Parallelism/Concurrency

Threads and Ownership

Goals For Today

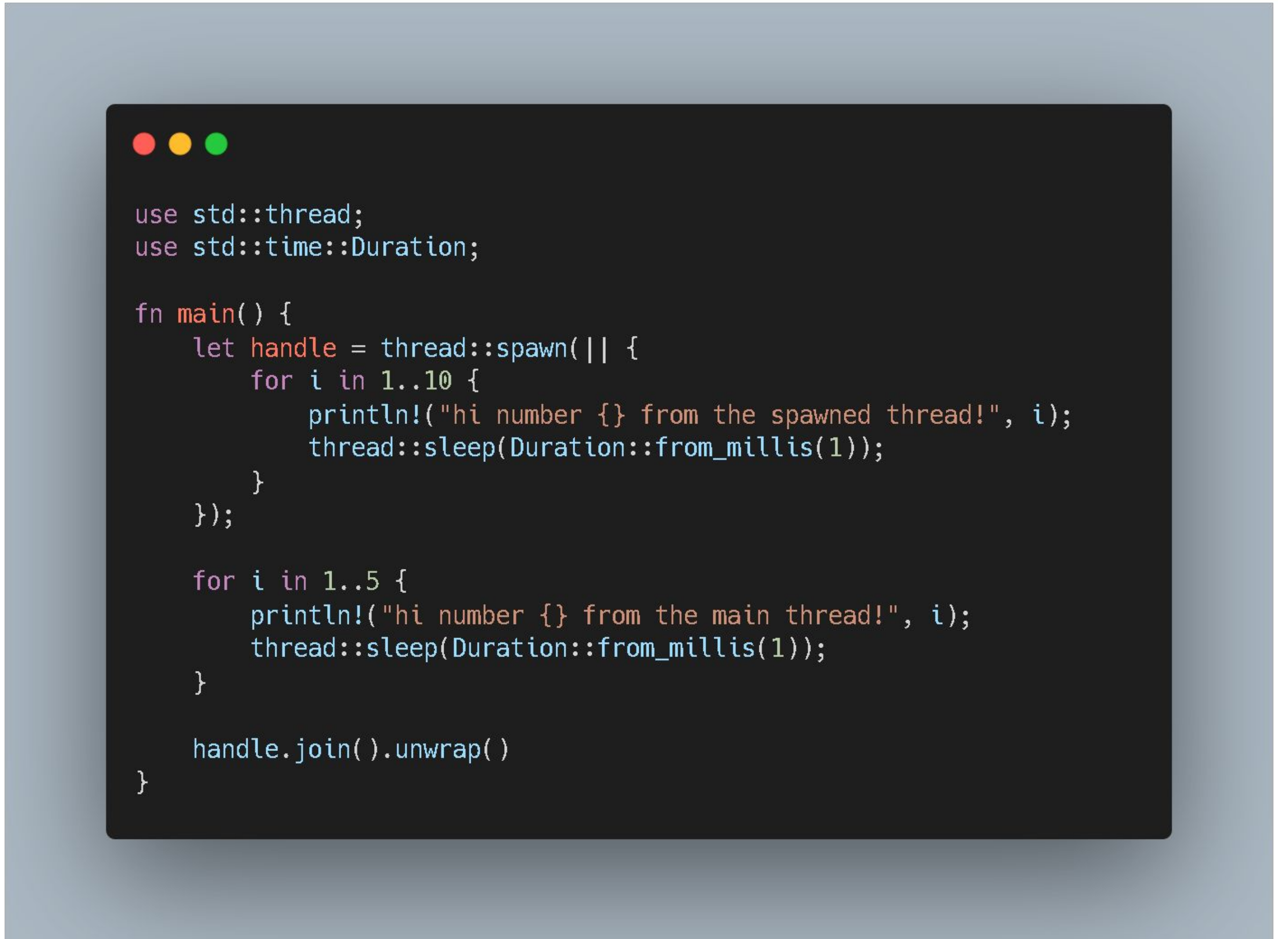
- Review
- Thread Ownership
- Thread Move
- MPSC concepts

