Products

Why can't I store a value and a reference to that value in the same struct?

Log in Sign up

Ask Question

Asked 6 years, 4 months ago

Active 9 months ago

Viewed 40k times

305

121

I have a value and I want to store that value and a reference to something inside that value in my own type:

```
struct Thing {
    count: u32,
}

struct Combined<'a>(Thing, &'a u32);

fn make_combined<'a>() -> Combined<'a> {
    let thing = Thing { count: 42 };

    Combined(thing, &thing.count)
}
```

Sometimes, I have a value and I want to store that value and a reference to that value in the same structure:

```
struct Combined<'a>(Thing, &'a Thing);

fn make_combined<'a>() -> Combined<'a> {
    let thing = Thing::new();

    Combined(thing, &thing)
}
```

Sometimes, I'm not even taking a reference of the value and I get the same error:

```
struct Combined<'a>(Parent, Child<'a>);

fn make_combined<'a>() -> Combined<'a> {
    let parent = Parent::new();
    let child = parent.child();

    Combined(parent, child)
}
```

In each of these cases, I get an error that one of the values "does not live long enough". What does this error mean?

rust reference lifetime borrow-checker

Share
Improve this question
Follow

edited Mar 7 at 1:01

trent formerly cl
20.1k ●7 ●42 ●72

asked Aug 30 '15 at 19:06

Shepmaster
305k ●59 ●824 ●1083

1

For the latter example, a definition of Parent and Child could help...
– Matthieu M.
Aug 31 '15 at 6:32

2

@MatthieuM. I debated that, but decided against it based on the two linked questions. Neither of those questions looked at the definition of the struct *or* the method in question, so I thought it would be best to mimic that to that people can more easily match this question to their own situation. Note that I *do* show the method signature in the answer.
– Shepmaster
Aug 31 '15 at 14:25

Add a comment

2 Answers

Active Oldest Votes

349

Let's look at a simple implementation of this:

```
struct Parent {
    count: u32,
}

struct Child<'a> {
    parent: &'a Parent,
}

struct Combined<'a> {
    parent: Parent,
    child: Child<'a>,
}

impl<'a> Combined<'a> {
    fn new() -> Self {
        let parent = Parent { count: 42 };
        let child = Child { parent: &parent };

        Combined { parent, child }
    }
}

fn main() {}
```

This will fail with the error:

```
error[E0515]: cannot return value referencing local variable `parent`
  --> src/main.rs:19:9
   |
17 |         let child = Child { parent: &parent };
   |                                     ------- `parent` is borrowed here
18 |
19 |         Combined { parent, child }
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^ returns a value referencing data owned by the current function

error[E0505]: cannot move out of `parent` because it is borrowed
  --> src/main.rs:19:20
   |
14 | impl<'a> Combined<'a> {
   |      -- lifetime `'a` defined here
...
17 |         let child = Child { parent: &parent };
   |                                     ------- borrow of `parent` occurs here
18 |
19 |         Combined { parent, child }
   |         -----------^^^^^^---------
   |         |          |
   |         |          move out of `parent` occurs here
   |         returning this value requires that `parent` is borrowed for `'a`
```

To completely understand this error, you have to think about how the values are represented in memory and what happens when you *move* those values. Let's annotate `Combined::new` with some hypothetical memory addresses that show where values are located:

```
let parent = Parent { count: 42 };
// `parent` lives at address 0x1000 and takes up 4 bytes
// The value of `parent` is 42
let child = Child { parent: &parent };
// `child` lives at address 0x1010 and takes up 4 bytes
// The value of `child` is 0x1000

Combined { parent, child }
// The return value lives at address 0x2000 and takes up 8 bytes
// `parent` is moved to 0x2000
// `child` is ... ?
```

What should happen to `child`? If the value was just moved like `parent` was, then it would refer to memory that no longer is guaranteed to have a valid value in it. Any other piece of code is allowed to store values at memory address 0x1000. Accessing that memory assuming it was an integer could lead to crashes and/or security bugs, and is one of the main categories of errors that Rust prevents.

This is exactly the problem that *lifetimes* prevent. A lifetime is a bit of metadata that allows you and the compiler to know how long a value will be valid at its **current memory location**. That's an important distinction, as it's a common mistake Rust newcomers make. Rust lifetimes are *not* the time period between when an object is created and when it is destroyed!

