WHY THERE IS NO FUNCTIONS OVERLOADING IN RUST?

Naive answer: because Rust is feature poor.

WHY THERE IS NO FUNCTIONS OVERLOADING IN RUST?

This is idiomatic decision. In languages with some kind of functions overloading it is common to have spammed multiple functions with same name, but doing very much different things (or just additional not obvious work) depending on given argument set.

WHY THERE IS NO FUNCTIONS OVERLOADING RUST?

Rust community decision was that if the function name should describe its behaviour, then only single behaviour should be described by single name. If the behaviour is possible to perform on multiple types, then it should be generalized out (using for exaple generic types or dynamic dispatch).

WHAT IS A TRAIT?

There is no problem with understanding that trait is encapsulating behaviour, but how that differs from interfaces (or C++ abstract classes)?

WHAT IS A TRAIT?

The core difference is relation between trait/interface and type. Making things simple: types are not directly related to any trait by they own. In contrast interfaces are bound to type in the place where type is defined, and the relation is rock solid.

WHAT IS A TRAIT?

CONSEQUENCE 1

In contrast to interface, when type `Foo` implementing trait `Bar` is imported, it doesn't give access to `Bar` implementation of `Foo` - `Bar` trait has to be imported on its own.

WHAT IS A TRAIT?

CONSEQUENCE 2

Dependency simplification. Consider two library crates: `Foo` and `Bar`. The `Foo` crate provides trait `Foo::Foo`, and the `Bar` crate provides `Bar::Bar` type, which has some implementation of `Foo::Foo` trait. However the user of the `Bar` crate doesn't need the `Foo` functionality at all, so he can use `Bar::Bar` without the trait implementation (using feature flags). It is commonly used to opt-in heavy `serde` dependency.

WHAT IS A TRAIT?

CONSEQUENCE 3

Deferred trait implementation. Consider previous example - the `Bar::Bar` implementator doesn't even need to know about existence of trait `Foo::Foo`, the implementation can be done by trait creator. This is also commonly used and know as "extension trait" pattern - the example is `Itertools` crate which using this technique adds additional functionality to `Iterator` trait.

WHAT IS A TRAIT?

CONCLUSION

Trait is more like a typeclass in Haskell, than an interface from OOP world.

POLYMORPHISM, TRAIT OBJECTS

Is there even polymorphism in Rust? Can be Rust OOP if it doesn't have interfaces?

POLYMORPHISM, TRAIT OBJECTS

WHAT IS A POLYMORPHISM?

Polymorphism is possibility to perform some behaviour on an object without knowing its exact type. In most mainstream languges it is achieved using interfaces, but it is not the only way.

POLYMORPHISM, TRAIT OBJECTS

POLYMORPHISM KINDS

Some people distinguish static polimorphism and dynamic polimorphism - the first one, when the type is actually know by compliler in compile time, but not expressed directly in signature, and the second, when type is unknown until the actuall runtime.

POLYMORPHISM, TRAIT OBJECTS

STATIC POLYMORPHISM

Static polymorphism is a simpler one, but applies only on compiled languages - in C++ it is handled with Templates (and in C++20 with Concepts). In Rust this is handled with Traits and generic types (commonly in combination with `impl Trait` syntax), and is called Monomorphisation (so for programmer it seems like type is Polymorphic, but compiler makes it Monomorphic at the end). This doesn't bring problems to people.

POLYMORPHISM, TRAIT OBJECTS

DYNAMIC POLYMORPHISM

Dynamic polymorphism is situation, when the type is not know even in runtime directly, so behaviour have to be provided next to data. Commonly it's solved by passing reference or pointer to the inteface (abstact class in C++), which contains both data, and interface implementation (via RTTI or vtables).

POLYMORPHISM, TRAIT OBJECTS

DYNAMIC POLYMORPHISM IN RUST

In Rust dynamic polimorphism is tackled using combination of Traits and Trait objects. The trait object is a type which is not exactly known, but guarantees to implement some trait, and its expressed with `dyn Trait` (so `dyn Iterator` is some type implementing an iterator). The issue is, that because the type has unknown size (it is not know at the end) it has to be hidden over some pointer (`Box` in most cases).

