Rust Workshop

Day 3

Recap of Day 2

Structs

```rust
struct Person {
    name: String,
    age: u8,
}
fn main() {
    let person = Person {
        name: String::from("John"),
        age: 35,
    };
    println!("Person's name: {}", person.name);
}
struct NonZeroByte(u8);
struct OnePossibleValue;
```

Methods

```rust
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
    fn new_square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
fn main() {
    let area = rect.area();
    let square = Rectangle::new_square(size);
}
```

Enums

```rust
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(u8, u8, u8),
}
enum Option<T> {
    None,
    Some(T),
}
```

Pattern Matching

```rust
match msg {
    Message::Quit => println!("bye bye!"),
    Message::Write(text) => println!("{}", text),
    Message::Move { x, y } => set_position(x, y),
    _ => {},
}
if let Some(num) = option_of_num {
    println!("number detected: {}", num);
}
while let Some(num) = vec_of_nums.pop() {
    println!("removed from vec: {}", num);
}
```

Project Organization

The *crate* is the unit of compilation, `main.rs` or `lib.rs` the *root* of the crate.

Above the crate: `Cargo.toml` defines a *package* for the build system (`cargo`).

Inside the crate: Code is structured in a *tree* of *modules*.

Modules & Visibility

```rust
pub fn carrot() {}
fruits::orange()                    // relative
crate::garden::fruits::orange() // absolute
super::garden::fruits::orange() // backwards
use std::collections::HashMap;
let m: HashMap; // type is now in scope
use std::collections::*;
```

Error Handling

```rust
enum AreaError {
    BadSeparator,
    BadInteger(String),
}
fn calculate_area(input: &str) -> Result<usize, AreaError> {
    let (left, right) = match input.split_once('x') {
        Some(t) => t,
        None => return Err(AreaError::BadSeparator),
    };
    Ok(parse_int(left)? * parse_int(right)?)
}
fn main() {
    match calculate_area(input) {
        Ok(area) => println!("the area is: {}", area),
        Err(AreaError::BadSeparator) => try_different_separator(),
        _ => give_up(),
    }
}
```

Advanced Features 1

book chapters 10 & 13

- generics
- traits
- lifetimes
- closures
- iterators

Generics

book chapter 10.1

```rust
let num: i32 = Some(42).unwrap();
let s: &str = Some("hello").unwrap();
```

Can we write `unwrap` ourselves?

The Problem: Duplication

```rust
fn my_unwrap_i32(maybe_int: Option<i32>) -> i32 {
    maybe_int.unwrap()
}

fn my_unwrap_i64(maybe_int: Option<i64>) -> i64 {
    maybe_int.unwrap()
}
```

`void *`

The Solution: Generics

```rust
fn my_unwrap<T>(maybe_int: Option<T>) -> T {
    maybe_int.unwrap()
}

// compiler copies my_unwrap for each type
my_unwrap(Some(42_i32));
my_unwrap(Some(42_i64));
```

Generics in Structs

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

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
    let mix = Point { x: 1.0, y: 10 };
    //                           ^^
    // mismatched types: expected float
}
```

Multiple Generic Type Parameters

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

fn main() {
    let mix = Point { x: 1.0, y: 10 };
    // inferred type: Point<f64, i32>
}
```

Generics in Enums

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

Generics in Methods

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

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

Performance?

Generics are resolved at compile time.

Generic code is essentially copy-pasted for every type parameter.

There is zero runtime cost to using generics.

(just like C++ templates)

Traits

book chapter 10.2

What are Traits?

German: Eigenschaft, Merkmal

Traits fulfill the same purpose as interfaces in other languages.

They enable polymorphism by specifying shared behavior.

(Rust does not have OOP-style class inheritance.)

Problem: `T` is useless

```rust
fn find_largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

compiler says:

> binary operation `>` cannot be applied to type `&T`

Define Shared Behavior

```rust
trait Comparable {
    fn is_greater_than(&self, other: &Self) -> bool;
}
```

Implementation for Concrete Types

```rust
impl Comparable for i32 {
    fn is_greater_than(&self, other: &Self) -> bool {
        self > other
    }
}
impl Comparable for i64 {
    fn is_greater_than(&self, other: &Self) -> bool {
        self > other
    }
}
```

Constrain Generic Type Parameters

```rust
fn find_largest<T: Comparable>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item.is_greater_than(largest) {
            largest = item;
        }
    }

    largest
}
```

Default Implementations

```rust
trait Comparable {
    fn is_greater_than(&self, other: &Self) -> bool;

    fn is_less_than_or_equal(&self, other: &Self) -> bool {
        !self.is_greater_than(other)
    }
}
```

Multiple Trait Bounds

```rust
fn find_largest<T: Comparable + Debug>(list: &[T]) -> &T {
    // ...
    println!("found {largest:?}!");
    largest
}
```

Where Clauses

