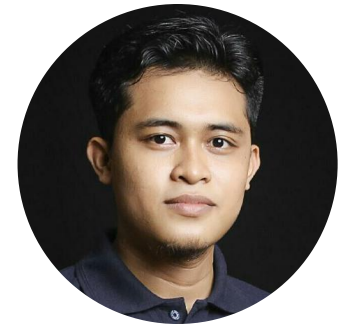
Rust Fundamentals

Basics of Rust
Part IV

AZZAM S.A

RustCourse
Oct. 13th, 2023

Azzam S.A



OSS devotee, speaker, and teacher.

Open sourceror. Namely Rust, Python, and Emacs.

azzamsa azzamsyawqi azzamsa.com

Course. Not talk!

Follow along!

- Rust Playground
- Exercises

▶ Show hints

Structs

```
struct Person {
    name: String,
    age: u8,
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    println!("{} is {} years old", peter.name, peter.age);

    peter.age = 28;
    println!("{} is {} years old", peter.name, peter.age);
```

- Unlike tuple, nordered field.
- Unlike tuple, Each piece of filed must have a name.
- Add more meaning to the data. No need to remember tuple number.

Field Init Shorthand

```
fn build_user(email: String, username: String) → User {
    User {
        active: true,
-        username: username,
-        email: email,
+        username,
+        email,
        sign_in_count: 1,
    }
}
```

- If parameter names and the struct field names are exactly the same.
- Less boilerplate.

Struct Update Syntax

```
fn main() {
    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

- ... specifies that the remaining fields should have the same value as the fields in the given instance.
- struct update syntax uses `=` like an assignment; it moves the data.

Tuple Structs

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

- If naming each field as in a regular struct would be verbose or redundant.
- Give the whole tuple a name.

Newtypes

```
struct PoundsOfForce(f64);

fn compute_thruster_force() → PoundsOfForce {
    todo!("Ask a rocket scientist at NASA")
}

fn main() {
    let force = compute_thruster_force();
}
```

- Encode additional information about the value in a primitive
- `PhoneNumber(String)` ensures the value is pre-validated, eliminating the need for validation on each use.

Unit-Like Structs Without Any Fields

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

- Useful when working with traits.

Constructors

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn new(name: String, age: u8) -> Person {
        // Some input processing and validation here.
        Person { name, age }
    }
}

fn main() {
    let bilbo = Person::new("Bilbo".to_string(), 27);
    println!("{bilbo:?}"); // debug print
}
```

```
impl Person {
-    fn new(name: String, age: u8) → Person {
+    fn new(name: String, age: u8) → Self {
        Person { name, age }
    }
}
```

- `new()` is just a convention.
- `Self` is alias of current struct name.
- Less code to refactor.

```rust
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn new(name: String, age: u8) -> Self {
        Person { name, age }
    }
    fn bot(name: String) -> Person {
        Person { name, age: 0 }
    }
}
```

- Multiple constructor is possible.
- Multiple impl Blocks is possible.

Default values

```
impl Default for Person {
    fn default() → Self {
        Self {
            name: "Bot".to_string(),
            age: 0,
        }
    }
}

fn main() {
    let person: Person = Default::default();
}
```

Enums

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4("127.0.0.1".to_string());
let loopback = IpAddr::V6("::1".to_string());
```

- Collect set of values under one type.
- Any IP address can be either a V4 or V6, **but not both at the same time.**
- The name of each enum variant also becomes a function that constructs an instance of the enum.
- Methods on enums is possible (just like struct).

Variant Payloads

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

- We can put data directly into each enum variant.
- Each variant can have different types and amounts of associated data (More flexible than struct).

Option and Result

```
enum Option<T> {
    Some(T),
    None
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

- The `Option<T>` enum and its variants are so useful that it's even included in the prelude.
- The compiler won't let us use an `Option<T>` value as if it were definitely a valid value.

Methods

```rust
struct Person {
    name: String,
}

impl Person {
    fn say_hello(&self) {
        println!("Hello, my name is {}", self.name);
    }
}

fn main() {
    let peter = Person {
        name: String::from("Peter"),
    };
    peter.say_hello();
}
```

- `self` represents the instance of the struct the method is being called on.
- It's very rare to use `self`. `&self` is more common.
- it is `self: Self` under the hood.
- Method receiver: `&self`, `&mut self`, `self`, `mut self`.

Pattern Matching

```
fn main() {
    let input = 'x';

    match input {
        'q'                   ⇒ println!("Quitting"),
        'a' | 's' | 'w' | 'd' ⇒ println!("Moving around"),
        '0'..='9'             ⇒ println!("Number input"),
        _                     ⇒ println!("Something else"), // Catch-All Patterns (must be last!)
    }
}
```

- Think of a match expression as being like a coin-sorting machine.
- `if` expression needs to return a Boolean value, `match` can return any type.
- An arm has two parts: a pattern and some code.
- Matches Are Exhaustive: the arms' patterns must cover all possibilities.

Destructuring Enums

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}

fn main() {
    let n = 100;

    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} divided in two is {half}"),
        Result::Err(msg) => println!("sorry, an error happened: {msg}"),
    }
```

Destructuring Structs

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };

    match foo {
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
    }
}
```

- Can bind to the parts of the values.

Destructuring Arrays

```
#[rustfmt::skip]
fn main() {
    let triple = [0, -2, 3];
    println!("Tell me about {triple:?}");

    match triple {
        [0, y, z] ⇒ println!("First is 0, y = {y}, and z = {z}"),
        [1, ..]   ⇒ println!("First is 1 and the rest were ignored"),
        _         ⇒ println!("All elements were ignored"),
    }
}
```

Match Guards

```
#[rustfmt::skip]
fn main() {
    let pair = (2, -2);
    println!("Tell me about {pair:?}");

    match pair {
        (x, y) if x == y     => println!("These are twins"),
        (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
        (x, _) if x % 2 == 1 => println!("The first one is odd"),
        _                    => println!("No correlation..."),
    }
}
```

`if let` expressions

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is configured to be {max}");
}
```

```
let config_max = Some(3u8);
match config_max {
    Some(max) ⇒ println!("The maximum is configured to be {max}"),
    _ ⇒ (),
}
```

- Get rid of `_ ⇒ ()`
- A less verbose way to handle values that match one pattern while ignoring the rest.
- Lose the exhaustive checking that match enforces.
- Can have `else`.

`while let` expressions

```
fn main() {
    let v = vec![10, 20, 30];
    let mut iter = v.into_iter();

    while let Some(x) = iter.next() {
        println!("x: {x}");
    }
}
```

Credits 🌟

- Mo's (mo8it) Comprehensive Rust 🦀
- rustlings 🦀