# INTRO TO RUST LANG

# CONCURRENCY: ASYNC/AWAIT

Benjamin Owad, David Rudo, and Connor Tsui

# Async is Complicated

- Rust is a **systems** programming language -- different design choices were made
    - Rust async != JavaScript async != C# async
- Async is still evolving both as a feature in Rust and as a programming paradigm
- Async is not a mutually exclusive feature, parallelism and concurrency can mix in Rust
- We're going to keep this lecture primarily focused on the high level details of using async rather than creating your own Futures

# What is Asynchronous Code?

- A *concurrent* programming model supported by many languages
  - All in slightly different forms under the hood
- Allows for a large number of tasks on a few OS threads
- Still preserves the "feel" of synchronous programming through `async/await` syntax

# Rust Async vs Other Concurrency Models

- OS threads
  - Very easy to express, but hard to synchronize and have overhead on startup
- Event driven programming
  - Can be performant with callbacks
  - Causes overly verbose non-linear control flow (debugging nightmare)
- Coroutines
  - Supports many tasks like async
  - Abstract away low-level details needed for systems programmers
- Actor Model
  - A fine solution for many distributed systems using message passing
  - Leaves practical issues such as control flow and retry logic up to the user

# What Makes Rust Async Special?

- Futures are inert
  - Futures **only make progress when polled**, dropping a future stops progress
- Async is zero-cost
  - Only pay for what you use (like iterators)
  - Async without heap allocation or dynamic dispatch
  - Great for low-resource systems
- Rust has no built-in runtime
  - Provided by community crates such as Tokio
- Single and Multithreaded runtimes are possible in Rust
  - Have different advantages/disadvantages

# Threaded Download

```rust
fn get_two_sites() {
    // Spawn two threads to do work.
    let thread_one = thread::spawn(|| download("https://www.foo.com"));
    let thread_two = thread::spawn(|| download("https://www.bar.com"));

    // Wait for both threads to complete.
    thread_one.join().expect("thread one panicked");
    thread_two.join().expect("thread two panicked");
}
```

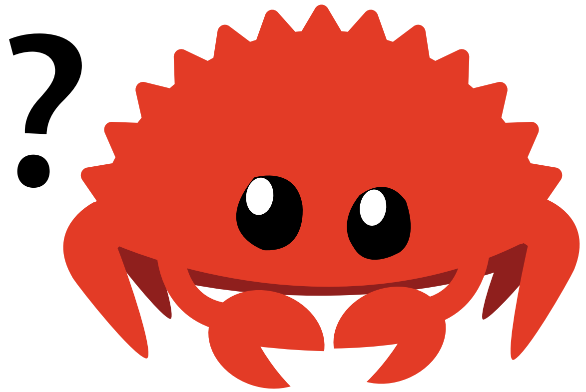
- This is pretty wasteful, let's use async instead!

# Async Download

```rust
async fn get_two_sites_async() {
    // Create two different "futures" which, when run to completion,
    // will asynchronously download the webpages.
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // Run both futures to completion at the same time.
    futures::join!(future_one, future_two);

    // Could've instead done:
    //  future_one.await;
    //  future_two.await;
    // But would've been slower since serial computation
}
```

# Another Async Example

```
async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is printed
    future.await; // printing should happen now?
}
```



# Another Async Example Error

```
5 | fn main() {
  |    ---- this is not `async`
6 |     let future = hello_world(); // Nothing is printed
7 |     future.await; // printing should happen now?
  |            ^^^^^ only allowed inside `async` functions and blocks
```

- We can only use `await` in async code blocks (which main isn't)
- We can fix this with an executor

# Another Async Example Fixed

```rust
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is printed
    block_on(future); // `future` is run and "hello, world!" is printed
}
```

- `block_on` blocks the current thread until the provided future has finished
- Other executors may provide more complex behavior
    - like scheduling multiple futures onto the same thread

# Futures Simplified

```
trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

- An async computation that can produce a value (even `()`)
- Above is a *simplified* version of the trait
- Futures are only advanced via the `poll` function

# Polling

- If a future completes it returns `Poll::Ready(result)`, else `Poll::Pending`
- The future arranges for the `wake()` function to be called when more progress can be made and makes the executor continue
    - This is how an executor is able to ensure progress without constant polling
- IMPORTANT: What would happen if we put a long blocking function in our future?

# Futures in depth

May not need to know all this

# Socket Read Future Example

```rust
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            // The socket has data -- read it into a buffer and return it.
            Poll::Ready(self.socket.read_buf())
        } else {
            // The socket does not yet have data.
            //
            // Arrange for `wake` to be called once data is available.
            // When data becomes available, `wake` will be called, and the
            // user of this `Future` will know to call `poll` again and
            // receive data.
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}
```

# Composing Futures Example

