Rust Fundamentals

Basics of Rust
Part 8

AZZAM S.A

RustCourse
Nov. 24th, 2023

Azzam S.A

OSS devotee, speaker, and teacher.

Open sourceror. Namely Rust, Python, and Emacs.

azzamsa azzamsyawqi azzamsa.com

Follow Along!

- Rust Playground
- Exercises

▶ Show hints

Traits

A way to abstract over types . They’re similar to interfaces in other languages.

```rust
trait Pet {
    fn name(&self) -> &str;
}

struct Dog {
    name: String,
}

struct Cat;

impl Pet for Dog {
    fn name(&self) -> &str {
        &self.name
    }
}

impl Pet for Cat {
    fn name(&self) -> &str {
        "The cat" // No name, cats won't respond to it anyway.
    }
}
```

```
fn greet(pet: &dyn Pet) {
    println!("Who's a cutie? {} is!", pet.name());
}

fn main() {
    let fido = Dog { name: "Fido".to_string() };
    greet(&fido);

    let floof = Cat;
    greet(&cfloof);
}
```

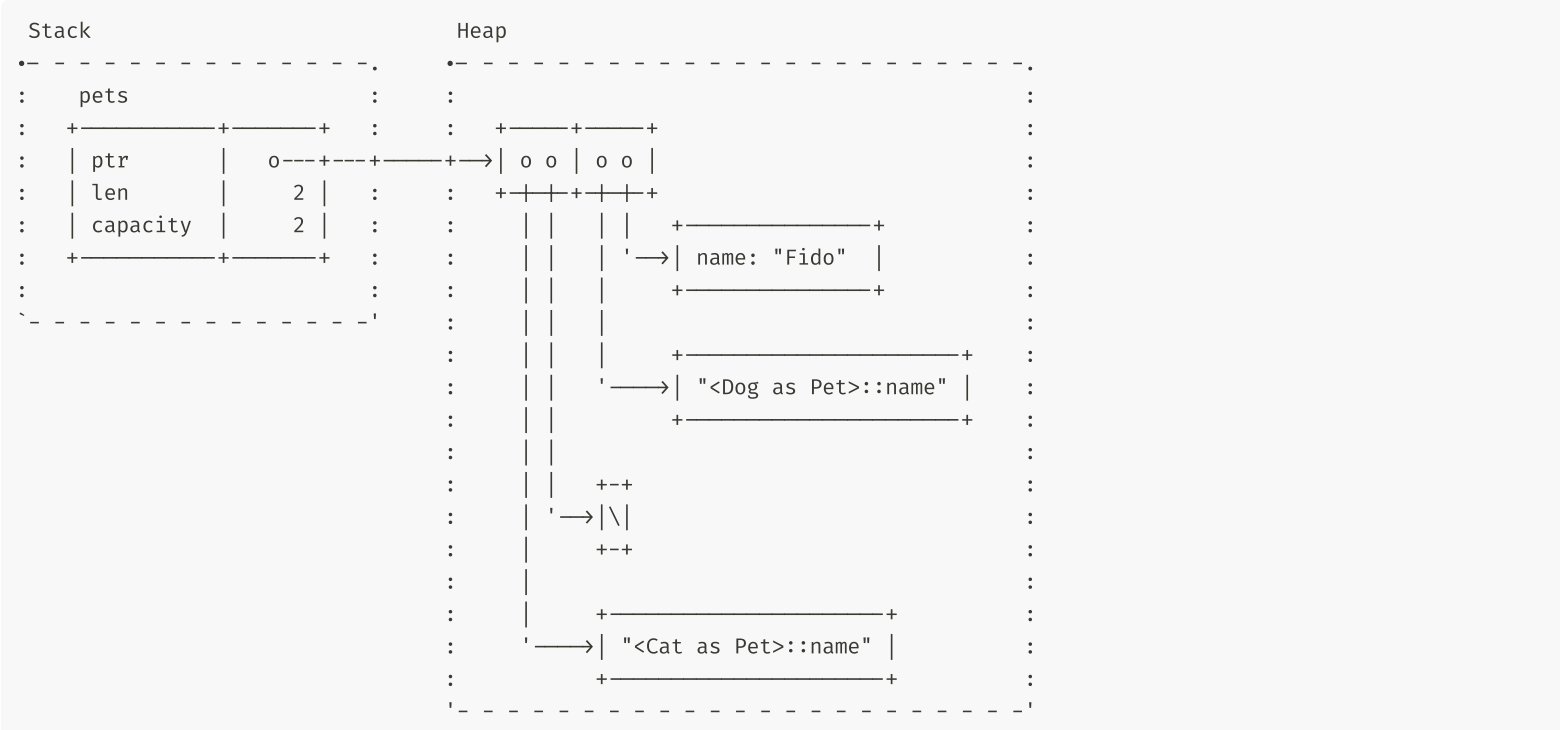
Trait Objects

Trait objects allow for values of different types, for instance in a collection.

```
fn main() {
    let fido = Dog {
        name: "Fido".to_string(),
    };
    let floof = Cat;

    let pets: Vec<Box<dyn Pet>> = vec![Box::new(fido), Box::new(floof)];
    for pet in pets {
        println!("Hello {}!", pet.name());
    }
}
```

- `pets` holds fat pointers to objects that implement `Pet`.
- The fat pointer consists of two components; 1) a pointer to the actual object, 2) a pointer to the virtual method table for the Pet implementation of that particular object.


Stack
Heap
pets
ptr
len
capacity
o
2
2
o o | o o
name: "Fido"
"<Dog as Pet>::name"
"<Cat as Pet>::name"

Why `Vec<Pet>` Doesn't Work

- types that implement a given trait may be of different sizes.

```
    println!(
        "{} {}",
        std::mem::size_of::<Dog>(),
        std::mem::size_of::<Cat>()
    );
    println!(
        "{} {}",
        std::mem::size_of::<&Dog>(),
        std::mem::size_of::<&Cat>()
    );
    println!("{}", std::mem::size_of::<&dyn Pet>());
    println!("{}", std::mem::size_of::<box<dyn Pet>>());
```

```
24 0

8 8 # Fixed size

16 # Fixed size
16
```

Deriving Traits

- Rust derive macros work by automatically generating code that implements the specified traits for a data structure.

```
struct Player {
    name: String,
}

