Discussion #10

11/10/2026

Today

Traits

Reminder: Corrections are next week! Grades for the midterm should be out by Wednesday

Today we will:

- Review traits
- Work through how traits work in code
- Practice implementing traits with an exercise

Submit today's exercise on Gradescope.

Traits Review

Using traits to define what a type can do

- In rust, a **trait** defines a capability that types can gain by implementing it
  - Example: Display is the capability “can be printed with {}”. String and i32 implement it, which lets us do println!("{}", name)
- A trait only declares method signatures; each type writes its own body with impl Trait for Type
- Functions can ask for a **capability** instead of one specific type, and the compiler checks if the passed type has that capability

Why traits?

Conceptual review

- When you need to write something down, what could you use?
  - To write **with:** Pen, pencil, quill, charcoal, crayon, paint
  - To write **on:** Paper, napkin, papyrus, your hand, the wall
- Anything that can write + anything that can be written on = a note
- Pen and paper may suit the job best, but if you only have a crayon and a napkin in a restaurant, **you can still complete the action**
- If you want to invent a new type of writing utensil, it can be used for all the same things as the others **as long as it has the writing functionality**

Worked-out example
Step 1: Define the traits

```
trait Writer {
    fn write(&self, text: &str) -> String;
}

trait Writable {
    fn write_to(&mut self, mark: String);
}
```

Types with the Writer trait have to have the write function, which outputs a string (a message)

Types with the Writeable trait have to have the write_to function, which can change self to record messages

Worked-out example
Step 2: Implement the traits

```
struct Pen;
impl Writer for Pen {
    fn write(&self, text: &str) -> String { format!("{} (in ink)", text) }
}

struct Paper { marks: Vec<String> }
impl Writable for Paper {
    fn write_to(&mut self, mark: String) { self.marks.push(mark); }
}
```

← Defines Pen

← Gives Pen the Writer trait

↑ Defines the write function specific to the Pen type. How we implement it is up to us as long as we return a String.

← Defines Paper

← Gives Paper the Writable trait

↑ Defines the write_to function specific to the Paper type. There's no output, so how we record the mark depends on the type (in this case we push new marks to the vec)

Worked-out example
Step 2.5: Implement the trait, but different!

Not every type with the Writer trait has to implement the Write function the same way.

Suppose we have a stamp that always says “APPROVED”:

```
struct Stamp;
impl Writer for Stamp {
    fn write(&self, _text: &str) -> String {
        String::from("APPROVED")
    }
}
```

This version ignores the input text; it is still a valid implementation of the Writer trait

Worked-out example
Step 3: Make a function

```
fn jot_down(tool: &impl Writer, surface: &mut impl Writable, text: &str) {
    surface.write_to(mark: tool.write(text));
}
```

jot_down has two inputs.

`tool: &impl Writer` Tool can be any type that writes (&impl Writer)

Surface can be any type that is writable; it also has to be mutable for us to be able to write to it (&mut impl Writeable) `surface: &mut impl Writable`

Worked-out example
Step 4: Try it out!

```
jot_down(tool: &Pen, surface: &mut paper, text: "dear diary");
jot_down(tool: &Stamp, surface: &mut paper, text: "dear diary");
jot_down(tool: &Crayon, surface: &mut napkin, text: "genius idea");

println!("Paper: {:?}", paper.marks);
println!("Napkin: {:?}", napkin.marks);
```

```
    Running `target\debug\discussion_10_traits.exe`
Paper: ["dear diary (in ink)", "APPROVED"]
Napkin: ["genius idea (in wax)"]
```

Trait Bounds

Requiring more than one capability

- fn f(item: &impl Describe) and fn f<T: Describe>(item: &T) both mean "any type that implements Describe"
- Use + to require several traits: T: Writable + Rippable
- A trait can build on another one (a **supertrait**): trait Writable: Rippable means every Writable type must also be Rippable
- Inside the function you can only use what the bounds promise

Traits you already use

Sorting, cloning, and copying

- **Clone:** For types that own data, like String and Vec: a.clone()
  - Duplicates so that the original value stays usable
- **Copy:** For small, simple types: numbers, bool, simple enums
  - Duplicates automatically on assignment so the original stays usable
  - Copy builds on Clone; every type that has Copy must also have Clone
- **PartialEq:** “are these two the same?” → ==
- **Ord:** “which one goes first?” → <, > and .sort()

Exercise: Yes, chef!

- Go to this url: https://bit.ly/4iz0R9z
- Complete the exercise (instructions included)
- Once you've completed the exercise, submit your program and its output on Gradescope
- If you finish early, feel free to work on Project 3, or to ask any questions you may have!