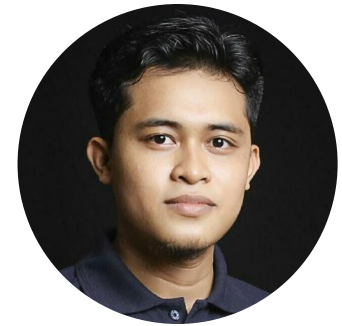
Rust Fundamentals

Basics of Rust
Part V

AZZAM S.A

RustCourse
Oct. 20th, 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

Standard Library

The common vocabulary types include:

- **Vec**: Store value next to each other.
- **String**: a collection of characters.
- **HashMap**: A hash map allows you to associate a value with a specific key.

Vectors

Creating a New Vector

```
let v: Vec<i32> = Vec::new();
```

❧ ❧ ❧

```
let v = vec![1, 2, 3];
```

- The data it contains is stored on the heap.
  - Doesn't need to be known at compile time. It can grow or shrink at runtime.
- Vectors can only store values of the same type.
- `` `vec!` `` is a convenient macro to create vectors with intial values.

Updating a Vector

```
let mut v = Vec::new();

v.push(5);
v.push(6);
```

Reading Elements of Vectors

```rust
let v = vec![1, 2, 3, 4, 5];

// 1
let third: &i32 = &v[2]; // returns reference
println!("The third element is {third}");

// 2
let third: Option<&i32> = v.get(2); // returns `Option<T>`
match third  {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

Choose your design:

1. Will cause panic for nonexistent element
2. Returns `None` without panicking.

Iterating Over the Values in a Vector

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{i}");
}
```

❧ ❧ ❧

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
println!("{v:?}");
```

- Iterating over a vector and mutating the value.

Strings

- Provided by Rust's standard library rather than coded into the core language.
- Growable, mutable, owned, UTF-8 encoded string type.

Creating a New String

```
let mut s = String::new();
```

```
let s = "initial contents".to_string();
let s = String::from("initial contents");
```

- Create `String` with initial data.
- No preferred choice; it is a matter of style and readability.

Updating a String

```
let mut s = String::from("foo");

s.push_str("bar") // take &str, don't take ownership
s.push('l'); // take char
```

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // `s1` has been moved, and can no longer be used.
```

```
fn add(self, s: &str) → String {
```

- We can only add a `&str` to a `String`; we can't add two String values together.
- But wait the type of `&s2` is `&String`?
  - The compiler can coerce the `&String` argument into a `&str` (deref coercion).

`format!` is more readable

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

❧ ❧ ❧

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{s1}-{s2}-{s3}"); // doesn't take ownership of any of its parameters.
```

Slicing Strings

```
fn main() {
    let original_string = "Hello, world!";

    // Create a slice of the string from index 7 to 12 (inclusive)
    let sliced_string = &original_string[7..=12];

    println!("Original String: {}", original_string);
    println!("Sliced String: {}", sliced_string); // returns "world!"
}
```

❧ ❧ ❧

```rust
fn main() {
    let hello = "hello";
    let s = &hello[0..1];

    println!("Sliced String: {}", s); // returns "h"
}
```

❧ ❧ ❧

```rust
fn main() {
    let hello = "Здравствуйте"; // Zdravstvuyte
    let s = &hello[0..1];

    println!("Sliced String: {}", s); // ⚠️
}
```

```
thread 'main' panicked at src/main.rs:3:19:
byte index 1 is not a char boundary; it is inside '3' (bytes 0..2) of `Здравствуйте`
```

```rust
fn main() {
    let char = "3";
    println!("The length of '3' is: {}", char.len()); // Output: 2

    let char = "h";
    println!("The length of 'h' is: {}", char.len()); // Output: 1
}
```

- Slicing string at a byte index that is not a valid character boundary (0 to 1).
- '3' (Cyrillic letter Ze) is multi-byte character.

❧ ❧ ❧

A Better approach

```
for c in "Зд".chars() {
    println!("{c}");
}
```

```
З
д
```

❧ ❧ ❧

```
fn main() {
    let hello = "Здравствуйте";
    println!("{}", hello.chars().nth(0).unwrap()); // returns З
}
```

HashMap

```rust
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Accessing Values in a Hash Map

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name).unwrap();
    println!("score: {:?}", score);
}
```

- `HashMap` is not defined in the prelude.
- `get()` returns `Option<&V>`.
- Unfortunately there is no `hashmap!` macro.

Iterate Over Each Key-Value Pair In a Hash Map

```
for (key, value) in &scores {
    println!("{key}: {value}");
}
```

Adding a Key and Value Only If a Key Isn't Present

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);

    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

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

```
{"Yellow": 50, "Blue": 10}
```

Modules

- A way to limit the amount of detail you have to keep in your head.

