

Rust for Embedded Developers

Day 3

Language 3

book chapters 10, 13, 15 – 18

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

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 arguments 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

```
1  fn main() {  
2      fn multiply(x: i32, y: i32) -> i32 { x * y }  
3      let multiply = |x: i32, y: i32| -> i32 { x * y };  
4      let multiply = |x: i32, y: i32|      { x * y };  
5      let multiply = |x      , y      |      { x * y };  
6      let multiply = |x      , y      |      x * y ;  
7  
8      // most concise:  
9      let multiply = |x, y| x * y;  
10 }
```

Closures as Arguments

```
1 fn main() {  
2     let x = 3;  
3  
4     let mut nums = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
5  
6     nums.retain(|elem| elem % x == 0);  
7  
8     println!("remaining: {:?}", nums); // [3, 6, 9]  
9 }
```

Mutating the Environment

```
1  fn main() {  
2      let mut nums = vec![1, 2, 3];  
3  
4      let mut push_seven = || nums.push(7);  
5  
6      for _ in 0..10 {  
7          push_seven();  
8      }  
9  
10     println!("nums: {:?}", nums);  
11 }
```

Forcing a Move

```
1 fn main() {  
2     let print_s;  
3     {  
4         let s = String::from("hello");  
5         print_s = || println!("{s}"); // ✗ `s` does not live long enough  
6         print_s = move || println!("{s}"); // ✓  
7     }  
8     print_s();  
9 }
```

The body of `print_s` only uses a shared reference to `s`.
But, the closure must take *ownership* of `s` to extend its lifetime.
That's what the `move` keyword does.

What is the Type of a Closure?

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

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

What is the Type of a Closure?

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

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

Doesn't work with capturing...

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

Another thing that doesn't work

```
1  let x = 3;  
2  let mut times_x = |y| x * y;  
3  
4  let x = 5;  
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
<code>Fn</code>	can be called and shared without restriction	captures only immutable references
<code>FnMut</code>	can be called many times but not shared	mutates captured values
<code>FnOnce</code>	can be called only once	consumes captured values

Closures have *unnamable* types.
We can only reason about them using trait bounds.

The Fn-Traits

```
1 let mut text_buffer = String::from("To whom it may concern\n\n");
2
3 // implements `Fn() -> usize`
4 let buf_len = || text_buffer.len();
5
6 // implements `FnMut(&str)`
7 let mut append_to_buf = |s| text_buffer.push_str(s);
8
9 // implements `FnOnce()`
10 let print_and_drop_buf = || drop(text_buffer);
```

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

Fn-Trait Example: `Vec::retain`

demo

```

1  fn main() {
2      let x = 3;
3
4      let mut nums: Vec<_> = (1..=16).collect();
5
6      my_retain(&mut nums, |elem| elem % x == 0);
7
8      println!("remaining: {:?}", nums);
9  }
10
11 fn my_retain<P>(nums: &mut Vec<i32>, predicate: P)
12 where
13     P: Fn(i32) -> bool,
14 {
15     for i in (0..nums.len()).rev() {
16         if !predicate(nums[i]) {
17             nums.remove(i);
18         }
19     }
20 }

```

Custom closures

demo

```
1  fn main() {
2      let mut env: i32 = 10;
3
4      let mut increment = MyIncrement { env: &mut env };
5
6      increment.call();
7      increment.call();
8
9      println!("{}", env);
10 }
11
12 struct MyIncrement<'env> {
13     env: &'env mut i32,
14 }
15
16 impl<'env> MyIncrement<'env> {
17     fn call(&mut self) {
18         *self.env += 1;
19     }
20 }
```

Iterators

book chapter 13.2

Processing a Series of Items

```
1  trait Iterator {  
2      type Item; // associated type (new syntax)  
3      fn next(&mut self) -> Option<Self::Item>;  
4  }
```

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.

