Simple nuanced lifetime dilemma where dereferencing a Box does not compile while constructing a new Box does

help

Keaudes April 16, 2025, 1:23pm 1

My apologies for the lack of a better title.

Compare

```
let mut word   = String::new();
let mut looker = Box::new(&word);
word += "hello";
*looker = &word; // panic!
```

to

```
let mut word   = String::new();
let mut looker = Box::new(&word);
word += "hello";
looker = Box::new(&word);
```

I have simplified the problem as much as possible and bottled it down to this.

The first example does not compile while the second one does. That is because in example (2), `looker`'s old value is implicitly destroyed before `word += "hello"` is performed. In example (1) on the contrary, `word` is still borrowed by `looker` since it is only the inside of the `Box` which is manipulated, keeping `looker`'s lifetime annotations intact. However, since I am only overwriting `looker`'s inner value, this code is practically harmless.

Is there a way to achieve example (1) without a compiler error, that is by only accessing `*looker`, not `looker`?

1 Like

The borrow checker is a bit smart and a bit stupid, and so is Polonius

nerditation April 16, 2025, 1:36pm 2

Keaudes:

> The first example panics

it should not compile at all, the borrow checker surely will reject this code! why do you have a panic instead? are you not showing the actual code, or something else was happening?

Keaudes April 16, 2025, 1:42pm 3

I'm sorry, I mixed up the terminology. It did not compile indeed.

jumpnbrownweasel April 16, 2025, 2:12pm 4

The error message explains the reason example (1) doesn't work:

```
error[E0502]: cannot borrow `word` as mutable because it is also borrowed as immutable
   --> test.rs:1458:5
    |
1457 |     let mut looker = Box::new(&word);
    |                               ----- immutable borrow occurs here
1458 |     word += "hello";
    |     ^^^^^^^^^^^^^^^ mutable borrow occurs here
1459 |     *looker = &word;
    |     --------------- immutable borrow later used here
```

I'm not sure exactly what you're trying to accomplish, but my guess is that you want to mutate the String while holding a reference to it in the Box. To do this you can't mutate the `word` variable since it is borrowed as explained by the error message above. But you can manipulate a `&mut String` stored inside the Box:

```
let mut word = String::new();
let looker = Box::new(&mut word);
looker.push_str("hello");
```

This is a strange thing to do, however. Can you explain more about what you're trying to accomplish? It is often a mistake to store references inside a Box or inside other data structures.

kornel April 16, 2025, 2:34pm 5

Rust's references exist primarily to control ownership and borrowing, and don't map cleanly to references in other languages that either pass objects by copy or by reference, so beware that your intuition about usage of references may be completely wrong.

Rust has the third option of passing by move that looks like passing by value but doesn't require deep copying. Rust has lots of types that hold their data "by reference" without copying it, but their syntax doesn't have anything looking like a reference, and they're not used with `&`.

For example, `String` stores the text data by reference and doesn't copy it when you pass `String` around. This makes `&String` redundant and creates two layers of indirection AND most importantly also creates a temporary loan and a strongly enforced restriction where the loan can be used.

Box is a "by reference" type, and passing it is equivalent to passing a pointer. `Box` containing a `&` type is pretty useless: it creates two layers of indirection, has all the overhead of boxing, but it's still keeping all of the restrictions of temporary loans. Temporary loans can't be extended, no matter where you put them. Putting a temporary loan in a long-lived storage like Box only makes the storage short-lived too. The shortest most restrictive loan always wins and infects everything it touches.

Unlike GC languages where by-reference objects are normal and can exist on their own, references in Rust are a special case. They can never exist by themselves, they always borrow from some other type (in exceptional cases that something may be the executable file or leaked memory). They are always devoid of any data. They can't store anything. Additionally, references can't exist together in the same data structure with the data they are borrowing. So if you try to put `&word` in a Box, there won't be any text stored in the box, `&word` has no words in it, and you're overlooking that `word` is the only thing here that actually contains any data, and `word` holds this data *by reference* without using `&` anywhere, because it owns the data instead of temporarily borrowing it.

6 Likes

kpreid April 16, 2025, 4:42pm 6

I suspect that what @Keaudes is trying to do has been misunderstood. This is not the typical example of trying to *use* a reference after its referent has been mutated. In particular, notice that the last line in both cases is *writing to* `looker`, replacing the old invalid reference with a new valid one without making use of the old one.

The question is why this kind of lifetime-changing assignment works with a plain variable, but not with the place inside of a `Box`.

(I don’t have an answer other than “probably nobody spent the effort to make it work”.)

2 Likes

quinedot April 16, 2025, 6:56pm 7

I don't think it's really `Box` specific. These also do not compile:

```
let mut word = String::new();
let mut looker = &mut &word;
word += "hello";
*looker = &word;

let mut word = String::new();
let mut looker = (&word,);
word += "hello";
looker.0 = &word;
```

Whereas these do.

```
let mut word = String::new();
let mut looker = &mut &word;
word += "hello";
looker = &mut &word;

let mut word = String::new();
let mut looker = (&word,);
word += "hello";
looker = (&word,);
```

When you overwrite all of `looker`, the compiler recognizes that any borrows within need not be kept active before the overwrite (at `word += "hello"`).

1 Like

The borrow checker is a bit smart and a bit stupid, and so is Polonius

Keaudes April 17, 2025, 4:56pm 9

@kpreid, @quinedot

kpreid:

> The question is why this kind of lifetime-changing assignment works with a plain variable, but not with the place inside of a `Box`.

Exactly!

Learning from the everyone's replies summed up, the answer is simply: the compilable example *reassigns* `looker`, allowing it to have a new lifetime, while the uncompilable example *modifies* `looker`, only reassigning its contents.
And, as of now, this sort of thing can't be done in Rust.

Keaudes April 17, 2025, 4:58pm 10

I will share my code, but I don't think it will clear up the question any more. The `&String` is really a placeholder for "a shared reference to a non-copy object".

jumpnbrownweasel:

> It is often a mistake to store references inside a Box or inside other data structures.

Unfortunately it is not up to me. The "`&'x String`" is actually a `ratatui::widgets::List<'x>`, and can be made from (for example, but not only from) an `Iterator` of `strs`, and then the `List` keeps the lifetime of these `strs`.
I am writing a terminal widget that combines a text input as a searchbox and a list of results. With every keystroke, the list is refined to matches that only start with the searchbox input. This in itself is no reason to mess around with lifetimes.

However, when the user presses `[Insert]`, I want to add the word that is currently in the searchbox to the list of results, thus inserting that element to the underlying source for `List`. In order to do that, I must: uncouple the `List` from that source, modify the source, recouple `List` with source.

I figured out a way to do this, which requires putting the widget inside of an `Option` and setting that to `None`. I dropped the Box aspect in its entirety since it did not really add any usefulness.

This is the code, but beware that it requires an understanding of the `ratatui` crate.

system Closed July 16, 2025, 4:59pm 11

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
How to visit the field of a mutable object after it's mutally borrowed help	7	257	April 9, 2023	
How to move a Box while holding a reference to the element inside it? help	13	2284	September 16, 2023	
How can I do an insert into a vec<Box<T>> while using &T? help	6	144	May 8, 2025	
A question about &■pointer and ref help	2	381	March 14, 2023	
*Box<T> vs *(Box<T>.deref())	7	571	August 15, 2023	

- Home
- Categories
- Guidelines
- Terms of Service

Powered by Discourse, best viewed with JavaScript enabled