Understanding the fundamental Rust - value doesn't live long enough

help

MOCKBA July 25, 2025, 8:02pm 1

I have the following fragment:

```
let line = match prev {
           None => {
                let Ok(len) = stdin.read(&mut buffer) else {break};
                String::from_utf8_lossy(&buffer[0..len])
           }
           Some(ref vec) => {
             String::from_utf8_lossy(&vec)}
       };
       prev = None;
   // do something with line
```

It will complain that `prev` can't be reassigned because its content then used. It's fine and understandable. However, if I try to clone `prev` value like:

```
String::from_utf8_lossy(&vec.clone())
```

It complains that a temporary value doesn't live long enough. So generally I need to define the clone vector outside. So question is: why Rust can't make a life time of the inner value long enough to get it out the block and assigned?

jumpnbrownweasel July 25, 2025, 8:13pm 2

Can you provide something we can try to compile and that reproduces the error? I guessed at the types for the variables you hadn't defined, and then didn't get a compile error, so I must have guessed wrong.

```
use std::io::Read;

fn f(mut prev: Option<Vec<u8>>) {
   let mut stdin = std::io::stdin();
   let mut buffer = [0_u8; 100];
   loop {
       let line = match prev {
           None => {
               let Ok(len) = stdin.read(&mut buffer) else {
                   break;
               };
               String::from_utf8_lossy(&buffer[0..len])
           }
           Some(ref vec) => String::from_utf8_lossy(vec),
       };
       prev = None;
       // do something with line
   }
}
```

jonh July 25, 2025, 9:57pm 3

@jumpnbrownweasel Only thing you miss is filling in the code of comment.

The important part is knowing `from_utf8_lossy` returns a `Cow`
Rather than a compulsory `clone` you can just add a call `into_owned`.
As you are wiping `prev` you can `take().unwrap()` it instead and avoid the clone.

MOCKBA:

> Rust can't make a life time of the inner value long enough to get it out the block and assigned?

It has code in simple assignment cases that do such

```
let _x = &String::new();
```

To persist out in the example though would involve the compiler backtracking to find a spot to place the variable. It isn't too easy to write as evaluation moves forward through code.

1 Like

MOCKBA July 25, 2025, 10:00pm 4

If you add something after the comment line as:

```
println!("{line}");
```

you should get an error. And then to fix the error, my question is comming.

1 Like

KillTheMule July 25, 2025, 10:24pm 5

This works: Rust Playground

By using `String::from_utf8_lossy` you're producing a value that keeps a reference to its input, which in turn has a reference to `prev`, which won't work, as the error says. Making it into an owned value makes it work, but of course the other match arm has to be adjusted as well. Incidentally, you can skip the `clone` call as well then.

jumpnbrownweasel July 25, 2025, 10:32pm 6

MOCKBA:

> If you add something after the comment line as:

Thanks, that caused the error. Could you assign `None` to `prev` *after* you're finished with `line`? That avoids the error, and also avoids allocating with `into_owned`.

@KillTheMule answered your question about *why* the error is occurring.

quinedot July 25, 2025, 10:48pm 7

MOCKBA:

> It complains that a temporary value doesn't live long enough. So generally I need to define the clone vector outside. So question is: why Rust can't make a life time of the inner value long enough to get it out the block and assigned?

I guess you're talking about something like this (and the fix you can apply.)

I recommend reading this article. Note that it was written before Edition 2024 which changed some temporary scopes involving `if let` and , but most of the things in the article still apply. Let me try to highlight some relevant sections...

> From the "In a `let` statement" section
>
> ```
> let a = f(&String::from('■'));
> …
> g(&a);
> ```
>
> Again: how long does the temporary `String` live?
>
> 1. The string gets dropped at the end of the let statement: after `f` returns, but before `g` is called. Or,
> 2. The string will be dropped at the same time as `a`, after `g` is called.
>
> This time, option 1 might work, depending on the signature of `f`. If `f` was defined as `fn f(s: &str) -> usize` (like `str::len`), then it's perfectly fine to immediately drop the `String` after the `let` statement.
>
> However, if `f` was defined as `fn f(s: &str) -> &[u8]` (like `str::as_bytes`), then a would borrow from the temporary `String`, so we'd get a borrow checking error if we keep a around for longer.
>
> With option 2, it'd compile fine in both cases, but we might keep a temporary around for much longer than necessary, which can waste resources or cause subtle bugs (e.g. a deadlock when a `MutexGuard` is dropped later than expected).
>
> This sounds like we might want to go for a third option: make it depend on the signature of `f`.
>
> However, Rust's borrow checker only performs a *check*; it does not influence the behavior of the code. [...] So, as a choice between only option 1 and 2, Rust went for option 1: drop the temporary at the end of the `let` statement.