```
1 // for-loop
2
3 fn main() {
4     let v = vec!['a', 'b'];
5
6     for item in v {
7         println!("next item: {}", item);
8     }
9 }
```

Before

```
fn main() {  
    let v = vec!['a', 'b'];  
  
    let mut iter = v.into_iter();  
  
    loop {  
        let Some(item) = iter.next() else {  
            break;  
        };  
        println!("next item: {}", item);  
    }  
}
```

After

```
fn main() {  
    let v = vec!['a', 'b'];  
  
    for item in v {  
        println!("next item: {}", item);  
    }  
}
```

Iteration and Borrowing

```
1  let nums = vec![1, 2, 3];
2
3  for elem in nums {
4      // elem has type `i32` (owned)
5  }
6  // nums has been consumed, cannot be used anymore
7
8  for elem in &nums {
9      // elem has type `&i32`, can only be read
10 }
11 // nums is still intact, can be used later
12
13 for elem in &mut nums {
14     // elem has type `&mut i32`, can be modified
15 }
16 // nums is modified, but can be used later
```

Writing an Iterator

demo

```
1  struct MyRange {
2      start: i32,
3      end: i32,
4  }
5
6  impl Iterator for MyRange {
7      type Item = i32;
8
9      fn next(&mut self) -> Option<Self::Item> {
10         if self.start >= self.end {
11             return None;
12         }
13         let result = self.start;
14         self.start += 1;
15         Some(result)
16     }
17 }
18
19 fn my_range(start: i32, end: i32) -> MyRange {
20     MyRange { start, end }
21 }
```

Interlude: Turbofish

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

```
1 fn main() {
2     println!("{}", "00123".parse().unwrap()); // ✗
3     // type annotations needed
4     // consider specifying the generic argument: `::<F>`
5
6     println!("{}", "00123".parse::i32().unwrap());
7
8     // why the double colon? why not this:
9     println!("{}", "00123".parse<i32>().unwrap());
10    // => syntax ambiguity with comparison operators `\_(\ツ)\_`
11 }
12
13 // for reference, from the standard library:
14 impl str {
15     fn parse<F: FromStr>(&self) -> Result<F, <F as FromStr>::Err>;
16 }
```

Iterator Adapters

demo

```
1 pub fn get_solution(input: &str) -> u32 {  
2     input  
3         .split("\n\n")  
4         .map(|elf| elf.lines().map(|line| line.parse::<u32>().unwrap()).sum())  
5         .max()  
6         .unwrap()  
7 }
```

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.

Practice

`exercises/day_3.md`

 closures & iterators

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?

```
1  trait Deref {  
2      type Target;  
3  
4      fn deref(&self) -> &Self::Target;  
5  }
```

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

```
1  trait Drop {  
2      fn drop(&mut self);  
3  }
```

Box

```
1  // pseudo-code for illustration
2
3  pub struct Box<T>(*mut T);
4
5  impl<T> Drop for Box<T> {
6      fn drop(&mut self) {
7          std::alloc::dealloc(self.0);
8      }
9  }
```

Recursive Types

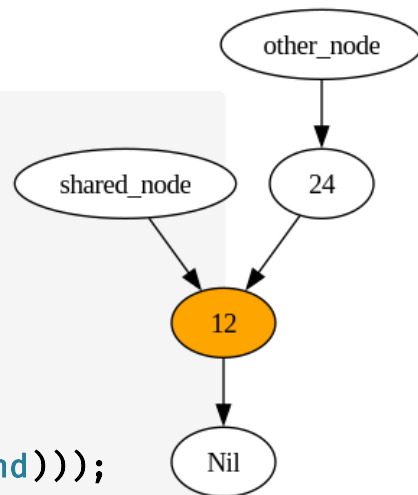
```
1  enum List {  
2      Node(i32, List), // ✗ error: infinite size  
3      End,  
4  }  
5  
6  enum List {  
7      Node(i32, Box<List>),  
8      End,  
9  }
```

Using the Deref trait

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

Shared Ownership

```
1  enum List {
2      Node(i32, Rc<List>),
3      End,
4  }
5  use List::*;
6
7  fn main() {
8      let shared_node = Rc::new(Node(12, Rc::new(End)));
9      // ref count: 1
10     {
11         let other_node = Node(24, Rc::clone(&shared_node));
12         // ref count: 2
13     }
14     // ref count: 1
15 } // ref count: 0 (shared_node gets dropped)
```



Interior Mutability

```
1  #[derive(Default)]
2  struct NewsWebsite {
3      free_articles_read: RefCell<usize>,
4  }
5  impl NewsWebsite {
6      fn read_article(&self) {
7          if *RefCell::borrow(&self.free_articles_read) >= 2 {
8              panic!("You have used up your free articles quota!")
9          }
10         *RefCell::borrow_mut(&self.free_articles_read) += 1;
11     }
12 }
13 fn main() {
14     let news_website = NewsWebsite::default();
15     news_website.read_article();
16     news_website.read_article();
17     news_website.read_article(); // panic! gotta buy a subscription...
18 }
```

Violating Borrowing Rules at Runtime

