Swapping two local references leads to lifetime error

Asked 3 years ago
Active 3 years ago
Viewed 599 times

7

I have two variables of type `&T`, `x` and `y`, which I swap locally inside a function:

```
pub fn foo<T: Copy>(mut x: &T) {
    let y_owned = *x;
    let mut y = &y_owned;
    for _ in 0..10 {
        do_work(x, y);
        std::mem::swap(&mut x, &mut y);
    }
}

fn do_work<T>(_x: &T, _y: &T) {}
```

This code fails to compile, giving the following error:

```
error[E0597]: `y_owned` does not live long enough
 --> src/lib.rs:3:22
  |
3 |         let mut y = &y_owned;
  |                     ^^^^^^^ borrowed value does not live long enough
...
8 |     }
  |     - borrowed value only lives until here
  |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the function body at 1:5...
 --> src/lib.rs:1:5
  |
1 | /     pub fn foo<T: Copy>(mut x: &T) {
2 | |         let y_owned = *x;
3 | |         let mut y = &y_owned;
4 | |         for _ in 0..10 {
... |
7 | |         }
8 | |     }
  | |_____^
```

I fail to see why it shouldn't work. `x` and `y` have different lifetimes, but why should the compiler require `y` to live as long as `x`? I am only modifying references locally inside `foo` and referenced objects are guaranteed to exist. Once `foo` returns, it doesn't matter if these `x` and `y` even existed, does it?

For larger context, I am implementing mergesort and want to swap the primary and auxiliary (temporary) arrays this way.

rust lifetime borrow-checker borrowing

edited Dec 19 '18 at 14:11

Peter Hall
41.4k ● 11 ● 90 ● 157

asked Dec 18 '18 at 14:56

kreo
1,895 ● 1 ● 12 ● 22

4 Answers

8

> Obviously, `x` and `y` have different lifetimes, but why should compiler require `y` to live as long as `x`?

Because of the signature of `std::mem::swap`:

```
pub fn swap<T>(x: &mut T, y: &mut T)
```

`T` is the type of the argument to `foo`, which is a reference of some lifetime *chosen by the caller of `foo`*. In the 2018 edition of Rust, the latest compiler gives a slightly more detailed error message in which it calls this lifetime `'1`. Calling `std::mem::swap` requires the type of `x`, `&'1 T`, to be the same as the type of `y`, but it can't shrink the lifetime of `x` to match that of `y` because the lifetime of `x` is chosen by the *caller*, not by `foo` itself. Vikram's answer goes into more detail on why the lifetime cannot be shrunk in this case.

> I am essentially only modifying references locally inside foo and referenced objects are guaranteed to exist

This is true, but it doesn't give you any freedom with respect to the lifetime of `x` inside `foo`. To make `foo` compile, you have to give the compiler another degree of freedom by making a new borrow of which the compiler can choose the lifetime. This version will compile (playground):

```
pub fn foo<T: Copy>(x: &T) {
    let mut x = &*x;
    ...
}
```

This is called *reborrowing*, and it happens implicitly in some cases, for example, to the receiver of a method call that takes `&mut self`. It does not happen implicitly in the case you presented because `swap` is not a method.

Share
Improve this answer
Follow
edited Dec 18 '18 at 16:16

answered Dec 18 '18 at 15:21

trent formerly cl
20.1k ● 7 ● 42 ● 72
Add a comment

▲
5
▼

It is helpful to compile this program with the latest stable toolchain on the 2018 Edition, since it improves the error message a bit:

```
error[E0597]: `y_owned` does not live long enough
 --> src/lib.rs:4:17
  |
1 | pub fn foo<T: Copy>(mut x: &T) {
  |                     - let's call the lifetime of this reference `'1`
...
4 |     let mut y = &y_owned;
  |                 ^^^^^^^^
  |                 |
  |                 borrowed value does not live long enough
  |                 assignment requires that `y_owned` is borrowed for `'1`
...
9 | }
  | - `y_owned` dropped here while still borrowed
```

What happens is:

- the input `x` is a reference with the arbitrary lifetime `'1` established by the caller.
- The variable `y` is a reference created locally, and as such it has lifetime "shorter" than `'1`.