```
pub struct AndThenFut<FutureA, FutureB> {
    first: Option<FutureA>,
    second: FutureB,
}

impl<FutureA, FutureB> SimpleFuture for AndThenFut<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if let Some(first) = &mut self.first {
            match first.poll(wake) {
                // We've completed the first future -- remove it and start on the second!
                Poll::Ready(()) => self.first.take(),
                Poll::Pending => return Poll::Pending, // Couldn't yet complete the first future
            };
        }
        // Now that the first future is done, attempt to complete the second.
        self.second.poll(wake)
    }
}
```

# Let's Talk Real Futures

```rust
trait Future {
    type Output;
    fn poll(
        // Note the change from `&mut self` to `Pin<&mut Self>`:
        self: Pin<&mut Self>,
        // and the change from `wake: fn()` to `cx: &mut Context<'_>`:
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

- `Pin` ensures that our futures are unmovable in memory
- `Context<'_>` holds info on the wake function as well as useful metadata
  - "Who" called the wake function
  - Value of type `Waker`
  - etc

# Waker

- Most futures do not complete on the first poll
- `Waker` is used to ensure the future is polled when it's ready to make progress
- `Waker` provides the following:
    - `wake()` to alert the executer that a task is ready to be polled
    - `clone()` so that it can be copied and stored

# Timer Example

```rust
pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

/// Shared state between the future and the waiting thread
struct SharedState {
    /// Whether or not the sleep time has elapsed
    completed: bool,

    /// The waker for the task that `TimerFuture` is running on.
    /// The thread can use this after setting `completed = true` to tell
    /// `TimerFuture`'s task to wake up, see that `completed = true`, and
    /// move forward.
    waker: Option<Waker>,
}
```

# Writing Our Future Implementation

```rust
impl Future for TimerFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        // Look at the shared state to see if the timer has already completed.
        let mut shared_state = self.shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

# TimerFuture Implementation

```rust
impl TimerFuture {
    /// Create a new `TimerFuture` which will complete after the provided timeout
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            // Signal that the timer has completed and wake up the latest task
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

## What Just Happened?

- Our TimerFuture launches a thread with access to a shared state variable
- In this thread, we sleep for a duration
- Once that time has passed we update the shared state `completed=true`
- We then tell the waker in our shared state to wake up the last future that polled it
- In practice, we would **never** use a thread for something like this

# Notable Takeaways

- Futures are a very powerful tool
- Futures and related functions can be implemented and managed in numerous ways
  - This is why Rust doesn't have a "default" runtime
- Futures are designed to be "interruptible", to enable efficient polling
  - Don't put large blocking code in async functions!
- While the previous future launched a thread, this is uncommon
  - IO related async code uses `epoll` or other related polling calls

# High Level Usage of Async/Await

You can wake up now

# `async` Blocks

```rust
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

- The `async` block results in a type of `Future<Output=u8>`
- `foo()` is also a type that implements `Future<Output=u8>`
  - `foo().await` will result in a value of type `u8`

# `async move`

```rust
fn move_block() -> impl Future<Output = ()> {
    let my_string = "foo".to_string();
    async move {
        // ...
        println!("{my_string}");
    }

    // println!("{my_string}"); will not compile
}
```

- Just like with closures, `move` gives an `async` block ownership of a variable
- Otherwise we had to handle future's that hold references

# `async` Lifetimes

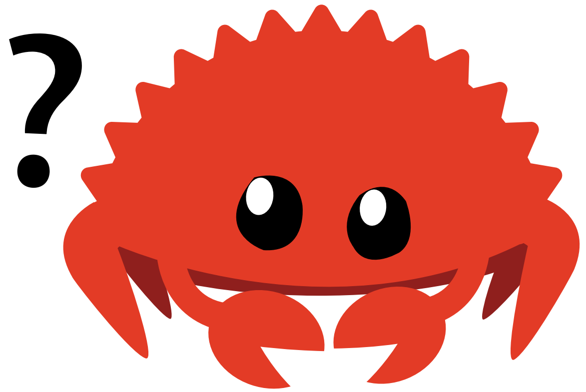
```rust
// This function:
async fn foo(x: &u8) -> u8 { *x }

// Is equivalent to this function:
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {
    async move { *x }
}
```

- Unlike typical functions, `async fn` are bounded by their argument's lifetimes
- This is because we're really putting a lifetime on the `Future` trait object

# `async` Lifetime Issues

```
fn foo() -> impl Future<Output = u8> {
    let x = 5;
    borrow_x(&x) // async function
}
```

- `async fn` must be `.await`ed while its non-static arguments are still valid
- Calling `await` immediately is one solution `foo.await`



# `async` Lifetime Solutions

