Parallelism & Concurrency

Goals For Today

- Introduce the Ideas of Parallelism & Concurrency

Don’t Forget!

- HW7 Due Tonight
- MP2 Due 3/4

- MP3 Releases 3/3

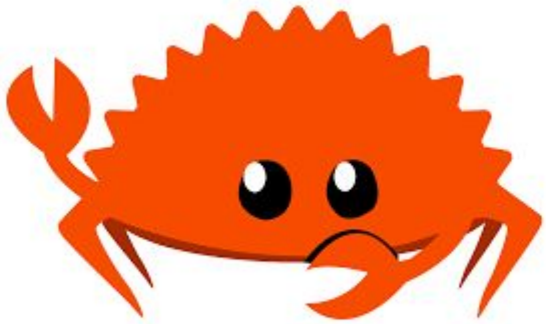
Don’t Forget!

- HW7 Due Tonight
- MP2 Due 3/4

- MP3 Releases 3/3

But First...

Could you explain how lowercase works for characters?



But First...

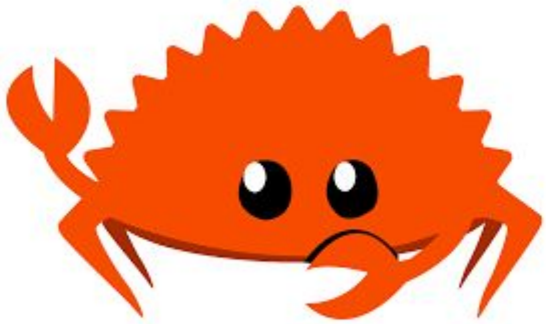
Could you explain how lowercase works for characters?

Converting

- `.to_uppercase()`
- `.to_lowercase()`

Checking

- `.is_uppercase()`
- `.is_lowercase()`



But First...

Could use another refresher on .unwrap, also why we use .as_ref instead of &?

Unwrap refresher:

- It's very common that certain operations may fail (`Result`)
  - Like if you try to read a file that doesn't exist
- We don't necessarily want to panic (and thus end execution), so instead of returning some type we can return `Result<T>`
- Then, we can check if what we return is `Ok(T)` or `Err(E*)`
- If it is `Ok(T)`, we probably want to use whatever T is
- So we can .unwrap() it

*We use E to represent an error type

But First...

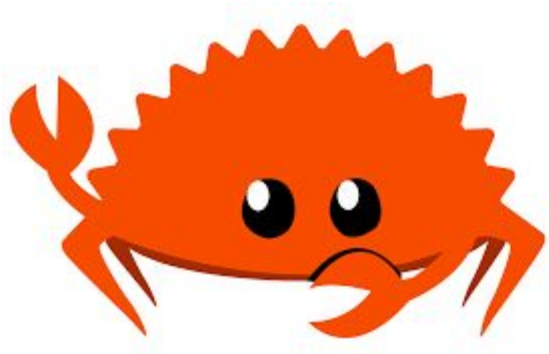
Could use another refresher on .unwrap, also why we use .as_ref instead of &?

Unwrap refresher:

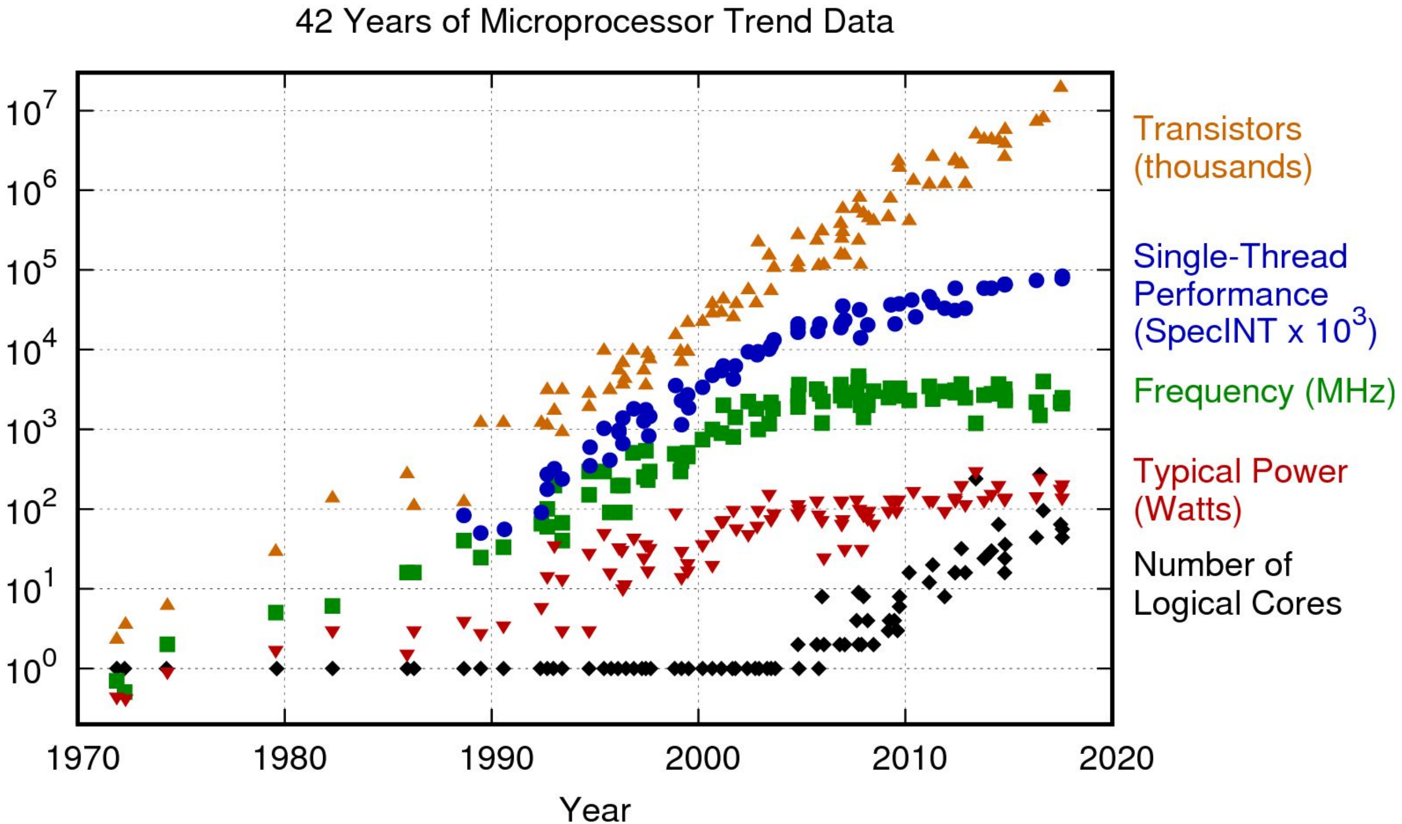
- It's very common that certain operations may ~~fail (Result)~~ return nothing (Option)
  - Like if you try to read a file that is empty
- We don't necessarily want to panic (and thus end execution), so instead of returning some type we can return ~~Result~~ Option<T>
- Then, we can check if what we return is ~~Ok~~ Some(T) or ~~Err(E)~~ None
- If it is ~~Ok~~ Some(T), we probably want to use whatever T is
- So we can .unwrap() it

But First...

Could you go over how to use clone? How does it differ to trying to use a reference?



Why Concurrency?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Why Concurrency?

"Scalable algorithms and libraries can be the best legacy we can leave behind from this era"

-David Kirk and Wen-mei W. Hwu's ECE 408 Slides

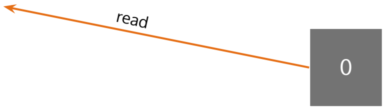
Why Not Concurrency?

Concurrency is **hard**.

Process 1:

```
let x = global_data;
global_data = x + 1;
```

Global Data

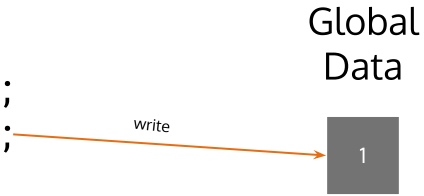


Why Not Concurrency?

Concurrency is **hard**.

Process 1:

```
let x = global_data;
global_data = x + 1;
```



Why Not Concurrency?

Concurrency is **hard**.

Process 1:

```
let x = global_data;
global_data = x + 1;
```

Global Data

0

Process 2:

```
global_data = 10;
```

Why Not Concurrency?

Concurrency is **hard**.

Process 1:

x=0

```
let x = global_data;
global_data = x + 1;
```

read

Global Data

0

Process 2:

```
global_data = 10;
```

Why Not Concurrency?

Concurrency is **hard**.

Process 1:

x=0

```
let x = global_data;
global_data = x + 1;
```

Global Data

10

write

Process 2:

```
global_data = 10;
```

Why Not Concurrency?