Don’t Forget

- MP3 Releases Today
- MP2 Due Tomorrow (3/3)

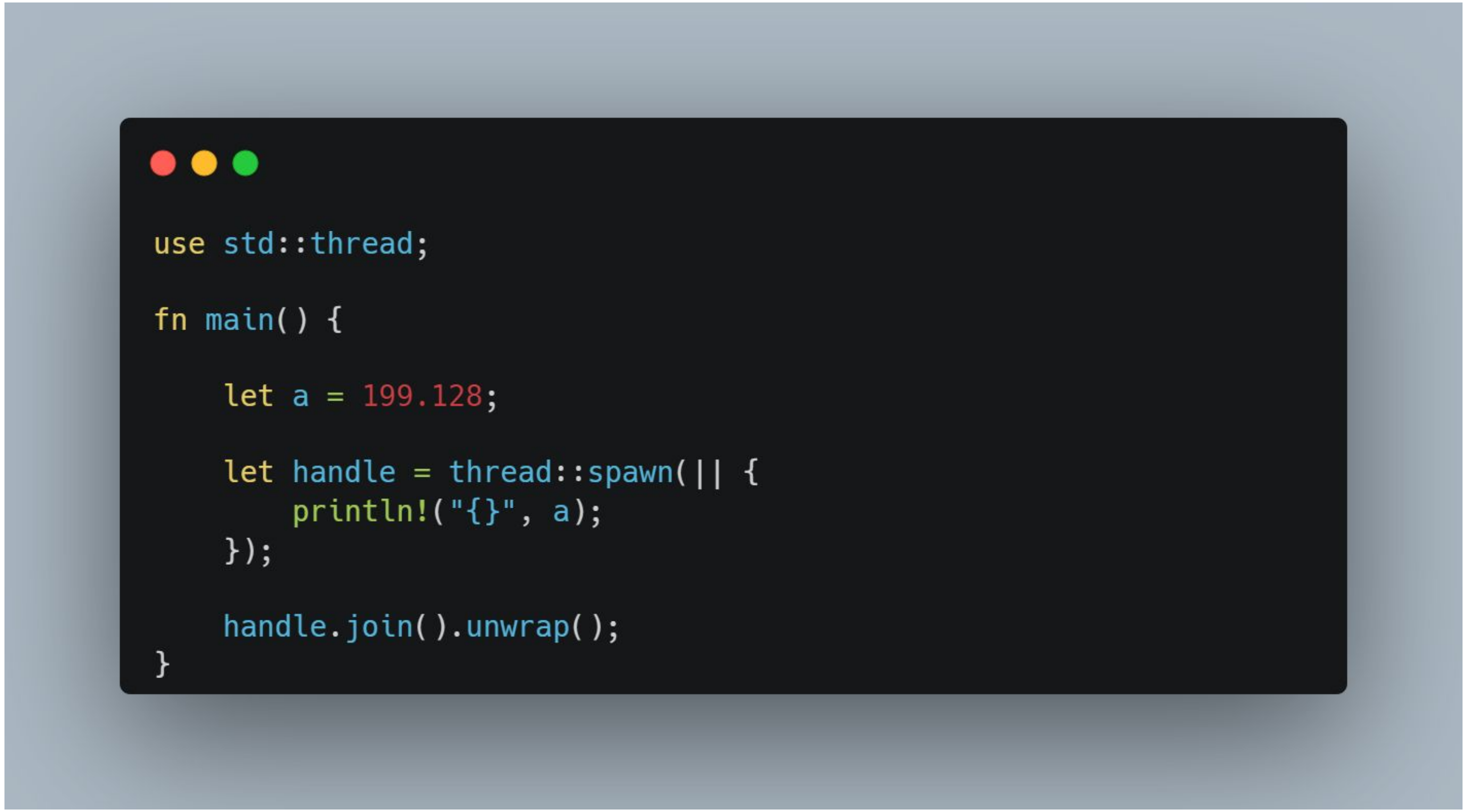
- HW9 and HW10 release today
    - Both will be due 3/11
- HW8 Due Tonight

