Lecture 2: Traits

Barinov Denis

February 28, 2023

barinov.diu@gmail.com

Operator ?

Consider the following structure:

```
struct Info {
    name: String,
    age: i32,
}
```

Operator ?

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = match File::create("my_best_friends.txt") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };
    if let Err(e) = file
        .write_all(format!("name: {}\n", info.name)
        .as_bytes()) {
        return Err(e)
    }
    if let Err(e) = file
        .write_all(format!("age: {}\n", info.age)
        .as_bytes()) {
        return Err(e)
    }
    Ok(())
}
```

Operator ?

We can use the ? operator to make the code smaller!

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt")?;
    file.write_all(format!("name: {}\n", info.name).as_bytes())?;
    file.write_all(format!("age: {}\n", info.age).as_bytes())?;
    Ok(())
}
```

Beautiful, isn't it?

We can use it for `Option` too!

In this lecture

- Traits
- Exotically Sized Types
- Standard library traits

Traits

Traits

Traits

In Rust, a trait is *similar to* an interface in other languages. It's the way how we can define *shared behavior* (i.e similarities between objects).

```rust
pub trait Animal {
    // No 'pub' keyword
    fn name(&self) -> String;
    fn noise(&self) -> String;
}
```

Traits

```rust
pub trait Animal {
    fn name(&self) -> String;
    fn noise(&self) -> String;

    // Traits can provide default method definitions
    fn talk(&self) {
        println!("{} says {}", self.name(), self.noise());
    }
}
```

Let's define some structures and implement this trait for them.

Traits

```rust
pub struct Sheep {
    name: String,
}

impl Animal for Sheep {
    fn name(&self) -> String {
        self.name.clone()
    }

    fn noise(&self) -> String {
        "baaaaah!".to_string()
    }
}
```

Traits

Usage example:

```
let sheep = Sheep {
    name: "Dolly".to_string(),
};
assert_eq!(sheep.name(), "Dolly");
sheep.talk();  // prints 'Dolly says baaaaah!'
```

Traits

```rust
pub struct Dog {
    name: String,
}

impl Animal for Dog {
    fn name(&self) -> String { self.name.clone() }

    fn noise(&self) -> String {
        "ruff!".to_string()
    }

    // Default trait methods can be overridden.
    fn talk(&self) {
        println!("Ruff! Don't call me doggo");
    }
}
```

Traits: `where` keyword

And here we'll have some troubles:

```
pub trait Animal {
    fn name(&self) -> String;
    fn noise(&self) -> String;

    fn talk(&self) {
        // Note: this clones &Self, not Self!
        // let cloned = self.clone();

        let cloned = (*self).clone();
        println!("{} says {}", self.name(), self.noise());
    }
}
```

Traits: `where` keyword

And here we'll have some troubles:

```rust
pub trait Animal {
    fn name(&self) -> String;
    fn noise(&self) -> String;

    fn talk(&self) {
        // error: no method named `clone` found for
        // type parameter `Self` in the current scope
        let cloned = (*self).clone();
        println!("{} says {}", self.name(), self.noise());
    }
}
```

Traits: `where` keyword

To add bounds to the type, use `where` keyword.

```
pub trait Animal
where
    Self: Clone
{
    fn name(&self) -> String;
    fn noise(&self) -> String;

    fn talk(&self) {
        // Compiles just fine!
        // Note: this clones Self, not &Self!
        let cloned = self.clone();
        println!("{} says {}", cloned.name(), cloned.noise());
    }
}
```

By default, Rust doesn't expect anything* from types! You should provide bounds.

Traits: `where` keyword

If we'll try to compile `Sheep` and `Dog` types, we'll see errors from the compiler:

```
error[E0277]: the trait bound `Sheep: Clone` is not satisfied
  --> src/main.rs:22:6
   |
22 | impl Animal for Sheep {
   |      ^^^^^^ the trait `Clone` is not implemented for `Sheep`
   |
note: required by a bound in `Animal`
  --> src/main.rs:3:11
   |
1  | pub trait Animal
   |           ------ required by a bound-in this
2  | where
3  |     Self: Clone
   |           ^^^^^ required by this bound in `Animal`
```

Traits: `where` keyword

You can also write trait bounds in generics:

```
trait Strange1<T: Clone + Iterator>
where  // You're not able to do this without 'where'!
    T::Item: Clone,
{
    fn smth(x: T);
}

trait Strange2<T>
where
    T: Clone + Iterator,
    T::Item: Clone,
{
    fn smth(x: T);
}
```

Note that you can add trait bounds only to generics with `where`!

