



**let's build
an HTTP
cache warmer**

<https://lab.localdomain.pl/rust/handout.pdf>





level 0
the command line

level 0: the command line

- **src/bin/level0.rs**
cargo run --bin level0 -- urls.txt
- **examples/level0.rs**
cargo run --example level0 -- urls.txt

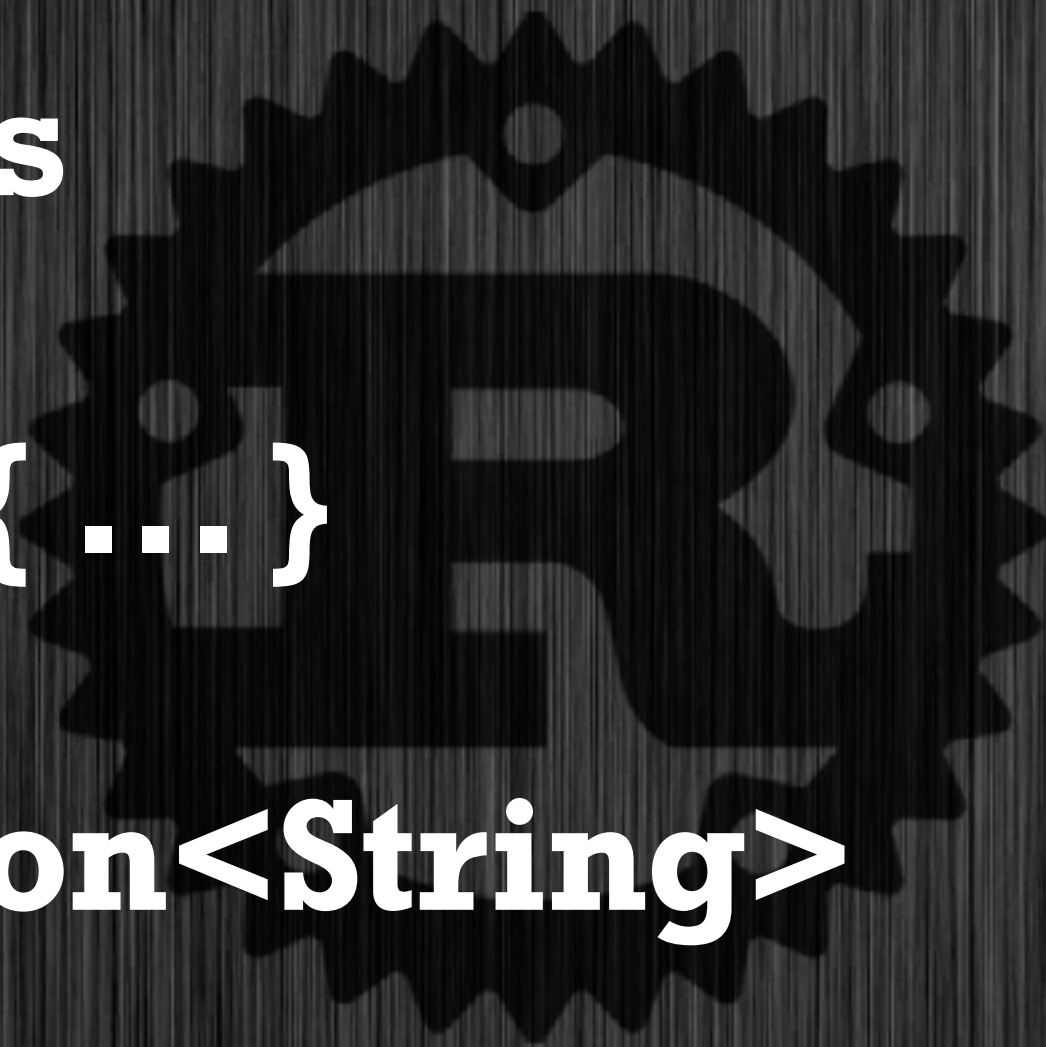


level 0: the command line

- **`std::env::args() -> Args`**

`impl Iterator for Args { ... }`

`fn nth(i: usize) -> Option<String>`



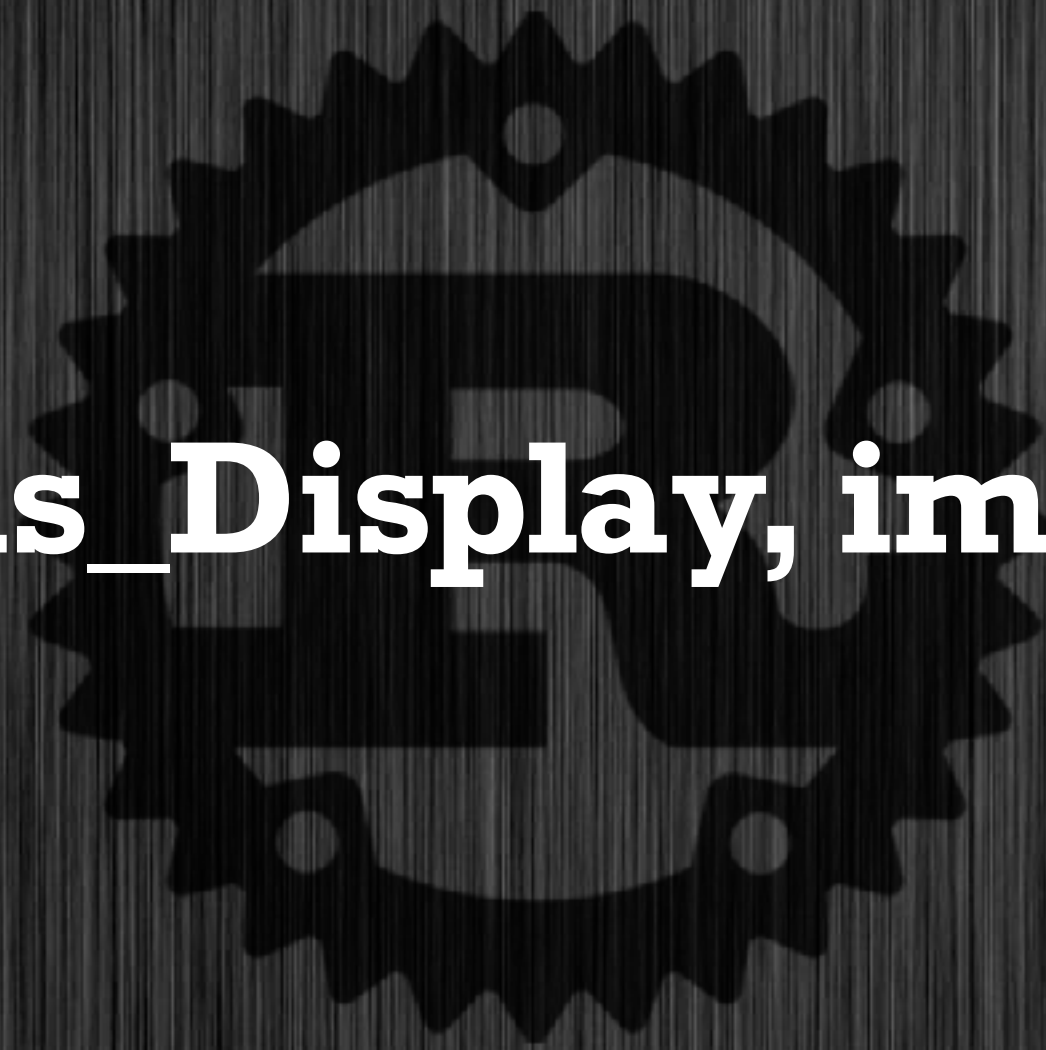
level 0: the command line

- **unwrap(), expect("msg")**
- **Result<T, E> -> T or panic**
- **Option<T> -> T or panic**



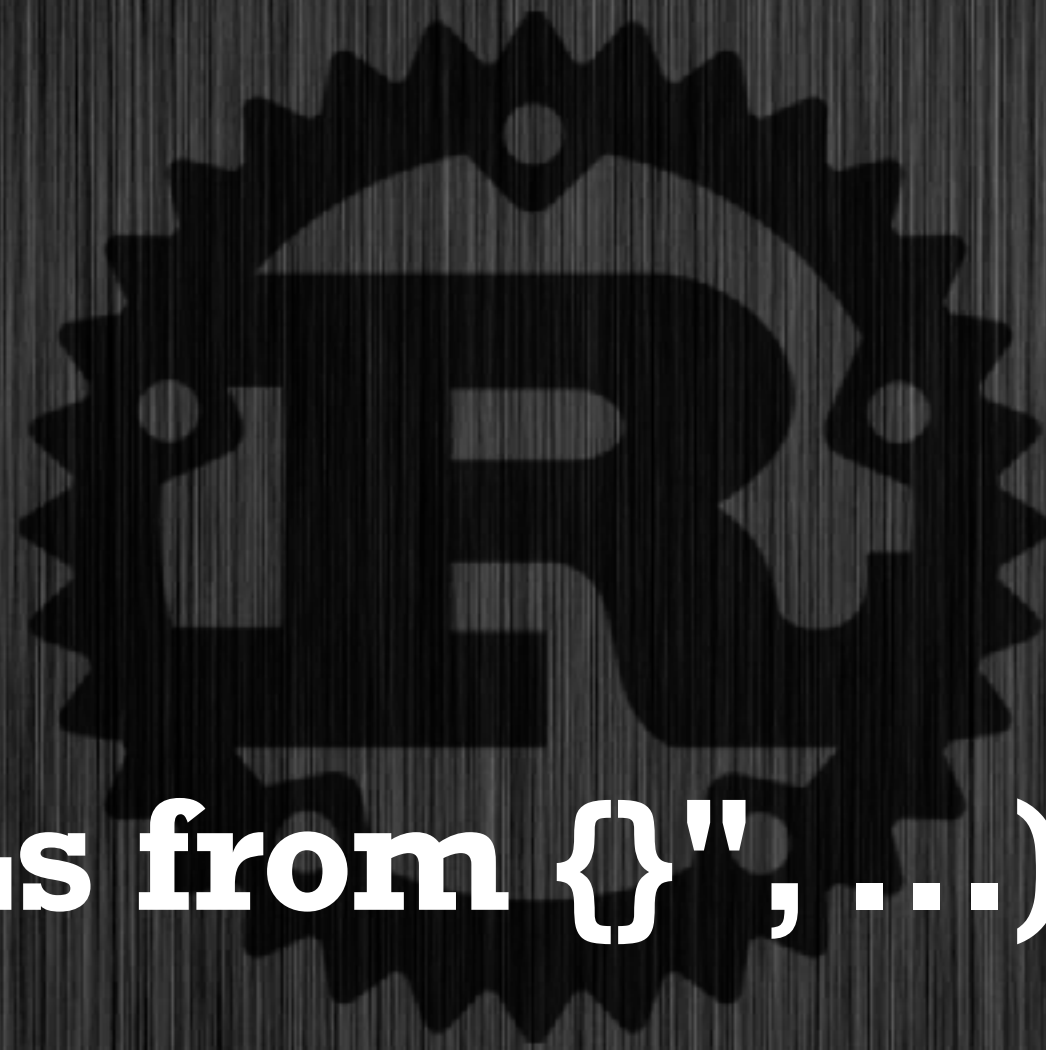
level 0: the command line

- **`println!("{}", {:?})", impls_Display, impls_Debug)`**



level 0: the command line

- **`std::env::args()`
`.nth(1)`
`.unwrap()`**
- **`println!("Loading URLs from {}", ...)`**

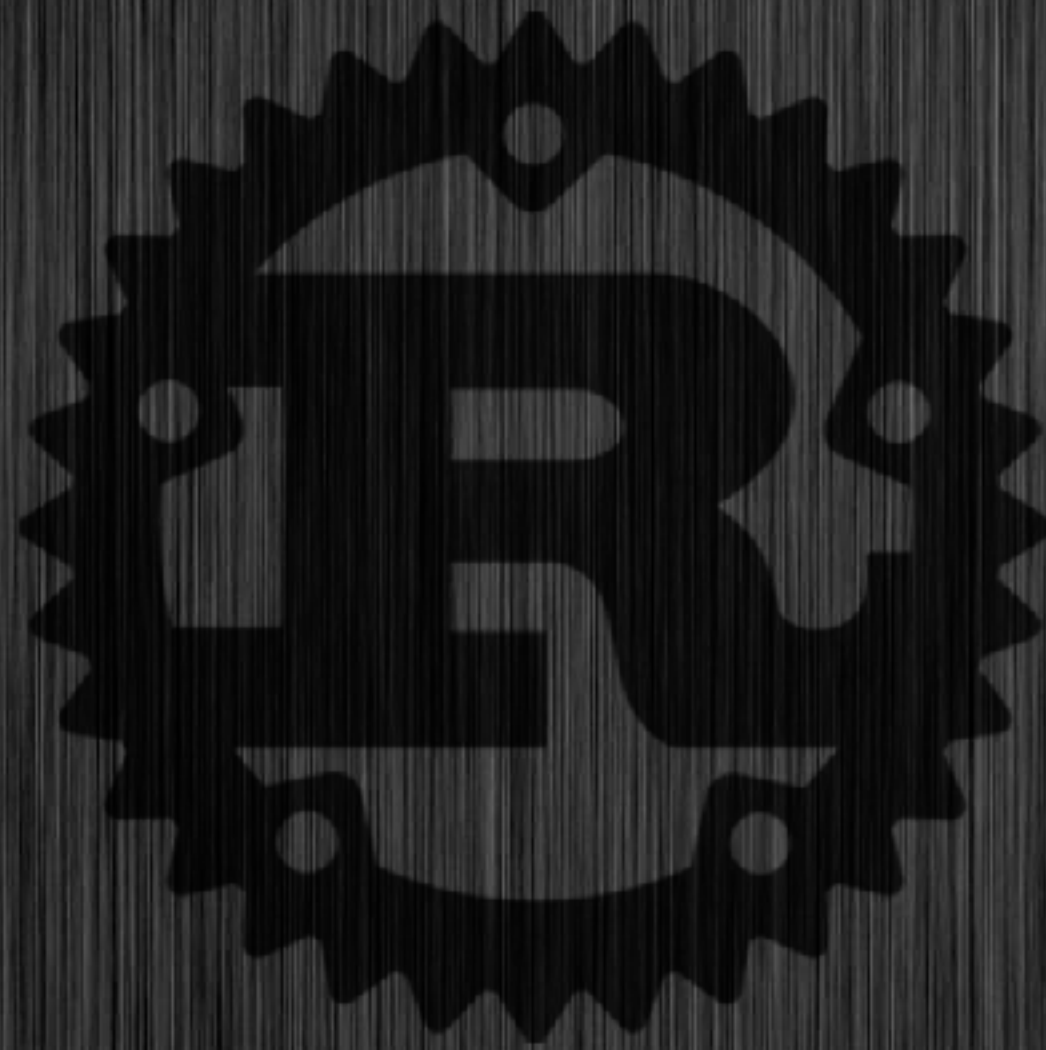




level 1
load a list of URLs
from a file

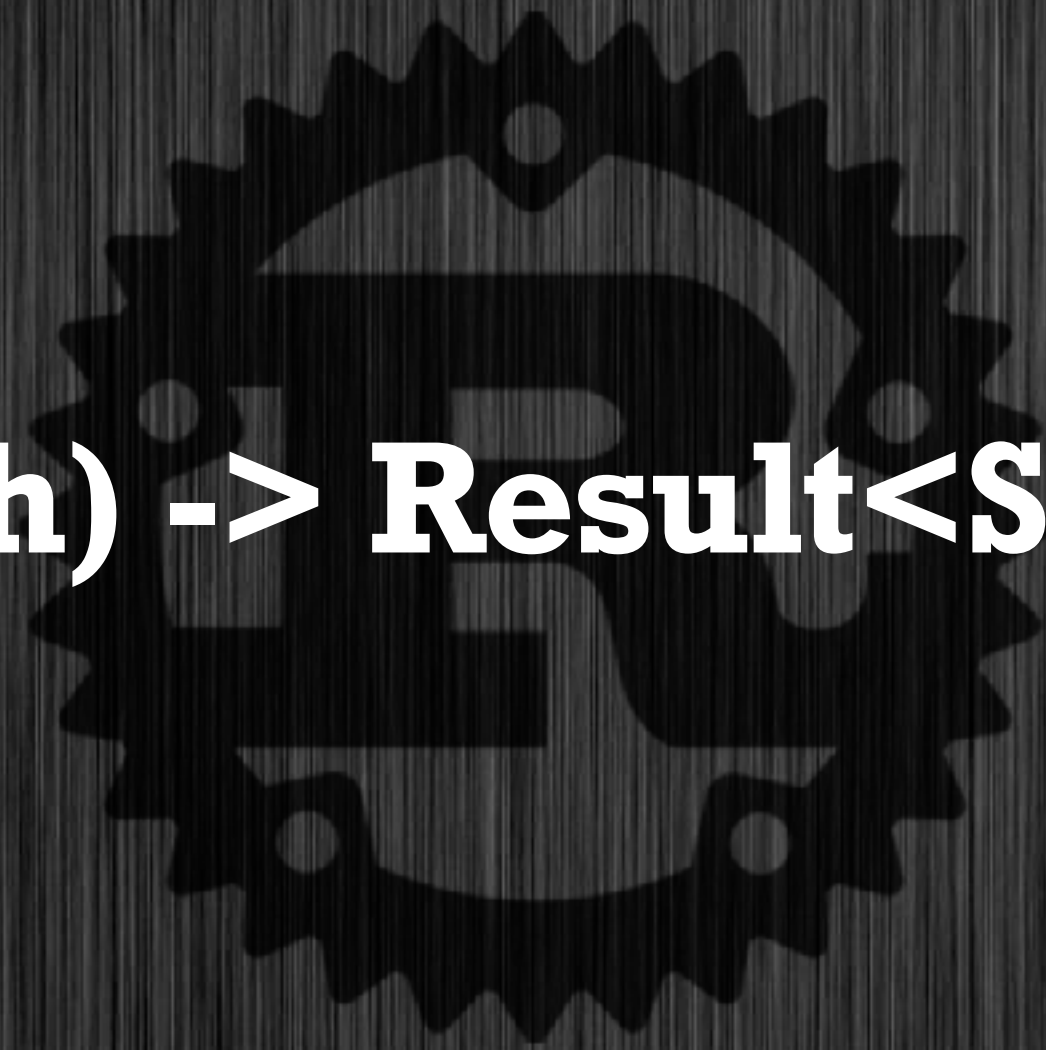
level 1: load a list of URLs from a file