Concurrency is **hard.**

Process 1:

x=0

```
let x = global_data;
global_data = x + 1;
```

write

Global Data

1

Process 2:

```
global_data = 10;
```

Why Not Concurrency?

Concurrency is **hard.**

Process 1:

x=0

```
let x = global_data;
global_data = x + 1;
```

write

Global Data

1

Process 2:

```
global_data = 10;
// I think it is == 10!
```

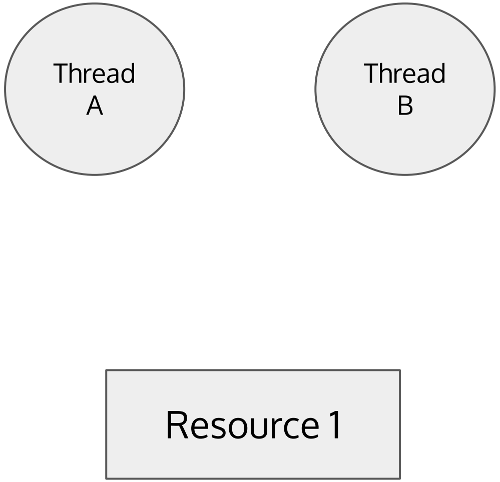
Why Not Concurrency?

Concurrency is **hard.**

Race	Thread A	Thread B	Resources
Time 1	Accesses 1	Accesses 1	Resource 1
Time 2	Resource 1 = 2	Resource 1 = 3	Resource 2
Time 3	Resource 1 += 1	Resource 1 += 1	Resource 3
Time 4	What is Resource 1?	What is Resource 1?	Resource 4

Common issues:

1. **Race Conditions**
2. Deadlocks
3. Livelocks
4. Starvation



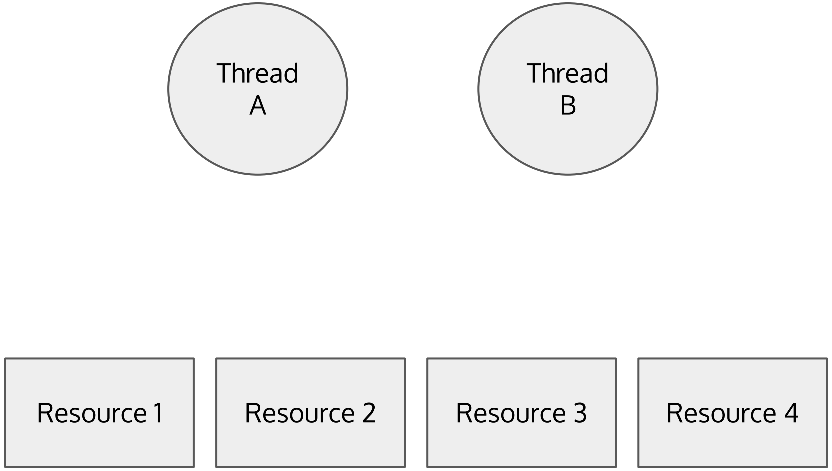
Why Not Concurrency?

Concurrency is **hard**.

Common issues:

1. Race Conditions
2. **Deadlocks**
3. Livelocks
4. Starvation

Deadlock Example	Thread A	Thread B	Resources
Time 1	Accesses 1 & 2	Accesses 3 & 4	Resource 1
Time 2	Attempts to Access 3	Denies Access to 3	Resource 2
Time 3	Denies Access to 1	Attempts Access to 1	Resource 3
Time 4	Try Again (Deadlock)	Try Again (Deadlock)	Resource 4



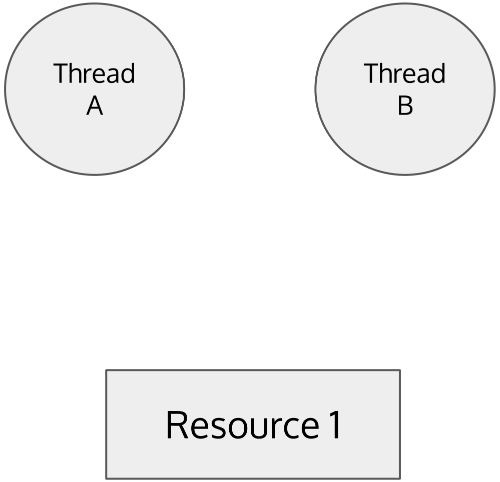
Why Not Concurrency?