Supertraits

Rust doesn't have "inheritance", but you can define a trait as being a superset of another trait:

```rust
trait Shape { fn area(&self) -> f64; }
trait Circle : Shape { fn radius(&self) -> f64; }
```

Same as:

```rust
trait Shape { fn area(&self) -> f64; }
trait Circle where Self: Shape { fn radius(&self) -> f64; }
```

It's an example of declaring Shape to be a supertrait of Circle.

Supertraits

We can use methods from another trait:

```
trait Shape { fn area(&self) -> f64; }

trait Circle: Shape {
    fn radius(&self) -> f64 {
        (self.area() /std::f64::consts::PI).sqrt()
    }
}
```

Supertraits

Usage:

```
fn print_area_and_radius<C: Circle>(c: C) {
    // Here we call the area method from
    // the supertrait `Shape` of `Circle`.
    println!("Area: {}", c.area());
    println!("Radius: {}", c.radius());
}
```

Fully Qualified Syntax

What if types have multiple methods named the same way, and Rust cannot understand what method to call?

```rust
struct Form {
    username: String,
    age: u8,
}

trait UsernameWidget {
    fn get(&self) -> String;
}

trait AgeWidget {
    fn get(&self) -> u8;
}
```

Fully Qualified Syntax

```
impl UsernameWidget for Form {
    fn get(&self) -> String {
        self.username.clone()
    }
}

impl AgeWidget for Form {
    fn get(&self) -> u8 {
        self.age
    }
}
```

Fully Qualified Syntax

Let's try to call `get`:

```
let form = Form {
    username: "rustacean".to_owned(),
    age: 28,
};

println!("{}", form.get());
```

Fully Qualified Syntax

```
error[E0034]: multiple applicable items in scope
  --> src/main.rs:35:25
   |
35 |     println!("{}", form.get());
   |                         ^^^ multiple `get` found
   |
note: candidate #1 is defined in an impl of the trait `UsernameWidget`
for the type `Form`
  --> src/main.rs:15:5
   |
15 |     fn get(&self) -> String {
   |     ^^^^^^^^^^^^^^^^^^^^^^^
note: candidate #2 is defined in an impl of the trait `AgeWidget`
for the type `Form`
  --> src/main.rs:21:5
   |
21 |     fn get(&self) -> u8 {
   |     ^^^^^^^^^^^^^^^^^^^
```

Fully Qualified Syntax

To solve the problem, one can call the method from a trait.

```rust
let form = Form {
    username: "rustacean".to_owned(),
    age: 28,
};

// println!("{}", form.get());

let username = UsernameWidget::get(&form);  // From trait
assert_eq!("rustacean".to_owned(), username);
let age = <Form as AgeWidget>::get(&form);  // FQS
assert_eq!(28, age);
```

Fully Qualified Syntax

```rust
let username = UsernameWidget::get(&form);
let age = <Form as AgeWidget>::get(&form);
```

Fully Qualified Syntax

```
let username = UsernameWidget::get(&form);
let age = <Form as AgeWidget>::get(&form);
```

Actually, this one is called Fully Qualified Syntax (previously called universal function call syntax), and it's the most generic way of using methods.

Fully Qualified Syntax

```rust
let username = UsernameWidget::get(&form);
let age = <Form as AgeWidget>::get(&form);
```

Actually, this one is called Fully Qualified Syntax (previously called universal function call syntax), and it's the most generic way of using methods.

The angle bracket can be omitted if the type expression is a simple identifier (as in the first line), but is required for anything more complex. The syntax `<T as Trait>` means that we require that `T` implements the trait `Trait`, and the method after the double colon refers to a method from that trait implementation.