- **cp level0.rs level1.rs**



level 1: load a list of URLs from a file

- **`std::fs::File::open(path) -> Result<Self, std::io::Error>`**



level 1: load a list of URLs from a file

- **std::io::{BufReader, BufRead}**

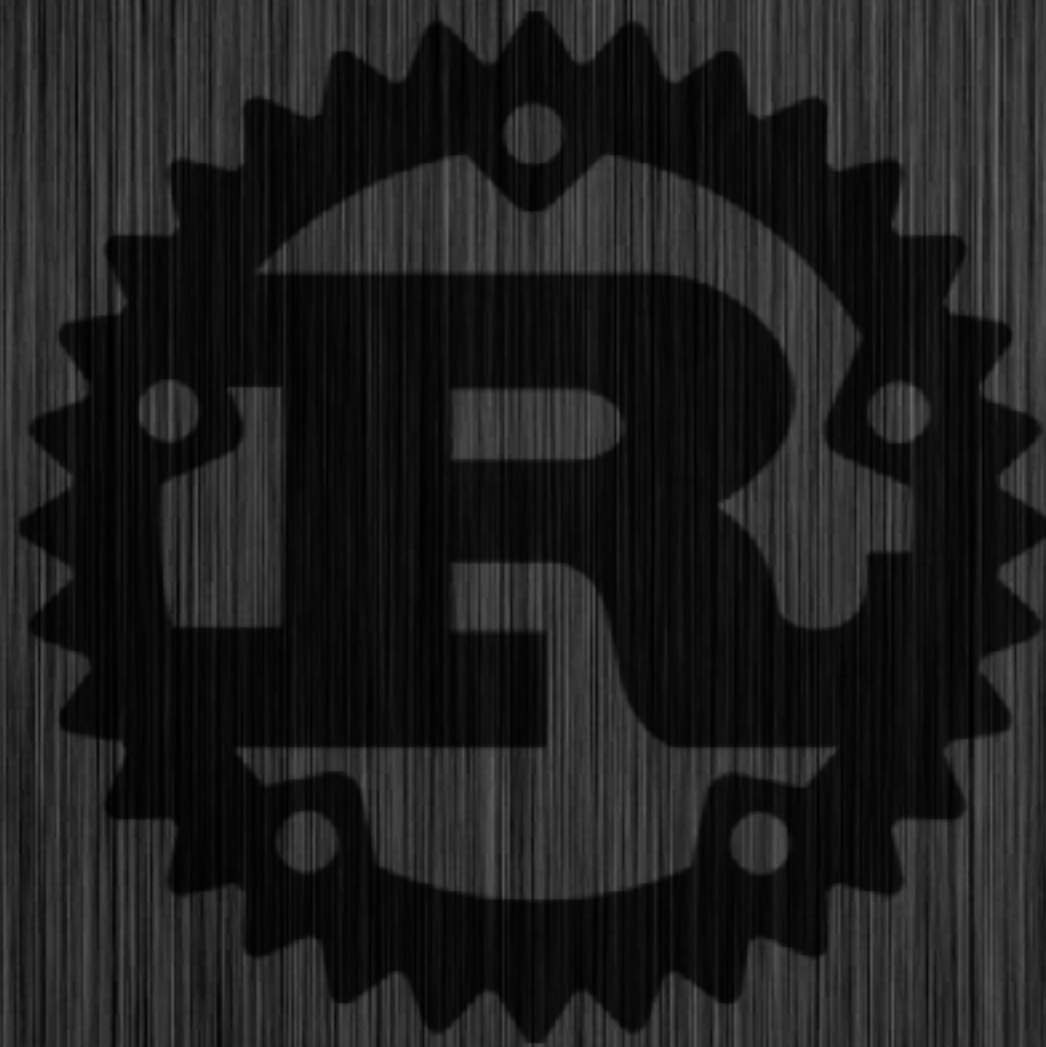
impl BufRead for BufReader { ... }

`use` both struct and trait

lines() -> impl Iterator<Item = Result<String, Error>>

level 1: load a list of URLs from a file

- **`Vec::new()`**
- **`Vec::push()`**
- **`for item in &vec { ... }`**



level 1: load a list of URLs from a file

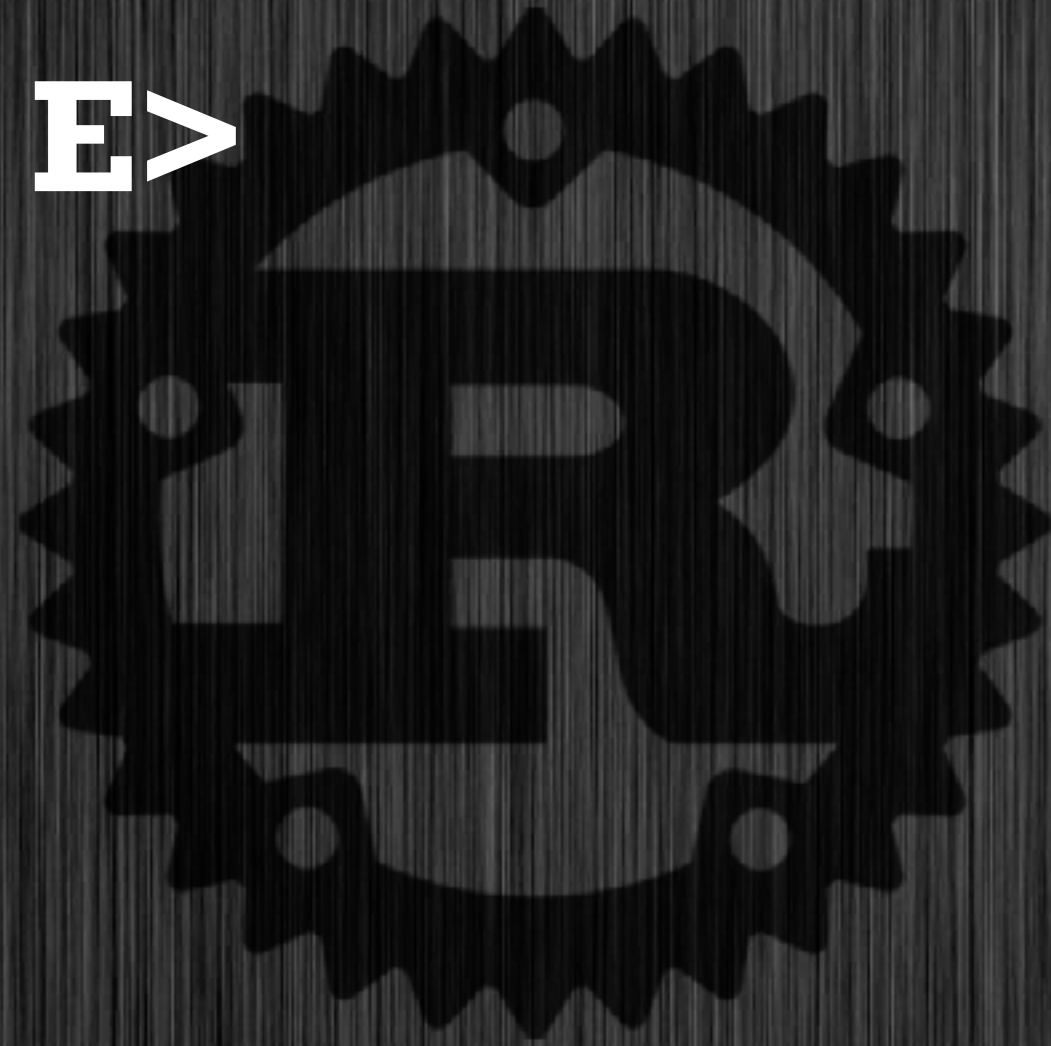
- **`std::fs::File::open(path) -> Result<Self, std::io::Error>`**
- **`std::io::{BufReader, BufRead}, BufRead::lines()`**
- **`Vec::new(), Vec::push()`**
- **`for item in &vec { ... }`**



level 2
basic error handling

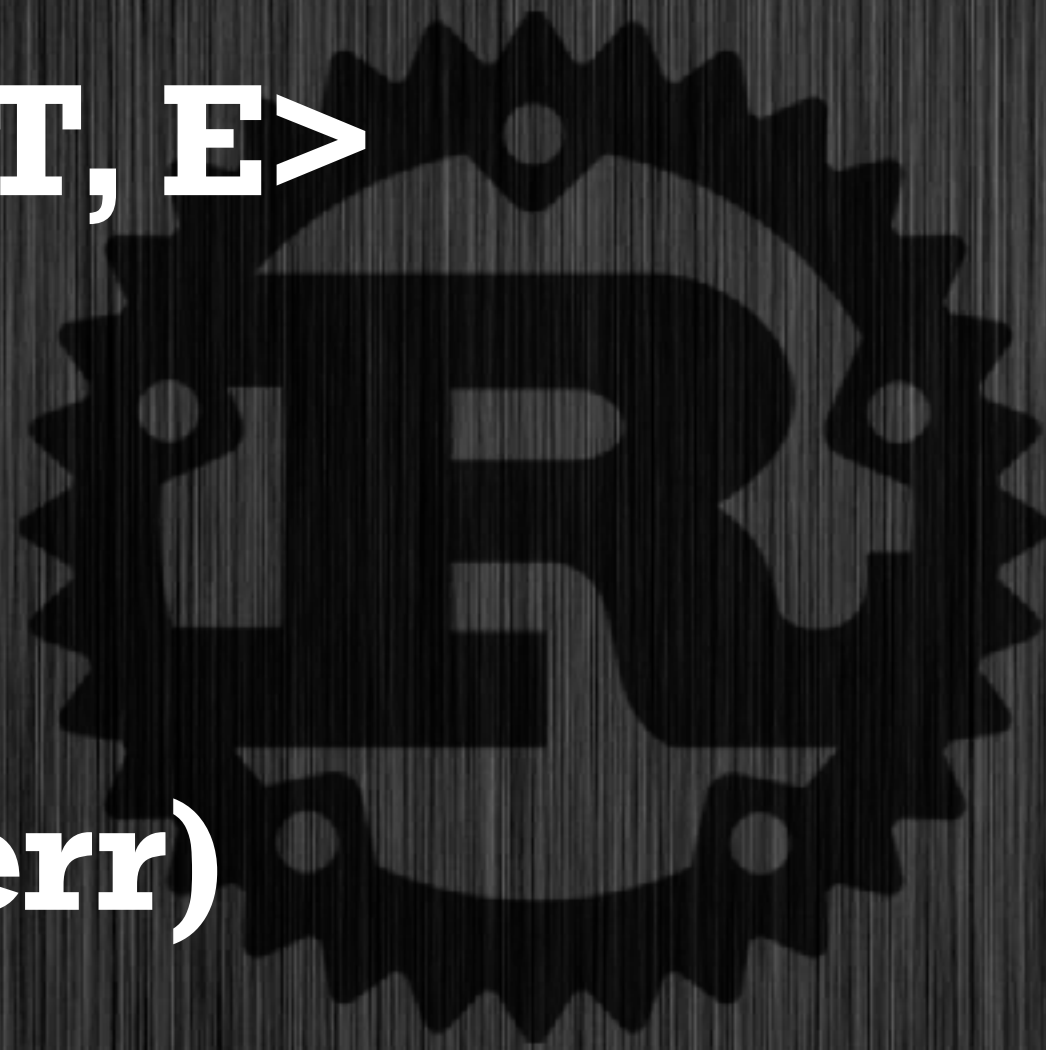
level 2: basic error handling

- **`fn main() -> Result<(), E>`**
- **`?`**
- **`std::io::Error`**



level 2: basic error handling

- **Option<T> -> Result<T, E>**
- **option.ok_or(err)**
- **option.ok_or_else(|| err)**



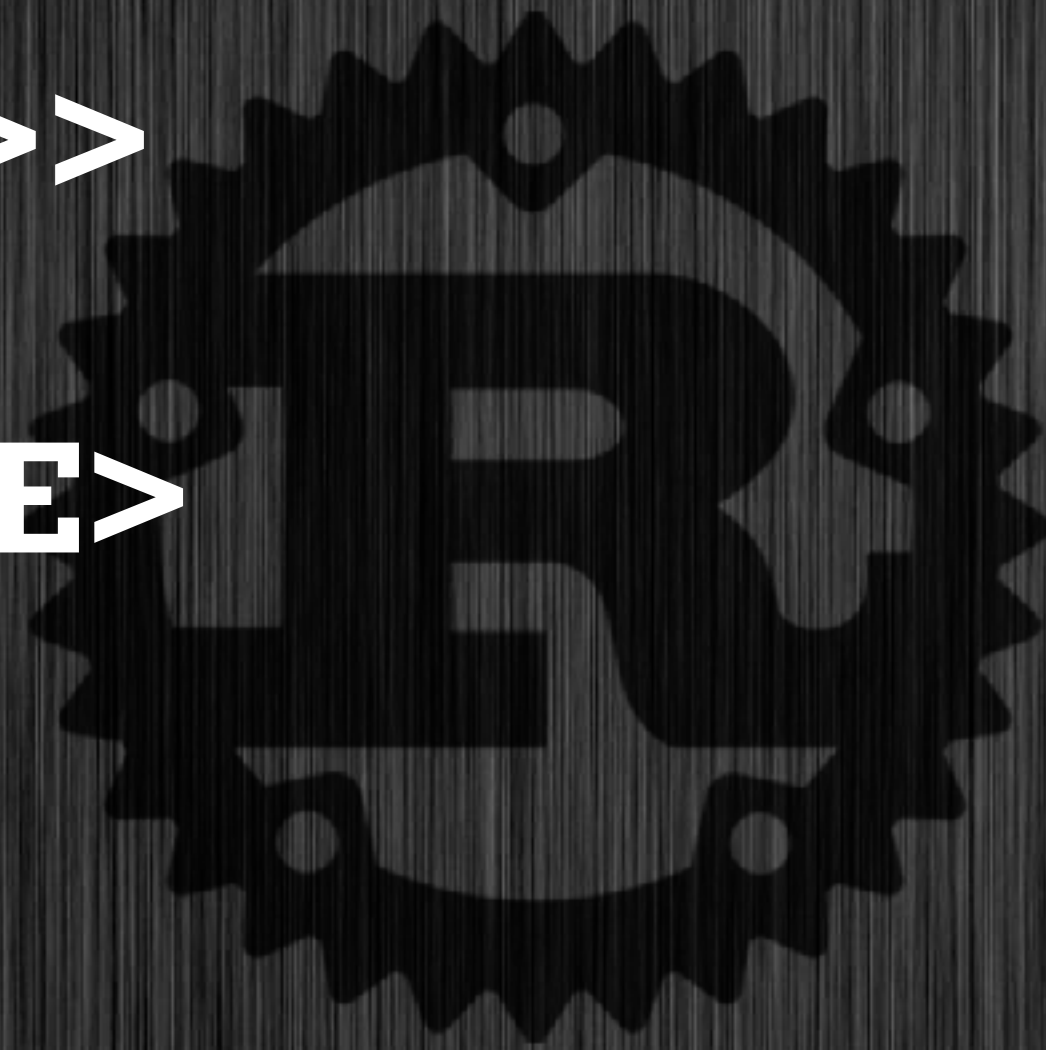
level 2: basic error handling

- **`std::io::{Error, ErrorKind}`**
- **`Error::new(ErrorKind::..., "Everything is on fire")`**



level 2: basic error handling

- have **`Vec<Result<T, E>>`**
- want **`Result<Vec<T>, E>`**
- **`Result::collect()`**

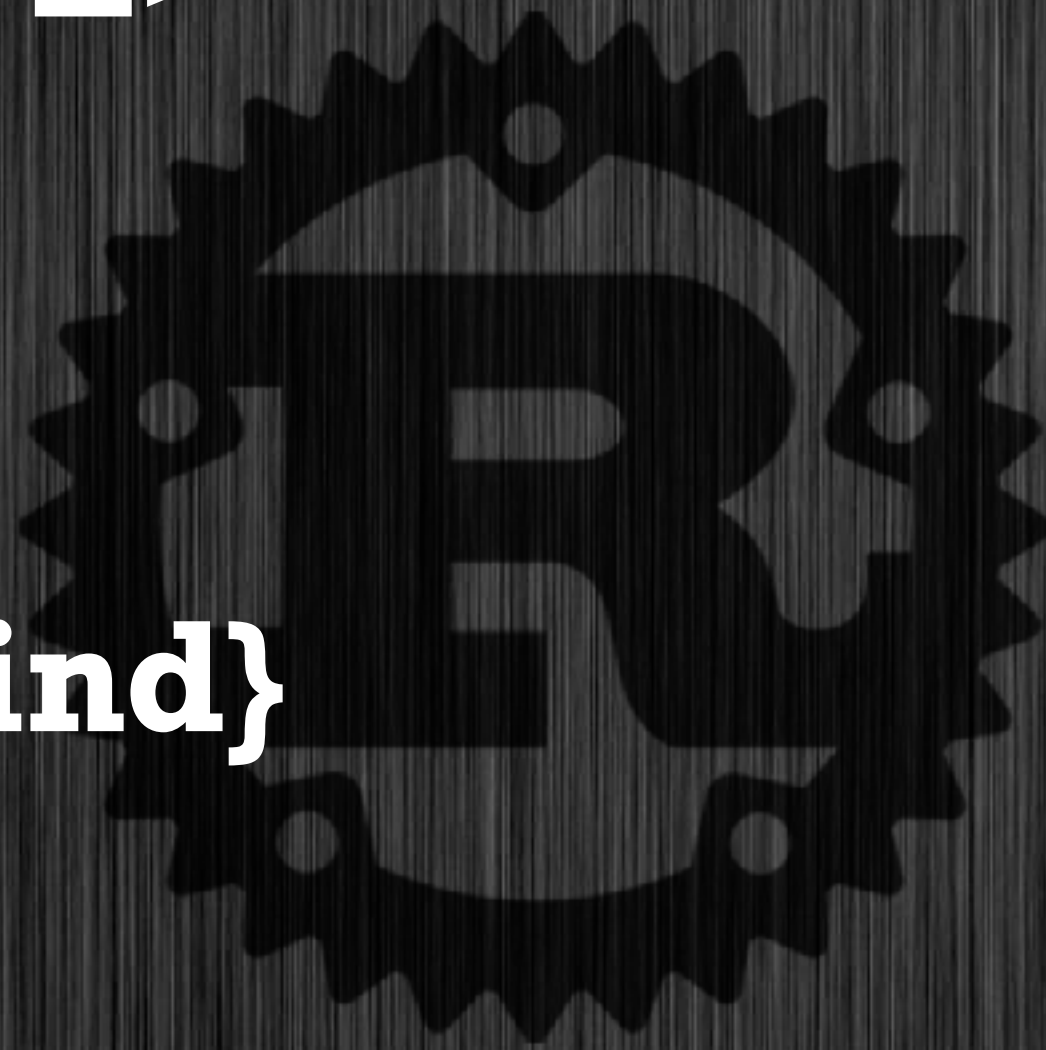


level 2: basic error handling

- have **IntoIterator**<Item = Result<T, E>>
- want **Result**<**FromIterator**<T>, E>
- **Result::collect()**

level 2: basic error handling

- **`fn main() -> Result<(), E>`**
- **`?`**
- **`std::io::{Error, ErrorKind}`**
- **`Result::collect()`**





