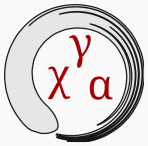

Geo's Notepad

Mostly Programming and Math

Rust Deep Dive: Borked Vtables and Barking Cats

Posted 2023-03-15 , Last Updated 2023-04-23

No, this post does not contain cruelty towards animals but only to our own sanity. We will explore a particular aspect of how Rust's trait objects work behind the scenes and take a deep dive down the rabbit hole. Sometimes it's good to be reminded that all the nice things we have as programmers are just sugar on top of ones and zeros in the imagination of some sand that we tricked into thinking.

Inspiration and Motivation

This post was inspired by this brilliantly titled video on the always entertaining and instructive Creel YouTube channel. In that video, the author shows how dynamic dispatch with inheritance works in C++ and how we can break it in interesting ways. We are going to take a look at how a similar thing can be achieved in Rust with trait objects. At the end of the post we're going to make this piece of code work:

```
let mut kitty: Box<dyn Pet> = Box::new(Cat::new("Kitty"));
// ... some magic ...
greet_pet(kitty);
```

and generate this output

```
You: Hello Kitty!
Kitty: Woof!
```

which indicates that something very peculiar is going on with our cat, because clearly it should go `"Meow!"` and not `"Woof!"`. The reason we snuck a `mut` in front of the kitty will become apparent once we work our evil magic. But first let's take a step back.

Dynamic Polymorphism: Meowing Cats and Barking Dogs

What we saw at work in our listing above is dynamic polymorphism. Wikipedia has the following to say about polymorphism in general:

> *In programming language theory and type theory, polymorphism is the provision of a single interface to entities of different types*

Dynamic polymorphism is the kind of polymorphism that happens at runtime, in contrast to e.g. static polymorphism with generics that happens at compile time. There are different ways of achieving dynamic polymorphism, but for this article I am interested in the kind of dynamic polymorphism that works with Rust's trait objects and pointers to them.

In a more object-oriented language[1] like C++ (or Java), the equivalent concept is dynamic dispatch through inheritance hierarchies. With it, we can call the methods of a derived class via a pointer (or reference) to its base class. Inheritance is the classic object oriented way of enabling dynamic polymorphism. In Rust we don't have inheritance but we have traits and trait objects. Let's look at a silly example that will accompany us through the rest of the post. We have a `Pet` trait and an implementor `Cat` like so. Feel free to skim the next part, because it's all just boilerplate and none of it will surprise you if you've ever implemented a trait.

```
trait Pet {
  fn name(&self) → String;
  fn sound(&self) → String;
}

struct Cat {
  life : u8, //keeping track of the 9 lives
  age : u8,
  my_name : String,
}

impl Cat {
  pub fn new(name : impl Into<String>) → Self {
    Self {
      my_name: name.into(),
      age : 0,
      life : 0,
    }
  }
}

impl Pet for Cat {
  fn name(&self) → String {
    &self.my_name
  }
  fn sound(&self) → String {
    "Meow!".into()
  }
}
```

We could also implement all kinds of other pet types, like `Dog`, `Bird`, and so on. You get the idea. Finally, with this boilerplate out of the way we can implement a function to greet a `Pet` trait object like so:

```
fn greet_pet(pet : Box<dyn Pet>) {
  println!("You: Hello {}", pet.name());
```

```
  println!("{}: {}", pet.name(), pet.sound());
}
```

This way, we can pass in the `kitty` instance from above and get a completely unsurprising output:

```
You: Hello Kitty!
Kitty: Meow!
```

Dynamic polymorphism using trait objects is what makes this code work. If we passed in a dog instance (assuming we have coded a `Dog` type), we would get an output such as `Woof!` upon greeting a `Dog` instance. The `greet_pet` function calls the correct `sound(...)` member function of either the cat or the dog or any other type for which we chose to implement the `Pet` trait.

How does it know which `sound(...)` member function to call *at runtime*? Because remember, this will even work for a vector of random instances of trait objects implementing the `Pet` trait [2]. So the compiler has no way to call the correct method *at compile time* as would be the case if we had used generics. So what's the magic here?

A Peek Behind the Curtain: Vtables

A *vtable* [3], or virtual function table, is what makes the magic above work. We'll take a step by step look at what those are and how they help to accomplish this. Bear with me, I promise there will be a nice graphic long before this is all over.

