```
1
2 let array = [1, 2, 3, 4, ...];
3
4
5
6
7
8
9
10
11
```



Speaker notes

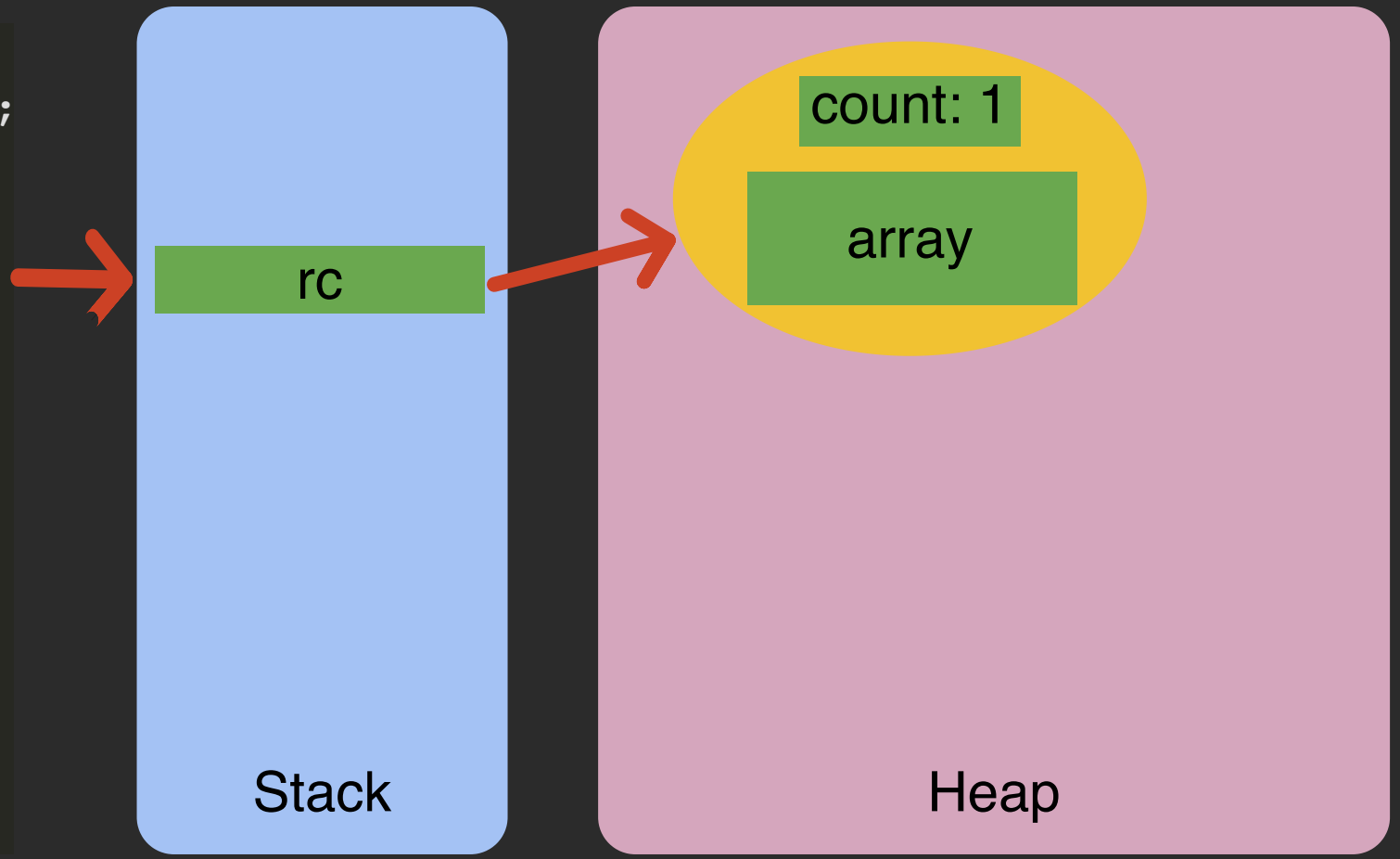
These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

Rc<>

```
1
2  let array = [1, 2, 3, 4, ...];
3
4
5  let rc = Rc::new(array);
6
7
8
9
10
11
```