Concurrency is **hard**.

Common issues:

1. Race Conditions
2. Deadlocks
3. **Livelocks**
4. Starvation

Livelock	Thread A	Thread B	Resources
Time 1	Accesses Resource 1	Requests Resource 1	Resource 1
Time 2	Requests Resource 1	Accesses Resource 1	Resource 2
Time 3	Accesses Resource 1	Requests Resource 1	Resource 3
Time 4	And so on.. (livelock)	And so on.. (livelock)	Resource 4



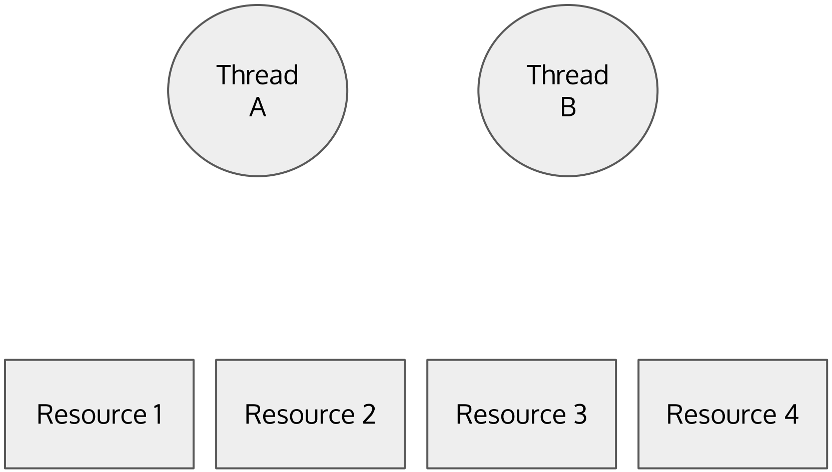
Why Not Concurrency?

Concurrency is **hard.**

Starvation	Thread A	Thread B	Resources
Time 1	Accesses 1, 2, 3, & 4	Politely Yields to A	Resource 1
Time 2	Uses 1, 2, 3, & 4	Requests Access	Resource 2
Time 3	Uses 1, 2, 3, & 4	Requests Access	Resource 3
Time 4	Continued Use (Starvation)	Continued Wait (Starvation)	Resource 4

Common issues:

1. Race Conditions
2. Deadlocks
3. Livelocks
4. **Starvation**



Why Not Concurrency?

Concurrency is **hard**.

Concurrency!

Concurrency is **hard**.

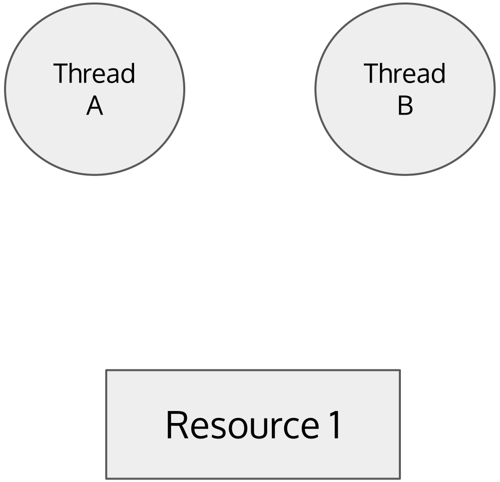
But Rust makes it **Easy.**

Concurrency!

Concurrency is hard.

But Rust makes it Easy.

Race	Thread A	Thread B	Resources
Time 1	Accesses 1	Accesses 1	Resource 1
Time 2	Resource 1 = 2	Resource 1 = 3	Resource 2
Time 3	Resource 1 += 1	Resource 1 += 1	Resource 3
Time 4	What is Resource 1?	What is Resource 1?	Resource 4



Concurrency!

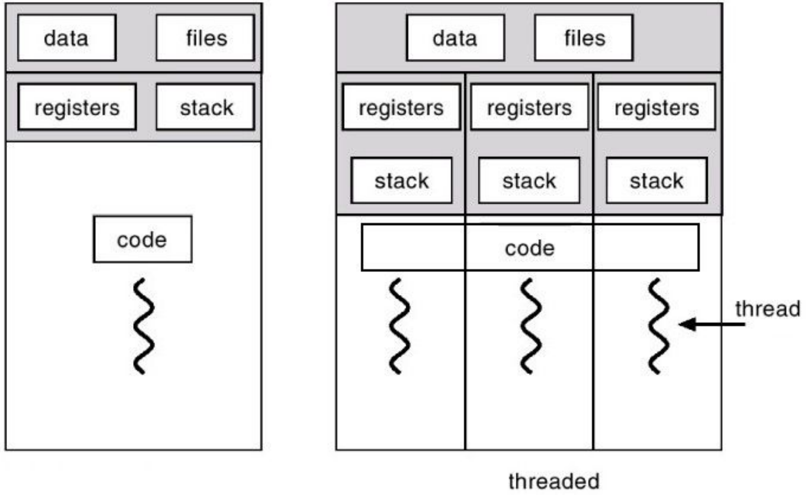
Concurrency is hard.