`impl` keyword

What if you need to accept any type that implements some trait? You can do the following:

```rust
fn func<T: MyTrait + Clone>(input: T) {
    // ...
}
```

`impl` keyword

...Or use special syntax!

One argument:

```
fn func(input: impl MyTrait + Clone) {
    // ...
}
```

Two arguments:

```
fn func(input: &impl MyTrait, output: &impl MyTrait) {
    // ...
}
```

One complex argument:

```
fn func(input: &(impl MyTrait + Clone)) {
    // ...
}
```

`impl` keyword

```
fn func<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 { ... }
```

Same as:

```
fn func<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{ ... }
```

Multiple `impl`

What if we want to implement additional methods for a type depending on if it has an implementation of some trait?

```rust
pub enum Option<T> {
    // ...
}

impl<T> Option<T> {
    // ..
}

impl<T> Option<T>
where
    T: Default
{
    // ...
}
```

`where` and selection

We can implement methods depending on whether the type has implementations of some traits.

```rust
pub enum Option<T> {
    // ...
}

impl<T> Option<T> {
    pub fn unwrap_or_default<T>(self) -> T
    where
        T: Default
    {
        // ...
    }
}
```

Exotically Sized Types

Exotically Sized Types

Most of the time, we expect types to have a statically known and positive size. This isn't always the case in Rust!

Currently, types can be:

- "Regular" (no formal name as far as lecturer knows)
- Dynamically Sized Types, DST
- Zero Sized Types, ZST
- Empty Types

Dynamically Sized Types

Dynamically Sized Type is a type which size is unknown at compile time.

There are only two kinds of DST's:

- Slices, either regular such as `[u8]` and `str`.
- Trait objects, such as `dyn Trait`.

Dynamically Sized Types

Dynamically Sized Type is a type which size is unknown at compile time.

There are only two kinds of DST's:

- Slices, either regular such as `[u8]` and `str`.
- Trait objects, such as `dyn Trait`.

Such types **do not** implement `Sized` marker trait. **By default, Rust "implements" it for all types it can!**

```rust
pub trait Sized {}
```

Dynamically Sized Types: Slices

Remember: Rust has strict type system. For instance, types `T` and `&T` are **different**.

All that time we've written `&str` instead of just `str` and that's for reason! Since the size of slice is not known at compile time, `str` is **unsized**, and it's a separate type.

Basically, `&str` is just a pointer to the beginning of the slice and its length, and it means the reference to the slice is sized.

The same stands true for `[u8]`, `[i64]` and others.

Dynamically Sized Types: Trait objects

Consider the following code:

```
trait Hello {
    fn hello(&self);
}

fn func(arr: &[Hello]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Dynamically Sized Types: Trait objects

Consider the following code:

```rust
trait Hello {
    fn hello(&self);
}

fn func(arr: &[Hello]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

No, since the compiler doesn't know which size every object that implements `Hello` have and therefore cannot put them in the slice.

Dynamically Sized Types: Trait objects

Consider the following code:

```rust
fn func<T: Hello>(arr: &[T]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Dynamically Sized Types: Trait objects

Consider the following code:

```
fn func<T: Hello>(arr: &[T]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Yes, since compiler knows which size every object have. It will generate unique instance of function for every `T`.

Dynamically Sized Types: Trait objects

Consider the following code:

```
fn func<T: Hello>(arr: &[T]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Yes, since compiler knows which size every object have. It will generate unique instance of function for every `T`.

But what if we need an array of objects that implement `Hello`?

Dynamically Sized Types: Trait objects

```rust
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

Dynamically Sized Types: Trait objects

```rust
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

- Keyword `dyn` creates a **trait object**: some object that implements `Hello`.

Dynamically Sized Types: Trait objects

```
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

- Keyword `dyn` creates a **trait object**: some object that implements `Hello`.
- `dyn Hello` is also an **unsized** type, since we don't know the size of the object that implements it.