Speaker notes

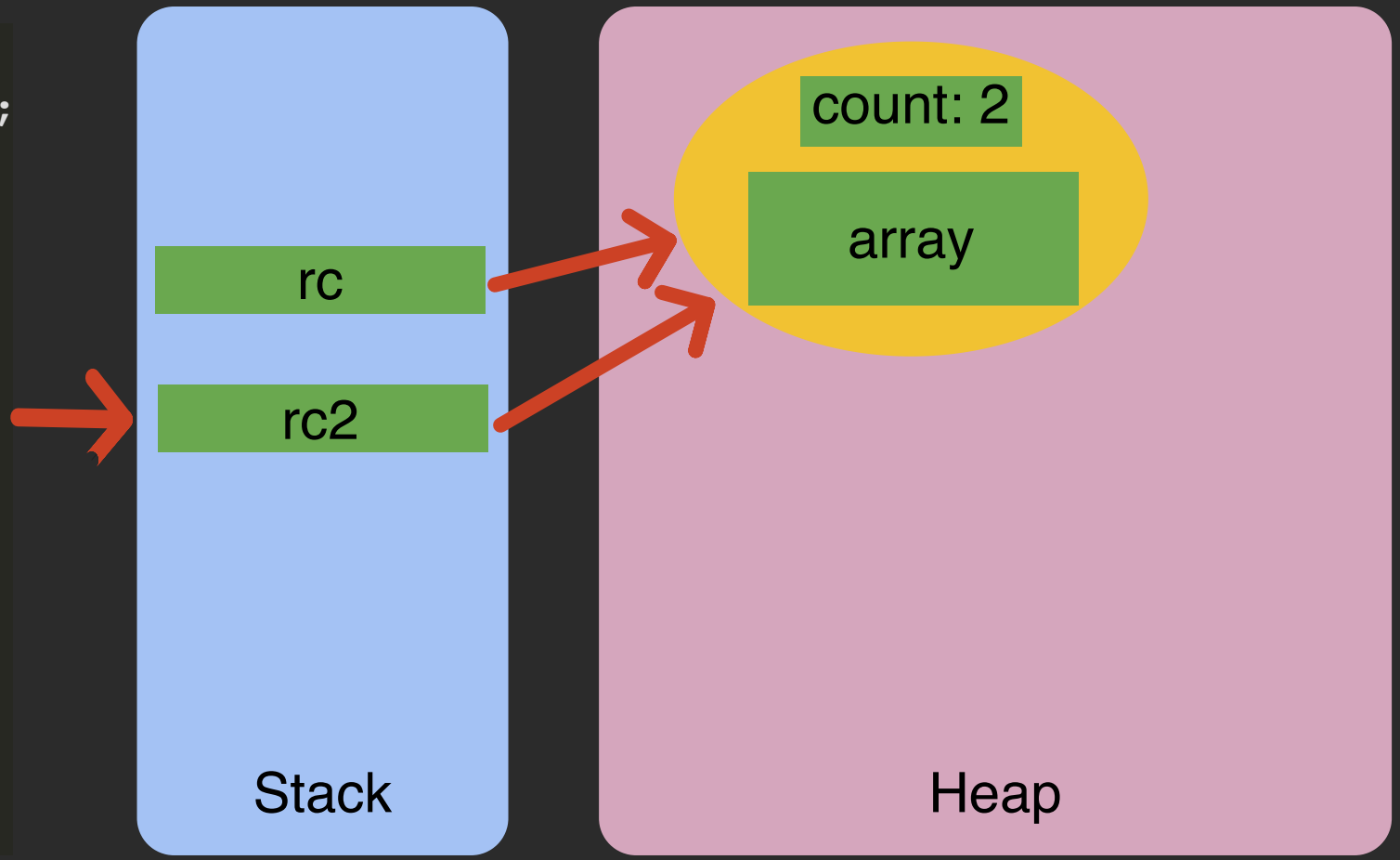
These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

Rc<>

```
1
2  let array = [1, 2, 3, 4, ...];
3
4
5  let rc = Rc::new(array);
6
7
8  let rc2 = rc.clone();
9
10
11
```