Review



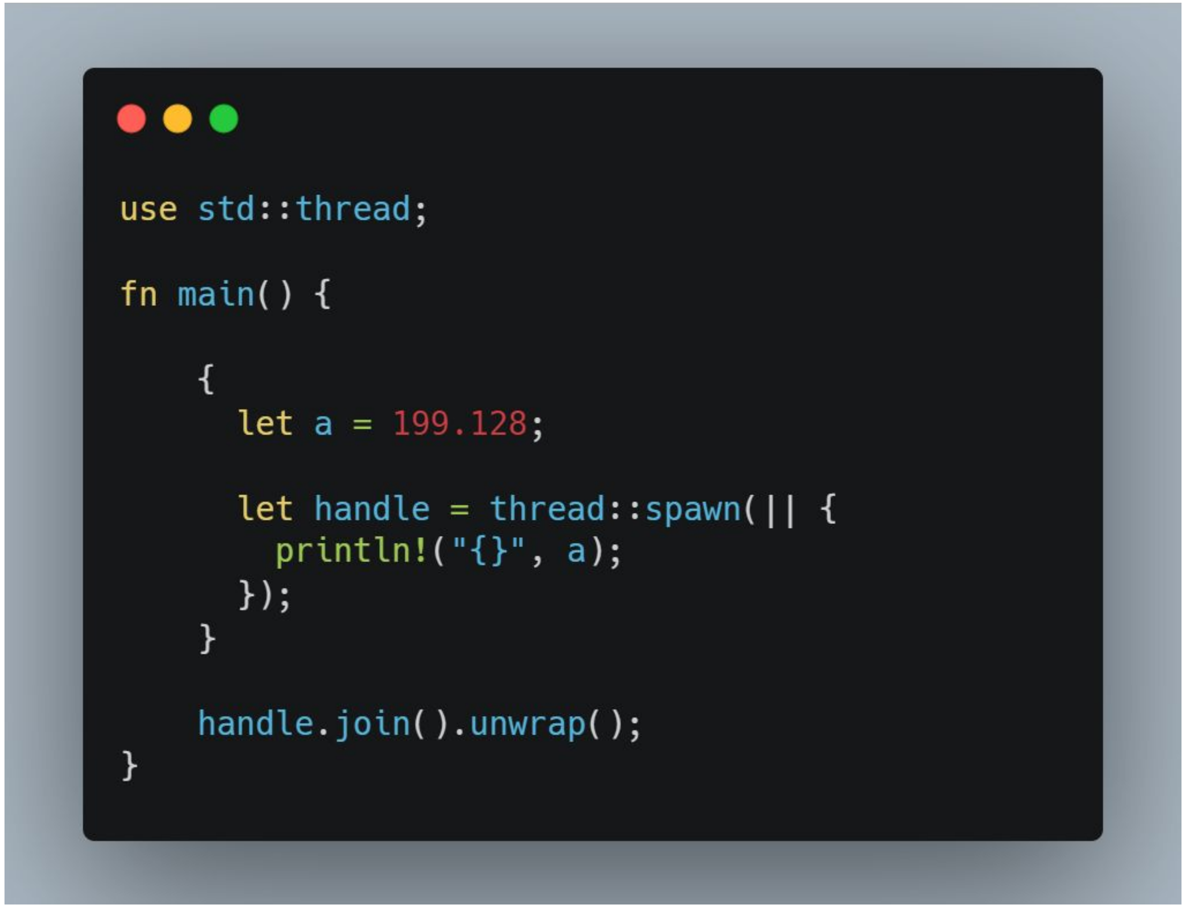
```rust
use std::thread;
use std::time::Duration;

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));
    }

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

Ownership with Threads



```rust
use std::thread;

fn main() {

    let a = 199.128;

    let handle = thread::spawn(|| {
        println!("{}", a);
    });