As an analogy, think of it this way: During a person's life, they will reside in many different locations, each with a distinct address. A Rust lifetime is concerned with the address you *currently reside at*, not about whenever you will die in the future (although dying also changes your address). Every time you move it's relevant because your address is no longer valid.

It's also important to note that lifetimes *do not* change your code; your code controls the lifetimes, your lifetimes don't control the code. The pithy saying is "lifetimes are descriptive, not prescriptive".

Let's annotate `Combined::new` with some line numbers which we will use to highlight lifetimes:

```
{                                          // 0
    let parent = Parent { count: 42 };     // 1
    let child = Child { parent: &parent }; // 2
                                           // 3
    Combined { parent, child }             // 4
}                                          // 5
```

The *concrete lifetime* of `parent` is from 1 to 4, inclusive (which I'll represent as `[1,4]`). The concrete lifetime of `child` is `[2,4]`, and the concrete lifetime of the return value is `[4,5]`. It's possible to have concrete lifetimes that start at zero - that would represent the lifetime of a parameter to a function or something that existed outside of the block.

Note that the lifetime of `child` itself is `[2,4]`, but that it **refers to** a value with a lifetime of `[1,4]`. This is fine as long as the referring value becomes invalid before the referred-to value does. The problem occurs when we try to return `child` from the block. This would "over-extend" the lifetime beyond its natural length.

This new knowledge should explain the first two examples. The third one requires looking at the implementation of `Parent::child`. Chances are, it will look something like this:

```
impl Parent {
    fn child(&self) -> Child { /* ... */ }
}
```

This uses *lifetime elision* to avoid writing explicit *generic lifetime parameters*. It is equivalent to:

```
impl Parent {
    fn child<'a>(&'a self) -> Child<'a> { /* ... */ }
}
```

In both cases, the method says that a `Child` structure will be returned that has been parameterized with the concrete lifetime of `self`. Said another way, the `Child` instance contains a reference to the `Parent` that created it, and thus cannot live longer than that `Parent` instance.

This also lets us recognize that something is really wrong with our creation function:

```
fn make_combined<'a>() -> Combined<'a> { /* ... */ }
```

Although you are more likely to see this written in a different form:

```
impl<'a> Combined<'a> {
    fn new() -> Combined<'a> { /* ... */ }
}
```

In both cases, there is no lifetime parameter being provided via an argument. This means that the lifetime that `Combined` will be parameterized with isn't constrained by anything - it can be whatever the caller wants it to be. This is nonsensical, because the caller could specify the `'static` lifetime and there's no way to meet that condition.

How do I fix it?

The easiest and most recommended solution is to not attempt to put these items in the same structure together. By doing this, your structure nesting will mimic the lifetimes of your code. Place types that own data into a structure together and then provide methods that allow you to get references or objects containing references as needed.

There is a special case where the lifetime tracking is overzealous: when you have something placed on the heap. This occurs when you use a `Box<T>`, for example. In this case, the structure that is moved contains a pointer into the heap. The pointed-at value will remain stable, but the address of the pointer itself will move. In practice, this doesn't matter, as you always follow the pointer.

Some crates provide ways of representing this case, but they require that the base address *never move*. This rules out mutating vectors, which may cause a reallocation and a move of the heap-allocated values.

- rental (no longer maintained or suppported)
- owning_ref
- ouroboros

Examples of problems solved with Rental:

- Is there an owned version of String::chars?
- Returning a RWLockReadGuard independently from a method
- How can I return an iterator over a locked struct member in Rust?
- How to return a reference to a sub-value of a value that is under a mutex?
- How do I store a result using Serde Zero-copy deserialization of a Futures-enabled Hyper Chunk?
- How to store a reference without having to deal with lifetimes?

In other cases, you may wish to move to some type of reference-counting, such as by using `Rc` or `Arc`.

More information

> After moving `parent` into the struct, why is the compiler not able to get a new reference to `parent` and assign it to `child` in the struct?

While it is theoretically possible to do this, doing so would introduce a large amount of complexity and overhead. Every time that the object is moved, the compiler would need to insert code to "fix up" the reference. This would mean that copying a struct is no longer a very cheap operation that just moves some bits around. It could even mean that code like this is expensive, depending on how good a hypothetical optimizer would be:

```
let a = Object::new();
let b = a;
let c = b;
```

Instead of forcing this to happen for *every* move, the programmer gets to *choose* when this will happen by creating methods that will take the appropriate references only when you call them.