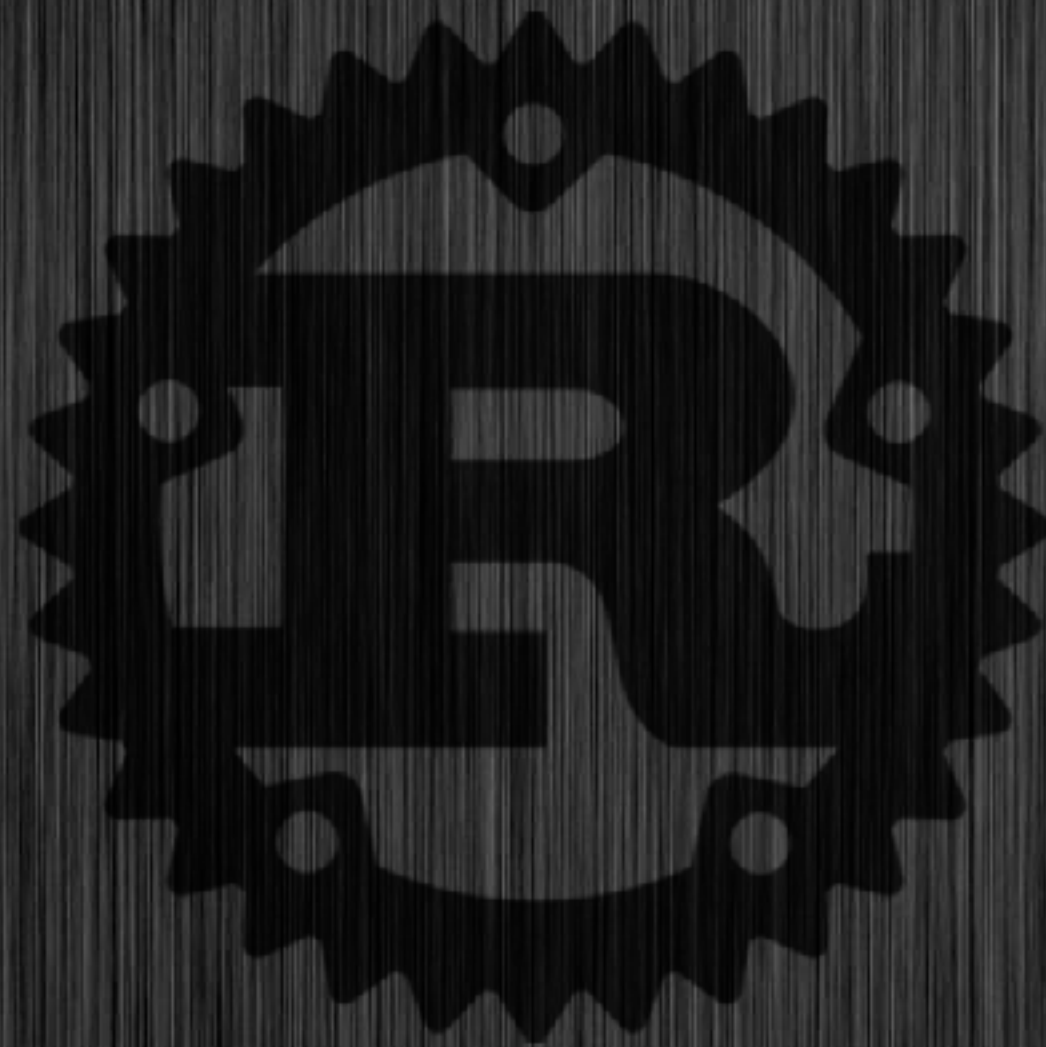
level 3
use an HTTP library

level 3: use an HTTP library

- **cargo add request**

- **Cargo.toml**

```
[dependencies]  
request = "0.11.12"
```



- **Cargo.lock**
- **cargo update**

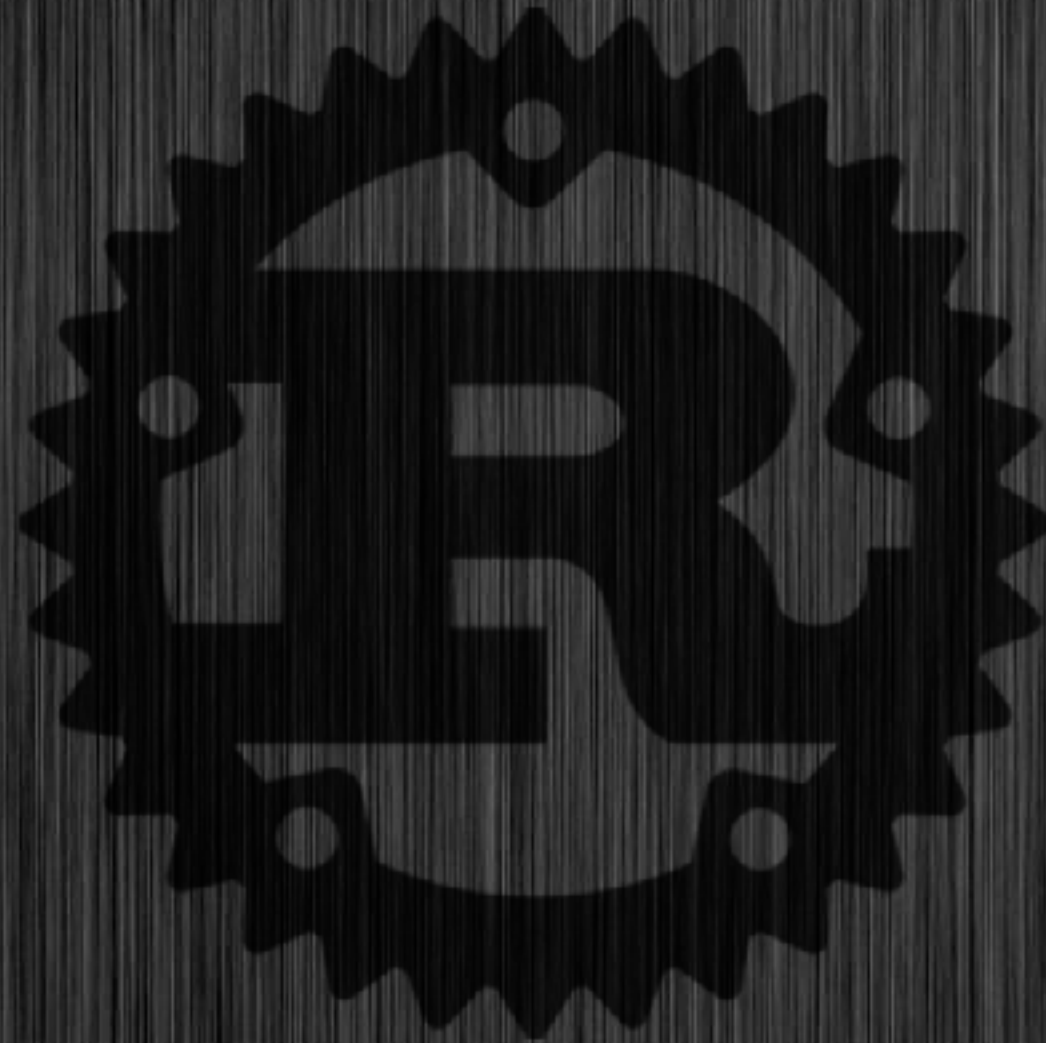
level 3: use an HTTP library

- **cargo add reqwest --features=blocking**

- **Cargo.toml**

[dependencies]

```
reqwest = {  
  version = "0.11.12",  
  features = ["blocking"]  
}
```



- **Cargo.lock**

- **cargo update**

level 3: use an HTTP library

- **let client = reqwest::blocking::Client::new();**
let resp = client.get(&url).send()?;
println!("{}", resp.status());

level 3: use an HTTP library

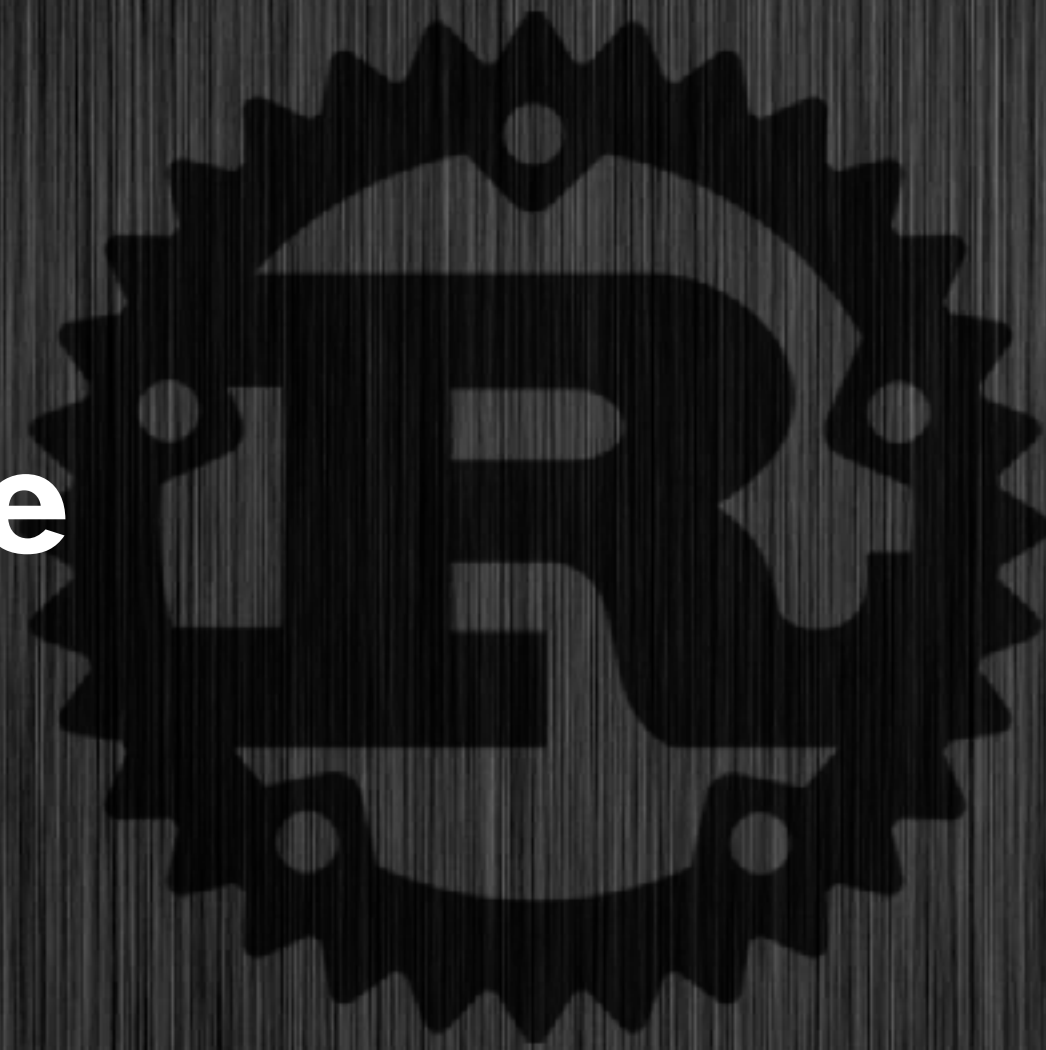
- **send(...) -> Result<
 request::blocking::Response,
 request::Error>**
- **request::Error != std::io::Error**
- **fn main() -> Result<(), Box<dyn Error>>**
- **?**



level 4
collect timings

level 4: collect timings

- **std::time::Instant**
::now()
.elapsed()
- **std::time::SystemTime**
::now()
.elapsed()
- **std::time::Duration (t2 - t1)**



level 4: collect timings

- **std::time::Instant (monotonic)**
::now() -> Instant
.elapsed() -> Duration
- **std::time::SystemTime (convertible to wall clock time)**
::now() -> SystemTime
.elapsed() -> Result<Duration, SystemTimeError>
- **std::time::Duration (t2 - t1)**

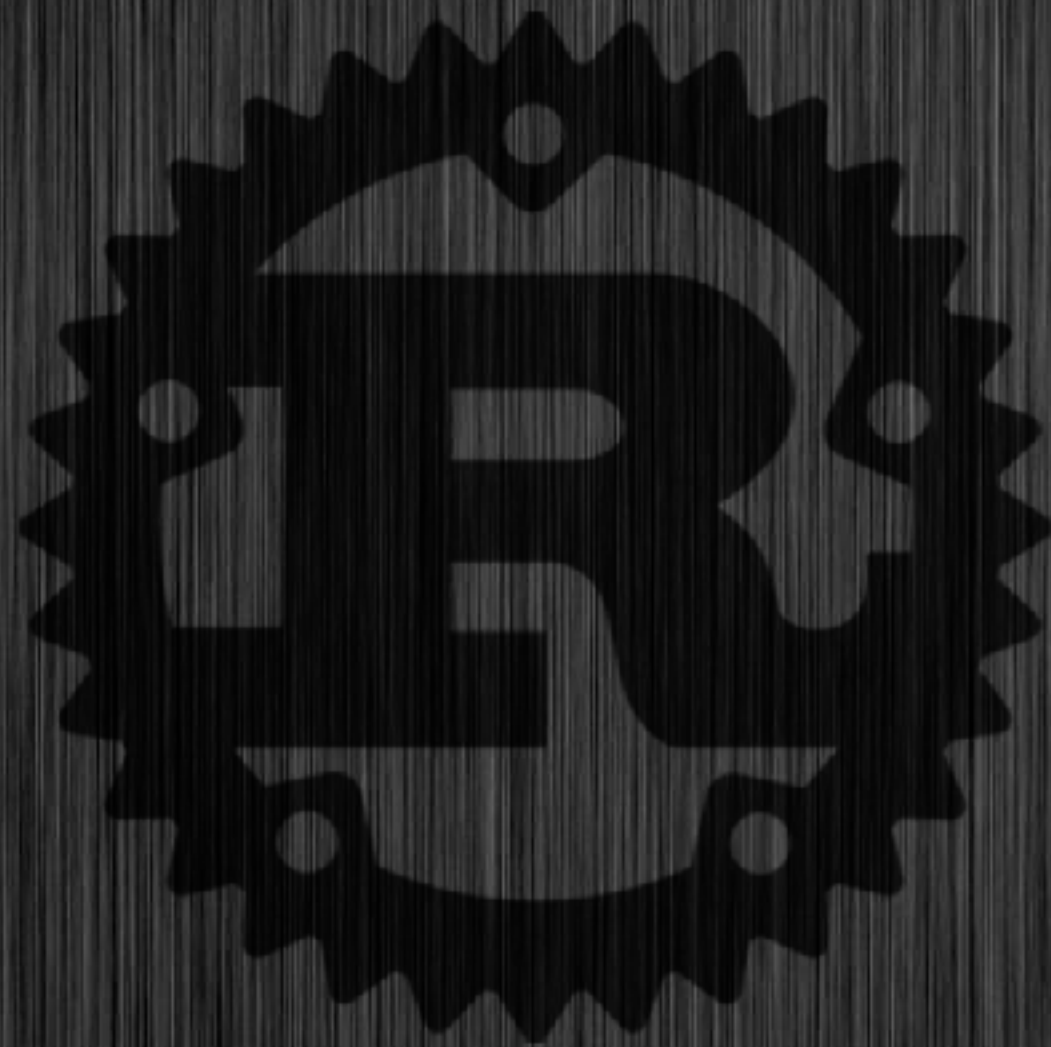
level 4: collect timings

- **struct Stats {
 elapsed: Duration,
 content_length: usize,
}**



sidebar: #[derive]able traits:

- **PartialEq, Eq**
- **PartialOrd, Ord**
- **Clone**
- **Copy**
- **Debug**
- **Default**
- **Hash**
- **custom derives (procedural macros)**



level 4: collect timings

- **#[derive(Debug)]** **// println!("{:?}", stats)**
struct Stats {
 elapsed: Duration,
 content_length: usize,
}
- **resp.text()?.len()**
- **fn get(client: &reqwest::blocking::Client, url: &str)**
-> Result<Stats, Box<dyn Error>>





level 5
aggregate
the timings

level 5: aggregate the timings

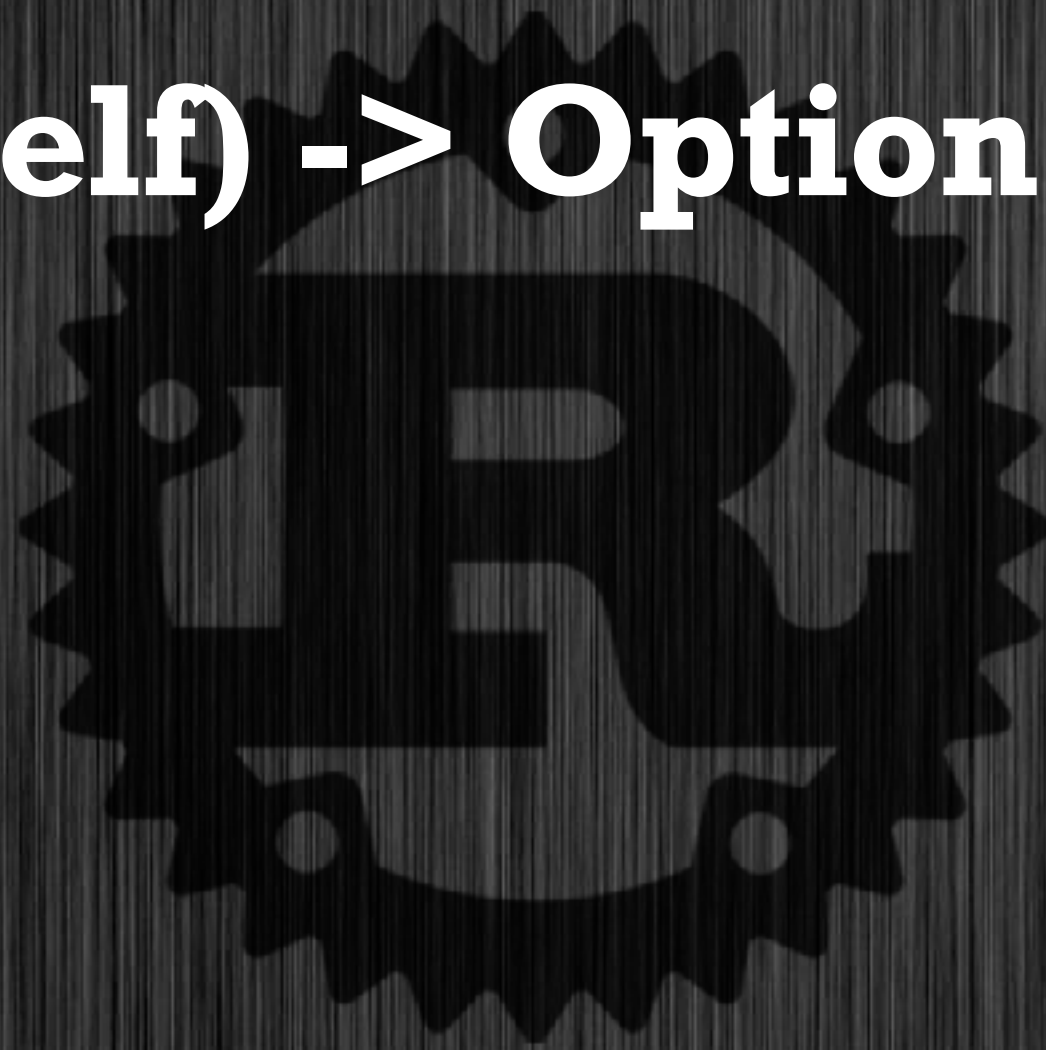
- **impl Stats {
 fn aggregate(&mut self, other: &Stats) {
 // ...
 }
}**
- **fn aggregate(self, other: &Stats) -> Self { ... }**
- **fn aggregate(left: &Stats, right: &Stats) -> Self { ... }**