But Rust makes it Easy.

An **operating system** manages many **processes***

Processes may have many **threads**

Threads are the execution of code.



*There's some distinction between processes and programs. Generally, we refer to programs as the code while processes are the things that execute that code. That is, your OS creates a process to execute a program.

Concurrency!

Concurrency vs **Parallelism**

- **Concurrency** is being able to switch tasks when we have some downtime
- **Parallelism** is being able to execute multiple tasks at the same time

System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	**1 s**
Level 1 cache access	0.9 ns	**2 s**
Level 2 cache access	2.8 ns	**7 s**
Level 3 cache access	28 ns	**1 min**
Main memory access (DDR DIMM)	~100 ns	**4 min**
Intel Optane memory access	<10 μs	**7 hrs**
NVMe SSD I/O	~25 μs	**17 hrs**
SSD I/O	50–150 μs	**1.5–4 days**
Rotational disk I/O	1–10 ms	**1–9 months**
Internet call: SF to NYC	65 ms	**5 years**
Internet call: SF to Hong Kong	141 ms	**11 years**

Concurrency!

Effective concurrency is highly dependent on effective communication.

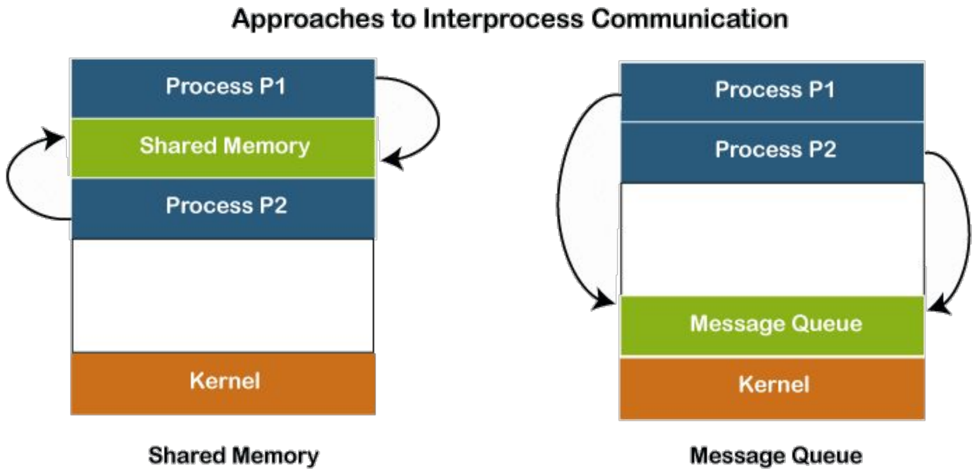
There’s two major ways of communicating that we’ll explore.

Concurrency!

Effective concurrency is highly dependent on effective communication.

There's two major ways of communicating that we'll explore.

Message Passing and **Shared Memory**



Concurrency Examples

Ok Eustis very cool but how do we actually do it?

Concurrency Examples

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Concurrency Examples - Closures

```
use std::thread;
use std::time::Duration;

fn main() {
   let closure = |num| -> i32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        return num + 1
    };

    println!("{}", closure(3));
}
```

Closures: **Anonymous** Functions that Can Capture Their Environment

Concurrency Examples - Closures

```rust
use std::thread;
use std::time::Duration;

fn main() {
   let closure = |num| -> i32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        return num + 1
    };

    println!("{}", closure(3));
}
```

Closures: Anonymous Functions that Can **Capture Their Environment**

Concurrency Examples - Closures

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let closure = |num| -> i32 {
          println!("calculating slowly...");
          thread::sleep(Duration::from_secs(2));
          return num + 1
     };

     println!("{}", closure(3));
}
```

Closures: Anonymous Functions that Can **Capture Their Environment**

- closures can capture values from the scope in which they're defined

Concurrency Examples - Closures

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let closure = |num| -> i32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        return num + 1
    };

    println!("{}", closure(3));
}
```