Speaker notes

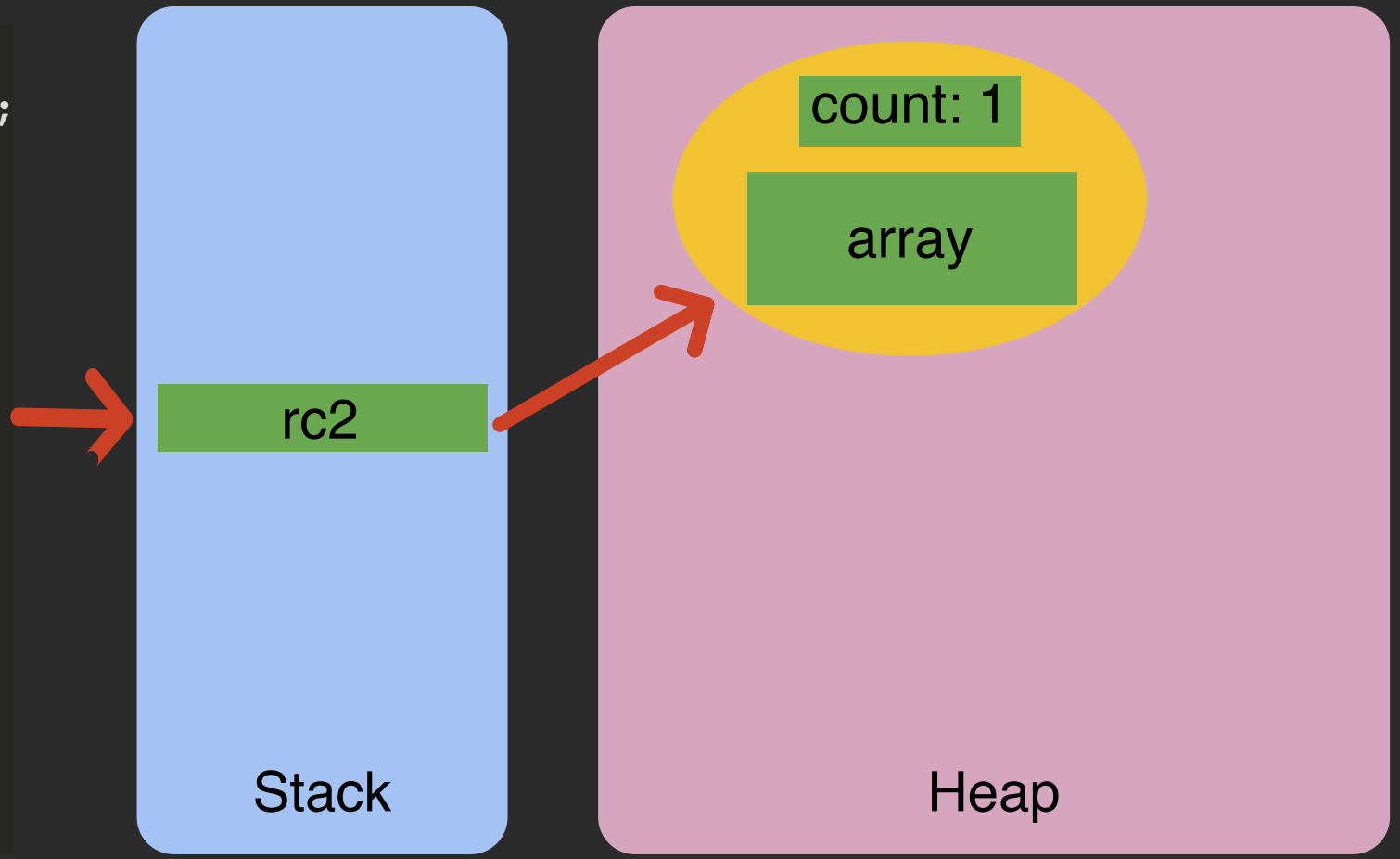
These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

Rc<>

```
1
2  let array = [1, 2, 3, 4, ...];
3
4
5  let rc = Rc::new(array);
6
7
8  let rc2 = rc.clone();
9
10 drop(rc);
11
```



