Why rust ignore lifetime checks on &str?

Asked 1 year, 1 month ago
Active 10 months ago
Viewed 244 times

7

1

```
fn main() {
    let strA = "a";
    let result;

    {
        let strB = "abc";
        result = longest(strA, strB); // Will return strB
    }

    println!("The longest string is {}", result); // result now point to strB!!
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

As I got from the Rust book

> 'a will get the concrete lifetime that is equal to the **smaller** of the lifetimes of x and y

So why strB is now visible outside of its scope?

rust lifetime borrow-checker ownership borrowing

Share
Improve this question
Follow

edited Feb 17 at 1:18

pretzelhammer
11.7k ● 14 ● 39 ● 88

asked Nov 27 '20 at 15:46

Basemm
1,075 ● 11 ● 15

Does this answer your question? Why do generic lifetimes not conform to the smaller lifetime of a nested scope?
– Steve Z
Mar 16 at 14:03

Add a comment

3 Answers

Active Oldest Votes

7

That's because all string literals have 'static lifetime. From the rust book:

> One special lifetime we need to discuss is 'static, which means that this reference can live for the entire duration of the program. All string literals have the 'static lifetime, which we can annotate as follows:
>
> ```
> let s: &'static str = "I have a static lifetime.";
> ```
>
> The text of this string is stored directly in the program's binary, which is always available. Therefore, the lifetime of all string literals is 'static

Share
Improve this answer

Follow

edited Dec 5 '20 at 15:42

answered Nov 27 '20 at 15:55

Mihir Luthra

4,675 ● 2 ● 11 ● 33

Add a comment

3

Fixed example:

```
fn main() {
    let strA = "a".to_string();
    let result;

    {
        let strB = "abc".to_string();
        result = longest(&strA, &strB); // Will return strB
    }

    println!("The longest string is {}", result); // compile error
}

fn longest<'a>(x &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Now produces a compiler error as expected:

```
error[E0597]: `strB` does not live long enough
  --> src/main.rs:7:33
   |
7  |         result = longest(&strA, &strB); // Will return strB
   |                                 ^^^^^ borrowed value does not live long enough
8  |     }
   |     - `strB` dropped here while still borrowed
9  |
10 |     println!("The longest string is {}", result); // result now point to strB!!
   |                                          ------ borrow later used here
```

playground

Rust was not "ignoring" the lifetimes of the string variables in your initial example. When you set a variable to a string literal that literal gets hardcoded into the executable binary and gets a `'static` lifetime which means it's valid for the entire duration of the program. If we add explicit type annotations to your initial example it should become clear why it compiles and works:

```
fn main() {
    let strA: &'static str = "a";
    let result;

    {
        let strB: &'static str = "abc";
        result = longest(&strA, &strB); // returns 'static str
    }

    println!("The longest string is {}", result); // prints result
}

fn longest<'a>(x &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

However when we call `to_string()` on a string literal we create an owned and heap allocated `String` which lifetime's is non-static and is scoped to whatever block its in, hence making the change makes the program no longer compile as expected.

Share

Improve this answer

Follow

answered Nov 27 '20 at 15:56

pretzelhammer

11.7k ● 14 ● 39 ● 88

Add a comment

1

the lifetime `'a` refers to the lifetime of the string buffer `str` not the reference to that buffer. So the lifetime of the `&str` `strB` is within the block. However, `"abc"` is a constant string. This means that the storage of the str buffer has `'static` lifetime, meaning it is guaranteed to outlast any other lifetime. As such it is valid for result to reference that buffer

even after `strB` no longer references it.

Share
Improve this answer
Follow

answered Nov 27 '20 at 15:53

user1937198
4,641 ● 4 ● 18 ● 31

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 ownership 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

0
About lifetime of Rust. Why the following code can be compiled

Why do generic lifetimes not conform to the smaller lifetime of a nested scope?

Related

What are the differences between Rust's `String` and `str`?

Does <'a, 'b: 'a> mean that the lifetime 'b must outlive the lifetime 'a?

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

Why do generic lifetimes not conform to the smaller lifetime of a nested scope?

Do lifetime annotations in Rust change the lifetime of the variables?

Lifetimes in Rust when using Strings

1
How is output lifetime of a function calculated?

Semantics of lifetime parameters

2
A simple test case to check my understanding of rust lifetimes

Hot Network Questions

- FEM for vector valued problems: reference request
- What does the numbers mean in this guitar tab if they already gave the chords to play?
- If we can get people to the moon and back, why are we so adamant that it's impossible to service James Webb at 4x that with a one way robotic vehicle?
- Do all observations arise from probability distributions?
- How am I able to simulate gears on my single speed?

more hot questions

Question feed

STACK OVERFLOW

Questions
Jobs
Developer Jobs Directory
Salary Calculator
Help
Mobile

PRODUCTS

Teams
Talent
Advertising
Enterprise

COMPANY

About
Press
Work Here
Legal
Privacy Policy
Terms of Service
Contact Us
Cookie Settings
Cookie Policy

STACK EXCHANGE NETWORK

Technology
Culture & recreation
Life & arts
Science
Professional
Business
API
Data

Blog
Facebook
Twitter
LinkedIn
Instagram