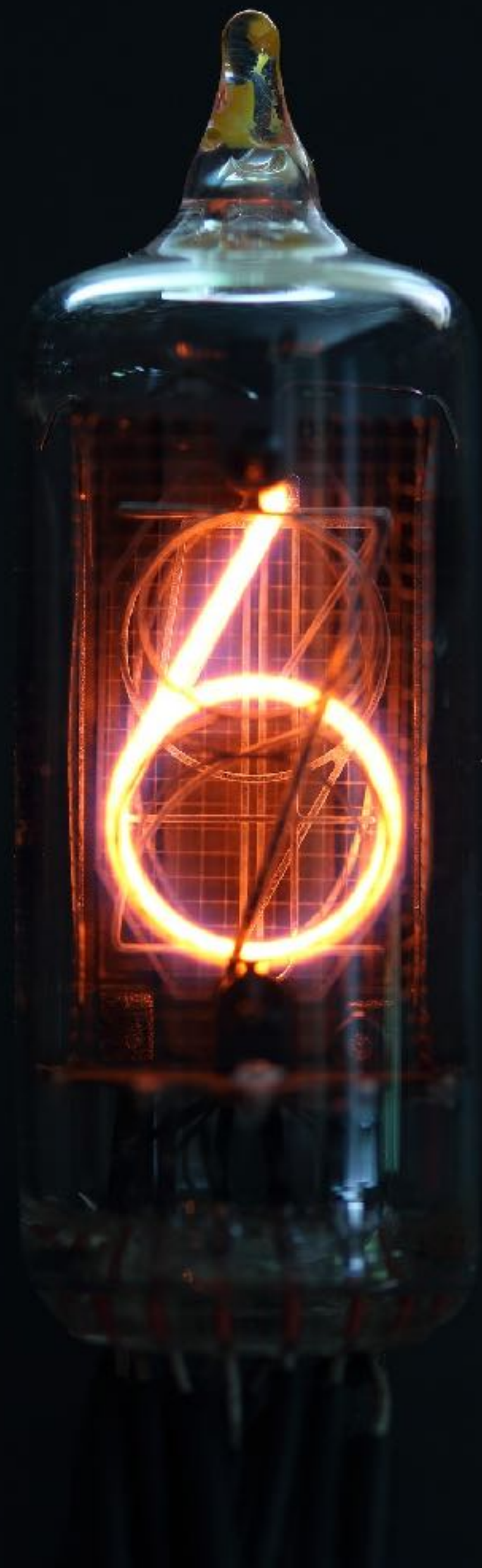
level 5: aggregate the timings

- **impl Stats {
 fn bytes_per_sec(&self) -> f64 {
 // self.content_length / self.elapsed
 }
}**
- **what if self.elapsed == 0?**

level 5: aggregate the timings

- **impl Stats {
 fn bytes_per_sec(&self) -> Option<f64> {
 // ...
 }
}**
- **.as_secs_f64()**
- **Option::unwrap_or_default()**





**level 6
testing**

level 6: testing

- **built into the language**
- **will look at it in more detail later**
- **just the absolute basics for now**



level 6: testing

- a test case is a function with the `#[test]` attribute
 - usually in a module with `#[cfg(test)]`
- a panic is a test failure
- `assert_eq!`, `assert_ne!`, `assert!` macros

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

level 6: testing

- **run the tests with cargo test**
- **it has several useful options**
 - **filtering test cases**
 - **controlling concurrency**
 - **controlling test output**





**level 7
callbacks**

level 7: callbacks

```
#include <iostream>
#include <string>

int main() {
    std::string hello("Hello, world!");

    auto lambda = [&](int i) {
        std::cout << hello << ", param = " << i << '\n';
    };

    lambda(1);
    lambda(2);
    std::cout << hello << '\n';
}
```

```
fn main() {
    let hello = String::from("Hello, world!");

    let lambda = |i| {
        println!("{}", param = {}, hello, i);
    };

    lambda(1);
    lambda(2);

    println!("{}", hello);
}
```

level 7: callbacks

```
#include <iostream>
#include <string>

int main() {
    std::string hello("Hello, world!");

    auto lambda = [=](int i) {
        std::cout << hello << ", param = " << i << '\n';
    };

    lambda(1);
    lambda(2);
    std::cout << hello << '\n';
}
```

```
fn main() {
    let hello = String::from("Hello, world!");

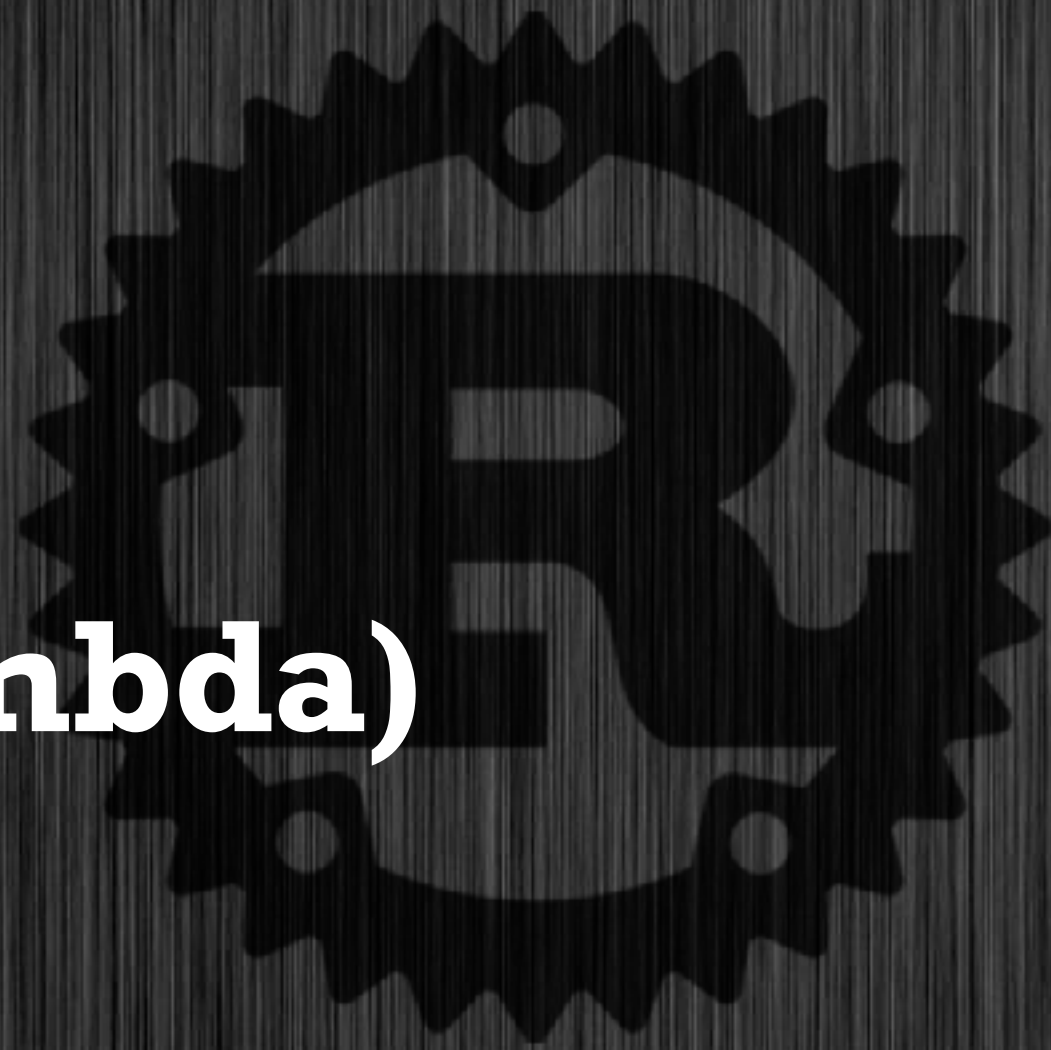
    let lambda = move |i| {
        println!("{}", param = {}", hello, i);
    };

    lambda(1);
    lambda(2);

    println!("{}", hello); // boom
}
```

level 7: callbacks

- **let lambda = |i| i+1;**
- **takes_callback(10, lambda)**



level 7: callbacks

- **let lambda: ??? = |i| i+1;**
- **fn takes_callback(i: i32, callback: ???) -> i32 {
 // ...
}**

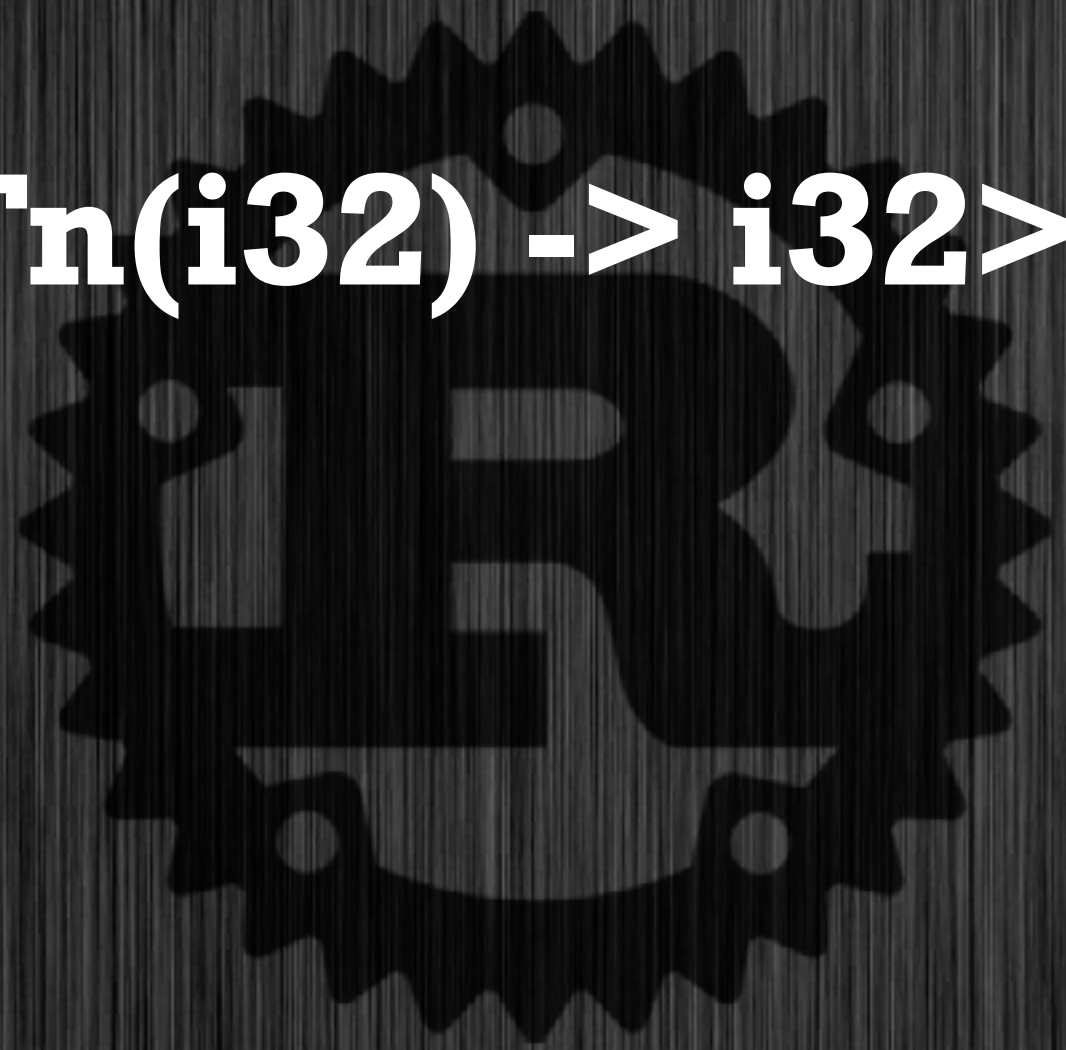
level 7: callbacks

- **let lambda:Voldemort = |i| i+1;**
Voldemort: Fn(i32) -> i32



level 7: callbacks

- **fn takes_callback<F: Fn(i32) -> i32>(i: i32, callback: F) -> i32 {
 callback(i)
}**



level 7: callbacks

- **fn takes_callback<F>(i: i32, callback: F) -> i32**
 where
 F: Fn(i32) -> i32
 {
 callback(i)
 }



level 7: callbacks

```
fn main() {  
    let hello = String::from("Hello, world!");  
  
    let lambda = |i| {  
        println!("{}", param = {}, hello, i);  
    };  
  
    lambda(1);  
    lambda(2);  
    println!("{}", hello);  
}
```

Fn(i32) -> ()

```
struct Voldemort<'a> {  
    hello: &'a String,  
}  
  
impl<'a> Voldemort<'a> {  
    fn call(&self, i: i32) {  
        println!("{}", param = {}, self.hello, i);  
    }  
}  
  
fn main() {  
    let hello = String::from("Hello, world!");  
    let lambda = Voldemort {  
        hello: &hello  
    };  
  
    lambda.call(1);  
    lambda.call(2);  
    println!("{}", hello);  
}
```

level 7: callbacks

```
fn main() {  
    let hello = String::from("Hello, world!");  
  
    let lambda = move |i| {  
        println!("{}", param = {}, hello, i);  
    };  
  
    lambda(1);  
    lambda(2);  
}
```

Fn(i32) -> ()

```
struct Voldemort {  
    hello: String,  
}  
  
impl Voldemort {  
    fn call(&self, i: i32) {  
        println!("{}", param = {}, self.hello, i);  
    }  
}  
  
fn main() {  
    let hello = String::from("Hello, world!");  
    let lambda = Voldemort {  
        hello,  
    };  
  
    lambda.call(1);  
    lambda.call(2);  
}
```

level 7: callbacks

```
fn main() {
    let mut hello = String::from("Hello, world!");

    let mut lambda = |i| {
        hello.push('!');
        println!("{}", param = {}, hello, i);
    };

    lambda(1);
    lambda(2);
    println!("{}", hello);
}
```

FnMut(i32) -> ()

```
struct Voldemort<'a> {
    hello: &'a mut String,
}

impl<'a> Voldemort<'a> {
    fn call(&mut self, i: i32) {
        self.hello.push('!');
        println!("{}", param = {}, self.hello, i);
    }
}

fn main() {
    let mut hello = String::from("Hello, world!");
    let mut lambda = Voldemort {
        hello: &mut hello
    };

    lambda.call(1);
    lambda.call(2);
    println!("{}", hello);
}
```

level 7: callbacks

```
fn main() {  
    let hello = String::from("Hello, world!");  
  
    let lambda = move |i| {  
        println!("{}", param = {}, hello, i);  
        drop(hello);  
    };  
  
    lambda(1);  
}
```

FnOnce(i32) -> ()

```
struct Voldemort {  
    hello: String,  
}  
  
impl Voldemort {  
    fn call(self, i: i32) {  
        println!("{}", param = {}, self.hello, i);  
        drop(self.hello);  
    }  
}  
  
fn main() {  
    let hello = String::from("Hello, world!");  
    let lambda = Voldemort {  
        hello,  
    };  
  
    lambda.call(1);  
}
```

level 7: callbacks

- **Fn()** takes **&self**: callable without restrictions
- **FnMut()** takes **&mut self**: borrows environment mutably
- **FnOnce()** takes **self**: callable only once

level 7: callbacks

- **get(&client, &url, |req_stats| {
 totals.aggregate(&req_stats);
 Ok()
})?;**

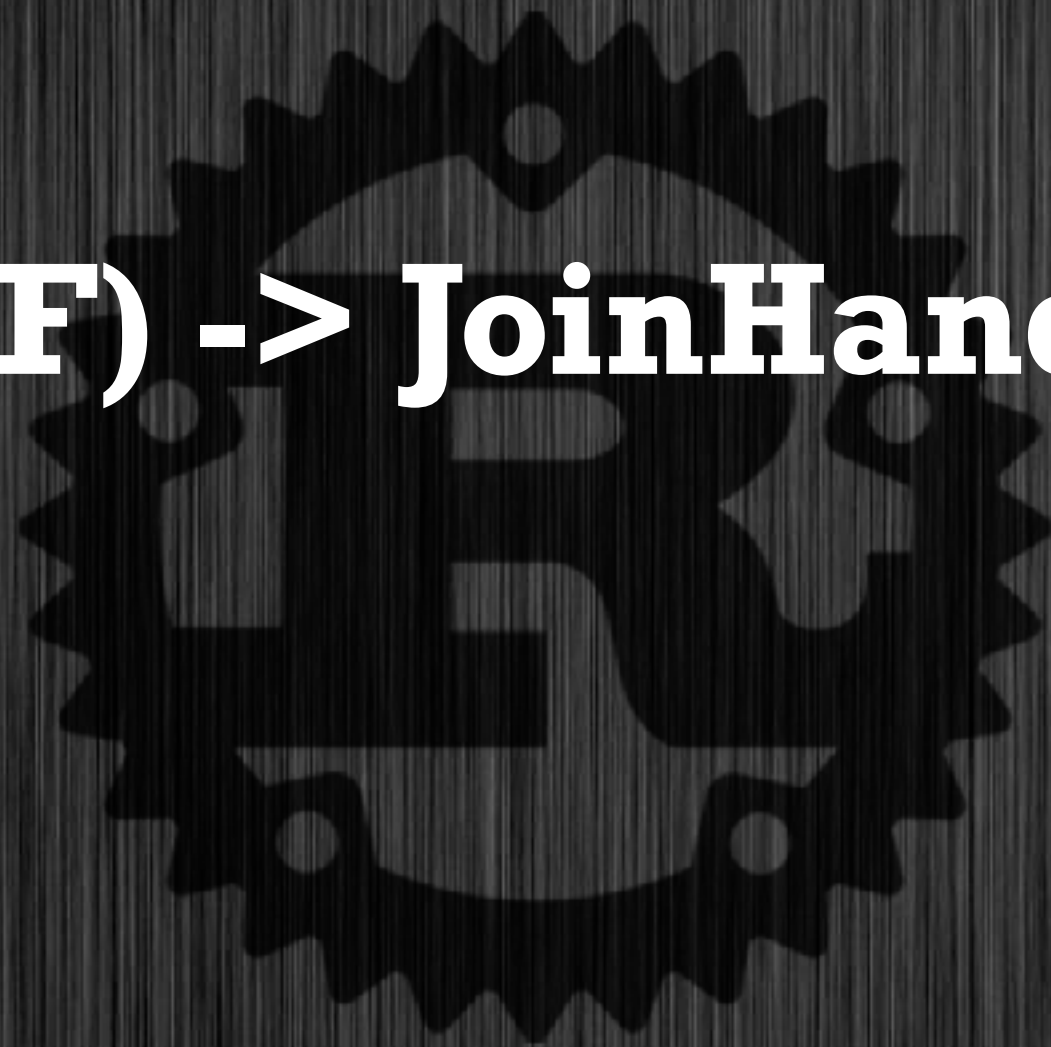




**level 8
threads**

level 8: threads

- **std::thread**
- **pub fn spawn<F, T>(f: F) -> JoinHandle<T>**
where
 F: FnOnce() -> T,
 F: Send + 'static,
 T: Send + 'static,



level 8: threads

- **Send trait**
can be accessed from multiple threads
****sequentially****
almost everything is Send
- **Sync trait**
can be accessed from multiple threads
****concurrently****
almost nothing is both Sync and mutable

level 8: threads

- **Send/Sync examples:**

using thread locals -> !Send

interior mutability without atomics:

compile-time borrow-checked (Cell<T>) -> Send, !Sync

runtime borrow-checked (Rc<T>) -> !Send, !Sync

- **T: Sync is equivalent to &T: Send**
(is it safe to borrow the object to a different thread?)