fn main() {
    let p1 = Player {
        name: "Gandalf".to_string(),
    };
    println!("{}", p1);
}
```

```
 1  error[E0277]: `Player` doesn't implement `std::fmt::Display`
   ⟶ src/main.rs:10:20
    |
 10 |     println!("{}", p1);
    |                    ^^ `Player` cannot be formatted with the default formatter
```

```
struct Player {
    name: String,
}

fn main() {
    let p1 = Player {
        name: "Gandalf".to_string(),
    };
    println!("{}", p1);
}
```

```
error[E0277]: `Player` doesn't implement `Debug`
  --> src/main.rs:10:22
   |
10 |     println!("{:?}", p1);
   |                      ^^ `Player` cannot be formatted using `{:?}`
   |

help: consider annotating `Player` with `#[derive(Debug)]`
   |
2  + #[derive(Debug)]
3  | struct Player {
```

```
#[derive(Debug)]
struct Player {
    name: String,
}

fn main() {
    let p1 = Player {
        name: "Gandalf".to_string(),
    };
    println!("{}", p1);
}
```

```
Player { name: "Gandalf" }
```

How To Implement `Debug` Manually?

```rust
use std::fmt;

struct Player {
    name: String,
}

impl fmt::Debug for Player {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Player {{ name: {} }}", self.name)
    }
}

fn main() {
    let p1 = Player {
        name: "Gandalf".to_string(),
    };
    println!("{:?}", p1);
}
```

```rust
#[derive(Debug)]
struct Player {
    name: string,
}

fn main() {
    let p1 = Player {
        name: "gandalf".to_string(),
    };
    let p2 = Player {
        name: "sauron".to_string(),
    };

    // the same error will occur with an if-else statement.
    assert_eq!(p1, p2);
}
```

```
error[e0369]: binary operation `==` cannot be applied to type `Player`
  --> src/main.rs:18:5
   |
18 |     assert_eq!(p1, p2);
   |     ^^^^^^^^^^^^^^^^^^
   |     |
   |     Player
   |     Player
   |

note: an implementation of `PartialEq` might be missing for `Player`
  --> src/main.rs:4:1
   |
4  | struct Player {
   | ^^^^^^^^^^^^^ must implement `PartialEq`

help: consider annotating `Player` with `#[derive(PartialEq)]`
   |
4  + #[derive(PartialEq)]
5  | struct Player {
```

Other Common Traits

```
#[derive(Clone, Default)]
struct Player {
    name: string,
}

fn main() {
    let p1 = Player::default();
    let p2 = p1.clone();
}
```