```
1 fn main() {  
2     let x = RefCell::new(1);  
3     let r1 = RefCell::borrow_mut(&x);  
4     let r2 = RefCell::borrow(&x); // panic! 🤪 (instead of compiler error)  
5 }
```

Shared Ownership + Interior Mutability

```
1  fn main() {
2      let x: Rc<RefCell<i32>> = Rc::new(RefCell::new(1));
3
4      let r1 = Rc::clone(&x);
5      let r2 = Rc::clone(&x);
6
7      *RefCell::borrow_mut(&r1) += 1;
8      *RefCell::borrow_mut(&r2) += 1;
9
10     println!("{}", x.borrow()); // 3
11 }
```

Reference Cycle == Memory Leak

demo

```

1  struct PrintWhenDropped(char);
2  impl Drop for PrintWhenDropped {
3      fn drop(&mut self) {
4          println!("drop called on {}!", self.0) }
5  }
6  struct GraphNode {
7      value: PrintWhenDropped,
8      neighbor: Option<Rc<RefCell<GraphNode>>>,
9  }
10 fn main() {
11     let a = Rc::new(RefCell::new(GraphNode {
12         value: PrintWhenDropped('a'),
13         neighbor: None,
14     }));
15     let b = Rc::new(RefCell::new(GraphNode {
16         value: PrintWhenDropped('b'),
17         neighbor: Some(Rc::clone(&a)),
18     }));
19     RefCell::borrow_mut(&a).neighbor.replace(Rc::clone(&b)); // reference cycle
20 }

```

Summary

smart pointers

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

Dynamic Dispatch

book chapter 18.2

Object-Oriented Programming

property	supported in Rust?	supporting feature
associate data and behavior	✓	methods
encapsulation	✓	modules
polymorphism	✓	traits
inheritance	✗	
dynamic dispatch	✓	?

Dynamic Dispatch

demo

```

1  trait Animal {
2      fn make_sound(&self);
3  }
4  struct Dog;
5  impl Animal for Dog {
6      fn make_sound(&self) {
7          println!("woof!") }
8  }
9  struct Cat;
10 impl Animal for Cat {
11     fn make_sound(&self) {
12         println!("meow!") }
13 }
14 fn get_animals() -> Vec<&'static dyn Animal> {
15     vec![&Dog, &Dog, &Cat]
16 }
17 fn main() {
18     for animal in get_animals() {
19         animal.make_sound();
20     }
21 }

```

Concurrency and Parallelism

book chapter 16

Spawning Threads

demo

Message Passing

```
1  fn main() {
2      // mpsc: Multiple Producers, Single Consumer
3      let (sender, receiver) = mpsc::channel();
4
5      thread::spawn(move || {
6          for message in ["hi", "from", "the", "thread"] {
7              sender.send(message).unwrap();
8              thread::sleep(Duration::from_secs(1));
9          }
10     });
11
12     let message = receiver.recv().unwrap();
13     println!("Got: {message}");
14
15     for message in receiver {
16         println!("Got: {message}");
17     }
18 }
```

Shared-State Concurrency

demo

```
1 fn main() {
2     let counter = Arc::new(Mutex::new(0));
3     let mut handles = vec![];
4
5     for _ in 0..10 {
6         let counter = Arc::clone(&counter);
7         let handle = thread::spawn(move || {
8             let mut num = counter.lock().unwrap();
9
10            *num += 1;
11        });
12        handles.push(handle);
13    }
14
15    for handle in handles {
16        handle.join().unwrap();
17    }
18    println!("Result: {}", *counter.lock().unwrap());
19 }
```

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

Rc

```
impl<T, A> !Send for Rc<T, A>  
where  
    A: Allocator,  
    T: ?Sized,
```

```
impl<T, A> !Sync for Rc<T, A>  
where  
    A: Allocator,  
    T: ?Sized,
```

Arc

```
impl<T, A> Send for Arc<T, A>  
where  
    T: Sync + Send + ?Sized,  
    A: Allocator + Send,
```