level 8: threads

- **'static lifetime for**
 - **owned objects**
 - **global constants** (`let hello: &'static str = "hello"`)
- **no borrowing across thread boundaries**
but see `std::thread::scope` (new in 1.63)

level 8: threads

- **Mutex<T>**
.lock() -> Result<MutexGuard, PoisonError>

PoisonError: a thread panicked while holding the mutex

- **mutex.lock().unwrap()**
- **can't be cloned**

level 8: threads

- **Mutex<T>** can't be cloned
so how do you share it across threads?
- **Arc<T>**: Atomic Reference Counter
- only gives out immutable references
that's OK, since **Mutex::lock()** takes **&self**
(but allows mutable access to the locked object)

level 8: threads

- **`Arc<Mutex<T>>`**
"a thread-safe T"
- **`let obj = Arc::new(Mutex::new(obj));`**
- **(for every thread)**
`let thread_obj = obj.clone();`
`thread::spawn(move || {`
 `let obj = thread_obj.lock().unwrap();`
 `// ...`
`}`

level 8: threads

- **Vec<JoinHandle<T>>**
most basic thread pool ever
- **for handle in vec_of_handles.into_iter() {
 handle.join()?;
}**



```
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: &mut i32)
{
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}

fn main() -> Result<(), &'static str>
{
    let mut counter = 0;
    let mut threads = Vec::new();

    for _ in 0 .. NTHREADS {
        threads.push(thread::spawn(|| spin(&mut counter)));
    }

    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }

    assert_eq!(counter, LOOPS * NTHREADS);
    Ok(())
}
```



```
use std::sync::{Arc, Mutex};
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: Arc<Mutex<i32>>)
{
    let mut counter = counter.lock().unwrap();
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}

fn main() -> Result<(), &'static str>
{
    let counter = Arc::new(Mutex::new(0));
    let mut threads = Vec::new();

    for _ in 0 .. NTHREADS {
        let counter = counter.clone();
        threads.push(thread::spawn(move || spin(counter)));
    }

    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }

    assert_eq!(*counter.lock().unwrap(), LOOPS * NTHREADS);
    Ok(())
}
```



```
use std::sync::{Arc, Mutex};
use std::thread;
```

```
const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;
```

```
fn spin(counter: Arc<Mutex<i32>>)
```

```
{
    let mut counter = counter.lock().unwrap();
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}
```

```
fn main() -> Result<(), &'static str>
{
```

```
    let counter = Arc::new(Mutex::new(0));
    let mut threads = Vec::new();
```

```
    for _ in 0 .. NTHREADS {
        let counter = counter.clone();
        threads.push(thread::spawn(move || spin(counter)));
    }
```

```
    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }
```

```
    assert_eq!(*counter.lock().unwrap(), LOOPS * NTHREADS);
    Ok(())
}
```

Mutex: protects an object with a lock



make a Send object Sync

```
use std::sync::{Arc, Mutex};
use std::thread;
```

```
const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;
```

```
fn spin(counter: Arc<Mutex<i32>>)
```

```
{
    let mut counter = counter.lock().unwrap();
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}
```

```
fn main() -> Result<(), &'static str>
{
```

```
    let counter = Arc::new(Mutex::new(0));
    let mut threads = Vec::new();
```

```
    for _ in 0 .. NTHREADS {
        let counter = counter.clone();
        threads.push(thread::spawn(move || spin(counter)));
    }
```

```
    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }
```

```
    assert_eq!(*counter.lock().unwrap(), LOOPS * NTHREADS);
    Ok(())
}
```

Arc: thread-safe reference counter

share a Sync object
across threads

```
use std::sync::{Arc, Mutex};
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: Arc<Mutex<i32>>)
{
    let mut counter = counter.lock().unwrap();
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}

fn main() -> Result<(), &'static str>
{
    let counter = Arc::new(Mutex::new(0));
    let mut threads = Vec::new();

    for _ in 0 .. NTHREADS {
        let counter = counter.clone();
        threads.push(thread::spawn(move || spin(counter)));
    }

    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }

    assert_eq!(*counter.lock().unwrap(), LOOPS * NTHREADS);
    Ok(())
}
```



**clone the Arc wrapper,
not the protected object**

**still protected
by one mutex**

```
use std::sync::{Arc, Mutex};
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: Arc<Mutex<i32>>)
{
    let mut counter = counter.lock().unwrap();
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}

fn main() -> Result<(), &'static str>
{
    let counter = Arc::new(Mutex::new(0));
    let mut threads = Vec::new();

    for _ in 0 .. NTHREADS {
        let counter = counter.clone();
        threads.push(thread::spawn(move || spin(counter)));
    }

    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }

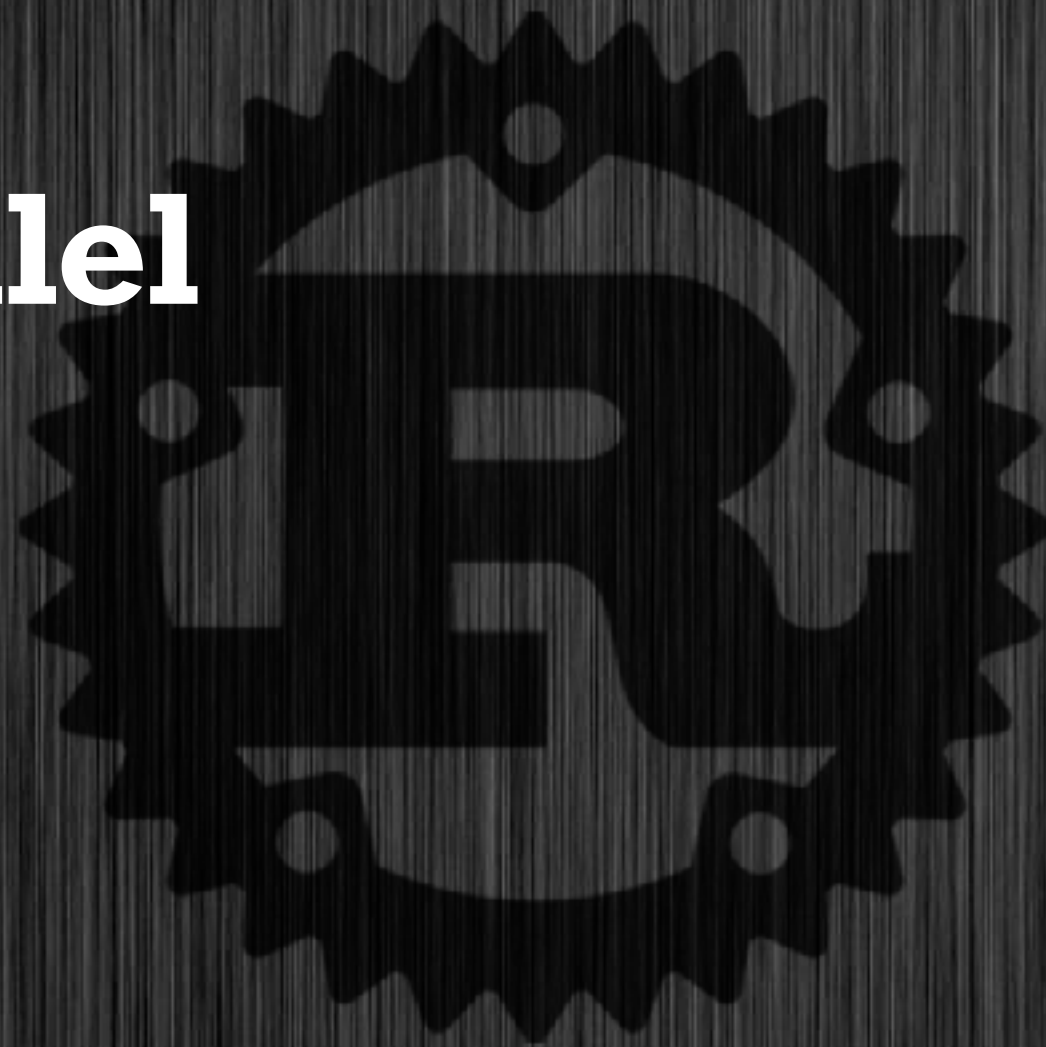
    assert_eq!(*counter.lock().unwrap(), LOOPS * NTHREADS);
    Ok(())
}
```



**cannot access the counter
without proper locking**

level 8: threads

- **fetch all URLs in parallel**
- **one thread per URL**
- **`Arc<Mutex<Stats>>`**



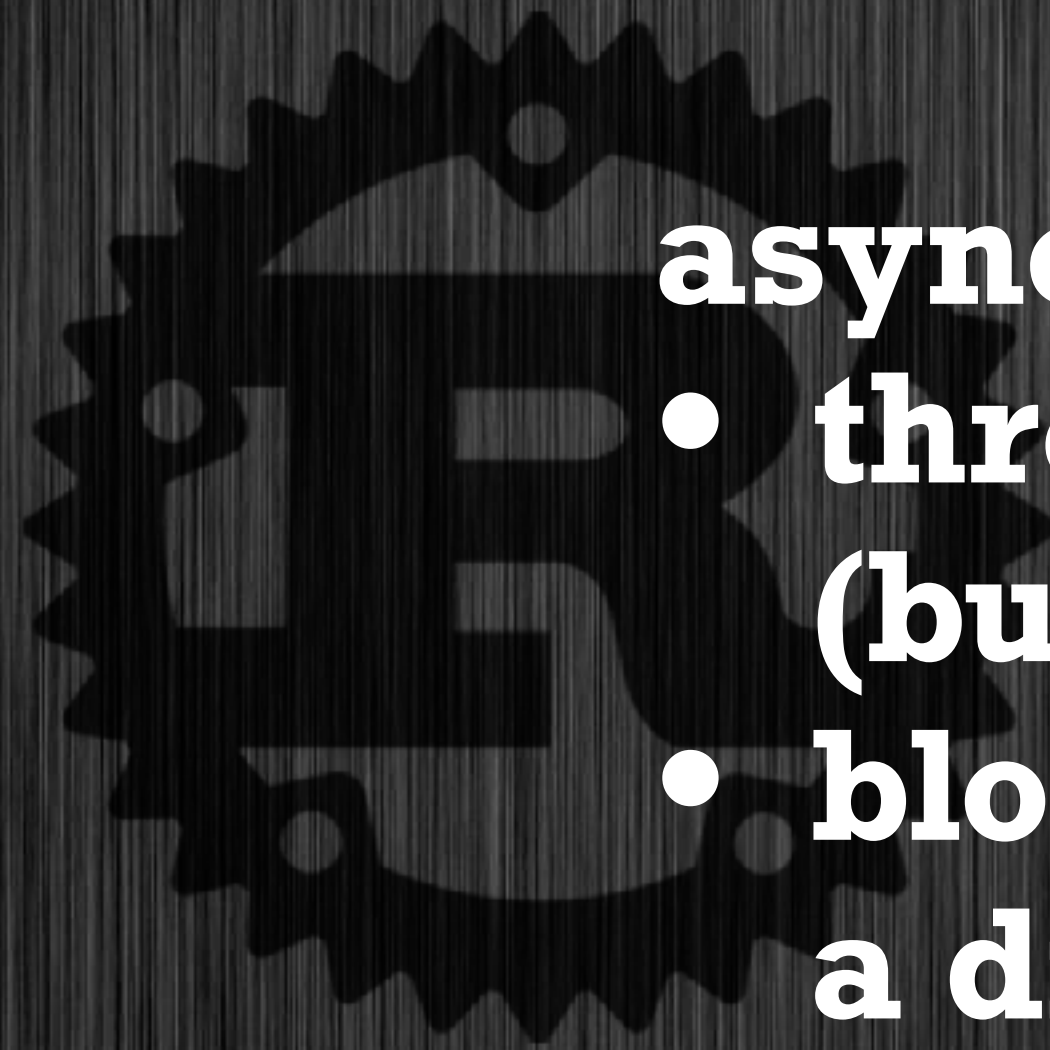


level 9
into the future(s)

level 9: into the future(s)

synchronous model

- **threads required for concurrency**
- **blocking yields to the OS kernel**
- **can be preempted**

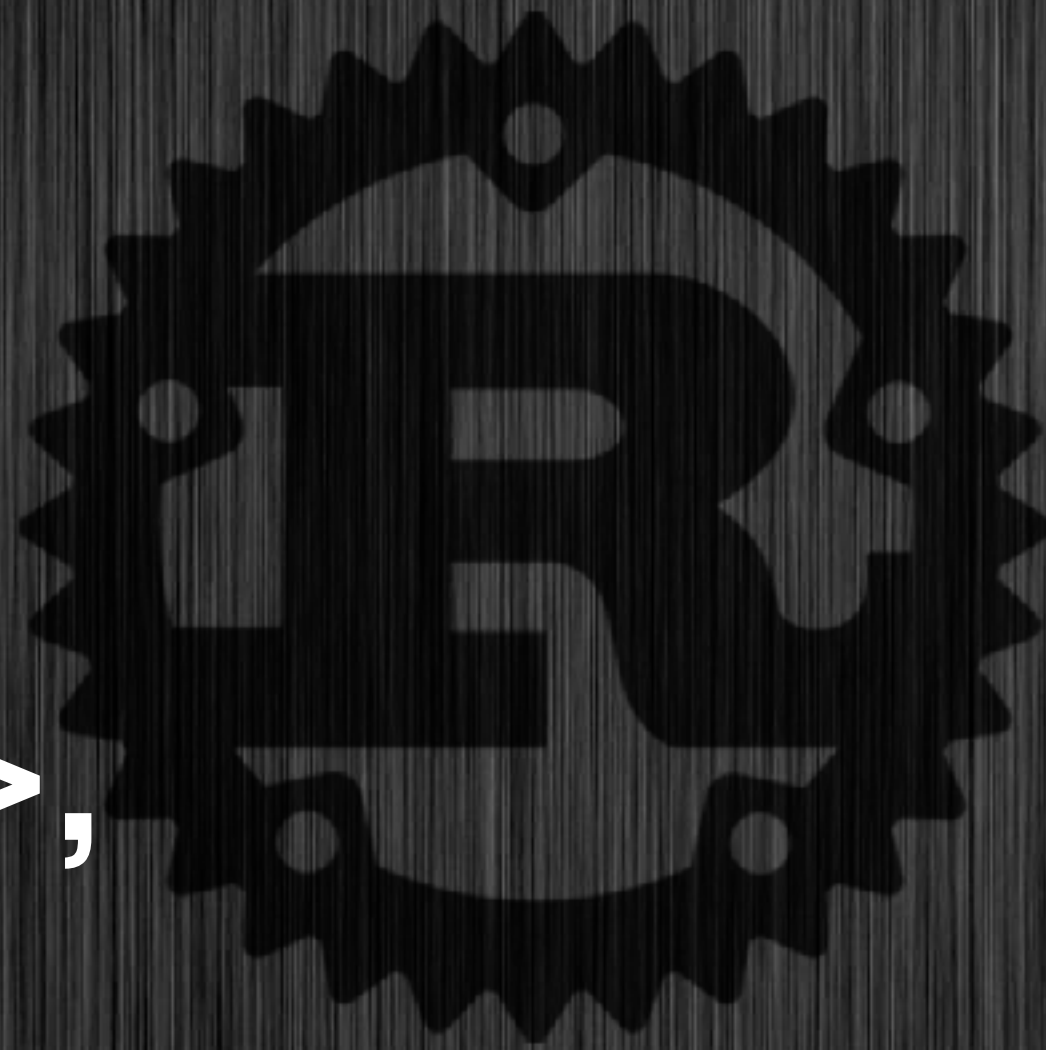


asynchronous model

- **threads optional (but supported)**
- **blocking yields to a different task**
- **cannot be preempted**

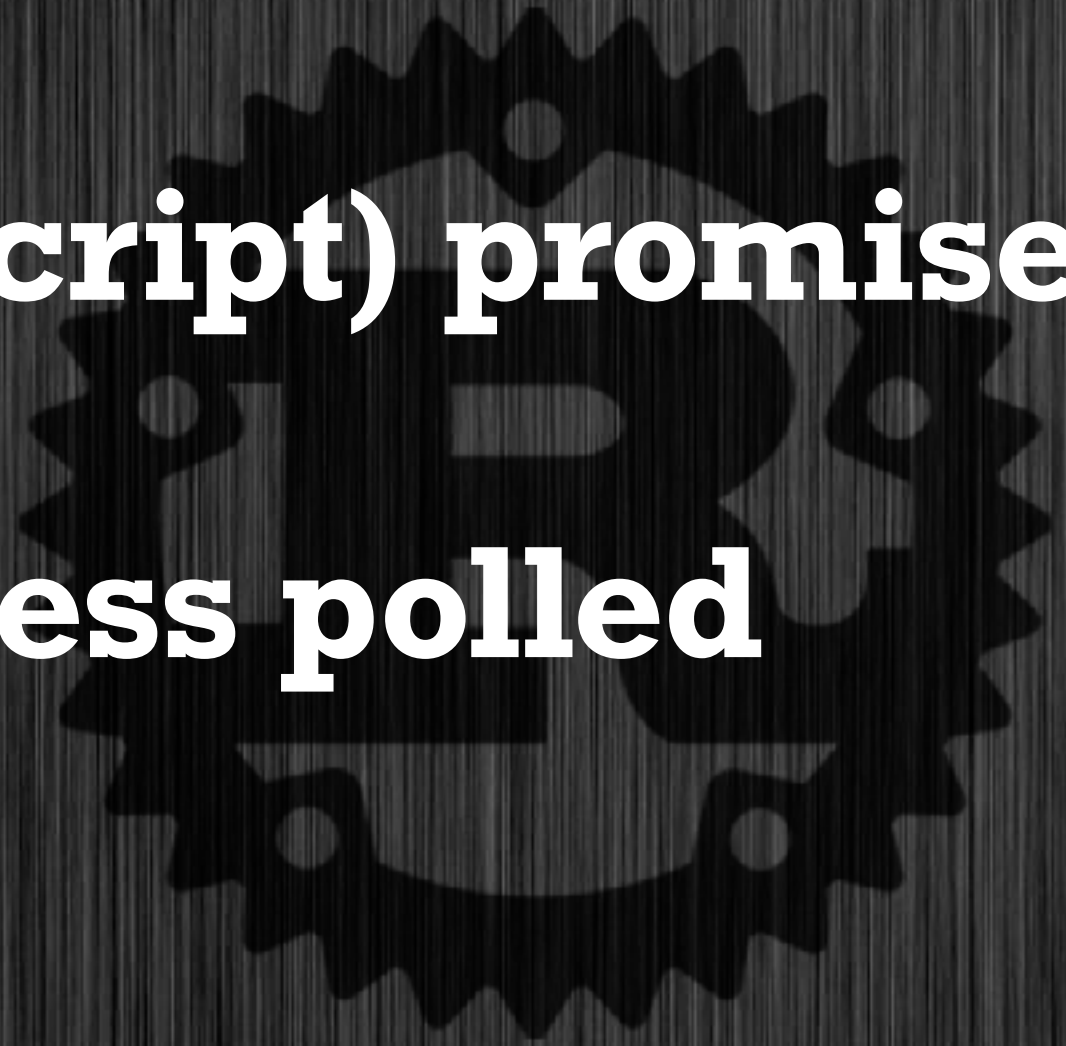
level 9: into the future(s)