Speaker notes

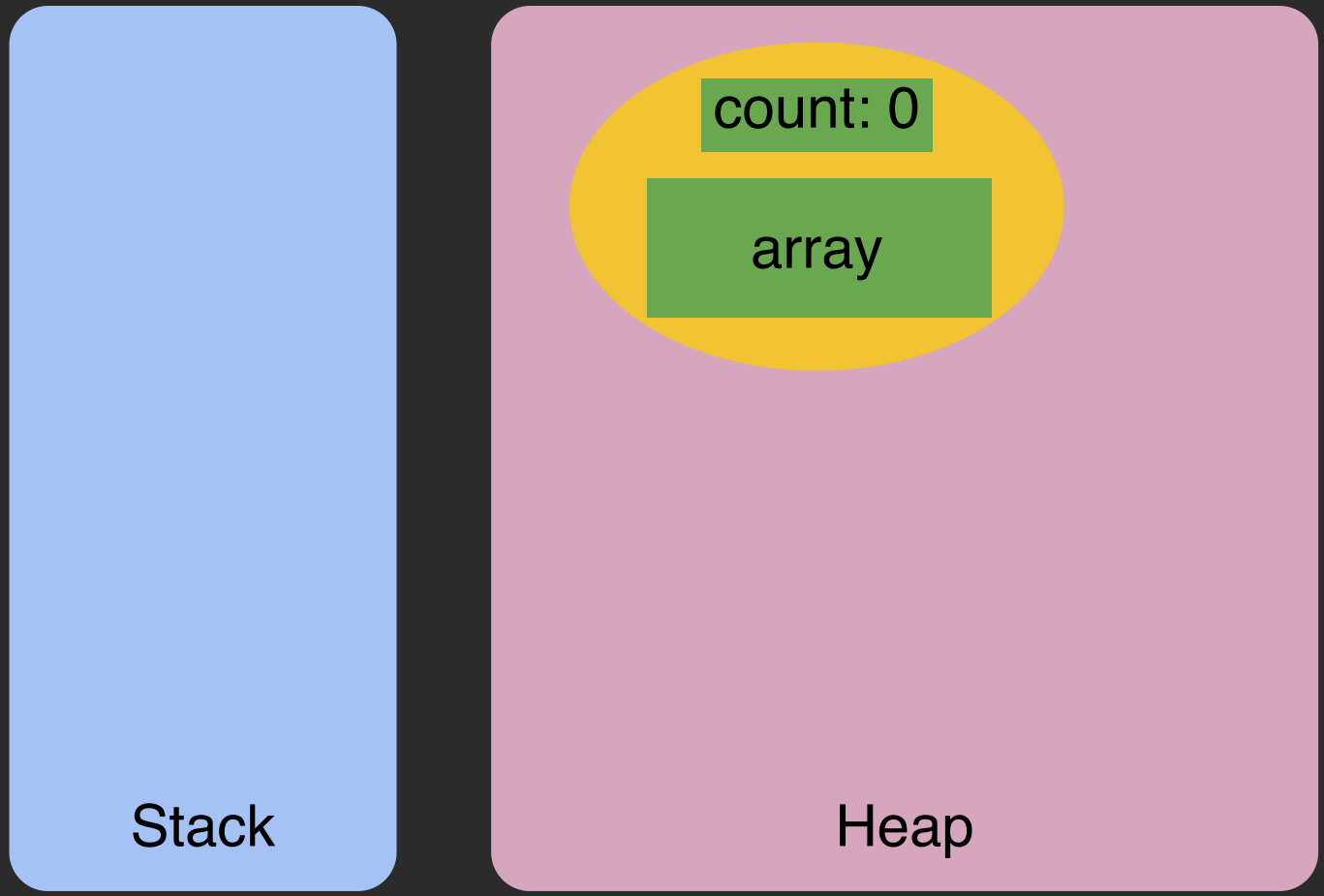
These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

Rc<>

```
1
2  let array = [1, 2, 3, 4, ...];
3
4
5  let rc = Rc::new(array);
6
7
8  let rc2 = rc.clone();
9
10 drop(rc);
11 drop(rc2);
```



