APPS@UCU

Rust #3: Collections, Iterators, and Traits

Sultanov Andriy

APPLIED SCIENCES FACULTY

Contents

Notes on inclusivity and safety

Inclusivity

We've already talked about our responsibility as programmers to use our powers to create good and safe software. Tech isn't a goal of its own, it's a tool (which we do enjoy using) to improve life for all of humanity.

We haven't talked about different kind of responsibility, responsibility inside of our own field, and especially inside open source community. Tolerating sexism, misogyny and outright abuse does not create safe and inclusive spaces, and we should be rightly alarmed.

Safety

Speaking about responsibility, Rust has a special notion of safety/unsafety. All the code we were writing on these lectures before and will write in the future is safe, with all the constraints and checks that Rust compiler imposes on us.

It is possible to skip those compiler checks, Rust has a special "safety hatch" for that, a notion of unsafety, for which the programmer and not the compiler should check semantic code correctness and ensure that unsafe code is wrapped in safe APIs.

Safety

Unsafe Rust allows you to:

- Dereference raw pointers
- Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator)
- Implement unsafe traits
- Mutate statics
- Access fields of unions

Safety

```
// SAFETY: This can be risky if called from separate threads, but `Cell`
// is `!Sync` so this won't happen. This also won't invalidate any
// pointers since `Cell` makes sure nothing else will be pointing into
// either of these `Cell`s.
unsafe {
    ptr::swap(self.value.get(), other.value.get());
}

// SAFETY: This can cause data races if called from a separate thread,
// but `Cell` is `!Sync` so this won't happen.
mem::replace(unsafe { &mut *self.value.get() }, val)
```

Collections in Rust

Collections
Most common collections

Rust's standard library implements some of the most common collections:

- **Sequences**: String, Vec, VecDeque, LinkedList
- **Maps**: HashMap, BTreeMap
- **Sets**: HashSet, BTreeSet
- **Misc**: BinaryHeap

In most cases, you are going to be fine with using just String, Vec, HashMap, HashSet.

Collections
String

Strings are dynamic (heap-allocated) UTF-8 collections!

```
// This dynamic string is allocated on the heap
let mut s = String::new();

// This string is compiled into the binary
let data = "utf-8 string";

let s = data.to_string();
let s = String::from("utf-8 string");

// String is a mutable growable type
s.push('a');
s.push_str("text");
s.push_str(data);
```

Collections
String

```rust
let s1 = String::from("Linux");
let s2 = String::from("Club");
let s3 = String::from("UCU");

// String concatenation
let s = format!("{}_{}@{}", s1, s2, s3);
assert_eq!(s, "Linux_Club@UCU");

let s = s1 + "_" + &s2 + "@" + &s3;
assert_eq!(s, "Linux_Club@UCU");

// YOU CAN'T INDEX A STRING IN RUST!
&s[0];
```

Collections
String

```
// You can slice it, but it will panic if you try
// to slice inside a UTF-8 character
&s[0..4];

// It's better to iterate over the array with these methods:
for c in s.chars() {
    println!("{}", c);
}

for b in s.bytes() {
    println!("{}", b);
}
```

Collections
String

```
// Rust's Strings implement Deref to &str, so it's
// better to take string slices as function parameters
fn takes_str(s: &str) { }

let s = String::from("Hello");

// Will deref from &String to &str
// This is known as Deref coercion
takes_str(&s);
```

Collections
Vector

Vectors are your typical growable generic containers:

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

// Macro initialization
let mut v = vec![1, 1, 1, 1, 1];
let mut v = vec![1; 5];

v.push(4);
let last_element = v.pop();

// Slicing
&a[1..4]
```

Collections
Vector

```rust
// Might cause Rust to panic if we access beyond boundaries
&v[0];

// Will never panic but is pretty ugly
match v.get(2) {
    Some(elm) => println!("{}", elm),
    None => println!("There is no such element"),
}

for i in &v {
    println!("{}", i);
}

for i in &mut v {
    *i += 50;
}
```

Collections
HashMap

HashMap is a 'dictionary' type that's generic over <K, V>:

```
// HashMap is not included in the prelude, you
// have to 'use' it. Might change in Rust 2021
use std::collections::HashMap;

// This will get inferred by the compiler as HashMap<String, i32>
let mut grades = HashMap::new();

grades.insert(String::from("Student1"), 100);
grades.insert(String::from("Student2"), 90);

// Will panic if the key is absent
grades["Student1"];

// Will not panic
grades.get("Student1");
```

Collections
HashMap

```rust
// Rust provides a nice way to set default values for keys
let student3 = String::from("Student3");

// Inserts a key only if it doesn't already exist
let new_entry = grades.entry(student3.clone()).or_insert(80);