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

Ownership with Threads



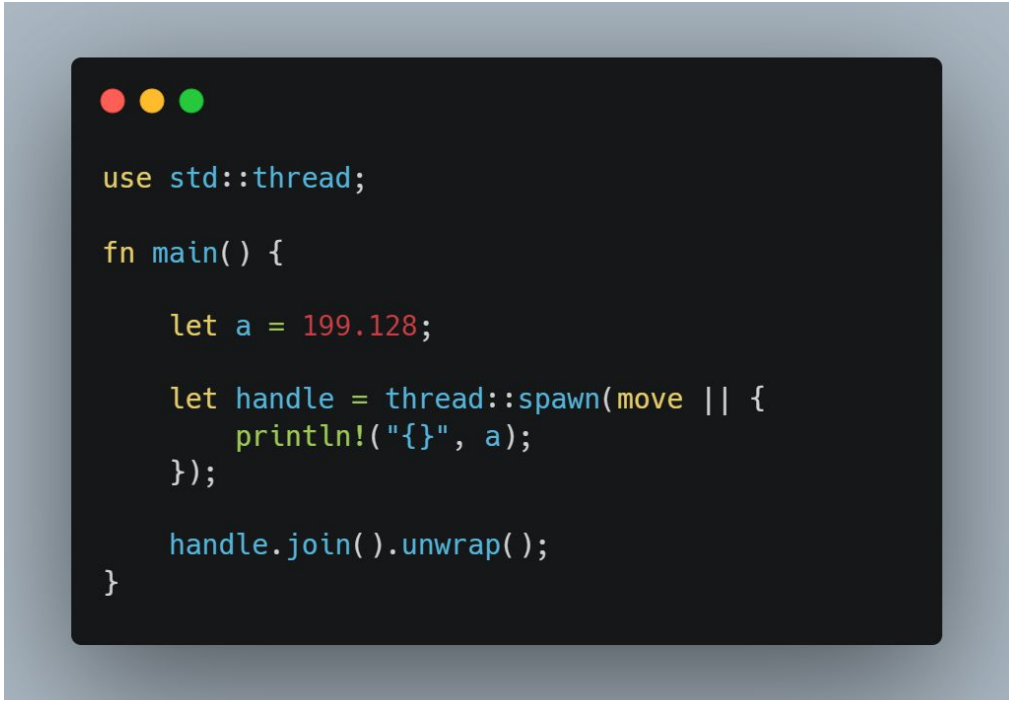
```rust
use std::thread;

fn main() {

    {
      let a = 199.128;

      let handle = thread::spawn(|| {
        println!("{}", a);
      });
    }

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

Ownership with Threads

We can use the `move` keyword to force the closure to take ownership values it uses.

Ownership with Threads

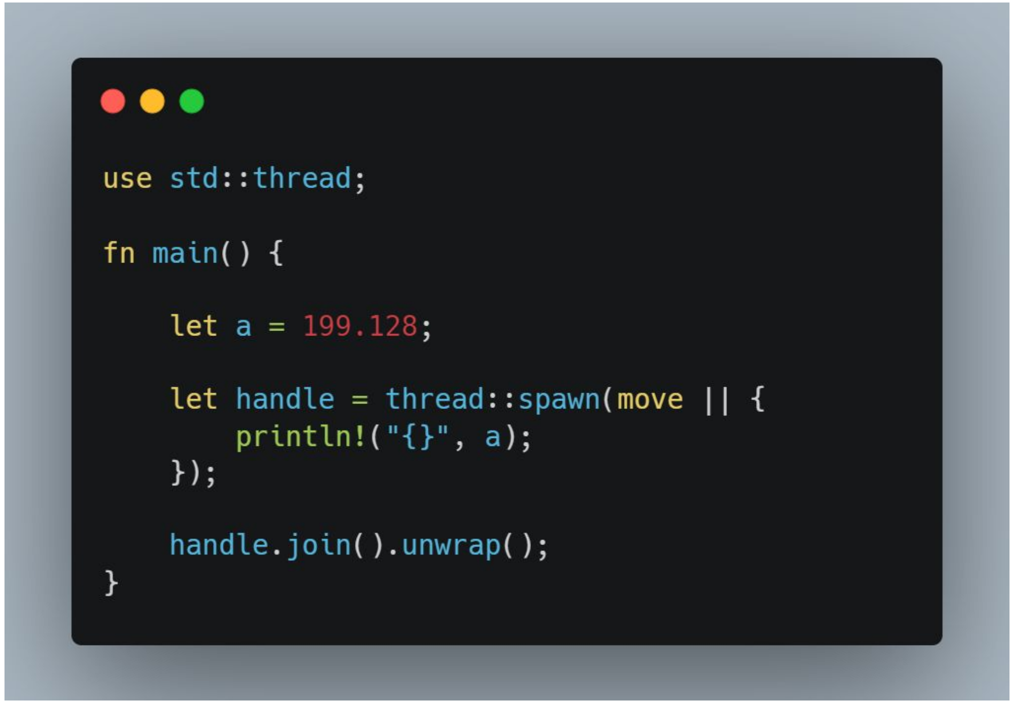
We can use the `move` keyword to force the closure to take ownership values it uses.