```
impl<T, A> Sync for Arc<T, A>  
where  
    T: Sync + Send + ?Sized,  
    A: Allocator + Sync,
```

RefCell

```
impl<T> Send for RefCell<T>  
where  
    T: Send + ?Sized,
```

```
impl<T> !Sync for RefCell<T>  
where  
    T: ?Sized,
```

Mutex

```
impl<T: ?Sized + Send> Send for Mutex<T> 1.0.0 · Source  
T must be Send for a Mutex to be Send because it is possible to acquire  
the owned T from the Mutex via into_inner.
```

```
impl<T: ?Sized + Send> Sync for Mutex<T> 1.0.0 · Source  
T must be Send for Mutex to be Sync. This ensures that the protected  
data can be accessed safely from multiple threads without causing data  
races or other unsafe behavior.
```

Fearless Concurrency

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

except...

Deadlocks! 😭

Practice

`exercises/day_3.md`

 concurrency & parallelism

Asynchronous Programming

book chapter 17

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

```
1 use tokio::time;
2
3 async fn do_stuff(name: &str) {
4     println!("{name:>5}: He...");
5     time::sleep(time::Duration::from_secs(1)).await;
6     println!("{name:>5}: ...llo...");
7     time::sleep(time::Duration::from_secs(1)).await;
8     println!("{name:>5}: ...world!");
9 }
10
11 #[tokio::main(worker_threads = 1)]
12 async fn main() {
13     let alice_task = tokio::spawn(do_stuff("Alice"));
14     do_stuff("Bob").await;
15     alice_task.await.unwrap();
16 }
```

async & await

```
1 async fn do_stuff() {  
2     println!("starting...");  
3     time::sleep(time::Duration::from_secs(0)).await;  
4     println!("done.");  
5 }  
6 // ...becomes:  
7 fn do_stuff() -> impl Future<Output = ()> {  
8     // ...  
9 }
```

The Future trait

```
1 pub trait Future {  
2     type Output; // return value once the future is finally done  
3  
4     // make future progress, asking it if it's done  
5     fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
6 }  
7  
8 pub enum Poll<T> {  
9     Ready(T),  
10    Pending,  
11 }
```

An implementation of Future

This is generated by the compiler, but want to have an intuition for it.

```
1  fn do_stuff() -> impl Future<Output = ()> {
2      println!("starting...");
3      time::sleep(time::Duration::from_secs(0)).await
4      println!("done.");
5  }
6  enum DoStuff {
7      Init,
8      Waiting, // <- corresponds to the await point
9      Done
10 }
11 impl Future for DoStuff {
12     type Output = ();
13     // fn poll() on next slide
14 }
```

An implementation of Future: poll

```
1  impl Future for DoStuff {
2      fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
3          match *self {
4              DoStuff::Init => {
5                  println!("starting...");
6                  *self = DoStuff::Waiting;
7                  Poll::Pending
8              }
9              DoStuff::Waiting => {
10                 if still_waiting() { // <- replace with actual wait logic
11                     return Poll::Pending;
12                 }
13                 println!("done.");
14                 *self = DoStuff::Done;
15                 Poll::Ready(())
16             }
17             DoStuff::Done => panic!("Stop calling me!"),
18     } } }
```

Writing an async runtime

demo

This is long and complicated ⚠

The point is not to actually be able to do this.

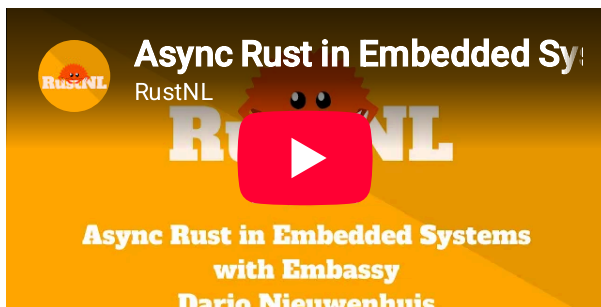
But having some idea of how a runtime is implemented
helps with knowing how to use it effectively.

The code is at:

<https://github.zhaw.ch/rust-wbk/custom-async-runtime-demo>
(one commit per step in the demo)

Recommended talk:

Async Rust in Embedded Systems with Embassy



(Too early) Lumatrix intro

<https://github.zhaw.ch/rust-wbk/lumatrix>

...in case the adventurous among you
want to start hacking on the weekend already...

Practice

`exercises/day_3.md`