Dynamically Sized Types: Trait objects

```
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

- Keyword `dyn` creates a **trait object**: some object that implements `Hello`.
- `dyn Hello` is also an **unsized** type, since we don't know the size of the object that implements it.
- `&dyn Hello` consists of pointer to the structure and the pointer to the *virtual table*, and it's sized. This reference is called **fat pointer**.

Dynamically Sized Types: Trait objects

Trait objects can be stored at any pointers:

```
impl Hello for &str {
    fn hello(&self) {
        println!("hello &str!");
    }
}

let x = "hello world";
let r1: &dyn Hello = &x;
let r2: Box<dyn Hello> = Box::new(x.clone());
```

Dynamically Sized Types: Trait objects

You cannot require more than one **non-auto** trait in trait objects, use supertraits instead.

```
let x = "hello world";

let r: &dyn Hello + World = &x; // World is some regular user trait,
                                // it won't compile!

trait HelloWorld: Hello + World {}
impl HelloWorld for &str {
    // ...
}

// Will compile just fine
let r: &dyn HelloWorld = &x;
```

Dynamically Sized Types: Trait objects

But you can require additional auto traits:

```rust
trait X {
    // ...
}

fn test(x: Box<dyn X + Send>) {
    // ...
}
```

Auto traits: Send, Sync, Unpin, UnwindSafe, and RefUnwindSafe.

Trait objects: object safety

Ok, let's compile the following code:

```
fn test(x: Box<dyn Clone + Send>) {
    // ...
}
```

Trait objects: object safety

```
error[E0038]: the trait `Clone` cannot be made into an object
 --> src/main.rs:1:16
  |
1 | fn test(x: Box<dyn Clone + Send>) {
  |                ^^^^^^^^^^^^^^^^^ `Clone` cannot be made
  |                                  into an object
  |
  = note: the trait cannot be made into an object because it
  requires `Self: Sized`
  = note: for a trait to be "object safe" it needs to allow
  building a vtable to allow the call to be resolvable dynamically
```

Trait objects: object safety

A trait is object safe if it has the following qualities:

- All supertraits must also be object safe
- Sized must not be a supertrait. In other words, it must not require Self: Sized
- It must not have any associated constants
- It must not have any associated types with generics
- About associated functions...

We can implement methods for trait objects!

```
impl dyn Example {
    fn is_dyn(&self) -> bool {
        true
    }
}

struct Test {}
impl Example for Test {}

let x = Test {};
let y: Box<dyn Example> = Box::new(Test {});

x.is_dyn() // Won't compile
y.is_dyn();
```

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.
- But when there are many types using generic function, code becomes bigger and CPU cannot fit it all into memory.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.
- But when there are many types using generic function, code becomes bigger and CPU cannot fit it all into memory.
- In this case, trait objects will be faster since single implementation would fit into cache line.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.
- But when there are many types using generic function, code becomes bigger and CPU cannot fit it all into memory.
- In this case, trait objects will be faster since single implementation would fit into cache line.
- **Answer**: only profiling can really help you with this question.

Standard library traits

Just a bit information about macros

In the first lecture, we mentioned that macros are a way of code generation in Rust. We can also use or even write a macro that will generate an implementation of trait automatically - `derive`.

Such type of macros is called **procedural macros**, whereas macros such as `println!` are **declarative**.

We'll discuss this in more detail a little bit later.

Default

Creates some default instance of T. Has a #[derive(Default)] macro.

```
pub trait Default {
    fn default() -> Self;
}
```

Many types in Rust have a constructor. However, this is specific to the type; Rust cannot abstract over "everything that has a `.new()` method".

To allow this, the `Default` trait was conceived, which can be used with containers and other generic types (e.g. `Option::unwrap_or_default()`).

Question: why this trait is not derived by default?

Clone

A trait for the ability to explicitly duplicate an object. Has a `#[derive(Clone)]` macro.

```
pub trait Clone {
    fn clone(&self) -> Self;

    // Note the default implementation!
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

`Copy`

Types whose values can be duplicated simply by copying bits. Has a `#[derive(Copy)]` macro.

It's **marker trait** and exists only to show the compiler that the type is special and can be copied by just copying bits of type representation.

```
pub trait Copy: Clone {}
```

By default, variable bindings have **"move semantics"**. However, if a type implements `Copy`, it instead has **"copy semantics"**.

Summary

- Traits syntax
- Dyn, impl keywords
- Some standard traits

Questions?