```rust
// hard to read
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {

// much better
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

Blanket Implementations

demo

```rust
use std::array;

fn main() {
    println!("{:?}", 42.clone_10_times());
    println!();
    println!("{:?}", String::from("hello").clone_10_times());
    println!();
    println!("{:?}", vec![1, 2].clone_10_times());
}

trait Clone10Times: Sized {
    fn clone_10_times(self) -> [Self; 10];
}

impl<T: Clone> Clone10Times for T {
    fn clone_10_times(self) -> [Self; 10] {
        array::from_fn(|_| self.clone())
    }
}
```

Useful Traits

`Debug`	string-representation for debugging
`Clone` & `Copy`	can be copied (cheaply)
`Default`	has default value (zero, empty string)
`PartialEq` & `Eq`	can check for equality (`==` , `!=`)
`PartialOrd` & `Ord`	can be ordered (`<` , `>` etc.)
`Hash`	can compute hash (for `HashMap` etc.)

not derivable: `Display` , `From` & `TryFrom`

Lifetimes

book chapter 10.3

Reminder:

Rust forbids invalid references

```rust
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Problem: Returning References

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

compiler says:

> missing lifetime specifier:
>
> this function’s return type contains a borrowed value,
>
> but the signature does not say whether it is borrowed from `x` or `y`

Recommended Solution

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

Cloning the string results in an additional heap allocation.

This is perfectly fine in 99% of situations.

Lifetime Annotations

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

Lifetime Annotations

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

There is some lifetime `'a`.

Lifetime Annotations

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

`x` lives at least for `'a`.

Lifetime Annotations

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

`y` lives at least for `'a`.

Lifetime Annotations

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

The returned reference lives at least for `'a`.

Lifetime Annotations

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

In essence:
The lifetime of the returned reference
is the shorter one of `x` and `y`'s lifetimes.

Limitations

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {/**/}

fn main() {
    let long;
    let x = String::from("looooong string");
    {
        let y = String::from("short str");
        long = longest(&x, &y);
    }
    println!("The longest string is: {}", long);
}
```

compiler says:

> y   does not live long enough

...but we know this would be OK at runtime.

Alternatively...

```rust
fn maybe_strip_prefix<'a>(x: &'a str, y: &str) -> &'a str {
    x.strip_prefix(y).unwrap_or(x)
}
```

The lifetime of `y` has no relation to the lifetime of the return value.

LT Annotations in Structs

```rust
struct StoringBorrowedData<'number, 'text> {
    n: &'number i32,
    s: &'text str,
}
```

Lifetime Elision Rules

(slightly simplified)

★ For a single parameter, its lifetime is assigned to all outputs.

```rust
fn foo(single_arg: &str) -> &str {}
// is the same as
fn foo<'a>(single_arg: &'a str) -> &'a str {}
```

★ For methods, the lifetime of `self` is assigned to all outputs.

```rust
impl Whatever {
    fn foo(&self, second_arg: &str) -> &str {}
    // is the same as
    fn foo<'a>(&'a self, second_arg: &str) -> &'a str {}
}
```

The `'static` Lifetime

```rust
static GREETING: &'static str = "hello world";

fn main() {
    let greeting: &'static str = "hello world";

    let answer: &'static i32;
    {
        let heap_alloc = Box::new(42);
        answer = Box::leak(heap_alloc); // explicit memory-leak
    }
    println!("answer: {}", answer);
}
```

Rust Easy Mode^{TM}

premature optimization is the root of all evil

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

`Clone` is your friend!

Closures

book chapter 13.1

What is a Closure?

Closures are inspired by functional programming, where anonymous functions are often created at runtime and passed around as argumets to and return values from other functions. They are sometimes called *lambdas* by other languages.

Unlike regular functions, closures can capture values from the scope in which they were defined.

Basic Syntax

```rust
fn main() {
        fn multiply(x: i32, y: i32) -> i32 { x * y }
    let multiply = |x: i32, y: i32| -> i32 { x * y };
    let multiply = |x: i32, y: i32|        { x * y };
    let multiply = |x     , y     |        { x * y };
    let multiply = |x     , y     |          x * y  ;

    // most concise:
    let multiply = |x, y| x * y;
}
```

Closures as Arguments

```rust
fn main() {
    let x = 3;

    let mut nums = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    nums.retain(|elem| elem % x == 0);

    println!("remaining: {:?}", nums); // [3, 6, 9]
}
```

Mutating the Environment

```rust
fn main() {
    let mut nums = vec![1, 2, 3];

    let mut push_seven = || nums.push(7);

    for _ in 0..10 {
        push_seven();
    }

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

Forcing a Move

actually a copy in this case

```rust
fn main() {
    let x_squared;
    {
        let x = 3;
        x_squared = || x * x; // `x` does not live long enough
        x_squared = move || x * x; // ✅
    }
    println!("{}", x_squared());
}
```

What is the Type of a Closure?

```rust
let square: ??? = |x| x * x;
```

We don’t need to write the type, but can we?

What is the Type of a Closure?