- **std::future::Future**
- **fn poll(
 self: Pin<&mut Self>,
 cx: &mut Context)
-> Poll<Self::Output>;**



level 9: into the future(s)

- **(Rust) future vs (JavaScript) promise**
- **futures do nothing unless polled**



level 9: into the future(s)

- **impl Future for MyFuture**
- **poll() must:**
 - **return Poll::Ready(T) without blocking, or**
 - **ensure cx.waker().wake() will get called eventually (usually by forwarding to another future)**

level 9: into the future(s)

an asynchronous function is effectively a state machine

```
fn handle_request(conn: &mut Connection) {  
    let req = conn.read_request();           // can block  
    let resp = format!("got: {}", req);  
    conn.send_response(resp)                 // can block  
}
```

every blocking point is a boundary between states

level 9: into the future(s)



```
enum HandleRequestFut {  
    Start { conn: &mut Connection },  
    ReadingRequest { conn: &mut Connection, req_fut: ReadRequestFut },  
    SendingResponse { send_fut: SendResponseFut },  
    End,  
}
```



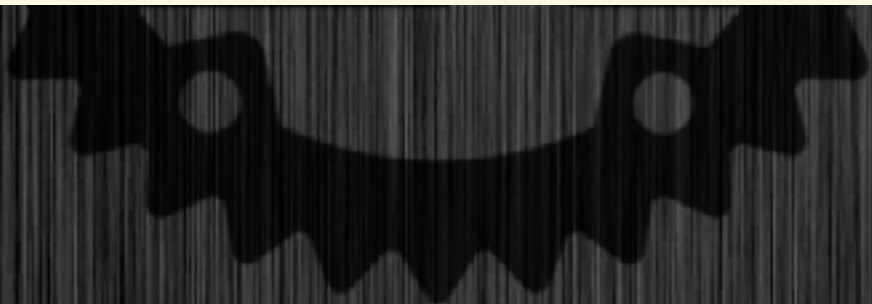
level 9: into the future(s)

```
impl Future for HandleRequestFut {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {
        loop {
            match self {
                HandleRequestFut::Start { conn } => todo!(),
                HandleRequestFut::ReadingRequest { conn, req_fut } => todo!(),
                HandleRequestFut::SendingResponse { ref send_fut } => todo!(),
                HandleRequestFut::End => todo!(),
            }
        }
    }
}
```

level 9: into the future(s)



```
HandleRequestFut::Start { conn } => {  
    let req_fut = conn.read_request();           // returns a future  
    *self = HandleRequestFut::ReadingRequest { conn, req_fut };  
}
```



level 9: into the future(s)

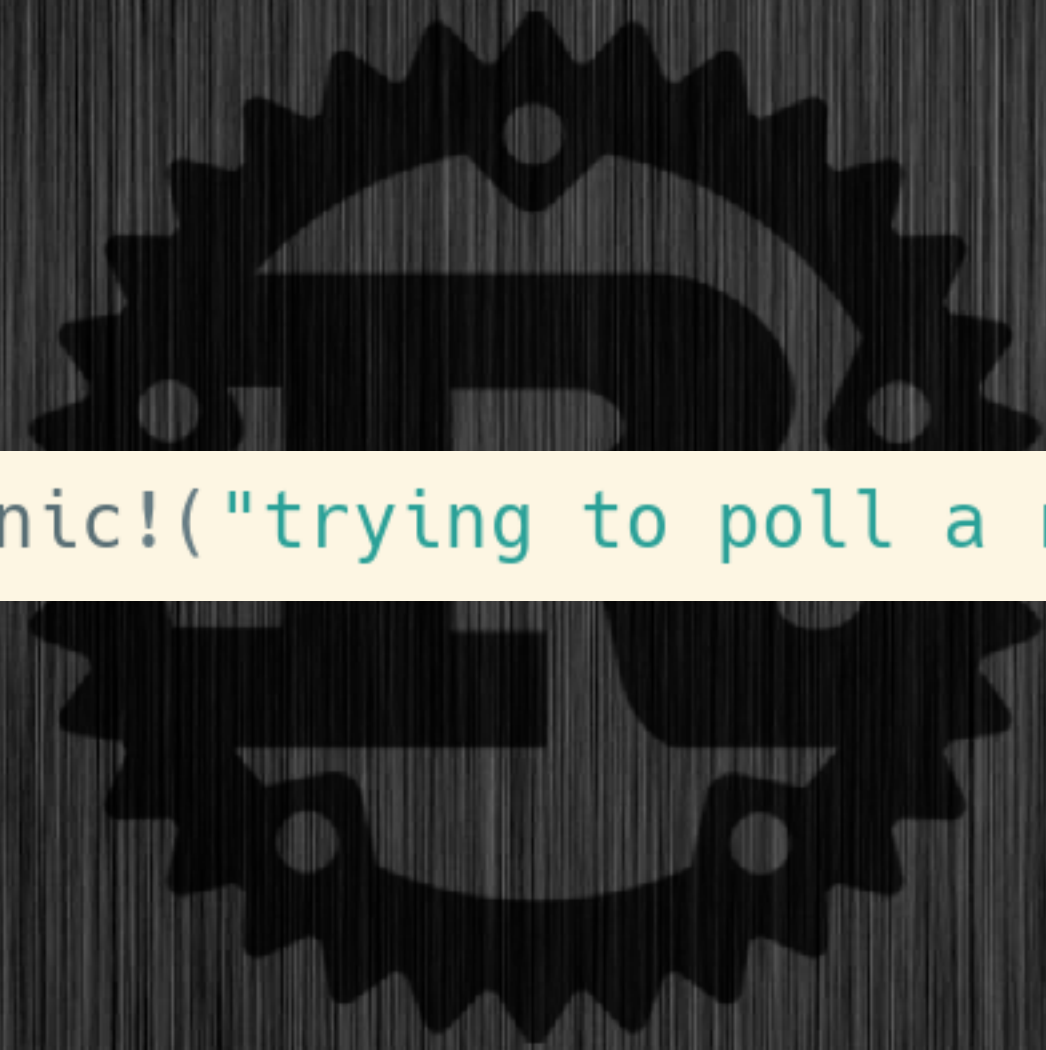
```
HandleRequestFut::ReadingRequest { conn, req_fut } => {  
    match req_fut.poll(cx) {  
        Poll::Pending => return Poll::Pending,  
        Poll::Ready(req) => {  
            let resp = format!("got: {}", req);  
            let send_fut = conn.send_response(resp); // returns a future  
            *self = HandleRequestFut::SendingResponse { send_fut }  
        }  
    }  
}
```

level 9: into the future(s)

```
HandleRequestFut::SendingResponse { ref send_fut } => {  
    match send_fut.poll(cx) {  
        Poll::Pending => return Poll::Pending,  
        Poll::Ready(_) => {  
            *self = HandleRequestFut::End;  
            return Poll::Ready(());  
        }  
    }  
}
```

level 9: into the future(s)

```
HandleRequestFut::End => panic!("trying to poll a resolved future"),
```



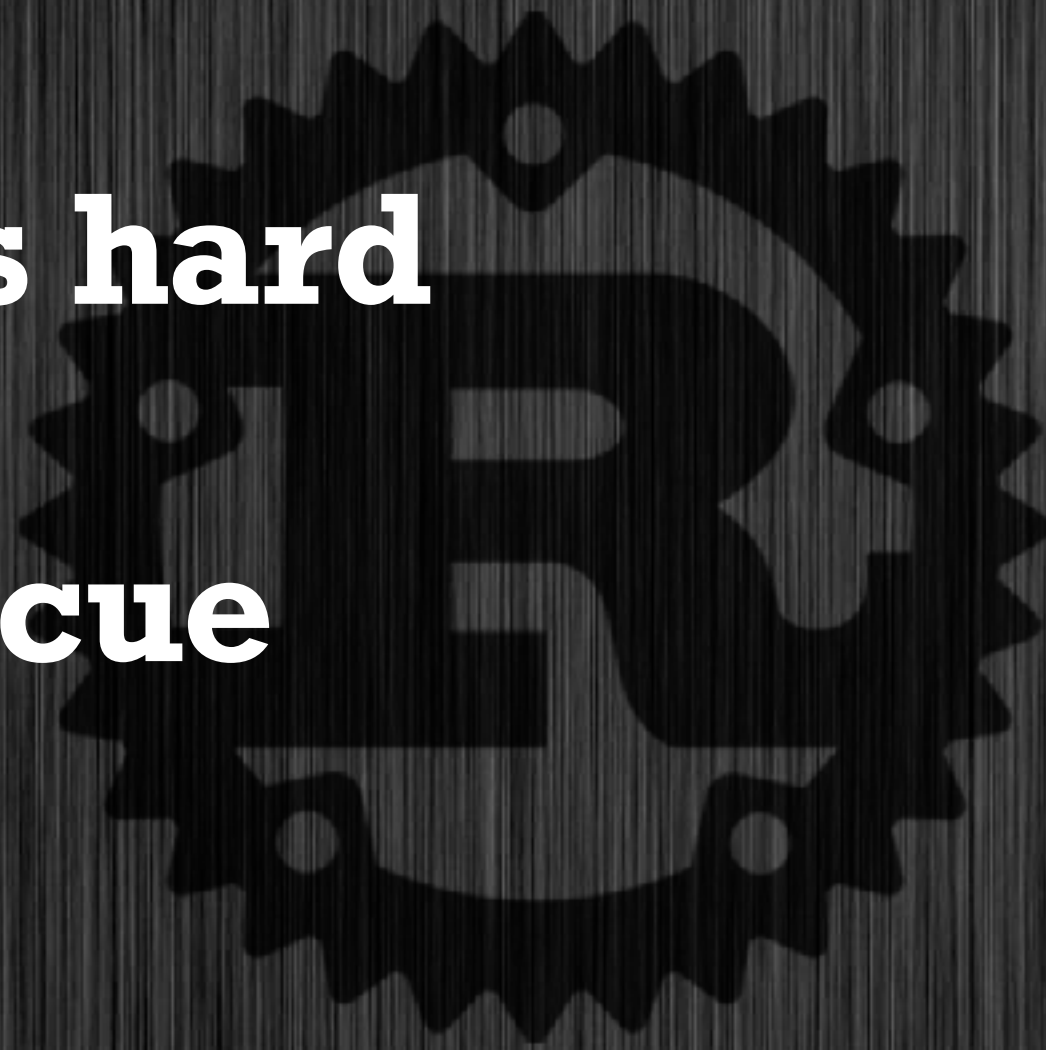
level 9: into the future(s)



```
fn handle_request(conn: &mut Connection) -> impl Future<Output = ()> {  
    HandleRequestFut::Start { conn }  
}
```

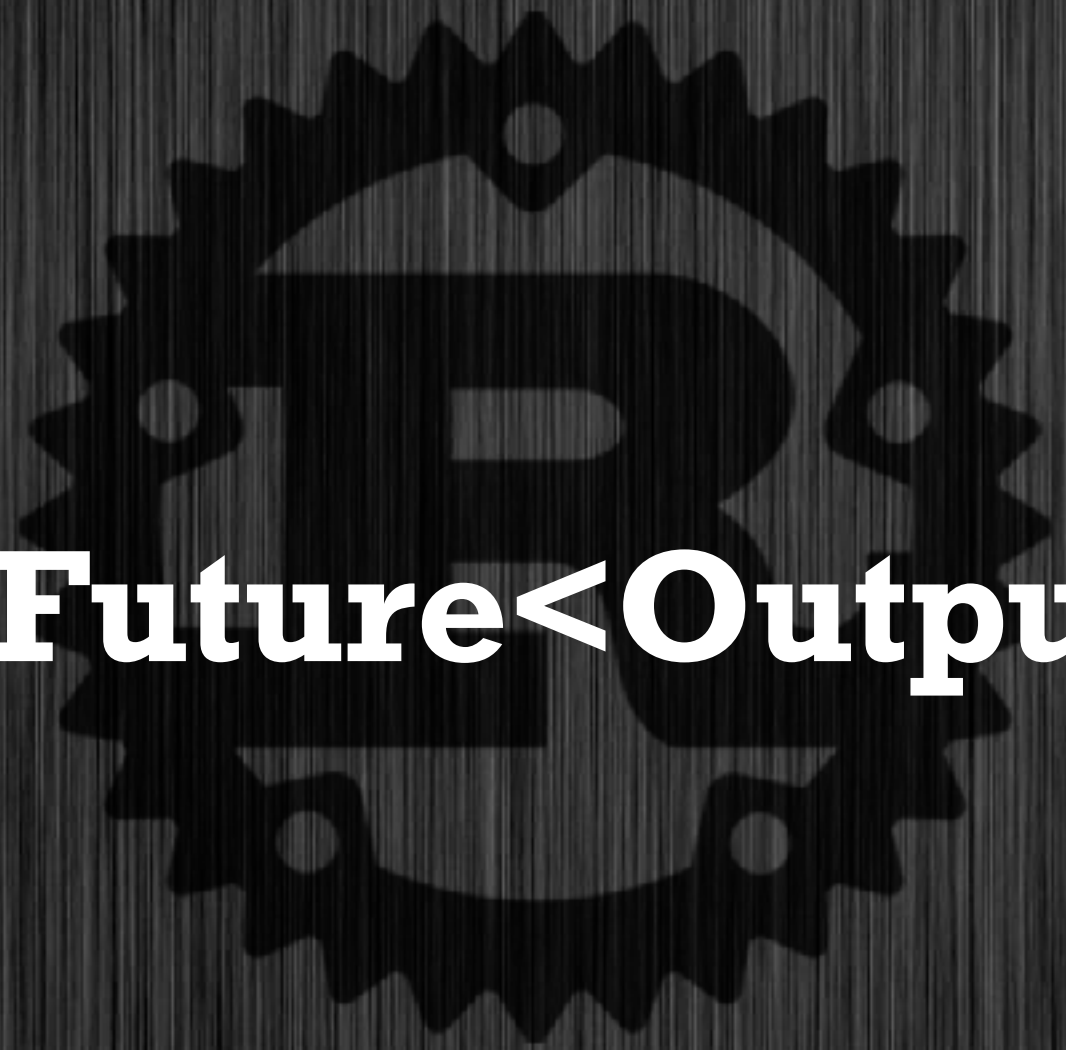
level 9: into the future(s)

- **impl Future by hand is hard**
- **async/await to the rescue**



level 9: into the future(s)

- **async fn foo(...)** -> **T**
- **fn foo(...)** -> **impl std::Future<Output=T>**

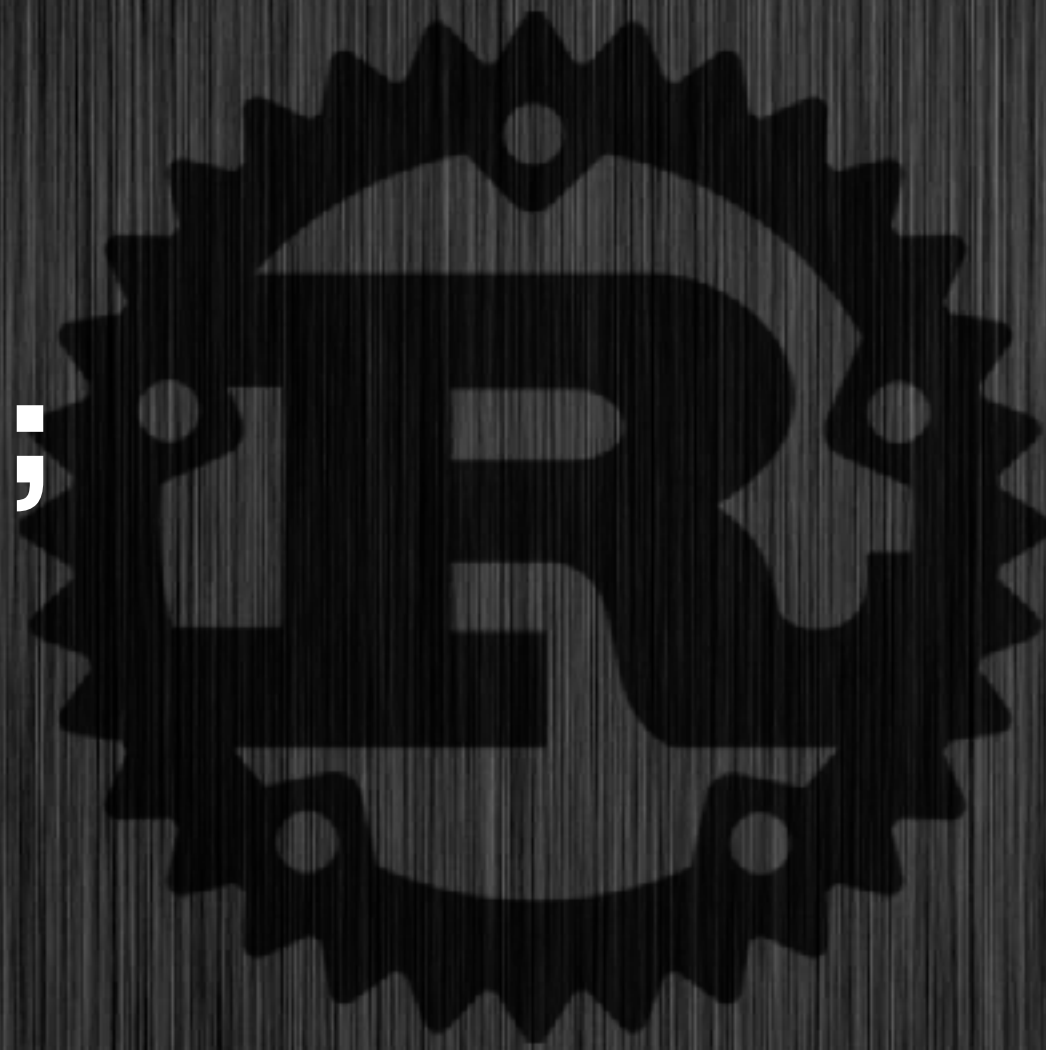


level 9: into the future(s)

