APPS@UCU

Rust #3: Collections, Iterators, and Traits

Sultanov Andriy

Contents

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

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.

TODO: talk about traits

TODO: talk about bounds, where

TODO: talk about lifetimes too

Iterators

Iterators

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

Thank you!