```rust
use std::thread;

fn main() {

    let a = 199.128;

    let handle = thread::spawn(move || {
        println!("{}", a);
    });

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

Ownership with Threads

We can use the `move` keyword to force the closure to take ownership values it uses.



```rust
use std::thread;

fn main() {

    let a = 199.128;

    let handle = thread::spawn(move || {
        println!("{}", a);
    });

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

Let’s do an example

We have some array of numbers, we want to sum all of the elements in this array.

We want each thread to sum up some portion of this array.

Messaging!

What does it mean to pass messages?

- Rather than sharing memory, each thread will have their own data
- They can communicate across a **channel**

Messaging!

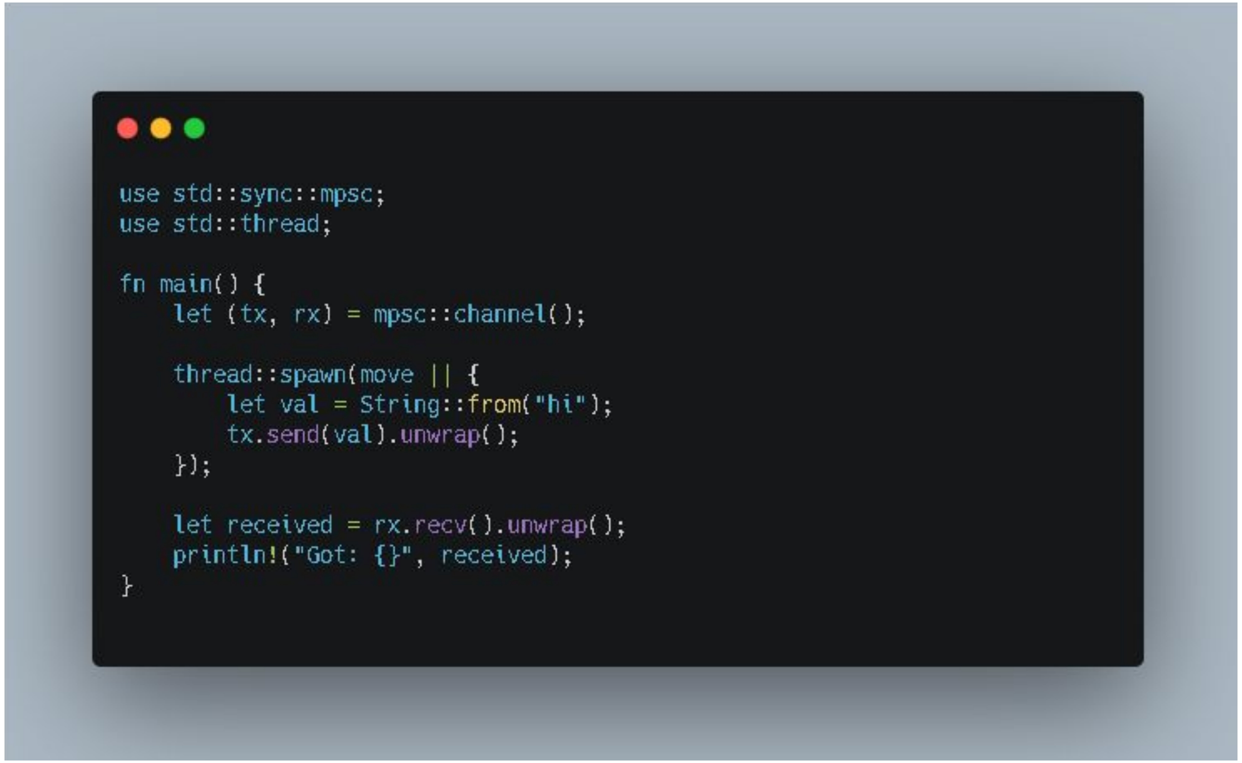