A type with a reference to itself

There's one specific case where you *can* create a type with a reference to itself. You need to use something like `Option` to make it in two steps though:

```
#[derive(Debug)]
struct WhatAboutThis<'a> {
    name: String,
    nickname: Option<&'a str>,
}

fn main() {
    let mut tricky = WhatAboutThis {
        name: "Annabelle".to_string(),
        nickname: None,
    };
    tricky.nickname = Some(&tricky.name[..4]);

    println!("{:?}", tricky);
}
```

This does work, in some sense, but the created value is highly restricted - it can *never* be moved. Notably, this means it cannot be returned from a function or passed by-value to anything. A constructor function shows the same problem with the lifetimes as above:

```
fn creator<'a>() -> WhatAboutThis<'a> { /* ... */ }
```

If you try to do this same code with a method, you'll need the alluring but ultimately useless `&'a self`. When that's involved, this code is even more restricted and you will get borrow-checker errors after the first method call:

```
#[derive(Debug)]
struct WhatAboutThis<'a> {
    name: String,
    nickname: Option<&'a str>,
}

impl<'a> WhatAboutThis<'a> {
    fn tie_the_knot(&'a mut self) {
       self.nickname = Some(&self.name[..4]);
    }
}

fn main() {
    let mut tricky = WhatAboutThis {
        name: "Annabelle".to_string(),
        nickname: None,
    };
    tricky.tie_the_knot();

    // cannot borrow `tricky` as immutable because it is also borrowed as mutable
    // println!("{:?}", tricky);
}
```

See also:

- [Cannot borrow as mutable more than once at a time in one code - but can in another very similar]

What about `Pin`?

[`Pin`], stabilized in Rust 1.33, has this [in the module documentation]:

> A prime example of such a scenario would be building self-referential structs, since moving an object with pointers to itself will invalidate them, which could cause undefined behavior.

It's important to note that "self-referential" doesn't necessarily mean using *a reference*. Indeed, the [example of a self-referential struct] specifically says (emphasis mine):

> We cannot inform the compiler about that with a normal reference, since this pattern cannot be described with the usual borrowing rules. Instead **we use a raw pointer**, though one which is known to not be null, since we know it's pointing at the string.

The ability to use a raw pointer for this behavior has existed since Rust 1.0. Indeed, owning-ref and rental use raw pointers under the hood.

The only thing that `Pin` adds to the table is a common way to state that a given value is guaranteed to not move.

See also:

- [How to use the Pin struct with self-referential structures?]

Share
Improve this answer
Follow
edited Dec 1 '20 at 19:33

answered Aug 30 '15 at 19:06

Shepmaster
305k • 59 • 824 • 1083

1

Is something like this (is.gd/wl2IAt) considered idiomatic? Ie, to expose the data via methods instead of the raw data.
– Peter Hall
Jan 4 '16 at 22:05

2

@PeterHall sure, it just means that `Combined` owns the `Child` which owns the `Parent`. That may or may not make sense depending on the actual types that you have. Returning references to your own internal data is pretty typical.
– Shepmaster
Jan 4 '16 at 22:42

What is the solution to the heap problem?
– derekdreery
Nov 14 '16 at 11:25

@derekdreery perhaps you could expand on your comment? Why is the entire paragraph talking about the *owning_ref* crate insufficient?
– Shepmaster
Nov 14 '16 at 13:57

2

@FynnBecker it's still impossible to store a **reference** and a value to that reference. `Pin` is mostly a way to know the safety of a struct containing a self referential **pointer**. The ability to use a raw pointer for the same purpose has existed since Rust 1.0.
– Shepmaster
Mar 4 '19 at 17:52

Show **7** more comments

▲

12

A slightly different issue which causes very similar compiler messages is object lifetime dependency, rather than storing an explicit reference. An example of that is the ssh2 library. When developing something bigger than a test project, it is tempting to try to put the `Session` and `Channel` obtained from that session alongside each other into a struct, hiding the implementation details from the user. However, note that the `Channel` definition has the `'sess` lifetime in its type annotation, while `Session` doesn't.

This causes similar compiler errors related to lifetimes.

One way to solve it in a very simple way is to declare the `Session` outside in the caller, and then for annotate the reference within the struct with a lifetime, similar to the answer in this Rust User's Forum post talking about the same issue while encapsulating SFTP. This will not look elegant and may not always apply - because now you have two entities to deal with, rather than one that you wanted!