// This is the same as writing this:
let new_entry = if grades.contains_key(&student3) {
    grades.get_mut(&student3)
} else {
    grades.insert(student3.clone(), 80);
    grades.get_mut(&student3)
};
```

Collections
HashMap

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

// Creation from an iterator
let grades = [("Student1", 100), ("Student2", 90)]
        .iter()
        .cloned()
        .collect::<HashMap<&str, i32>>();
```

Collections
HashSet

HashSet is a set that's generic over <T>:

```
// Once again, HashSet is not included in the prelude
use std::collections::HashSet;

let mut books = HashSet::new();

// Add some books.
books.insert("The Making of the Indebted Man".to_string());
books.insert("Introduction to Civil War".to_string());

if !books.contains("The Concept of the Political") {}

// Remove an item
books.remove("Time, Labor, and Social Domination");
```

Collections
HashSet

```
// Iterating
for book in &books {
    println!("{}", book);
}

// Creation from an iterator
let students = ["Student1", "Student2", "Student3"]
        .iter()
        .cloned()
        .collect::<HashSet<&'static str>>();
```

Generics and Traits

Generics are a common way to generalize types and functionalities. This can reduce code duplication and allow for different and user-defined types to be used with generic functions.

Generics
Generic structs and enums

We've already seen a few examples of generic types in Rust (as opposed to concrete types):

```
// Option is generic over type T
pub enum Option<T> {
    None,
    Some(T),
}

// Result is generic over types T, E
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Generics
Generic structs and enums

Let's write one of these ourselves:

```
struct Point<T> {
    x: T,
    y: T,
}

// A generic impl block also has special syntax
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

We can also be more specific with impl blocks:

```
impl Point<f32> {
    fn dist(&self, p: Point<f32>) -> f32 {
        ((&self.x - &p.x).powf(2f32) + (&self.y - &p.y).powf(2f32)).sqrt()
    }
}
```

In Rust, generics can be used in a lot of cases, let's first look at generic functions.

Let's imagine we have two functions that are basically the same except for parameter types:

```
fn adder(a: i32, b: i32) -> String {
    format!("{} + {} = {}", a, b, a + b)
}

fn adder(a: f64, b: f64) -> String {
    format!("{} + {} = {}", a, b, a + b)
}
```

Let’s rewrite these functions into a generic function:

```
fn adder<T>(a: T, b: T) -> String {
    format!("{} + {} = {}", a, b, a + b)
}
```

This doesn’t compile though…

Rust, at compile-time, needs to be sure that you won't be able to call these functions with a type that won't have the needed functionality.

Rust has a concept that allows to tell the compiler about type's functionality, and to share that functionality between several types - traits.

Traits
General

What’s the type functionality? Basically - just the methods we can call on that type, and traits allow us to group those methods in specific sets.

Let’s imagine we have several possible types of students - kindergarten, school and university students. We want to be able to get a report on them - with their names, grades etc.

Traits
Example

We define an 'interface' of our shared functionality with method signatures without any implementation. We can also provide default functionality for a trait:

```
trait Report {
    fn general_report(&self) -> String;
    fn grades(&self) -> &HashMap<String, i32> {
        &self.grades
    };
}

struct SchoolStudent {
    name: String,
    grades: HashMap<String, i32>,
    teacher: String,
    classes: Vec<String>,
}
```

Traits
Example

Let's implement this trait for one of our student types:

```
impl Report for SchoolStudent {
    fn general_report(&self) -> String {
        format!(
            "Student: {},
        Their teacher: {},
        Takes these classes: {:?},
        Their grades: {:?}",
            &self.name, &self.teacher, &self.classes, &self.grades
        )
    }
}
```

Traits
Traits as parameters

When we want to be sure that the type we take as a parameter has the needed functionality, we can use the impl Trait syntax:

```
fn get_report(student: &impl Report) {
    println!("{}", student.general_report());
}

// This is equivalent to this, now with generics:
fn get_report<T: Report>(student: &T) {
    println!("{}", student.general_report());
}
```

Traits
Traits as parameters

We can also specify having multiple traits implemented as a requirement:

```
fn get_report(student: &(impl Report + Display)) {
    println!("{}", student.general_report());
}

// This is equivalent to this, now with generics:
fn get_report<T: Report + Display>(student: &T) {
    println!("{}", student.general_report());
}
```

If your function is generic over several types with different trait bounds, it's better to use where syntax:

```
fn function<T, U>(t: &T, u: &U) -> String
where
    T: Report + Clone,
    U: Display + Clone,
{
}
```

Let's return to an earlier example of a generic function:

```
fn adder<T>(a: T, b: T) -> String {
    format!("{} + {} = {}", a, b, a + b)
}
```