Orphan Rule And Coherence

- we can’t implement external traits on external types.

```rust
use std::fmt::display;

// Both are not local
impl<t> Display for Vec<t> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        // implementation goes here
    }
}

fn main() {}
```

```
error: only traits defined in the current crate can be implemented for types defined outside of the crate
  --> src/main.rs:4:1
   |
4  | impl<t> Display for Vec<t> {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   | |                     |
   | |                     `Vec` is not defined in the current crate
   | impl doesn't use only types from inside the current crate
   |
   = note: define and implement a trait or new type instead
```

Default Implementations

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Traits as Parameters

- We can avoid using concrete type in function Parameters

```
- fn greet(pet: &dyn Pet) {
+ fn greet(pet: &impl Pet) {
      println!("Who's a cutie? {} is!", pet.name());
  }
```

`impl Pet` is syntatic sugar for trait bound syntax `<P: Pet>(pet: P)`.

Some of the benefits of `impl Trait`:

- Convenient
- Concise

Trait Bound Syntax Vs `Impl Trait`

```
// Can have *different* types.
pub fn notify(item1: &impl Summary, item2: &impl Summary) {

// Force both parameters to have the *same* type.
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

Multiple Trait Bounds with the + Syntax

Both are valid Rust code.

```
pub fn notify(item: &(impl Summary + Display)) {

pub fn notify<T: Summary + Display>(item: &T) {
```

Clearer Trait Bounds with where Clauses

```
// The function signature hard to read
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) → i32 {

// This function's signature is less cluttered
fn some_function<T, U>(t: &T, u: &U) → i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

- Declutters the function signature if you have many parameters.

Returning Types That Implement Traits

```
fn make_dog() -> impl Pet {
    Dog {
        name: "Fido".to_string(),
    }
}

fn main() {
    greet(&make_dog());
}
```

- Returns a value without naming the concrete type.
- Allows you to work with types which you cannot name.
  - `impl IntoResponse`.

The `impl Trait` is a bit different in the different positions.

- In function parameter:
  - `impl Trait` is like an anonymous generic parameter with a trait bound.
- In function return type:
  - it means a concrete type that implements the trait, without naming the type.

```
fn get_x(name: impl Display) -> impl Display {
    format!("Hello {name}")
}
```

Can’t Return Different Types with `impl Trait`

```
fn returns_noisemaker(switch: bool) -> impl NoiseMaker {
    if switch {
        Dog
    } else {
        Cat
    }
}
```

```
help: you could change the return type to be a boxed trait object
   |
19 | fn returns_noisemaker(switch: bool) -> Box<dyn NoiseMaker> {
   |                                        ~~~~~~~           +
help: if you change the return type to expect trait objects, box the returned expressions
   |
23 ~         Box::new(Dog)
24 |     } else {
25 |         // Box::new(Cat)
26 ~         Box::new(Cat)
   |
```

Solutions

- Using a `Box`

```
fn returns_noisemaker(switch: bool) → Box<dyn NoiseMaker> {
    if switch {
        Box::new(Dog)
    } else {
        Box::new(Cat)
    }
}
```

- Using an enum

```
enum NoiseSource {
    Dog(Dog),
    Cat(Cat),
}

fn returns_noisemaker(switch: bool) → NoiseSource {
    if switch {
        NoiseSource::Dog(Dog)
    } else {
        NoiseSource::Cat(Cat)
    }
}
```

Using Trait Bounds to Conditionally Implement Methods

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Using Trait Bounds to Conditionally Implement Methods

```rust
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Using Trait Bounds to Conditionally Implement Methods

```
fn main() {
    // Example with Pair of integers
    let pair_of_integers = Pair::new(5, 10);
    pair_of_integers.cmp_display();

    // Example with Pair of strings
    let pair_of_strings = Pair::new("hello", "world");
    pair_of_strings.cmp_display();
}
```

Using Trait Bounds to Conditionally Implement Methods

```
struct Employee;

fn main() {
    let pair = Pair::new(Employee, Employee);
}
```

Using Trait Bounds to Conditionally Implement Methods

```
struct Employee;

fn main() {
    let pair = Pair::new(Employee, Employee);
    pair.cmp_display();
}
```

```
error[E0599]: the method `cmp_display` exists for struct `Pair<Employee>`, but its trait bounds were not satisfied
  --> src/main.rs:38:10
   |
3  | struct Pair<T> {
   | -------------- method `cmp_display` not found for this struct
...
34 | struct Employee;
   | ----------------
   | |
   | doesn't satisfy `Employee: PartialEq`
   | doesn't satisfy `Employee: PartialOrd`
   | doesn't satisfy `Employee: std::fmt::Display`