Concurrency Examples - Closures

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let closure = |num| -> i32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        return num + 1
    };

    println!("{}", closure(3));
}
```

```rust
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

Concurrency Examples

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 2 from the main thread!
hi number 6 from the spawned thread!
hi number 3 from the main thread!
hi number 7 from the spawned thread!
hi number 4 from the main thread!
hi number 8 from the spawned thread!
```

Concurrency Examples

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap()
}
```

Concurrency Examples

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..9 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }


}
```

That’s All Folks!

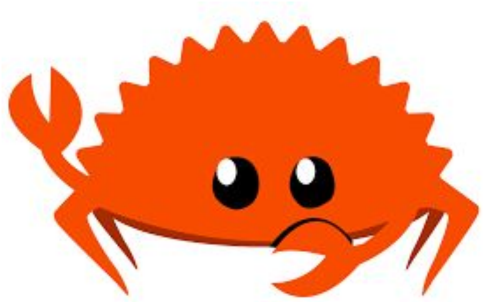
See you on Thursday :)

But First...

Could use another refresher on .unwrap, also why we use .as_ref instead of &?

Why do we use .as_ref instead of &?

- This one is a sneaky difference that enters a large tangent, but here we go.
- When we are talking about the difference, we really care about the difference in the behavior of the **Borrow** and **AsRef** traits.



*The Docs: https://doc.rust-lang.org/std/convert/trait.AsRef.html

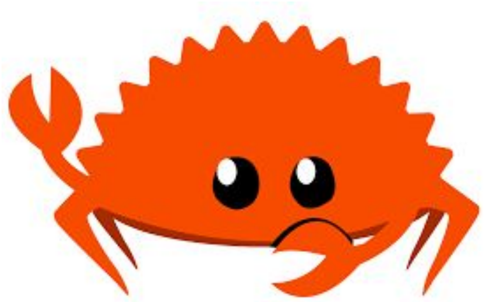
But First...

Could use another refresher on .unwrap, also why we use .as_ref instead of &?

Why do we use .as_ref instead of &?

From the docs*:

- **AsRef** has the same signature as **Borrow**, but **Borrow** is different in few aspects:
  - Unlike AsRef, Borrow has a blanket impl for any T, and can be used to accept either a reference or a value.
  - Borrow also requires that Hash, Eq and Ord for borrowed value are equivalent to those of the owned value. For this reason, if you want to borrow only a single field of a struct you can implement AsRef, but not Borrow.
    - (x == y) == (&x == &y)



*The Docs: https://doc.rust-lang.org/std/convert/trait.AsRef.html

But First...

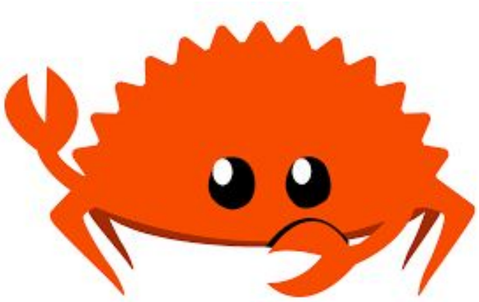
Could use another refresher on .unwrap, also why we use .as_ref instead of &?

Why do we use .as_ref instead of &?

```rust
fn is_hello<T: AsRef<str>>(s: T) {
   assert_eq!("hello", s.as_ref());
}

let s = "hello";
is_hello(s);

let s = "hello".to_string();
is_hello(s);
```



*The Docs: https://doc.rust-lang.org/std/convert/trait.AsRef.html

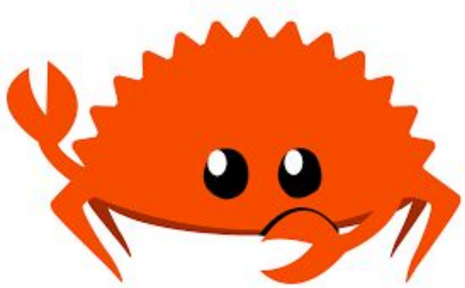
But First...

Could use another refresher on .unwrap, also why we use .as_ref instead of &?

Why do we use .as_ref instead of &?

Choose **Borrow** when you want to abstract over different kinds of borrowing, or when you're building a data structure that treats owned and borrowed values in equivalent ways, such as hashing and comparison.

Choose **AsRef** when you want to convert something to a reference directly, and you're writing generic code.



*https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/borrow-and-asref.html