Hidden Vtables

When we implement the `Pet` trait for `Cat`, the compiler generates a vtable instance, which is a hidden data structure that it puts into our program's binary. This is our `Pet`-vtable for `Cat`. There is also going to be a vtable instance for every other type that implements the `Pet` trait, meaning a `Pet`-vtable for `Dog`, for `Bird` and so on. Before we go any further let me emphasize that we are entering territory that is dangerous, evolving and not intended to be messed with. The

specifics of how all the things described in this post are implemented in the Rust compiler might change at any point in time [4].

We'll get to the specific layout of a vtable below, but for now suffice it to say that a vtable is a contiguous piece of storage in memory that is (mostly) an array of function pointers. The `Pet`-vtable for `Cat` contains function pointers to the implementations of the trait methods for `Cat`, while the `Pet`-vtable for `Dog` contains pointers to the implementations for `Dog`, and so on. And this helps us solve dynamic dispatch at runtime if, for every trait object `Box<dyn Pet>`, we keep track of which vtable is associated with a particular instance of a trait object. Let's now look at how that association is made.

Hidden Pointers to Vtables

A naive approach would be to store the whole vtable instance as part of a trait object. But that would be wasteful for multiple reasons: First of all this approach would add a number of pointer members to each instance, which will waste precious memory. Secondly, not every instance of a cat requires its own vtable instance. All trait related functions pointers of one type would point to the same functions anyways. Specifically, for our cat example the function pointer for `sound` will always point to the code for the `<Cat as Pet>::sound` function. That is true for all instances of `Cat`. Thus, we only need one vtable *per type*, so it makes sense to create one global instance of this vtable and refer to it through pointers. Both the Rust compiler and the commonly used C++ compilers do it like that, but there is a crucial difference in how they keep track of the *pointer* to the vtable. In C++, it is common to make the pointer to the vtable a hidden member of each instance of a class or struct [5]. The Rust compiler goes a different route and uses so called fat pointers [6].

Fat Pointers

Maybe you've heard of fat pointers in the context of slices, where a slice is really just a tuple of two elements [7]: the first element is the pointer to the beginning of the data and the second element is the length of the slice. But if you're like me you will be (or already were) surprised to learn that the pointer types `Box<T>`, `&T`,

and `&mut T` are different from the pointer types `Box<dyn Trait>`, `&dyn Trait`, and `&mut dyn Trait`. The latter are fat pointers. They, again, consist of two elements: their first element is the pointer to the actual data (the `T` instance) and the second is the pointer to the associated vtable instance (the `Trait`-vtable for type `T`).

(Fat) Pointer and Vtable Memory Layout

We now have all the pieces together to understand how pointers to trait objects work and how to mess with them. Before we start doing that though, let me summarize what we saw so far in a graphic:

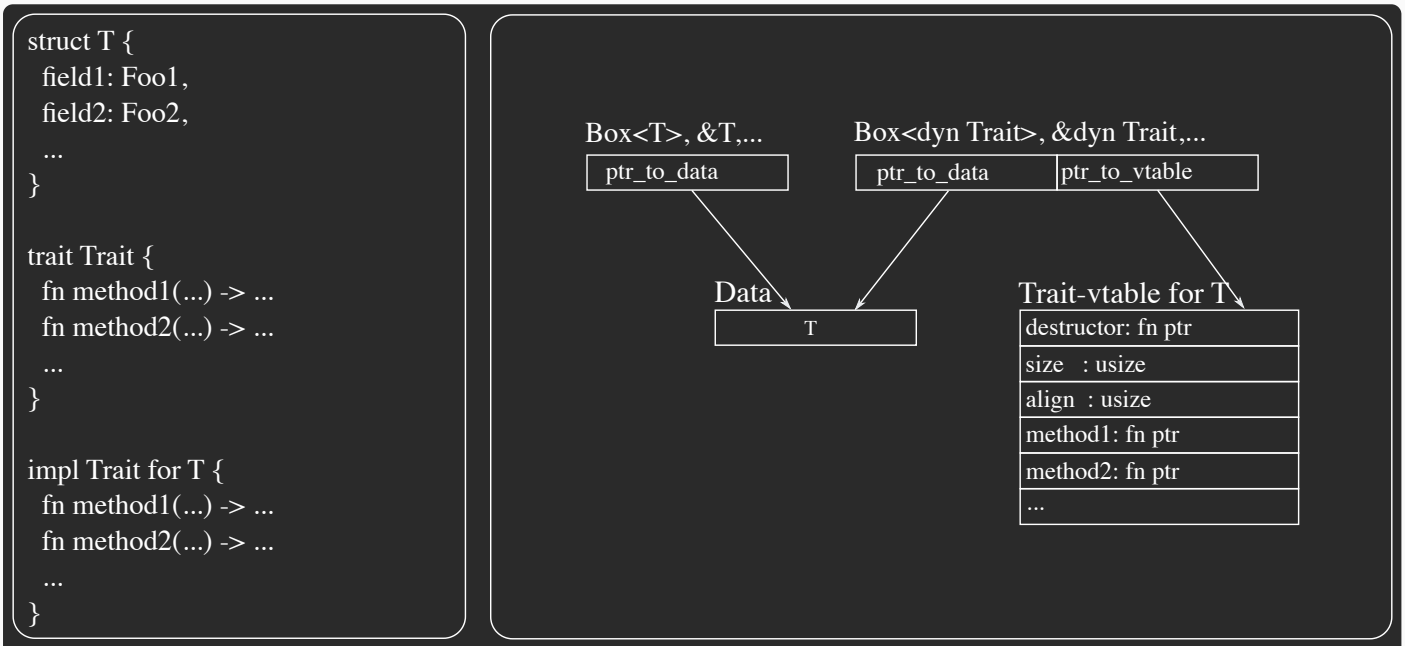

Figure 1. Pointer and vtable layout for a type implementing a single trait. Plain old pointers just store the address of the data. The pointers to dyn Trait objects store both the address of the data as well as a pointer to the vtable. There is one global vtable instance for each type T implementing trait Trait. Meaning all dynamic cats point to the same vtable, while all dynamic dogs point another one.

The figure was adapted from Ralph Levien's now out of date container cheat sheet. We see that the vtable contains a function pointer to the destructor, then `usize` fields for size and alignment, respectively, and finally the pointers to the member functions in order of declaration [8].

Some words of warning: this figure is accurate enough at the time of writing with the Rust compiler version 1.68.0, but it does not show the full picture. If supertraits get involved, the vtable gets more complicated to accomodate the

planned trait upcasting feature. The most comprehensive documentation on the current vtable layout I could find is here. But it also pays to look at the rustc source code along with this helpful answer on a reddit thread on the topic. So much for the nitty-gritty details, now let's try to get our hands dirty and venture even deeper into don't-try-this-in-prod territory.

Fun With Vtables

It's time to revisit the code from above and see if we can't make our kitty go woof. Let's first create a data structure for our vtable so that we can manipulate it with some more finesse.

```
#[repr(C)]
#[derive(Copy, Clone)]
struct PetVtable {
    drop : fn(*mut c_void),
    size : usize,
    align : usize,
    sound : fn(*const c_void) -> String,
    name : fn(*const c_void) -> String,
}
```

By making the struct `#[repr(C)]`, we make sure that it has the same memory layout as the `Pet`-vtable. For the function pointers we have basically copied the signature of the member functions, with one important difference. We have made the `&self` parameter a pointer to void, so that we can reuse the structure for any implementor of `Pet`. Still, we have *some* degree of type safety by choosing appropriate integer and function pointer types for the members. Now let's revisit the code from above and see how to mess with the vtable.

```
const POINTER_SIZE : usize = std::mem::size_of::<usize>();

fn main() {
    unsafe {
        // (1)
        let mut kitty : Box<dyn Pet> = Box::new(Cat::new("Kitty"));
```

```
        // (2)
        let addr_of_data_ptr = &mut kitty as *mut _ as *mut c_void as usiz
        // (3)
        let addr_of_pointer_to_vtable = addr_of_data_ptr + POINTER_SIZE;
        // (4)
        let ptr_to_ptr_to_vtable = addr_of_pointer_to_vtable as *mut *cons
        // (5)
        let mut new_vtable = **ptr_to_ptr_to_vtable;
        // (6)
        new_vtable.sound = bark;
        // (7)
        *ptr_to_ptr_to_vtable = &new_vtable;

        greet_pet(kitty);
    }
}

fn bark(_this : *const c_void) -> String {
    "Woof!".to_string()
}
```

This will produce a barking cat as teased above. Try this example on the playground.

```
You: Hello, Kitty!
Kitty: Woof!
```

