Rust Workshop

Day 4

Recap of Day 3

Generics

```rust
fn my_unwrap<T>(maybe_val: Option<T>) -> T {
    maybe_val.unwrap()
}
impl<T> Option<T> {
    fn unwrap(self) -> T {
        // ...
    }
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Traits + Bounds

```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)
    }
}
fn find_largest<T>(list: &[T]) -> &T
where
    T: Comparable,
{ /**/ }
```

Lifetime Annotations

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

// recommended
fn longest(x: &str, y: &str) -> String {}
```

Closures

```
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);
}
```

Iterators

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let mut iter = numbers.into_iter();
    while let Some(num) = iter.next() { /**/ }
    // equivalent:
    for num in numbers { /**/ }
}
```

Advanced Features 2

book chapters 15, 16, 17 & async

- Smart Pointers
- Concurrency & Parallelism
- Dynamic Dispatch
- Asynchronous Programming

Smart Pointers

book chapter 15

What is a Smart Pointer?

Smart pointers are data structures that act like a pointer
but also have additional metadata and capabilities.

We've already seen them! `Vec` and `String` are smart pointers.
They store their length and capacity as metadata.
They have the capability to grow and shrink.

What does a smart pointer point to?

```rust
trait Deref {
    type Target;

    fn deref(&self) -> &Self::Target;
}
```

What happens when I'm done with a smart pointer?

```rust
trait Drop {
    fn drop(&mut self);
}
```

Box

```
// pseudo-code for illustration

pub struct Box<T>(*mut T);

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        std::alloc::dealloc(self.0);
    }
}
```

Recursive Types

```
enum List {
    Node(i32, List), // ❌ error: infinite size
    End,
}

enum List {
    Node(i32, Box<List>),
    End,
}
```

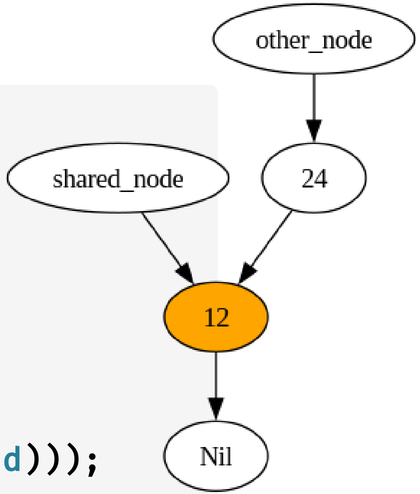
Using the `Deref` trait

```
fn main() {
    let boxed_int = Box::new(5);
    let reference: &i32 = &boxed_int;
    let copy: i32 = *boxed_int;
}
```

Shared Ownership

```rust
enum List {
    Node(i32, Rc<List>),
    End,
}
use List::*;

fn main() {
    let shared_node = Rc::new(Node(12, Rc::new(End)));
    // ref count: 1
    {
        let other_node = Node(24, Rc::clone(&shared_node));
        // ref count: 2
    }
    // ref count: 1
} // ref count: 0 (shared_node gets dropped)
```



Interior Mutability

```rust
#[derive(Default)]
struct NewsWebsite {
    free_articles_read: RefCell<usize>,
}
impl NewsWebsite {
    fn read_article(&self) {
        if *RefCell::borrow(&self.free_articles_read) >= 2 {
            panic!("You have used up your free articles quota!")
        }
        *RefCell::borrow_mut(&self.free_articles_read) += 1;
    }
}
fn main() {
    let news_website = NewsWebsite::default();
    news_website.read_article();
    news_website.read_article();
    news_website.read_article(); // panic! gotta buy a subscription...
}
```

Violating Borrowing Rules at Runtime

```
fn main() {
    let x = RefCell::new(1);
    let r1 = RefCell::borrow_mut(&x);
    let r2 = RefCell::borrow(&x); // panic! 😱 (instead of compiler error)
}
```

Shared Ownership + Interior Mutability

```rust
fn main() {
    let x: Rc<RefCell<i32>> = Rc::new(RefCell::new(1));

    let r1 = Rc::clone(&x);
    let r2 = Rc::clone(&x);

    *RefCell::borrow_mut(&r1) += 1;
    *RefCell::borrow_mut(&r2) += 1;

    println!("{}", x.borrow()); // 3
}
```

Reference Cycle == Memory Leak

demo

```rust
struct PrintWhenDropped(char);
impl Drop for PrintWhenDropped {
    fn drop(&mut self) {
        println!("drop called on {}!", self.0) }
}
struct GraphNode {
    value: PrintWhenDropped,
    neighbor: Option<Rc<RefCell<GraphNode>>>,
}
fn main() {
    let a = Rc::new(RefCell::new(GraphNode {
        value: PrintWhenDropped('a'),
        neighbor: None,
    }));
    let b = Rc::new(RefCell::new(GraphNode {
        value: PrintWhenDropped('b'),
        neighbor: Some(Rc::clone(&a)),
    }));
    RefCell::borrow_mut(&a).neighbor.replace(Rc::clone(&b)); // reference cycle
}
```

Summary

smart pointers

- put data on the heap with `Box`
- share ownership with `Rc`
- allow interior mutability with `RefCell`
- watch out for reference cycles 🙂

Concurrency and Parallelism

book chapter 16

Spawning Threads

demo

Message Passing