Turns out the rental crate or the owning_ref crate from the other answer are the solutions for this issue too. Let's consider the owning_ref, which has the special object for this exact purpose: `OwningHandle`. To avoid the underlying object moving, we allocate it on the heap using a `Box`, which gives us the following possible solution:

```
use ssh2::{Channel, Error, Session};
use std::net::TcpStream;

use owning_ref::OwningHandle;

struct DeviceSSHConnection {
    tcp: TcpStream,
    channel: OwningHandle<Box<Session>, Box<Channel<'static>>>,
}

impl DeviceSSHConnection {
    fn new(targ: &str, c_user: &str, c_pass: &str) -> Self {
        use std::net::TcpStream;
        let mut session = Session::new().unwrap();
        let mut tcp = TcpStream::connect(targ).unwrap();

        session.handshake(&tcp).unwrap();
        session.set_timeout(5000);
        session.userauth_password(c_user, c_pass).unwrap();

        let mut sess = Box::new(session);
        let mut oref = OwningHandle::new_with_fn(
            sess,
            unsafe { |x| Box::new((*x).channel_session().unwrap()) },
        );

        oref.shell().unwrap();
        let ret = DeviceSSHConnection {
            tcp: tcp,
            channel: oref,
        };
        ret
    }
}
```

The result of this code is that we can not use the `Session` anymore, but it is stored alongside with the `Channel` which we will be using. Because the `OwningHandle` object dereferences to `Box`, which dereferences to `Channel`, when storing it in a struct, we name it as such. **NOTE:** This is just my understanding. I have a suspicion this may not be correct, since it appears to be quite close to discussion of `OwningHandle` unsafety.

One curious detail here is that the `Session` logically has a similar relationship with `TcpStream` as `Channel` has to `Session`, yet its ownership is not taken and there are no type annotations around doing so. Instead, it is up to the user to take care of this, as the documentation of handshake method says:

> This session does not take ownership of the socket provided, it is recommended to ensure that the socket persists the lifetime of this session to ensure that communication is correctly performed.
>
> It is also highly recommended that the stream provided is not used concurrently elsewhere for the duration of this session as it may interfere with the protocol.

So with the `TcpStream` usage, is completely up to the programmer to ensure the correctness of the code. With the `OwningHandle`, the attention to where the "dangerous magic" happens is drawn using the `unsafe {}` block.

A further and a more high-level discussion of this issue is in this Rust User's Forum thread - which includes a different example and its solution using the rental crate, which does not contain unsafe blocks.

Your Answer

Not the answer you're looking for? Browse other questions tagged rust reference lifetime borrow-checker or ask your own question.

The Overflow Blog

- Sequencing your DNA with a USB dongle and open source code
- Don’t push that button: Exploring the software that flies SpaceX rockets and...

Featured on Meta

- Providing a JavaScript API for userscripts
- Congratulations to the 59 sites that just left Beta

Linked

How to store rusqlite Connection and Statement objects in the same struct in Rust?

How can I provide a reference to a struct that is a sibling?

4
How to design a struct when I need to reference to itself

Return a reference together with the referenced object in Rust

2
Return references to parts of one struct as fields of another struct

2
Can I transfer ownership of local variables and references to them to a returned iterator?

Reference to the field in the same/containing structure

Is it possible to call a parent struct's methods from a child struct?

0
How do I construct and return two values, one of which references the other

2
Cannot borrow as mutable more than once a at a time / Cannot infer an appropriate lifetime parameter

See more linked questions

Related

3618
What are the differences between a pointer variable and a reference variable in C++?

How do I implement the Add trait for a reference to a struct?

"borrowed value does not live long enough" when using the builder pattern

Why can't I return an &str value generated from a String?

Why is the bound `T: 'a` required in order to store a reference `&'a T`?

Rust, how to return reference to something in a struct that lasts as long as the struct?

1
Why does the compiler assume the returned reference has the same lifetime as the struct?

Hot Network Questions

Seeing oneself in an abstract painting
What does this entry on the Rocinante's pilot quick-menu mean?
How much of the English history in this Decameron story has any basis in fact?
Sample without replacement from 1 to N and stop when the value is less than the previous one
How to salvage bitter homemade mustard?
more hot questions

Question feed