```
let square: fn(i32) -> i32 = |x| x * x;
```

This is a *function pointer* and occupies space in memory.

Doesn’t work with capturing…

```rust
let x = 3;
let times_x: fn(i32) -> i32 = |y| x * y;
// ❌ error: expected fn pointer, found closure
```

Another thing that doesn’t work

```rust
let x = 3;
let mut times_x = |y| x * y;

let x = 5;
times_x = |y| x * y;
```

```
mismatched types
expected closure, found a different closure
  = note: no two closures, even if identical, have the same type
```

The `Fn`-Traits

Trait	Informal Meaning	Connection to Ownership Rules
`Fn`	can be called and shared without restriction	captures only immutable references
`FnMut`	can be called many times but not shared	mutates captured values
`FnOnce`	can be called only once	moves captured values into closure

Closures have *unnamable* types.
We can only refer to them via the traits they implement.

The `Fn`-Traits

```rust
let mut text_buffer = String::from("To whom it may concern\n\n");

// implements `Fn() -> usize`
let buf_len = || text_buffer.len();

// implements `FnMut(&str)`
let mut append_to_buf = |s| text_buffer.push_str(s);

// implements `FnOnce()`
let print_and_drop_buf = move || println!("{text_buffer}");
```

Note: `Fn` is a "superset" of `FnMut`, which in turn is a "superset" of `FnOnce`.

`Fn`-Trait Example: `Vec::retain`

demo

```rust
fn main() {
    let x = 3;

    let mut nums: Vec<_> = (1..=16).collect();

    my_retain(&mut nums, |elem| elem % x == 0);

    println!("remaining: {:?}", nums);
}

fn my_retain<P>(nums: &mut Vec<i32>, predicate: P)
where
    P: Fn(i32) -> bool,
{
    for i in (0..nums.len()).rev() {
        if !predicate(nums[i]) {
            nums.remove(i);
        }
    }
}
```

Iterators

book chapter 13.2

Processing a Series of Items

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

For a type to be an iterator, it needs to...

- define the type of the items it iterates over.
- provide a method to get the next item in the series.

```rust
// manually calling next

fn main() {
    let v = vec!['a', 'b'];

    let mut iter = v.into_iter();

    // a
    let item = iter.next().unwrap();
    println!("next item: {}", item);

    // b
    let item = iter.next().unwrap();
    println!("next item: {}", item);

    // crash
    let item = iter.next().unwrap();
    println!("next item: {}", item);
}
```

Before

```rust
fn main() {
    let v = vec!['a', 'b'];

    let mut iter = v.into_iter();

    loop {
        let item = iter.next();
        if item.is_none() {
            break;
        }
        let item: char = item.unwrap();
        println!("next item: {}", item);
    }
}
```

After

```rust
fn main() {
    let v = vec!['a', 'b'];


    for item in v {




        println!("next item: {}", item);
    }
}
```

Iteration and Borrowing

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

for elem: i32 in nums {
    // elem is deallocated
}
// nums is destroyed

for elem: &i32 in nums.iter() {
    // can only read from elem
}
// nums is still intact

for elem: &mut i32 in nums.iter_mut() {
    // can modify value of elem
}
// nums is still intact
```

Writing an Iterator

demo

```rust
struct MyRange {
    start: i32,
    end: i32,
}

impl Iterator for MyRange {
    type Item = i32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.start >= self.end {
            return None;
        }
        let result = self.start;
        self.start += 1;
        Some(result)
    }
}

fn my_range(start: i32, end: i32) -> MyRange {
    MyRange { start, end }
}
```

Interlude: Turbofish

check out https://turbo.fish – it's fun

```rust
fn main() {
    println!("{}", "00123".parse().unwrap()); // ❌
    // type annotations needed
    // consider specifying the generic argument: `::<F>`

    println!("{}", "00123".parse::<i32>().unwrap());

    // why the double colon? why not this:
    println!("{}", "00123".parse<i32>().unwrap());
    // => syntax ambiguity with comparison operators ¯\_(ツ)_/¯
}

// for reference, from the standard library:
impl str {
    fn parse<F: FromStr>(&self) -> Result<F, <F as FromStr>::Err>;
}
```

Iterator Adapters

demo

```rust
pub fn get_solution(input: &str) -> u32 {
    input
        .split("\n\n")
        .map(|elf| elf.lines().map(|line| line.parse::<u32>().unwrap()).sum())
        .max()
        .unwrap()
}
```

Performance?

Iterators and their adapters are heavily optimized.
Sometimes, they are even faster than hand-coded loops.

Rule of Thumb:

Do not pick one over the other based on performance speculations.
If performance really matters, you need to benchmark.

Hint for Practice Session

```rust
// a test that ensures an expected panic occurs
#[test]
#[should_panic]
fn expected_panic_occurs() {
    let v = vec![1, 2, 3];
    v[10];
}
```

Practice 🧑‍💻

`rust-exercises/day_3/README.md`