Speaker notes

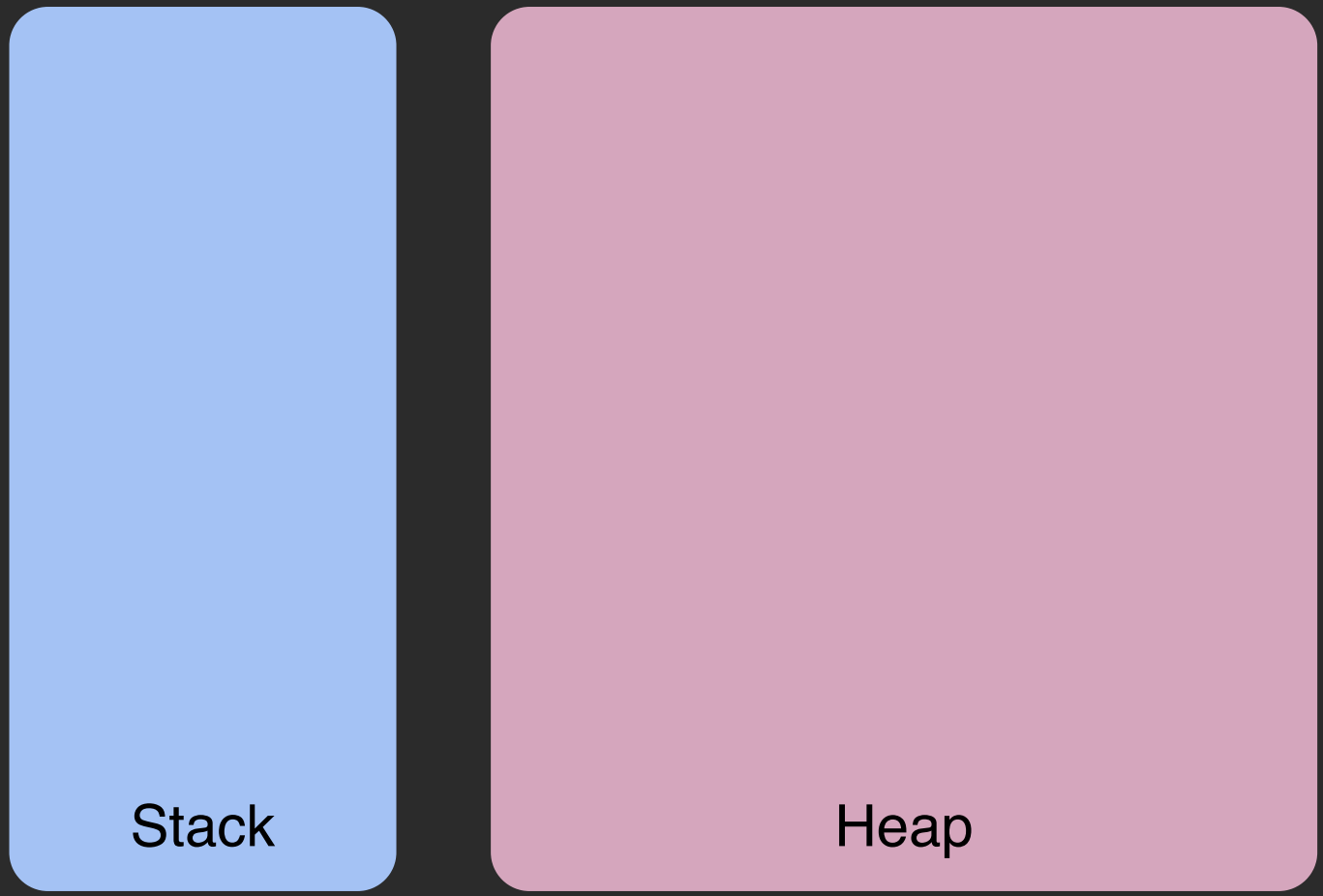
These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

Rc<>

```
1
2  let array = [1, 2, 3, 4, ...];
3
4
5  let rc = Rc::new(array);
6
7
8  let rc2 = rc.clone();
9
10 drop(rc);
11 drop(rc2);
```



