Trying to understand the basics of Rust

help

7urkm3n April 6, 2025, 4:35am 1

Im coming from Golang, since Go has pointers reference based. Can someone explain me the part of what im missing to understand it.

1. Different strings: str and String

```
    let s = "123";
   let s2 = String::from("123");

println!("{:p}", &s);
println!("{:p}", &s2);

// logs:
// 0x7ffca6304d98
// 0x7ffca6304da8
```

Since im printing out reference of both strings and they are actually different variables but somehow prints the same pointer address, WHY ?

UPDATED
Terminal log issue, fixed restarting it...

2) Also, If i use && its different:

```
println!("{:p}", &&s);
println!("{:p}", &&s2);

// logs:
// 0x7ffca6304123
// 0x7ffca6304abc
```

3) Also I can do this: What does it mean ?

```
println!("{:p}", &&&&&&&&&&s);
```

Cerber-Ursi April 6, 2025, 4:50am 2

7urkm3n:

> Different strings: str and String

In short:

- `str` is a blob of UTF-8 bytes;
- `&str` is a temporary view into this blob;
- `&'static str` is a view that does never expire (for example, if this blob of bytes is stored in the binary);
- `String` is a container that allocated this blob of bytes and is responsible for deallocating it when it's not used.

7urkm3n:

> somehow prints the same pointer address

It's not the same, it's `...d98` vs `...da8`. But this should not be relied on - if some value isn't used, Rust may reuse its position on stack for another value as optimization.

7urkm3n:

> What does it mean ?

Exactly what is written - it's a borrow of a borrow of a ... of a borrow of a blob of bytes. References are first-class values, they can be used wherever any other value can, including taking reference to them.

2 Likes

7urkm3n April 6, 2025, 5:00am 3

Thanks, yeah just realized its different. Terminal log issue, previously was printing the same fixed that restarting it.

So multiple &&& is a borrow of a borrow ... and doesnt matter how long its.

Its clear now.

Michael-F-Bryan April 6, 2025, 12:06pm 4

7urkm3n:

> So multiple &&& is a borrow of a borrow ... and doesnt matter how long its.

Yep. When you have multiple &'s, the compiler automatically inserts a temporary variable for you.

```
let x = 42;
let x_ptr = &&&&42;

// is equivalent to

let x = 42;
let temp_x_ptr1 = &x;
let temp_x_ptr2 = &x_ptr1;
let temp_x_ptr3 = &x_ptr2;
let x_ptr = &x_ptr3;
```

That explains why adding a & changes the address - you are pointing to a different variable on the stack.

4 Likes

jumpnbrownweasel April 6, 2025, 2:20pm 5

7urkm3n:

> Different strings: str and String

A longer explanation is in this chapter of The Book:

doc.rust-lang.org

What is Ownership? - The Rust Programming Language

1 Like

kornel April 6, 2025, 5:12pm 6

You will not be able to learn anything useful about references from behavior of `{:p}`. It's a wrong way to inspect what Rust is doing.

In Rust, objects don't have an identity attached to their address. Objects are movable, so they can exist at different addresses at different times, or not even have any address at all. The same address can be reused for different objects, even within the same function.

& has a semantic meaning of borrowing, and in many cases it's actually implemented as a pointer, but *it doesn't have to be*.
Rust is designed around zero-cost abstractions. It's designed to be used with an optimizer, and the optimizer can optimize references out of existence. This happens regularly to local variables and when methods with & arguments are inlined.

Data that used to be behind a reference can be copied into registers, and registers don't even have an address.

Merely asking about "what's the address of this thing?" changes how the program behaves. It may prevent optimizations that would remove the references from the actual program. It may force data to be copied from registers back to memory just for the purpose of getting some temporary address to print for you, even though that address is useless and not used for any other purpose than printing it.

And it's not just counter-productive with optimizations. In an unoptimized program, you also get nonsense. In the dev/debug builds, Rust only cares about generating code quickly, not about quality of the code. It will create useless temporary variables, it will needlessly move values around. So the same data will have various different addresses that can change, and don't mean much. You're just seeing implementation details, and in the dev/debug mode these aren't even details that anybody pays attention to, because they're all meant to be optimized out of existence in normal release builds.

And finally, Rust has pretty sophisticated auto-dereferencing that peels layers of &s whenever you call methods on it. In non-edge-case situations when you call `(&&&&&p).some_method()`, you're not even using `&&&&&p`, and the hypothetical address of the reference-to-reference-to-reference... doesn't matter, because it's dereferenced even before it starts to exist.

`{:p}` makes sense in specific cases, such as heap-allocated types and raw pointers, but in many other cases you can get either junk or fiction.

10 Likes

jonh April 6, 2025, 5:54pm 7

```
    let b1: &'static [u8; 3] = b"123";
   let b2: &'static [u8] = b"123";
```

Rust has a missing type. `str` is just a `[u8]` with additional restriction of being limited to utf-8. There is no replacement for the array so only a slice can be used.

system Closed July 5, 2025, 5:55pm 8

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
Strings - References and Derefrences help	4	138	November 5, 2025	
Reference vs pointer help	7	453	June 8, 2025	
How come Strings don't need & unlike String literals? help	8	632	September 23, 2020	
Undestanding str and ownership help	7	526	December 10, 2023	
[Solved] Difference between &str and String help	3	1008	September 7, 2020	