```rust
fn main() {
    // mpsc: Multiple Producers, Single Consumer
    let (sender, receiver) = mpsc::channel();

    thread::spawn(move || {
        for message in ["hi", "from", "the", "thread"] {
            sender.send(message).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    let message = receiver.recv().unwrap();
    println!("Got: {message}");

    for message in receiver {
        println!("Got: {message}");
    }
}
```

Shared-State Concurrency

demo

```rust
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

`Send` and `Sync`

- Types that can be sent (move ownership) across thread-boundaries are `Send`.
- Types where references to them can be sent across thread-boundaries are `Sync`.

Intuitively, they can be read by multiple threads at the same time.

- Most normal types are both `Send` and `Sync`.
- `Rc` is NEITHER `Send` NOR `Sync`.
- `RefCell` IS `Send`, but it is NOT `Sync`.
- `Mutex` IS `Sync`, even if its contained type is only `Send`.

`Send` and `Sync` are **auto traits**, meaning the compiler implements them for you where appropriate.

Fearless Concurrency

With Rust, you can write concurrent programs
without having to be afraid of bugs like data races. 🥳

except...

Deadlocks! 😢

Dynamic Dispatch

book chapter 17.2

Object-Oriented Programming

property	supported in Rust?	supporting feature
associate data and behavior	✅	methods
encapsulation	✅	modules
polymorphism	✅	traits
inheritance	❌	
dynamic dispatch	✅	❓

Dynamic Dispatch

demo

```rust
trait Animal {
    fn make_sound(&self);
}
struct Dog;
impl Animal for Dog {
    fn make_sound(&self) {
        println!("woof!") }
}
struct Cat;
impl Animal for Cat {
    fn make_sound(&self) {
        println!("meow!") }
}
fn get_animals() -> Vec<&'static dyn Animal> {
    vec![&Dog, &Dog, &Cat]
}
fn main() {
    for animal in get_animals() {
        animal.make_sound();
    }
}
```

Asynchronous Programming

book chapter still being worked on!

Asynchronous Programming, or async for short,
is a *concurrent programming model*.

For practical purposes, it’s an alternative to OS threads.

Disadvantages of OS threads

A single thread has relatively large overhead, including its own stack.

Consequently, they are not well-suited for massive IO-bound workloads.
(e.g. high-traffic web servers)

Scheduling is done by the OS, implying the overhead of a context-switch.

Scheduling is preemptive, which is less efficient than cooperative.

Async by comparison

Essentially zero overhead, not even heap allocations.

Perfectly suited for massive IO-bound workloads.

Scheduling is cooperative and works without context-switches.

...but more difficult to use!

Async hello world

demo

```rust
use tokio::time;

async fn do_stuff(name: &str) {
    println!("{name:>5}: He...");
    time::sleep(time::Duration::from_secs(1)).await;
    println!("{name:>5}: ...llo...");
    time::sleep(time::Duration::from_secs(1)).await;
    println!("{name:>5}: ...world!");
}

#[tokio::main(worker_threads = 1)]
async fn main() {
    let alice_task = tokio::spawn(do_stuff("Alice"));
    do_stuff("Bob").await;
    alice_task.await.unwrap();
}
```

Async & Embedded

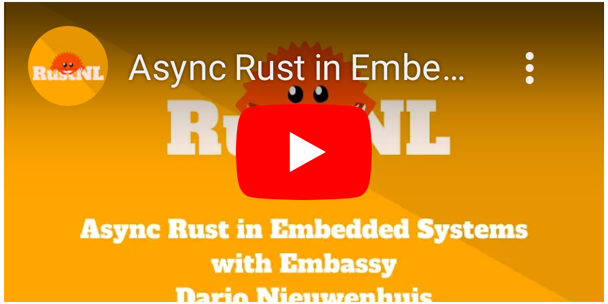
```rust
use embassy_nrf::gpio::{AnyPin, Input, Level, Output, OutputDrive, Pin, Pull};
use embassy_time::{Duration, Timer};

// Declare async tasks
#[embassy_executor::task]
async fn blink(pin: AnyPin) {
    let mut led = Output::new(pin, Level::Low, OutputDrive::Standard);

    loop {
        // Timekeeping is globally available, no need to mess with hardware timers.
        led.set_high();
        Timer::after_millis(150).await;
        led.set_low();
        Timer::after_millis(150).await;
    }
}
```

Recommended talk:

Async Rust in Embedded Systems with Embassy



Congratulations!

🥳 🧑‍🎓 🥳

You now have a solid grasp of all the tools available in Rust.

It’s time to start building stuff!

Outlook Day 5 & 6

It’s all about practical skills now!

- libraries, APIs, documentation
- automated testing and deployment
- a variety of practice projects

Heads-up:
Bring your LED-Matrix!

Next week, will get hands on with practical projects.
Among other options, you can program the LED-Matrix in Rust!

Practice 🧑‍💻

`rust-exercises/day_4/README.md`