Speaker notes

These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

Rc<> vs Lifetimes

```rust
struct First {
    image: Rc<Vec<u8>>,
}

struct Second {
    image: Rc<Vec<u8>>,
}
```

```rust
let image = load_image();

let img_rc = Rc::new(image);

let ms1 = First {
    image: img_rc.clone()
};
let ms2 = Second {
    image: img_rc.clone()
};

drop(img_rc);

// First & Second carry on existing
```

Speaker notes

These sort of lifetime references are the best method of sharing data, as all the checks regarding allocation and deallocation can be performed at compile time. However with some more complex data-structures the management of lifetimes can get complicated; a common case is when building a graph with cyclic relationships. In this case we need some help.

Rc stands for Reference Counter, and is a utility in the standard library that holds data for us, handing out references to that data and keeping track of them. We won't go too deep into it right now, but given what we've learned so far it's not hard to imagine how it would be implemented using boxes and drop(). One important feature it has is the ability to keep both strong and weak references, allowing cyclic data structures.

Rc<> is interesting in the context of garbage collection, in that it moves the GC from compile time to runtime, while still leveraging the compile-time constraints.

But generally speaking you should only use Rc if you know exactly why you need it; leaving clean-up to the compiler is more efficient and Rust-like.

Threads

Speaker notes

Everything we have looked at so far has implicitly single-threaded. But one of the promises of Rust is fearless concurrency, so we should look at that. Understanding this is also necessary to understanding Async, at least at the moment.

Threads

```rust
use std::thread;

struct Config {
    mangos: bool,
    avatar: Vec<u8>
}

let config = Config {
    mangos: true,
    avatar: load_avatar(),
};
```

Speaker notes

OK, so let’s look at simple real-world example. Say we have a multi-threaded application that loads its config at startup and then shares this with the worker threads. We’ll also assume that for the purposes of illustration that it contains a large image, so we don’t want to copy it.

Threads

'config

```
1  fn main() {
2      let config = Config {
3          mangos: true,
4          avatar: load_avatar(),
5      };
6      let ref1 = &config;
7
8      let backend_thread = thread::spawn(move || backend(ref1) );
9
10     backend_thread.join().unwrap();
11 }
```

'?

```
error[E0597]: `config` does not live long enough
  |
  | let config = Config {
  |     ------ binding `config` declared here
...
  | let ref1 = &config;
  |            ^^^^^^^ borrowed value does not live long enough
```

Speaker notes

OK, we know the most efficient method of sharing is to use a reference.
Because the config is read on startup and never modified, we should be able to share it with the threads safely, right?

However even though we join the thread, effectively waiting until they are shutdown before ending the program the compiler has no way of guaranteeing that this will occur properly.
Threads can potentially live on after the main program has finished.

Threads & Rc<>

```rust
let config = Config {
    mangos: true,
    avatar: load_avatar(),
};
let rc = Rc::new(config);
let rc2 = rc.clone();

let backend_thread = thread::spawn(move || backend(rc2) );
```

```
| let backend_thread = thread::spawn(|| backend(rc) );
|                      ------------- --^^^^^^^^^^^^^^^^^^
|                      |             |
|                      |             `Rc<Config>` cannot be sent between threads safely
|                      required by a bound introduced by this call
|
= help: the trait `std::marker::Send` is not implemented for `Rc<Config>`
```

Speaker notes

We've already looked at a tool to help with this. If we move the config on the heap we can ensure it lives as long as necessary. Rc<> moves data to the heap with Box, and also allows us to a low-cost sharing mechanism that will also clean up after us.

OK, that didn't work. The underlying reason is that Rc is not thread-safe; it increments and decrements counters, but doing that across multiple threads creates race-conditions. But the compiler expresses this in a different way; with Send, Sync, and 'static.

Traits & Markers

Speaker notes

Traits are most commonly explained as being a method of specifying an API and allowing types to opt into them.

Traits as Constraint Marker

```rust
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}
```

Speaker notes

If we look at the definition of the thread spawn() function, we can see that there are constraints on the type of functions that can be called and what they can return.
FnOnce() is the formal type of our Rust closures, and T is the return type.
As see can see they must be Send and static.
Send means that they must be able to be transferred between threads safely; in practice this means non unprotected mutable data.
We'll come back to static in a minute.