Generics and Traits
Bounds

Why didn't it compile? Well, the compiler is actually quite helpful:

```
error[E0369]: cannot add `T` to `T`
 --> src/main.rs:2:37
  |
2 |     format!("{} + {} = {}", a, b, a + b)
  |                             - ^ - T
  |                             |
  |                             T
  |
help: consider restricting type parameter `T`
  |
1 | fn adder<T: std::ops::Add<Output = T>>(a: T, b: T) -> String {
  |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Generics and Traits
Bounds

Let's look at that std::ops::Add:

```
// The addition operator +.
pub trait Add<Rhs = Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Let's add a bound on our generic function:

```
fn adder<T: std::ops::Add<Output = T>>(a: T, b: T) -> String {
    format!("{} + {} = {}", a, b, a + b)
}
```

But this doesn't compile, again...

Generics and Traits
Bounds

Once again, Rust compiler saving our souls:

```
error[E0277]: `T` doesn't implement `std::fmt::Display`
 --> src/main.rs:2:29
  |
2 |     format!("{} + {} = {}", a, b, a + b)
  |                             ^ `T` cannot be formatted with the default formatter
  |
  = note: required by `std::fmt::Display::fmt`
help: consider further restricting this bound
  |
1 | fn adder<T: std::ops::Add<Output = T> + std::fmt::Display>(a: T, b: T) -> String {
  |                                       ^^^^^^^^^^^^^^^^^^^
```

Alright, let's restrict it even further, now also considering that we don't want to borrow our arguments, we want to copy them:

```
fn adder<T>(a: T, b: T) -> String
where
    T: std::ops::Add<Output = T> + std::fmt::Display + Copy,
{
    format!("{} + {} = {}", a, b, a + b)
}
```

The standard library implements a lot of traits which allow it to reason about the functionality that certain types might have. The Copy one we used is a good example.

```
#[derive(Copy, Clone)]
struct OurType {
    a: i32,
}
```

This allows the Rust compiler to understand whether to move the object out or to copy it.

If the type implements a Copy trait, it's usually just called Copy. For example, i32 is Copy, because it implements the trait and won't move out:

```
fn main() {
    let s: i32 = 64;
    let u = s;
    println!("{}", s);
}

// Does not implement Copy, will move out
fn main() {
    let s: String = "text".to_string();
    let u = s;
    println!("{}", s);
}
```

The standard library provides a lot of traits with default implementations which you can derive, among them:

- **Debug** - Debug formatting using :?
- **PartialEq** and **Eq** - For ! = and == implementations
- **PartialOrd** and **Ord** - For orderings using <, >, <=, >=
- **Hash** - Allows to map an instance to a value

There are also 'marker' traits without any implementations, but they are beyond the scope of this lecture.

Lifetimes

Lifetimes

We've already talked about lifetimes a little, but so far we've mainly relied on the compiler to figure out lifetimes for us. Sometimes it's impossible for the compiler to reason about lifetimes, and we are forced to help it out.

This is where generic lifetimes come in. Yes, Rust has a lot of generic stuff - your functions can operate on generic types, 'generic' trait-implementing types and lifetimes.

Lifetimes

Let’s remind ourselves how Rust compiler reasons about lifetimes:

```
fn main() {
    let a;

    {
        let b = 5;
        a = &b;
    }

    println!("a: {}", a);
}
```

Lifetimes

And now in the case of a simple function:

```
fn main() {
    println!("{:?}", first_word("word1 word2"))
}

fn first_word(s: &str) -> Option<&str> {
    s.split(' ').next()
}
```

Lifetimes

If the references do not contain explicit lifetime annotations, they are instead called implicit. We can write simple functions without explicitly annotating lifetimes of our parameters (just how we can omit types sometimes), and the function will be figured out by the compiler like this:

```
fn main() {
    println!("{:?}", first_word("word1 word2"))
}

fn first_word<'a>(s: &'a str) -> Option<&'a str> {
    s.split(' ').next()
}
```

Lifetimes

Generally, when it is possible, Rust compiler does not need our help and is able to figure out the lifetimes on its own.

It uses several rules for figuring out the lifetimes for functions:

- Each parameter that is a reference gets its own lifetime
- If there is one input lifetime, this lifetime is assigned to the return value
- If there are several input lifetimes, but one is self, its lifetime is assigned to the return value

Lifetimes

Let’s try to apply these rules in a more complex case:

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

Rust compiler is not able to figure out the lifetime of the returned reference with these rules:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

It will complain and try help us:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value,
  but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ^^^^    ^^^^^^^     ^^^^^^^     ^^^
```

Lifetimes

Let's introduce a single named lifetime parameter:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

One special lifetime in Rust is 'static:

```
let s: &'static str = "text";
```

It means that the reference will live for the entire duration of the program (indeed, string literals are just embedded into the program binary).

The compiler will often suggest you use this lifetime as an annotation, but you should only use it when it makes sense!

Lifetimes

Rust functions can often get pretty messy when you use generic types, lifetimes and trait bounds:

```
fn longest<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
where
    T: std::fmt::Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Iterators and Closures

Closures

Closures are a common concept in functional programming, they are similar to lambda functions. In Rust, closures are anonymous inline functions which can be bound to variables. They are 'lazily' executed - will only start working when you call them.

```
let square = |x| { x * x };
println!("{}", square(3));
```

Closures

Rust will infer the types of closure parameters and return values, but we could include them explicitly, like in a function definition:

```
let square = |x: i32| -> i32 { x * x };
println!("{}", square(3));
```

Closures

Rust allows some variation with closures syntax,
but they are still similar to functions:

```
fn  add_one   (x: u32) -> u32 { x + 1 }
let add_one = |x: u32| -> u32 { x + 1 };
let add_one = |x|             { x + 1 };
let add_one = |x|               x + 1  ;
```

Closures

Why are they called closures? Because they 'close over' their environment, they can capture variables in their definition scope:

```
let a = 5;
let b = 32;
let c = |x: i32| x + a;
```

Closures

Closures will figure out which variables they need and what they need to do with them, and will only borrow or move those. You should be aware of their behavior:

```
let a = 5;
let b = 32;
let c = |x: i32| x + a;

let d = &mut a; // Error - can't borrow as mutable when
                // borrowed as immutable
println!("{}", b); // Works fine - closure did not capture it
```

Closures

You can force the closure to move the variables it needs.
It might not actually take ownership of them if types are Copy!

```
let a = 5;
let c = move |x: i32| x + a;

let d = &mut a;
```

This is often used with closures that run asynchronously
and will need to survive longer than the current scope.

Closures

Under the hood, closures are actually done using several traits:

```
pub trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

pub trait FnMut<Args>: FnOnce<Args> {
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

Closures

So, your nice closure syntax will actually desugar to Rust creating a struct with all the variables you capture, and implementing the trait it needs on your struct.

Every closure is an FnOnce, since it can be called once, closures that do not move their environment into them, but only borrow it mutably also are FnMut, and closures that only borrow it immutably are also Fn.

Closures

You can take or return closures from functions using these traits as bounds:

```
fn test_func<A, B, F>(self, f: F) -> Vec<B>
    where F: FnMut(A) -> B;
{}

fn box_up_your_closure_and_move_out() -> Box<Fn(i32) -> i32> {
    // local stuff
    Box::new(move |x| x * local)
}
```

If you want to return closures, you will have to put them on the heap (since their size is not known at compile time) and move your environment into them.

Iterators

Iterators in Rust are not that different from your typical iterators in other languages. To be an Iterator you will have to implement an Iterator trait with a single required method:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Everything works thanks to Rust features: traits and algebraic types. While your iterator has values to return, it will return Some(Item), and once it runs out of them, it will start returning None.

Iterators

Everything works as you would expect:

```
let a = [1, 2, 3];

let mut iter = a.iter();

assert_eq!(Some(&1), iter.next());
assert_eq!(Some(&2), iter.next());
assert_eq!(Some(&3), iter.next());
```

Iterators

You can also create mutable iterators, and iterators that will consume your collection:

- **.iter()** - iterates over &T
- **.iter_mut()** - iterates over &mut T
- **.into_iter()** - iterates over T

Iterators

Rust provides syntactic sugar for them:

```
let mut values = vec![41];

for x in values.iter() {
    assert_eq!(*x, 42);
}

for x in &values { // same as `values.iter()`
    assert_eq!(*x, 42);
}
```

Iterators

```rust
let mut values = vec![41];

for x in values.iter_mut() {
    *x += 1;
}

for x in &mut values { // same as `values.iter_mut()`
    *x += 1;
}
```

Iterators

```
let mut values = vec![41];

for x in values.into_iter() {
    *x += 1;
}

for x in values { // same as `values.into_iter()`
    *x += 1;
}
```

Iterators

Once you implement the .next() method, you will get default implementations of a lot of adapters:

```
let a = [1, 2, 3];
let a1 = [1, 2, 3];

a.iter().map(|x| 2 * x)
        .filter(|x| x.is_positive())
        .max();

a.iter().zip(a1.iter());

a.iter().enumerate();

let data = vec![vec![1, 2, 3, 4], vec![5, 6]];
let flattened = data.into_iter().flatten().collect::<Vec<u8>>();
```

Iterators

Rust's iterators are heavily optimized, so you shouldn't feel bad about using a bunch of adapters one over another.

They are essential for idiomatic collection processing, and you should spend some time looking over their official documentation and learn which adapters you need!

Thank you!