POLYMORPHISM, TRAIT OBJECTS

CONSEQUENCE

In contrast to C++, Rust solves both types of polymorphism via same entity called Trait. The type (or trait) implementator delivers single implementation, and it is up to user which one would he use at the end.

DROP AND MEMORY GUARANTEES

Drop is just a destructor.

DROP AND MEMORY GUARANTEES

No, it is not.

DROP AND MEMORY GUARANTEES

The most difference is a guarantees - destructors are guaranteed to be called when object goes out of scope. In Rust there us no such guarantee - object can be easly `std::forget` (or `Box:::leak`ed). This is perfectly safe and valid.

DROP AND MEMORY GUARANTEES

CONSEQUENCE

Rust does not prevent from memory leaks. It is not part of Rust memory safety guarantees.

DROP AND MEMORY GUARANTEES

RUST MEMORY GUARANTEES

Rust guarantees, that using safe code it is impossible to introduce exploitable memory bugs. Memory leak is not exploitable on its own. Exploitable are:

- Data races
- Dereferencing unowned memory (which might cause SEGFAULT, but not always do)
- Use after free

MOVE BY DEFAULT

Does that mean, that its equivalent of calling `std::move` on everything?

MOVE BY DEFAULT

No, it doesn't.

MOVE BY DEFAULT

COPY TRAIT

"By default" doesn't mean "always". There is a `Copy` trait which means, that obects of type implemented it are copied instead of moved.

MOVE BY DEFAULT

RUST MOVE IS NOT C++ STD::MOVE

Rust move is a semantic move - whatever is moved out, then in worst case `memcpy` is invoked, and noone in the scope has access to the object. It is assumed, it would be droped later, whoever received ownership of moved object. In C++ move involves calling move constructor (which might do much more than `memcpy`) and destructor. In Rust, *nothing* is dropped while moving.

LIFETIME

What is lifetime? Is it just a scope in which given variable lifes?

LIFETIME

The valid answer would be: no, but good topic for entire talk (or rather book).

LIFETIME

Lifetime is a scope, which depending on context:

- Is guaranteed to be outliven by an object
- Constraints an object to not outlive it

LIFETIME

GUARANTEE

```rust
fn foo<'a>(arg: &'a u32) {
    // ...
}
```

It is guaranted, that whatever is a lifetime 'a, arg would outlife it.

LIFETIME

CONSTRAINT

```
let obj: &'a = ...;
```

It is required, that whatever is a lifetime `'a`, `obj` will not outlife it.

LIFETIME

COMBINED

```
fn foo(arg: &'a u32) -> &'a u32 { /* ... */ }
```

It is guaranteed, taht whatever `arg` is given, the result of a function would not outlife an `arg`.

LIFETIME

LIFETIME AS GENERICS ARE CONSIDERED AS CONSTRAINTS

```
struct<'a> Foo<'a> {
    field: &'a u32,
}
```

Whatever lifetime `'a` is, it is guaranteed, that a `field` cannot outlive it, and in consequence whole object of struct `Foo` cannot outlife it.

COPY TRAIT

Copy trait is a solution for case when there are problems with moving an objects:

```rust
#[derive(Debug)]
struct X(u32);

fn main() {
    let x = X(5);

    (|| drop(x))();

    println!("{:?}", x);
}
```

COPY TRAIT

No, its not. My opinion is, that Rust compiler makes bad job here giving it as a hint:

```
|           - move occurs because `x` has type `X`,
|             which does not implement the `Copy` trait
```

COPY TRAIT

Copy is a performance related trait - if copying type is as cheap as copying reference, there is no reason to borrowin the type, and in such cases `Copy` should be implemented. If copying is heavy, then only `Clone` trait should be implemented, and explicit `.clone()` should be called to perform expesive copy.