Okay, now let's take this step by step: ① We create a pointer to a `Pet` trait object with a cat instance. From the section above we know that `kitty` consists of two elements right next to each other in memory: the pointer to the data and the pointer to the vtable. ② Now we find out the address of `kitty` and store it as an integer (because we can, since the value of a pointer is just an address). It's important to note that we really care about the address of `kitty` and *not* the address of the cat instance that it points to. The address of `kitty` is also the address of the data pointer. ③ Now we add one pointer size in bytes to this value, which will give us the address of the pointer to the vtable. ④ Finally we

cast this address into a pointer. Note that the type we now have is a *pointer to pointer* to `PetVtable`. If you are confused why we need two pointer indirections, bear with me. I'll explain later. ⑤ Here, we copy the vtable into a stack variable, which is possible because our vtable derives `Copy`. Since we have two pointer indirections, we have to dereference twice. ⑥ Next, we make the function pointer for the `sound` member function point to the `bark` function, which is just a free function that has the correct signature. It takes a void pointer as its first argument, which is the reference to `self`, aka the pointer-to-data part of the fat pointer. ⑦ Finally we make the pointer to the vtable point to the newly created vtable. When passed to the `greet_pet` function the kitty will now bark.

References or Boxes

The code above works just the same if we had used references instead of boxes. The memory layout for both pointer types is the same. The only difference is where the *pointed-to* element is stored.

(Im)mutable Vtables

In the code above, we used two pointer indirections to manipulate the vtable. We copied the existing vtable into a new one, manipulated the new vtable and then set the pointer-to-vtable to point to the new vtable. Couldn't we just grab a mutable pointer to the vtable itself and make the `sound` function point somewhere else?

Let's think about what it would mean if we could do that: we could change the vtable for *all* `Cat` instances in our program present and future, because only one vtable instance is created for the `Cat` type. All trait object instances point to the same vtable. If it was possible to manipulate it, then this could turn into all kinds of nightmares quickly (debugging, security, you name it). This is why the compiler places all vtables into a special *read only* section of the program binary, which will make it a runtime error to try to write to it. That's a good thing.

We're Not in Kansas Anymore

Before we end this adventure, let's try and break the whole thing some more. In the code above, we have made the function pointer members type safe. Granted, we *did* use pointers to void, but we *have* restricted the function signatures. Very reasonable, but why would we restrict ourselves like that? We've already been naughty and treated pointers as numbers, so let's alter our vtable structure a bit and remove any semblance of typesafety from it.

```
#[repr(C)]
#[derive(Copy, Clone)]
struct RawPetVtable {
    drop : usize,
    size : usize,
    align : usize,
    sound : usize,
    name : usize,
}
```

Now we can stick any address (any number really) into the function pointers. Let's now create two functions that do not obey the expected signature at all:

```
fn bark2() -> String {
    "Woof!".to_string()
}

fn add(a : usize, b : usize) -> String {
    format!("{} + {} = {}", a, b, a + b)
}
```

The `bark2` function barks but it takes no arguments. We expect the `sound` member function to be called with exactly one argument, which is the address of `self`. The `add` function is even wackier in this context. It takes not one but two arguments. The first argument is the address of self surely, but where does the second argument come from? Well, as Darth Vader once said to poor Lando: "I am altering the deal. Pray I don't alter it any further..." At least we give the program the expected return value!

I've got a link to the playground with the code here. There's no new surprises in there, so feel free to explore the code for yourself. Let's take a look at the outputs. If we assign the `bark2` function address to the `sound` member of the vtable we get the following output:

```
You: Hello, Kitty!
Kitty: Woof!
```

That's surprising, isn't it? Not the `"Woof!"`, we're already used to that. However, the fact that the program even runs without crashing might surprise us. The code just works even though the function signature is missing an argument. Now let's look at what happens if we put the `add` function address into the `sound` member. That output is not deterministic but it will look something like this:

```
You: Hello, Kitty!
Kitty: 93842120210704 + 93842100797168 = 187684221007872
```

We learn two things. Our kitty is surprisingly good at math and the program still runs. Good grief, why? Well the thing is: function pointers are just addresses to code and code is just ones and zeros. Typesafety is an illusion that exists while the program is being compiled but not after. The CPU just executes the instructions that it was told to jump to. Those will just go ahead and operate on the data that is found at the location of the function arguments, whether someone put something useful in there or not.

Conclusion

As promised we took a deep dive into one particular aspect of trait objects, but in many respects we only scratched the surface. As promised, we did go down some rabbit holes to remind us that, as programmers, we really just live in a world of ones and zeros with a whole lot of sugar poured on top of it [9].

I know there is some confusion around traits and trait objects and I wish I could say that this deep dive cleared all (or even any) of that up, but I don't think that's

the case. If anything this probably left you more confused... all I can hope for is that some fun was had while reading this. I sure had fun writing it.