As such, you cannot pass the reference in `y` to `x`, even if it may seem safe to, because `x` expects something that lives at least for the lifetime indicated by the caller.

One possible solution is to create a second copy of the value behind x, and borrow that locally.

```
pub fn foo<T: Copy>(x: &T) {
    let mut x = &*x;
    let mut y = &*x;
    for _ in 0..10 {
        do_work(x, y);
        std::mem::swap(&mut x, &mut y);
    }
}
```

Share
Improve this answer
Follow
answered Dec 18 '18 at 15:18

E_net4 the flagger
22.8k ● 11 ● 80 ● 117
Add a comment

▲
4
▼

Mutable references are invariant over the type they refer to. If you have `&'a mut T`, then it is invariant over `T`. The signature of `swap()` expects same types with same lifetimes on both the input arguments. i.e they both are mutable references to `T`.

Let's look at your problem:

The argument to `foo()` is `&T` and with lifetimes it will be `foo<'a, T: Copy>(mut x: &'a T)` and this lifetime is given by the caller. Within the function you have a local variable

`y_owned` and you take a reference to it with some local lifetime. So at this point we have `&'a T` which is the input argument with lifetime set by the caller and `&'local y_owned` with some local lifetime. All good!

Next, you call `swap()` and pass to it, mutable references (`&mut &T` and `&mut &y_owned`) to the aforementioned references. Now, here's the catch; Since they are mutable references and as mentioned they are invariant over what they point to; `x` which is `&'a T` will not shrink down to the scope of the function call , as a result `y` which is `&'local y_owned` will also now be expected to be `&'a y_owned` , which is not possible, since `'a` goes beyond `y_owned` , hence it complains that `y_owned` does not live long enough.

For more, please refer this

Share
Improve this answer
Follow
edited Dec 18 '18 at 17:29

answered Dec 18 '18 at 15:40

vikram2784
549 ● 5 ● 13

Add a comment

3

The lifetime information of a reference is part of its type. Since Rust is a statically typed language, the lifetime of a reference variable can't dynamically change at runtime.

The lifetime of the reference `x` is specified by the caller, and it must be longer than everything that is created inside the function. The lifetime of `y` is the lifetime of a variable local to the funciton, and as such is shorter than the lifetime of `x`. Since the two lifetimes don't match, you can't swap the variables, since you can't dynamically change the type of a variable, and the lifetime is part of its type.

Share
Improve this answer
Follow
answered Dec 18 '18 at 15:21

Sven Marnach
511k ● 113 ● 891 ● 798

Add a comment

Your Answer

Post Your Answer

By clicking "Post Your Answer", you agree to our terms of service, privacy policy and cookie policy

Not the answer you're looking for? Browse other questions tagged rust lifetime borrow-checker borrowing 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

Why is the mutable reference not moved here?

Related

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

Factory method: instance does not live long enough

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

Taking closures with explicit lifetimes as arguments in Rust

3

Value does not live long enough with a generic function that creates a container, adds items to it, then iterates over the items

0

Lack of lifetime of line from buffered reader prevents splitting line

Problems with Tuple's lifetime in rust.

Infer an appropriate lifetime for methods including interior references

`*arg0` does not live long enough - wasm_bindgen

Hot Network Questions

- How to convince clan leaders and Party Cadres to give up their power?
- Is Elon Musk really exploiting a loophole to avoid taxes?
- What did John the Baptist say about Jesus that resulted in many Jews in Jesus' time to put faith in him? John 10:41-42
- Is it acceptable to omit "about" in this sentence? "I love everything (about) math."
- Split polyline to equal parts using QGIS

more hot questions

Question feed