APPS@UCU

Rust #3: Collections, Iterators, and Traits

Sultanov Andriy

APPLIED SCIENCES FACULTY

Contents

1. Collections in Rust
2. Generics and Traits
3. Iterators

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 timber_resources = [("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: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: required by `std::fmt::Display::fmt`
  = note: this error originates in a macro (in Nightly builds, run with -Z macro-backtrace for
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);
}
```

Generics and Traits
std Traits

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.
TODO: talk about lifetimes too

Iterators

Iterators

TODO: talk about iterators, closures, functional programming etc.

Thank you!