Endnotes

1. I mean that C++ is a *more* object-oriented (OO) language than Rust, not that C++ is a purely an OO language. Further, I don't want to imply that Rust is an OO language *at all*. ↩

2. Not every trait in Rust can be made into a trait object. The key concept here is object safety. In this article, we are only concerned with object safe traits. ↩

3. Also called virtual *method* tables, but they also go by many other names. Pronounced "veeh-table". Vtables are commonly used in C++ compilers for dispatching to the method of a derived class via a pointer to its base class. The `virtual` keyword plays an important role in dynamic dispatch via inheritance in C++, hence the V in vtable. Our animal example in C++ would be calling the `Cat::sound` member function via a pointer to super class `Pet`, where the `Cat` class derives from `Pet`, which has a `virtual` member function `sound()`. I'll leave it at that for now and I urge anyone interested in the C++ aspects to check out the aforementioned video on the Creel YouTube channel. ↩

4. It's also important that vtables aren't really a *language feature* but an *implementation detail* of the compiler, as reddit user u/myrrlyn pointed out here. It might change in future versions of rustc and might be even different in other rust compilers, once they become available. ↩

5. The vtable pointer in C++ may e.g. be placed at the beginning of an object. This is why you must not rely on the fact that the address of an object is also the address of its first member in C++. ↩

6. I'm not going to pretend to understand *why* Rust chose fat pointers over thin ones, but if you are interested, here and here are some insightful discussions on the topic. ↩

7. A *twople*,... get it? Sorry about that... ↩

8. If you're wondering why a vtable contains size and alignment, you're not alone. Let me also link a nifty crate that takes on the very niche problem of providing `&dyn Fn` but without the vtable. ↩

9. Even those ones and zeros are abstractions the physical realities involving electrons. ↩

pointers rust trait vtable

Comments

- zeenix:

  Great post!

I think the "former" and "latter" are inverted in the paragraph about pointers and fat pointers? 🤔

> *from from the pointer types Box, &dyn Trait, and &mut dyn Trait. While the former are really just pointers 8, the latter are also fat pointers.*

Also I not noticed a typo: extra "from".

(#1483823338 - 2023-03-25T13:28:27Z)

- geo-ant:

  Thank you kindly @zeenix. I removed the duplicate from. I'm not a native speaker and I looked up the former / latter thingy again and I believe I used it correctly. If you think the phrasing is unclear, I'll try and come up with something better.

  (#1483923323 - 2023-03-25T21:12:10Z)

- zeenix:

  You got it! I think the confusing bit (for me at least) was that Box isn't a pointer. Maybe it's just my C background overshadowing my Rust knowledge but I think only references can be called pointers. 🤔

  (#1483943051 - 2023-03-25T23:06:47Z)

- zeenix:

  In any case this small correction will make things clearer: "former are" -> "former is".

  (#1483943325 - 2023-03-25T23:08:25Z)

- geo-ant:

  > *You got it! I think the confusing bit (for me at least) was that Box isn't a pointer. Maybe it's just my C background overshadowing my Rust knowledge but I think only references can be called pointers.* 🤔

  Hey, I'll think of a way to rephrase that parapraph, I think it's confusing after all.

  One more thing though. And I am honestly not trying to argue with you, just trying to understand your point and I really appreciate that you took the time to comment. Why do you not consider Box a pointer? The rust doc calls it a pointer type. Is it because it contains more members than just the raw pointer (i.e. the allocator)?

(#1484170604 - 2023-03-26T17:50:27Z)

- zeenix:

> *One more thing though. And I am honestly not trying to argue with you, just trying to understand your point and I really appreciate that you took the time to comment.*

Don't worry about it. It'd be perfectly fine to argue. :) I try not to take things personally. We're all just learners here w/ different way of looking at things.

> *Why do you not consider Box a pointer? The rust doc calls it a pointer type. Is it because it contains more members than just the raw pointer (i.e. the allocator)?*

No. I guess it boils down to the fact that I think there is a difference between pointer type and pointer. I'd consider Box (as well as Arc, Rc, Cell etc) pointer types but not pointers. E.g here:

```
struct Str<'a> {
    pointer: &'a str,
}
```

`

I'd call `Str` a pointer type but not a pointer, while I'd call `Str::pointer` a pointer. I hope that clarifies things. :+1:

(#1484207877 - 2023-03-26T20:05:02Z)

- geo-ant:

I understand and that makes sense. I'll try to rephrase the paragraph, but I still have to consolidate how I think about these things. Thanks for taking the time to discuss this. I hope the Rust community will stay as welcoming and positive even as it grows more :)

(#1485824076 - 2023-03-27T20:31:15Z)

You can comment on this post using your GitHub account.

Join the discussion for this article on this ticket. Comments appear on this page instantly.