- **trait SomeTrait {
 async fn foo(...) -> T;
}**

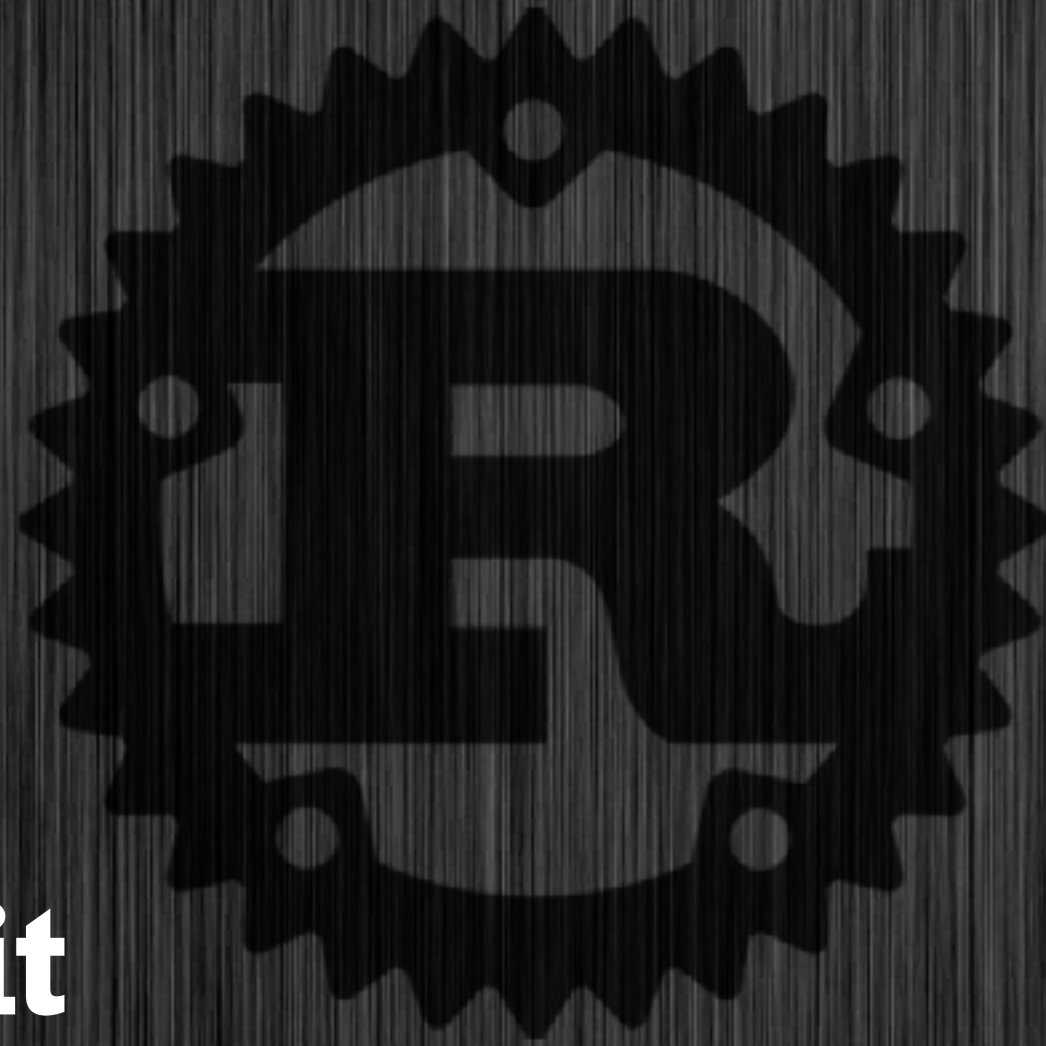
is not possible

- **yet**



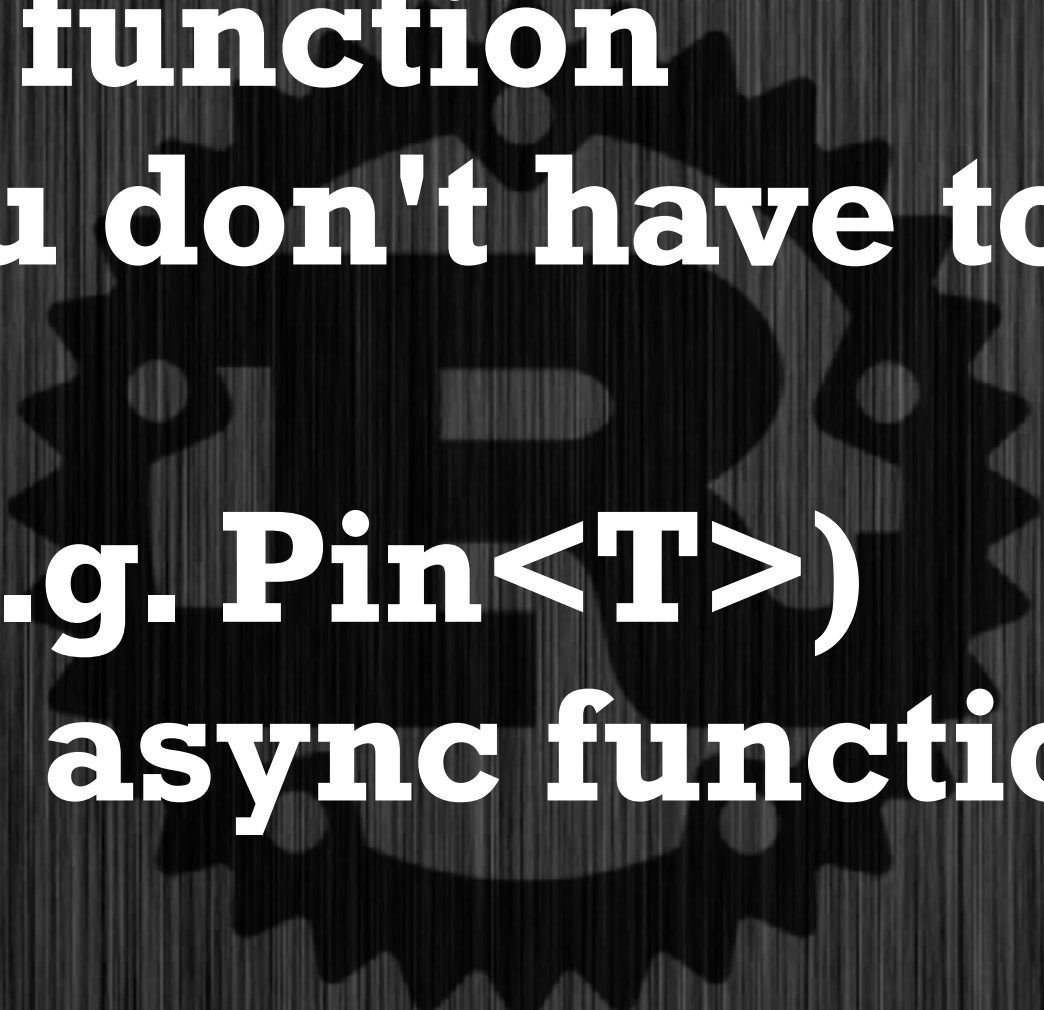
level 9: into the future(s)

- **await**
- **postfix operator**
- **result_of_future?.await**
- **future_of_result.await?**

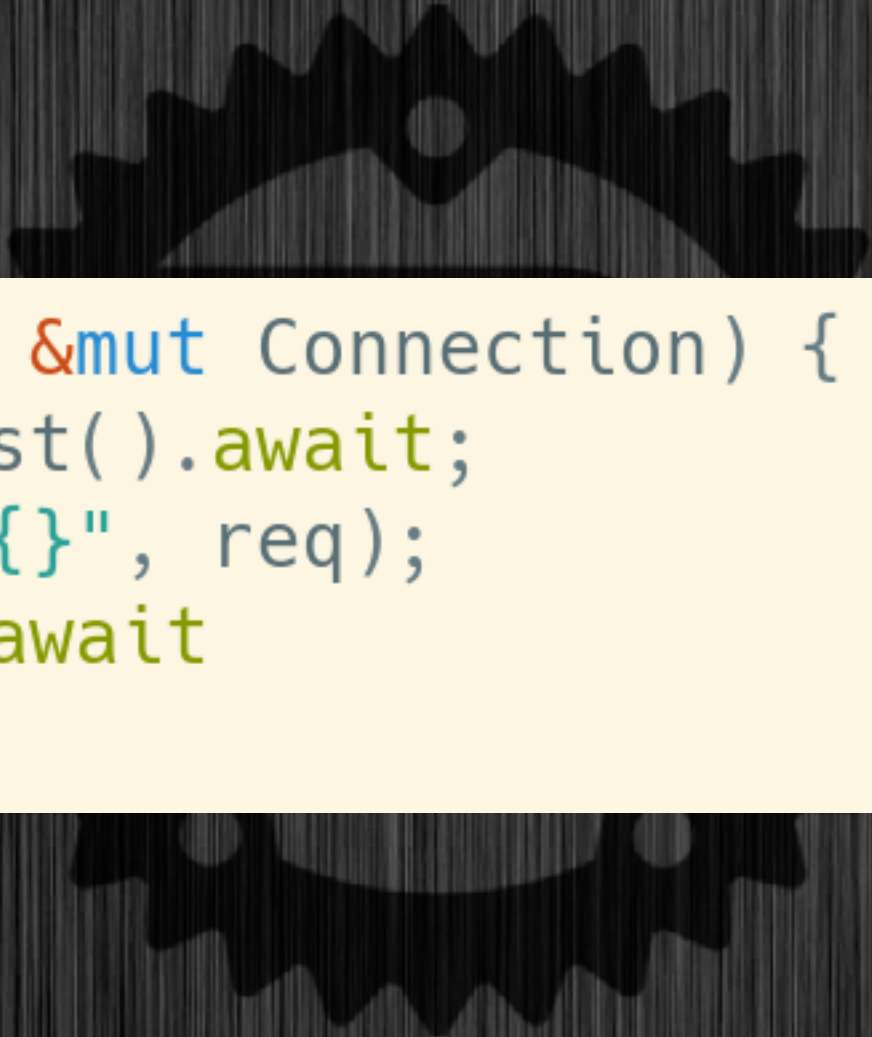


level 9: into the future(s)

**`async fn` converts your function
to a state machine so you don't have to
magic under the hood (e.g. `Pin<T>`)
lets you safely borrow in async functions
took years to design**



level 9: into the future(s)



```
async fn handle_request(conn: &mut Connection) {  
    let req = conn.read_request().await;  
    let resp = format!("got: {}", req);  
    conn.send_response(resp).await  
}
```

level 9: into the future(s)



level 9: into the future(s)

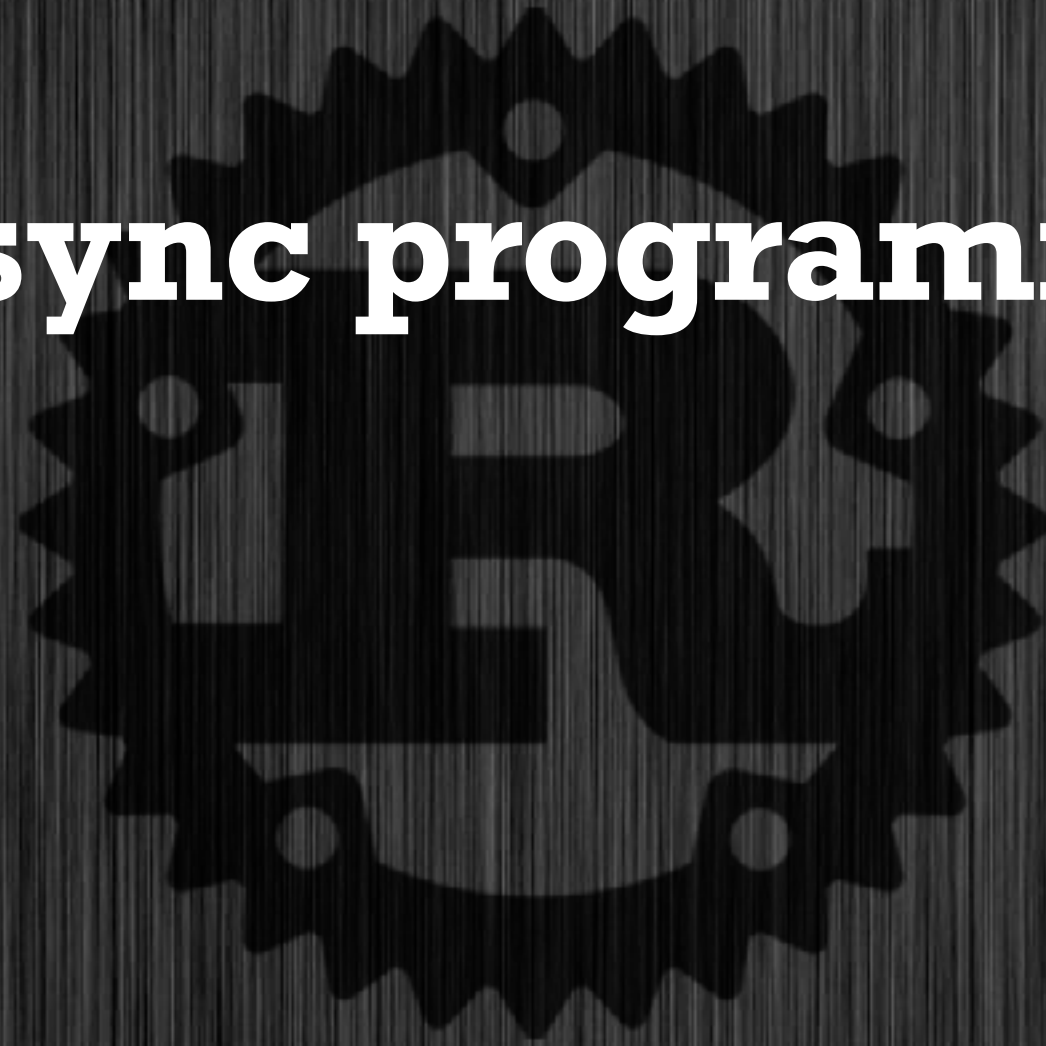
- **tokio is an executor for futures with `epoll()` based I/O**

```
let mut runtime = tokio::runtime::Runtime::new()?;  
let result = runtime.block_on(future);
```

- **`#[tokio::main]`
`async fn main() { /* ... */ }`**

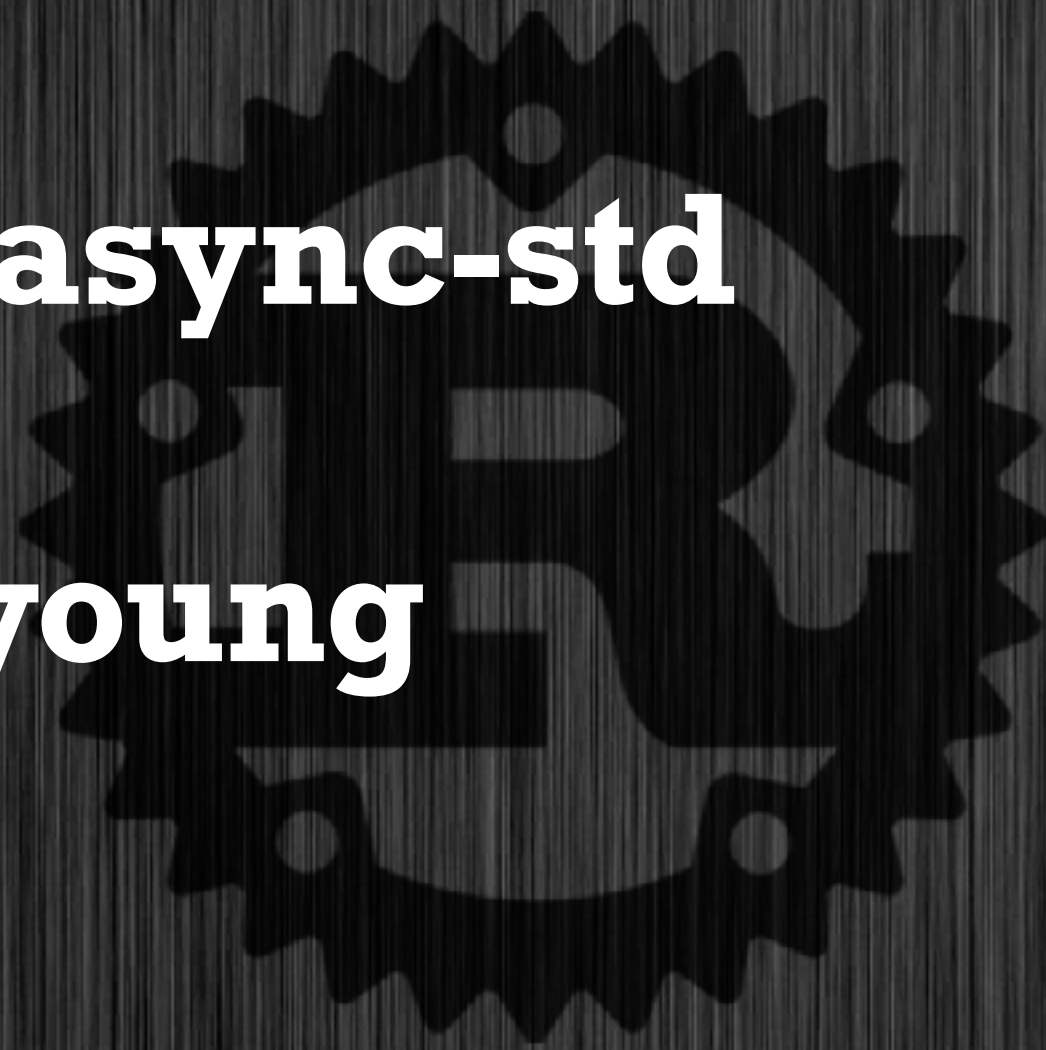
level 9: into the future(s)

- **tokio is a library for async programming**
codecs, protocols, ...