UNPIN TRAIT

The general problem occurs, when one have async function like this:

```rust
async fn bar<S: Stream>(stream: S) {
    while let Some(n) = stream.next().await {
        // ...
    }
}
```

UNPIN TRAIT

And the problem is again a solution proposed by compiler is generic, but not the best one:

```
error[E0277]: the trait bound `S: std::marker::Unpin`
             is not satisfied
 --> src/main.rs:4:32
  |
3 | async fn bar<S: Stream>(stream: S) {
  |              -- help: consider further restricting
  |                 this bound: `S: std::marker::Unpin +`
4 |     while let Some(n) = stream.next().await {
  |                                ^^^^ the trait
  |                 `std::marker::Unpin` is not implemented
  |                 for `S`
```

UNPIN TRAIT

The problem is, that people doesn't understand what is an `Unpin` trait, so instead of considering how to make a type `Unpin`, they just put type bound:

```rust
async fn bar<S: Stream + Unpin>(stream: S) {
    while let Some(n) = stream.next().await {
        // ...
    }
}
```

UNPIN TRAIT

`Unpin` trait means, that it doesn't matter if it is pinned to memory, or moved. The only types which are not `Unpin`, are those, which might contain self reference (so the types generated by `async` function). Self referenced types care if they are moved in memory.

UNPIN TRAIT

But how to make type which cares if it is moved in memory to not care anymore? You can just pin it! If type cannot be moved in memory, then it nevermore cares if it will be! The better solution for this case would be:

```
async fn bar<S: Stream>(stream: S) {
    futures::pin_mut!(stream);
    while let Some(n) = stream.next().await {
        // ...
    }
}
```

UNPIN TRAIT

Sometimes it is not enaugh to pin a type to stack, because it actually need to transfer ownership, in this case it is possible to pin it to heap:

```rust
async fn bar<S: Stream>(stream: S) {
    Box::pin(stream);
    while let Some(n) = stream.next().await {
        // ...
    }
}
```

UNPIN TRAIT

CARE ABOUT ZERO-COST

The issue with pinning to heap with `Box` is, that it introduces overhead, so in this case it might be better, to add a trait bound. If user would have type which is already `Unpin`, then he would just pass it - in other case, he would `Box` it himself. It is also a benefit of pinning to stack when possible - if `Unpin` bound is introduced, calee would need to `Box` any not `Unpin` argument - but it is often not needed.

SYNC TRAIT

As `Send` trait is commonly understanded - it is implemented for types which can be sended between threads, the `Sync` trait seems to be an issue. The common sense is "marks type, which can be somehow shared between threads, but tbh I have no idea why I cannot share a type between threads".

SYNC TRAIT

The actual misunderstanding is: "are there any non-sync types?". The answer is: yes, any type, which allows unsynchronized internal mutability. Two most common safe non `Sync` types are:

- `Cell`
- `RefCell`

SYNC TRAIT

The reason why it is often missed is that internal mutability should be avoided in general, so even if one used it, he might just not correlate facts. But why is that not safe for those types to be `Sync`? Those types allows to mutate internals without synchronization, having only shared reference. So sharing reference to `RefCell` between threads would allow to mutate same object from two different thread in the same time.

ASSOCIATED VS GENERIC TYPES

I see there is a problem with distinguishing difference between associated and generic types in traits.

ASOCIATED VS GENERIC TYPES

GENERICS

```
trait<T> Foo<T> {
    // ...
}
```

Note, that the Foo trait can be implemented multiple times for single type Bar, every time with different T. T is a parameter of implementation.

ASSOCIATED VS GENERIC TYPES

GENERICS

```
trait Foo {
    type T;
}
```

Note, that the `Foo` trait can be implemented once for single type `Bar`, and `T` is fixed. `T` is a consequence of implementation.

ASSOCIATED VS GENERIC TYPES

General rule of thumb - use generic, when it makes sense to have implemented the trait with multiple generic types which are implementation parameters. Use associated type when T has strong connection to a type for each trait implementation, so there is no point to have implementation with different types.

ASSOCIATED VS GENERIC TYPES

REAL LIFE EXAMPLE

In futures crate, there are two complement traits, but implemented with different approach:

- `Stream` trait has an associated `Stream::Item` type, because whatever is a stream of types, should be a stream of uniform types.
- `Sink<T>` is a type which is generic over input, because it might make sense, that given "output pipe" can accept different types of objects, as long as it can later uniform them.

THANK YOU

FIND ME ON:

- Rust Wrocław: www.rust-wroclaw.pl
- Github: www.github.com/hashedone