Send + Sync

```rust
pub unsafe auto trait Send {
    // empty
}
```

Data is Send if it can be sent between threads safely.

This means No Unsafe Mutable.

```rust
pub unsafe auto trait Sync {
    // Empty
}
```

Data is Sync if it can be shared between threads safely.

It promises that data is Mutable Safely. i.e. Mutexes & Atomics.

Speaker notes

Although it’s not required here you’ll often see Send used in conjunction with Sync. Sync is the next level, as states that access to your type will be synchronised between threads; usually this means using mutexes.

Send + Sync

```rust
pub unsafe auto trait Send {
    // empty
}
```

```rust
pub unsafe auto trait Sync {
    // Empty
}
```

The compiler will auto-apply Send/Sync if all fields are Send/Sync.

Data is Send if it can be sent between threads safely.

This means No Unsafe Mutable.

Data is Sync if it can be shared between threads safely.

It promises that data is Mutable Safely. i.e. Mutexes & Atomics.

Speaker notes

Luckily you rarely need to implement these; the compiler will automatically assign these to your as long as the types you use to implement your data structures are themselves Send and/or Sync.

'static

```rust
fn get_name() -> &'static str {
    "Steve"
}
```

```rust
struct StaticData {
    avatar: Vec<u8>,
    uuid: String,
}

struct NotStatic<'a> {
    image: &'a Vec<u8>,
}
```

"A reference lifetime `'static` indicates that the data pointed to by the reference lives for the remaining lifetime of the running program."

"A `'static` trait bound means the type does not contain any non-static references."

Speaker notes

You'll probably have noticed that the thread spawn constraints included 'static.
This is slightly confusing, as it's not the same thing as 'static on e.g. string literals, which means that it's baked into the program.
'static in this context means that all the components of the type are self-contained; Owned types such as Vec or String, or other statics.

Threads and Rc

```rust
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    // ...
}

let rc = Rc::new(config);

let backend_thread = thread::spawn(|| backend(rc) );
```

Speaker notes

The problem we have is that Rc is neither Send nor Sync.
In fact its implementation explicitly states that it is not.

Threads and Rc

```
1  impl<T> !Send for Rc<T> {}
2
3  impl<T> !Sync for Rc<T> {}
```

Speaker notes

The problem we have is that Rc is neither Send nor Sync. In fact its implementation explicitly states that it is not.

However there is another related type Arc, short for Atomic Reference Counter. It acts much like Rc, but its internal shared data is wrapped in atomic primitives.

Threads and Rc

```
let config = Config {
    mangos: true,
    avatar: load_avatar(),
};
let arc1 = Arc::new(config);
let arc2 = rc.clone();

let backend_thread = thread::spawn(move || backend(arc1) );
let frontend_thread = thread::spawn(move || frontend(arc2) );
```

Speaker notes

The standard library provides a large number of thread-safe primitives you can use, including various channel implementations, which are another method of communicating between threads.

Arc is Atomic

```
1 struct ArcInner<T> {
2     strong: atomic::AtomicUsize,
3
4     weak: atomic::AtomicUsize,
5
6     data: T,
7 }
```

```
1 unsafe impl<T: Sync + Send> Send for Arc<T> {}
2 unsafe impl<T: Sync + Send> Sync for Arc<T> {}
```

Async

Async

https://emschwartz.me/async-rust-can-be-a-pleasure-to-work-with-without-send-sync-static/

https://emschwartz.me/async-rust-can-be-a-pleasure-to-work-with-without-send-sync-static/

Conclusion & Links

Rustonomicon

https://doc.rust-lang.org/nomicon/send-and-sync.html

Rust Design Patterns

https://rust-unofficial.github.io/patterns/idioms/coercion-arguments.html

Evan Schwartz

https://emschwartz.me/async-rust-can-be-a-pleasure-to-work-with-without-send-sync-static/

Like and Subscribe!

ANDROID APP ON Google play

Download on the App Store

goto;

https://haltcondition.net

@tarkasteve@hachyderm.io

@tarka.haltcondition.net