...
38 |     pair.cmp_display();
   |          ^^^^^^^^^^^^ method cannot be called on `Pair<Employee>` due to unsatisfied trait bounds
14 | impl<T: Display + PartialOrd> Pair<T> {
   |         -------   ----------  ------
   |         |         |
   |         |         unsatisfied trait bound introduced here
   |         unsatisfied trait bound introduced here
   = note: the following trait bounds were not satisfied:
           `Employee: PartialEq`
           which is required by `Employee: PartialOrd`
```

But Rust Compiler is Always Helpful

```
668 | pub trait Display {
    | ^^^^^^^^^^^^^^^^^
help: consider annotating `Employee` with `#[derive(PartialEq, PartialOrd)]`
    |
34  + #[derive(PartialEq, PartialOrd)]
35  | struct Employee;
```

Conditionally Implement a Trait For Any Type That Implements Another Trait

The impl block in the standard library looks similar to this code:

```
// Blanket implementation for any type T that implements Display
impl<T: Display> ToString for T {
    --snip--
}
```

```
let s = 3.to_string();
```

Blanket Implementation Example

```rust
use std::fmt::Debug;

trait PrintInfo {
    fn print_info(&self);
}

// Blanket implementation for any type T that implements Debug
impl<T: Debug> PrintInfo for T {
    fn print_info(&self) {
        println!("Type: {:?}, Debug Info: {:?}", std::any::type_name::<T>(), self);
    }
}

fn main() {
    // Example with i32
    let number = 42;
    number.print_info();

    // Example with String
    let text = String::from("Hello, Rust!");
    text.print_info();
}
```

Using Trait Bounds to Conditionally Implement Methods

- Use generic type parameters to avoid duplication but limit to particular behavior.
- All the correct behavior checked at compile time.

Supertrait

```
trait Equals {
    fn equals(&self, other: &Self) -> bool;

    fn not_equals(&self, other: &Self) -> bool {
        !self.equals(other)
    }
}

#[derive(Debug)]
struct Centimeter(i16);

impl Equals for Centimeter {
    fn equals(&self, other: &Centimeter) -> bool {
        self.0 == other.0
    }
}

fn main() {
    let a = Centimeter(10);
    let b = Centimeter(20);

    println!("{a:?} equals {b:?}: {}", a.equals(&b));
    println!("{a:?} not_equals {b:?}: {}", a.not_equals(&b));
}
```

```
trait Equals {
    fn equals(&self, other: &Self) → bool;

-    fn not_equals(&self, other: &Self) → bool {
-        !self.equals(other)
-    }
}


+trait NotEquals {
+    fn not_equals(&self, other: &Self) → bool {
+        !self.equals(other)
+    }
+}
```

```
error[E0599]: no method named `equals` found for reference `&Self` in the current scope
  --> src/main.rs:10:15
   |
10 |         !self.equals(other)
   |               ^^^^^^
   |
   = help: items from traits can only be used if the type parameter is bounded by the trait
help: the following trait defines an item `equals`, perhaps you need to add a supertrait for it:
   |
8  | trait NotEquals: Equals {
   |                  ++++++++
help: there is a method with a similar name
   |
10 |         !self.not_equals(other)
   |               ~~~~~~~~~~
```

```
+// The NotEquals trait is using the Equals trait as a supertrait.
+// This means that any type implementing NotEquals must also implement Equals.
+// trait NotEquals {
+trait NotEquals: Equals {
+    fn not_equals(&self, other: &Self) -> bool {
+        !self.equals(other)
+    }
+}

#[derive(Debug)]
struct Centimeter(i16);

impl Equals for Centimeter {
    fn equals(&self, other: &Centimeter) -> bool {
        self.0 == other.0
    }
}

+impl NotEquals for Centimeter {}
```

```
trait NotEquals {
    fn not_equals(&self, other: &Self) -> bool;
}

impl<T> NotEquals for T
where
    T: Equals,
{
    fn not_equals(&self, other: &Self) -> bool {
        !self.equals(other)
    }
}
```

Credits 🌟

- "The Rust Programming Language, 2nd Edition" by Steve Klabnik, and Carol Nichols
- Mo's (mo8it) Comprehensive Rust 🦀
- rustlings 🦀