```
mod garden {
    pub fn clean() {
        println!("In the garden module");
    }
}
```

The compiler will look for the module's code in these places:

- Inline, within curly brackets that replace the semicolon following `` `mod garden` ``
- In the file `` `src/garden.rs` ``
- In the file `` `src/garden/mod.rs` ``

Visibility

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

- Module items are private by default (hides implementation details).
  - In Rust, all items (functions, methods, structs, enums, modules, and constants) are private to parent modules by default.
  - Think of the privacy rules as being like the back office of a restaurant.
- Some modules are siblings, if they are in the same level.
- Making the module public doesn’t make the items within public as well.

Paths

Paths are resolved as follows:

1. As a relative path
2. As an absolute path

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

- The crate name to start from the crate root is like using `/` in filesystem root in your shell.

```rust
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

- `super` is like starting a filesystem path with the `..` syntax.

Making Structs and Enums Public

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}
```

- If we make the struct public, the struct's fields will still be private.

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

- In contrast, if we make an enum public, all of its variants are then public.

Bringing Paths into Scope with the use Keyword

```
use std::collections::HashSet;
use std::mem::transmute;
```

Creating Idiomatic use Paths

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

- Specifying the parent module makes it clear that the function isn't locally defined.

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

- For structs, enums, and other items with `use`, it's idiomatic to specify the full path.

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    --snip--
}

fn function2() -> io::Result<()> {
    --snip--
}
```

- The exception if we have two items with the same name.

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    --snip--
}

fn function2() -> IoResult<()> {
    --snip--
}
```

- Providing new names with the `as` keyword.

Exercises

Vec1

```rust
// Your task is to create a `Vec` which holds the exact same elements as in the
// array `a`.

fn array_and_vec() -> ([i32; 4], Vec<i32>) {
    let a = [10, 20, 30, 40]; // a plain array
    let v = // TODO: declare your vector here with the macro for vectors

    (a, v)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_array_and_vec_similarity() {
        let (a, v) = array_and_vec();
        assert_eq!(a, v[..]);
    }
}
```

```
error: expected `;`, found `}`
  ⟶ src/lib.rs:9:11
   |
9  |     (a, v)
   |           ^ help: add `;` here
10 | }
   | - unexpected token

error[E0425]: cannot find value `v` in this scope
 ⟶ src/lib.rs:9:9
  |
9 |     (a, v)
  |         ^ help: a local variable with a similar name exists: `a`

error[E0308]: mismatched types
 ⟶ src/lib.rs:5:23
  |
5 | fn array_and_vec() → ([i32; 4], Vec<i32>) {
  |    ————————————      ^^^^^^^^^^^^^^^^^^^^ expected `([i32; 4], Vec<i32>)`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
  |
  = note:  expected tuple `([i32; 4], Vec<i32>)`
          found unit type `()`
```

```rust
fn array_and_vec() -> ([i32; 4], Vec<i32>) {
    let a = [10, 20, 30, 40]; // a plain array
    let v = vec![10, 20, 30, 40];

    (a, v)
}
```

Others

- `vecs2.rs`

vec2

```
fn vec_loop(mut v: Vec<i32>) -> Vec<i32> {
    for i in v.iter_mut() {
        // Fill this up so that each element in the Vec `v` is
        // multiplied by 2.
+        *i *=  2
    }

    // At this point, `v` should be equal to [4, 8, 12, 16, 20].
    v
}

fn vec_map(v: &Vec<i32>) -> Vec<i32> {
    v.iter().map(|num| {
        //  Do the same thing as above - but instead of mutating the
        // Vec, you can just return the new number!
+        num * 2
    }).collect()
}
```

Credits 🌟

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