What does it mean to pass messages?

- Rather than sharing memory, each thread will have their own data
- They can communicate across a **channel**

Channels

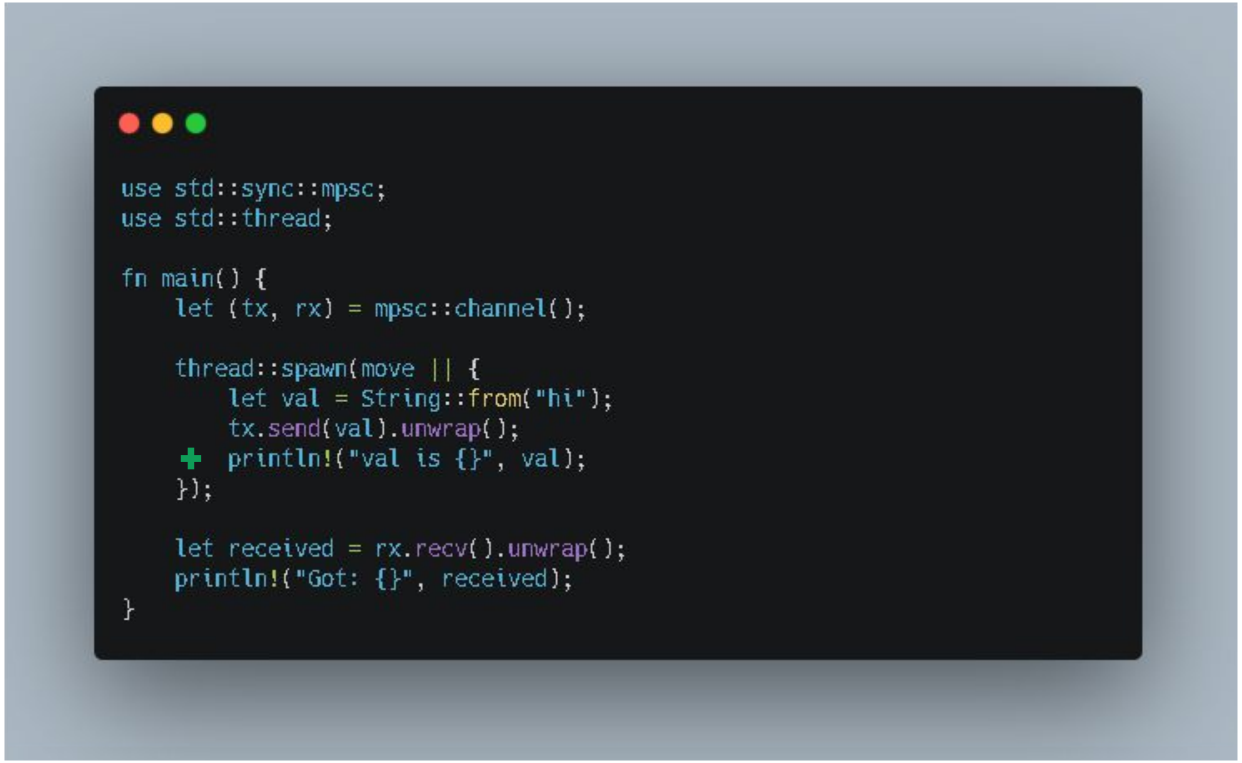
- Channels have **transmitters** and **receivers**
- Channels are closed if either is dropped

Messaging!



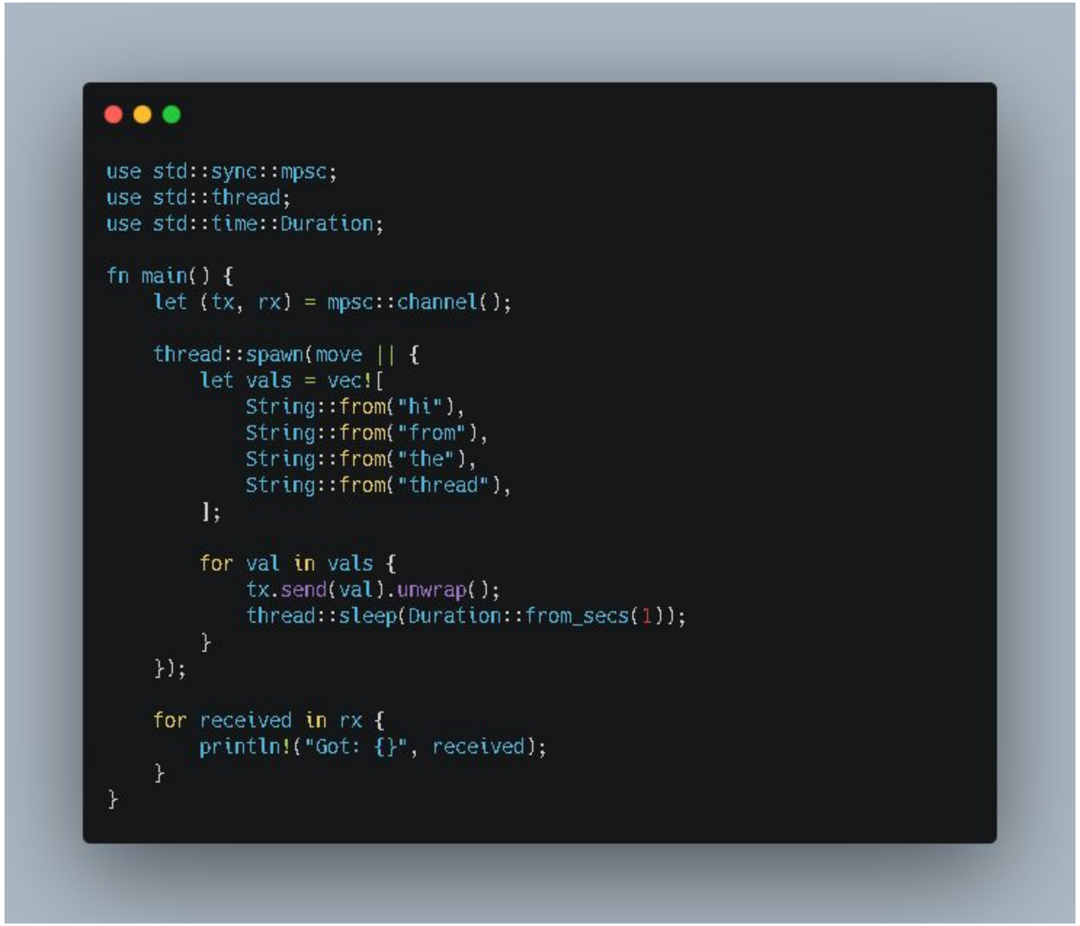
```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Messaging!