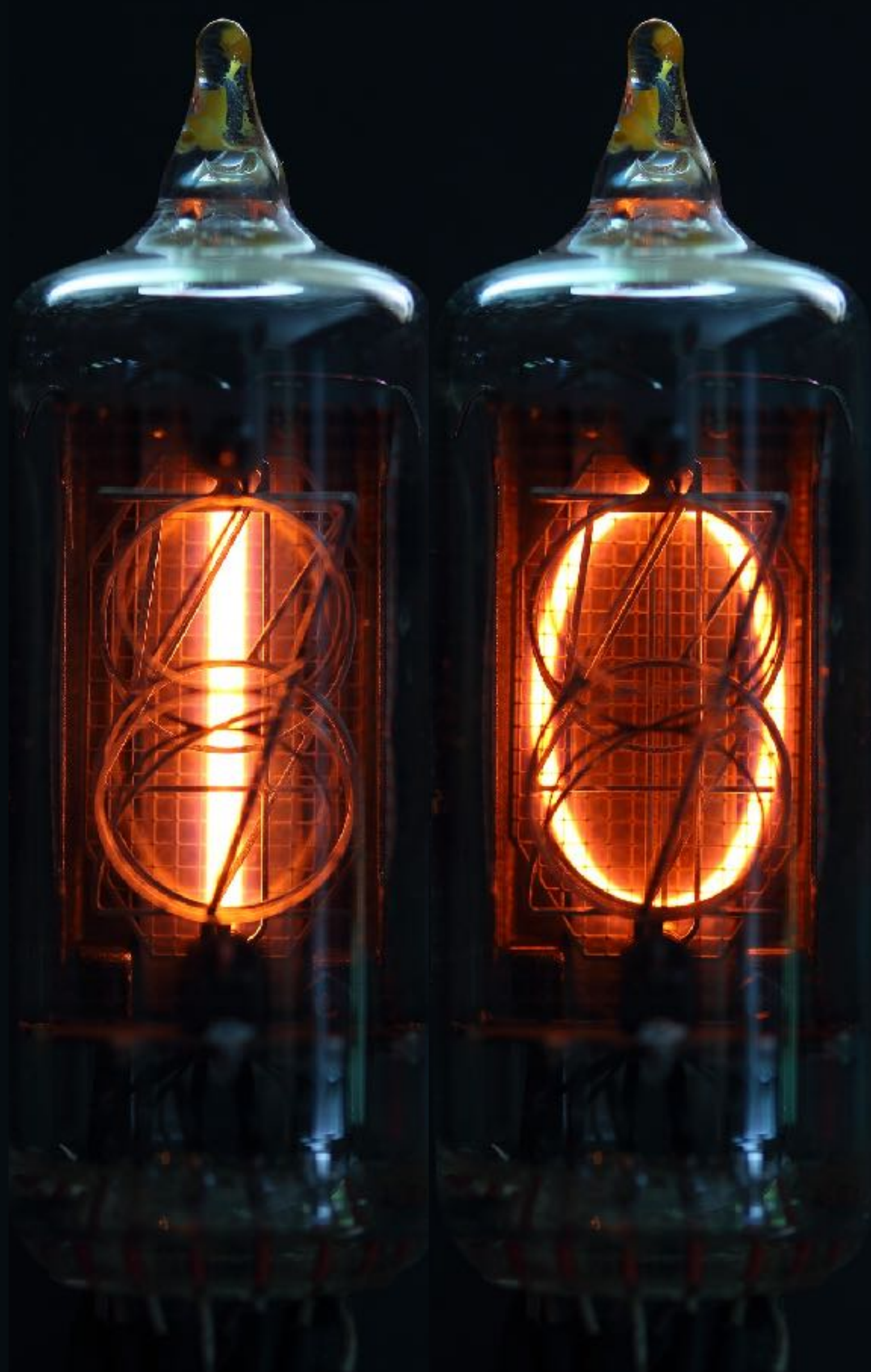
level 9: into the future(s)

- **tokio vs futures::io vs async-std**
- **the ecosystem is still young**



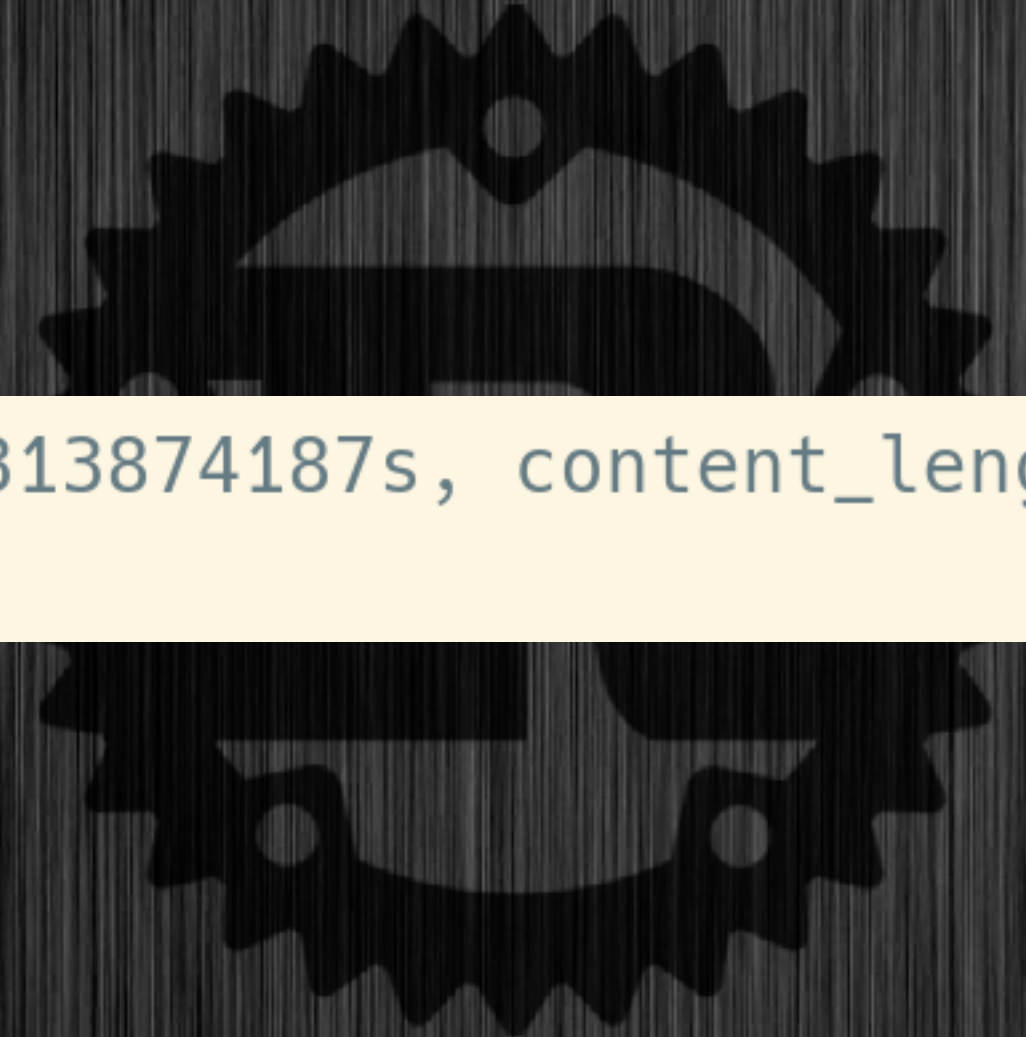
level 9: into the future(s)

- **reqwest::blocking::Client -> reqwest::Client**
- **#[tokio::main]**
- **tokio = { version = "1", features = ["macros"] }**

Two glowing vacuum tubes, likely Nixie tubes, are shown on the left side of the image. They are cylindrical with a glass envelope and a metal base. Inside, a complex grid of wires is visible, and a bright orange-red light emanates from the center of each tube. The tubes are set against a dark background.

**level 10
actual
concurrency**

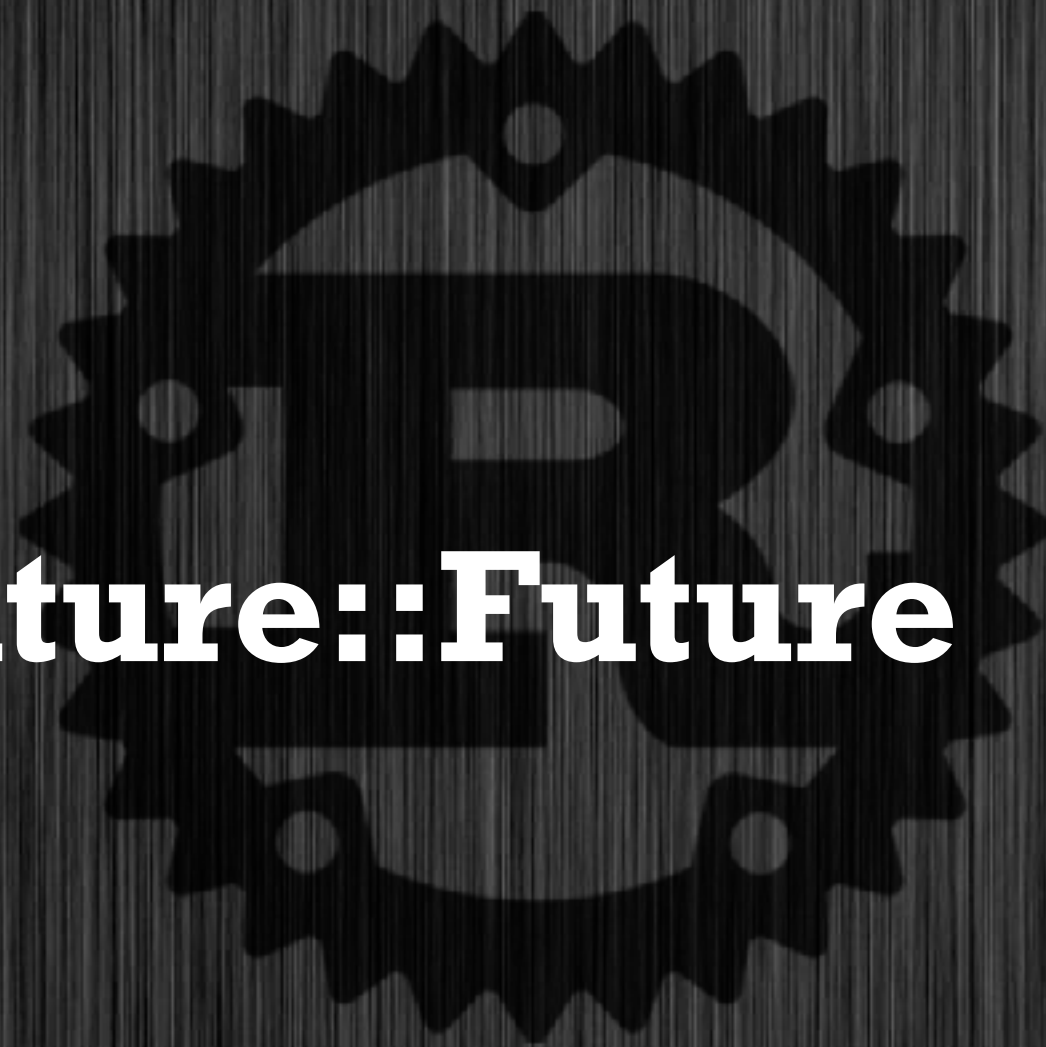
level 10: actual concurrency



```
total Stats { elapsed_time: 2.313874187s, content_length: 7605 } (3286.70 bytes/sec)
wall clock time: 2.317808863s
```

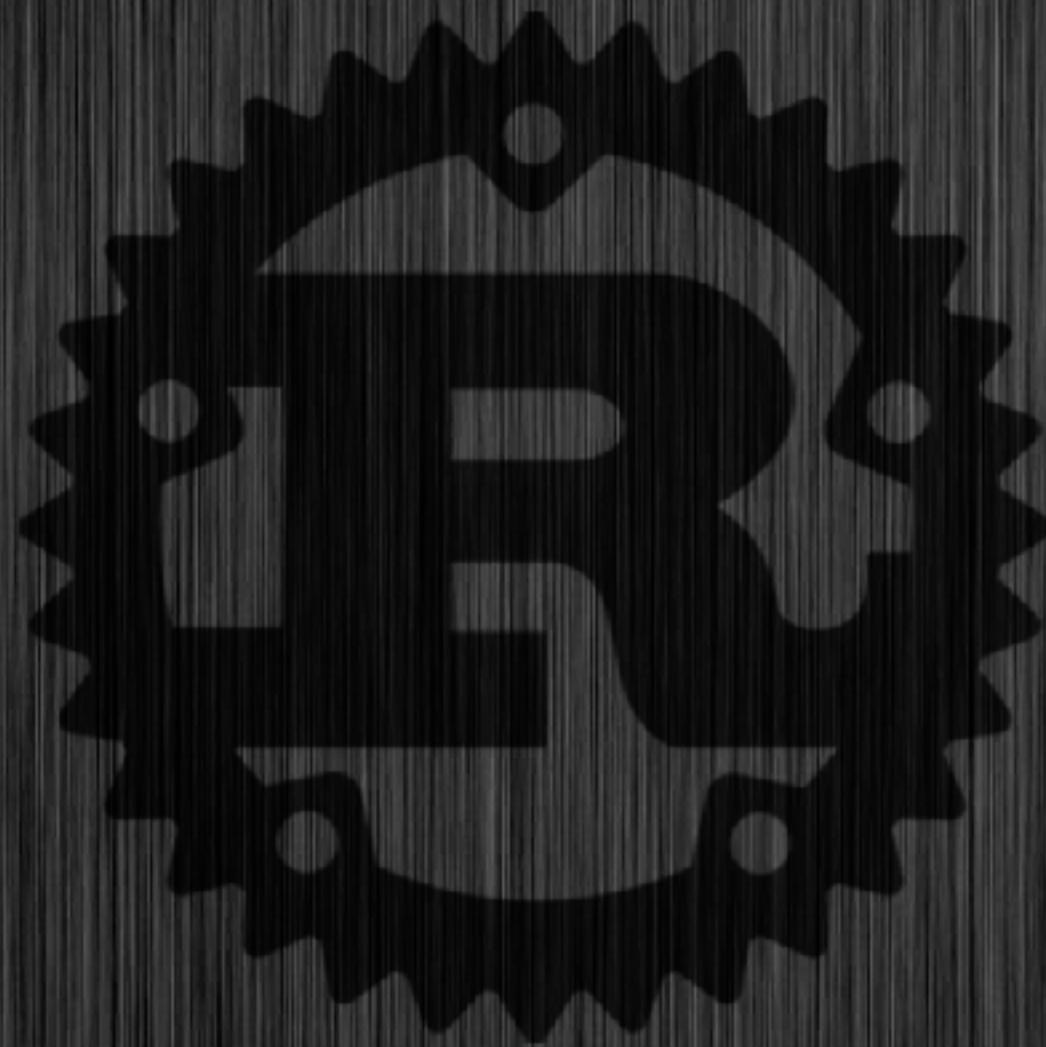
level 10: actual concurrency

- **futures crate**
- **builds on top of `std::future::Future`**



level 10: actual concurrency

- **futures::FutureExt**
 - **map()**
 - **then()**
 - **...**



level 10: actual concurrency

- **futures::stream::Stream**
- **asynchronous iterator**
 - **listening socket yields a stream of connections**
 - **receiving from a channel yields a stream of messages**
- **futures::stream::StreamExt**

level 10: actual concurrency

- **futures::sink::Sink**
- **asynchronous consumer of a Stream**
- **futures::sink::SinkExt**



level 10: actual concurrency

- **futures::stream::FuturesUnordered**
 - **a collection of futures -> a stream of results**
 - **.push(fut)**
- 

level 10: actual concurrency

- **futures::stream::StreamExt::buffered(n: usize)**
futures::stream::StreamExt::buffered_unordered(n: usize)
- **stream of futures -> stream of results**
- **futures::stream::iter(impl Iterator) -> impl Stream**

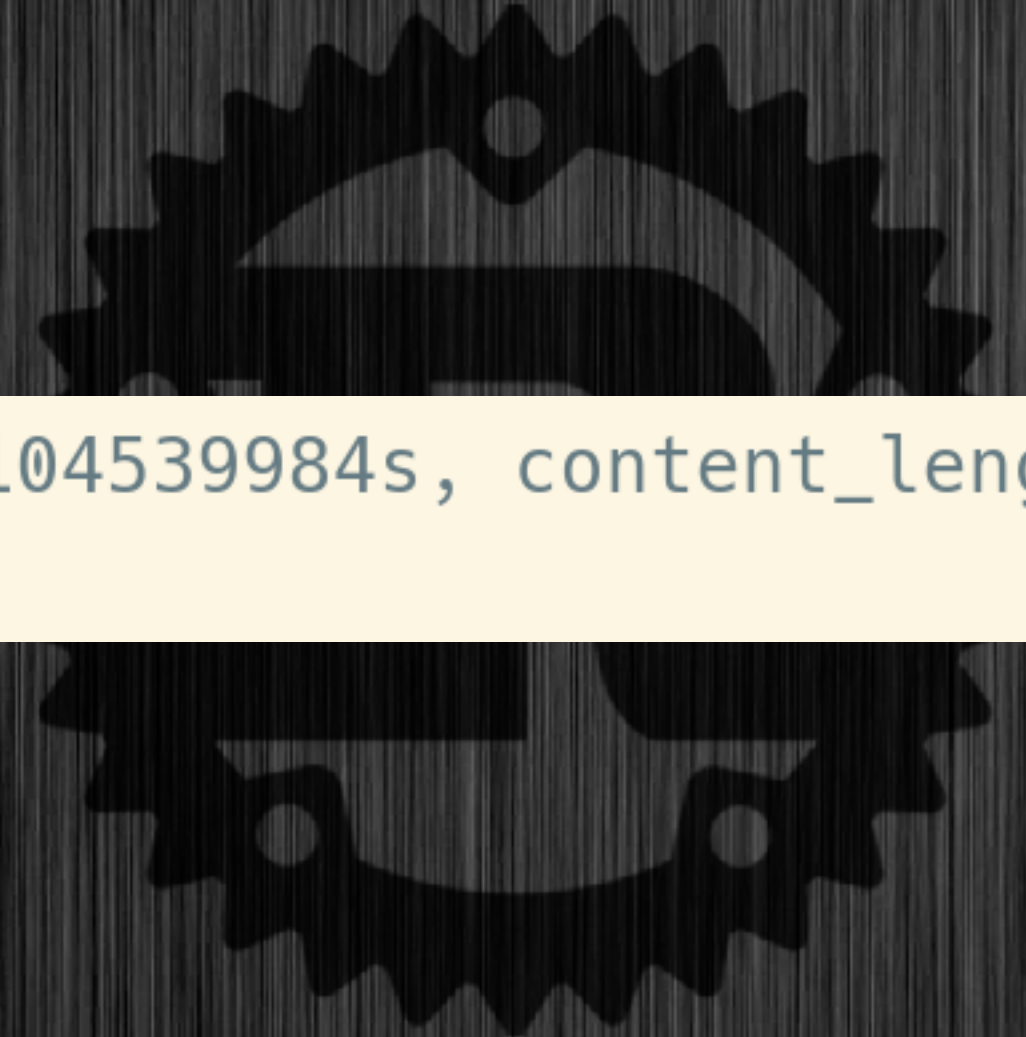
level 10: actual concurrency

- **consuming a stream**

while let Some(item) = stream.next().await { /* ... */ }

- **no ``for async item in stream`` (yet?)**
- **`.next()` is implemented in `StreamExt`**

level 10: actual concurrency



```
total Stats { elapsed_time: 7.104539984s, content_length: 7605 } (1070.44 bytes/sec)
wall clock time: 516.617451ms
```

level 10: actual concurrency

- **futures::stream::StreamExt**
- **futures::stream::FuturesUnordered**
- **futures = "0.3"**