> From the "Temporary lifetime extension" section
>
> Temporary lifetime extension does not apply to all temporaries that appear in a let statement, as we've already seen: the temporary string in `let a = f(&String::from('■'));` does not outlast the let statement.
> [...]
> The rules for which temporaries in a let statement get their lifetimes extended is documented in the Rust Reference, but effectively comes down to those expressions where you can tell from just the syntax that extending the lifetime is necessary, independent of any types, function signatures, or trait

implementations

And then the article goes on to suggest new `super let` functionality in the language. That is now an unstable feature. It works for the example, though it's not a huge improvement IMO.[1]

AFAIK there's not yet something that works like `super let`, but in match patterns directly. ■■

1 Like

jumpnbrownweasel July 25, 2025, 10:59pm 8

quinedot:

and the fix you can apply

Very nice.

MOCKBA July 26, 2025, 12:21am 9

Indeed, a cause of the problem was that `line` isn't owning the value. I had actually `into()', but doings some refactoring, I forgot why it was there, and simple removed it. It's a silly mistake. Anyway, the thread is really helpful and uncovers more hidden Rust for me.

MOCKBA July 26, 2025, 12:31am 10

quinedot:

(and the fix you can apply.)

It's awesome that you can just just reserve a space for a variable like `let vec;`. If I knew that before, I could simplify code in other places too.

doublequartz July 26, 2025, 5:51am 11

I'm pretty sure that almost every programming language allow declaring variables without also assigning a value to it; at least, it's hard for me to name a language where that isn't the case.

To be fair, it's reasonable for you to have not known it possible, given that I couldn't find any explicit mention in the Book. This is disappointing, but I can easily see why it has failed you. It's exactly because the feature is so universal, and so immediately obvious to certain people, that they don't find it as something to be taught.

Tangentially:

It's also disappointing that searching for a Rust keyword is so difficult. Not only does `mdbook` indiscriminately match English "let" just like Rust `let`, it also matches any substring like "de**let**e", without any way to opt out.

Anyway, if you're feeling eager to discover Rust features of various obscurity, I encourage you to read a wider variety of Rust-related information, as well as Rust code in the wild. While you don't need more than one tutorial covering basics of Rust which you already understand, digging deeper toward more specific domains is necessary to keep on finding new insights.

Some of the links in `awesome-rust` or Little Book of Rust Books might suit you well. If you work on a project with dependencies, reading the source code of said dependencies might help with your project in addition to increasing your Rust expertise.

jumpnbrownweasel July 26, 2025, 7:34am 12

doublequartz:

I'm pretty sure that almost every programming language allow declaring variables without also assigning a value to it; at least, it's hard for me to name a language where that isn't the case.

Some languages don't support this, e.g., in Go when the initializer expression is omitted the variable is initialized to zero/nil.

doublequartz:

I couldn't find any explicit mention in the Book

There is some mention in the reference:

doc.rust-lang.org

Variables - The Rust Reference

MOCKBA July 27, 2025, 1:00am 13

doublequartz:

Some of the links in `awesome-rust` or Little Book of Rust Books might suit you well.

Thanks for the links. However I stopped reading books since it's the job of AI now to make my vibe coding more efficient. From other side,

jumpnbrownweasel:

> There is some mention in the reference:

my university professor taught me to read only original design documents and references. So it looks like @jumpnbrownweasel studied at the same professor.

doublequartz:

> work on a project with dependencies, reading the source code of said dependencies might help with your project in addition to increasing your Rust expertise.

I completely agree with that observation, it's what I do continuously. Although I do not always agree with authors of 3rd party solutions and rewrite them in my way. It's the cause I do not use Cargo.

Sorry for a little of topic talk.

system Closed October 25, 2025, 1:01am 14

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic		Replies	Views	Activity
New to rust... error: "borrowed value does not live long enough" help	5	603	January 24, 2023	
Why does this code NOT fail to compile?	8	790	November 10, 2021	
Vector lifetime rust help	14	476	February 4, 2025	
Creates a temporary which is freed while still in use Again :) help	3	31745	September 10, 2019	
How to understand "temporary lifetime extension" correctly? help	10	5451	April 20, 2022	