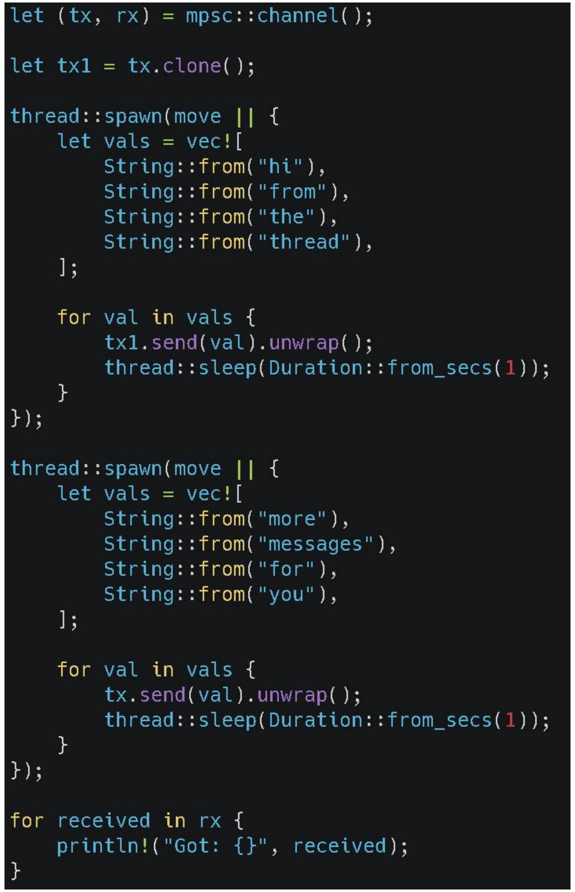
```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
  +     println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Messaging!



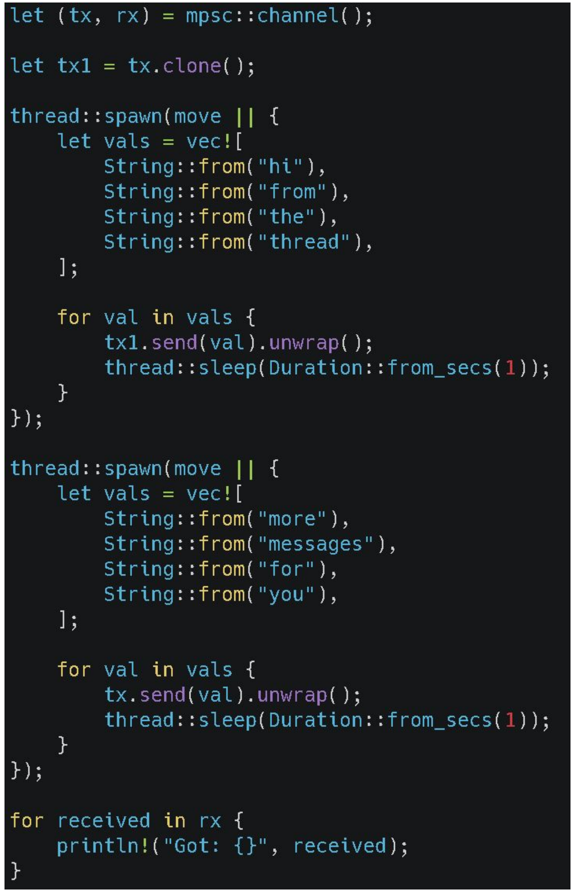
```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Messaging!



```rust
let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();

thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Got: {}", received);
}
```

Messaging!



```
let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();

thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Got: {}", received);
}
```

Let’s do an example

We have some array of numbers, we want to sum all of the elements in this array.

We want each thread to sum up some portion of this array.

Now with messaging!

That’s all for now!

See you next episode.

Reminders:

extra/0 credit practice problems that are always open - mention in lecture when each are possible

add github repo with example code from lecture

add easy EC points to MPs - showcase interesting extensions during lecture

whenever we give lecture give them a chapter to follow along with

student interaction in lecture like steltzer- email students if they do well

mention common pitfalls of MPs after due date

partial credit until 1 week after 50%

emphasize early that you will get whatever you put in - lots of opportunities to do more