```rust
fn good() -> impl Future<Output = u8> {
    async {
        let x = 5;
        borrow_x(&x).await
    }
}
```

- Another is to use an `async` block to bundle the arguments with an `async fn` call
- This is now a `'static` future

# Streams

```rust
trait Stream {
    /// The type of the value yielded by the stream.
    type Item;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

- Very similar to a `Future` but returns multiple values instead
- Functionally like an iterator
  - `poll` returns `Ready(Some(T))` or `Ready(None)` when the stream is done

# Streams in Channels

```rust
async fn send_recv() {
    const BUFFER_SIZE: usize = 10;
    let (mut tx, mut rx) = mpsc::channel::<i32>(BUFFER_SIZE);

    tx.send(1).await.unwrap();
    tx.send(2).await.unwrap();
    drop(tx);

    // `StreamExt::next` is similar to `Iterator::next`, but returns a
    // type that implements `Future<Output = Option<T>>`.
    assert_eq!(Some(1), rx.next().await);
    assert_eq!(Some(2), rx.next().await);
    assert_eq!(None, rx.next().await);
}
```

- This is a small teaser for asynchronous channels

# Executing Multiple Futures at a Time

Sometimes `.await` isn't enough

# Who Was Paying Attention?

```
async fn get_book_and_music() -> (Book, Music) {
    let book_future = get_book();
    let music_future = get_music();
    (book_future.await, music_future.await)
}
```

Which will finish executing first?

- `book_future`
- `music_future`
- This is non-deterministic
- All of the above

# Who Was Paying Attention?

```
async fn get_book_and_music() -> (Book, Music) {
    let book_future = get_book();
    let music_future = get_music();
    (book_future.await, music_future.await)
}
```

- Remember, futures are inert
- Rust won't do any work until they're actively `.await`ed
- This means `book_future` and `music_future` will be polled to completion in series rather than concurrently
  - Note: polled to completion concurrently IS NOT running concurrently

# What We Really Want is `join!`

```rust
use futures::join;

async fn get_book_and_music() -> (Book, Music) {
    let book_fut = get_book();
    let music_fut = get_music();
    join!(book_fut, music_fut)
}
```

- We still get a tuple containing the output of each Future
- But now we've "joined" them to be polled together

# `select!`

```
use futures::{future, select};

async fn count() {
    let mut a_fut = future::ready(4);
    let mut b_fut = future::ready(6);
    let mut total = 0;

    loop {
        select! {
            a = a_fut => total += a,
            b = b_fut => total += b,
            complete => break,
            default => unreachable!(), // never runs (futures are ready, then complete)
        };
    } // value at end of loop should be 10
}
```

- This runs multiple futures, but quits polling other futures after the first responds
- `select` follows the syntax `<pattern> = <expression> => <code>`

# Spawning

Here's a common asynchronous scenario:

- Imagine we have a web server that needs to accept connections
  - We don't want to block the main thread
- `async_std::task::spawn` will create and run a new task that handles connections
  - It takes a Future and returns a `JoinHandle` which can be `.await`ed
  - Note that `async_std` is

# Spawning Example

```
async fn process_request(stream: &mut TcpStream) -> Result<(), std::io::Error>{
    stream.write_all(b"HTTP/1.1 200 OK\r\n\r\n").await?;
    stream.write_all(b"Hello World").await?;
    Ok(())
}

async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").await.unwrap();
    loop {
        // Accept a new connection
        let (mut stream, _) = listener.accept().await.unwrap();
        // Now process this request without blocking the main loop
        task::spawn(async move {process_request(&mut stream).await});
    }
}
```

- Note that `spawn` requires an asynchronous runtime!

# The Power of Async Runtime

ft. Tokio

# Why Use Async Runtimes?

- Writing code that primarily manages multiple IO operations
- Interfacing with libraries that depend on an async runtime
- Need non-blocking versions of std library api functions for your async code

# When is Using Async Runtimes Bad?

- Trying to speed up CPU-bound computations
  - Just use threads or Rayon
- Reading a lot of files
  - OSes tend to not provide async file APIs
  - A thread pool will serve just as well
- Sending a single web request
  - Async runtimes are meant to help manage multiple tasks at a time
  - Use reqwest instead

# Async Message Passing

```rust
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(32);
    let tx2 = tx.clone();

    tokio::spawn(async move {
        tx.send("sending from first handle").await.unwrap();
    });

    tokio::spawn(async move {
        tx2.send("sending from second handle").await.unwrap();
    });

    while let Some(message) = rx.recv().await {
        println!("GOT = {}", message);
    }
}
```

# Why Async Message Passsing?

- An option for maintaining shared state
- A convenient way to link async code with sync code
  - Async server handling sends data to sync processing thread
- Most libraries provide tailored channels for specific use cases
  - Ex: Tokio `mpsc`, `oneshot`, `broadcast`, `watch`

# Mutex With Async

- Within an async runtime, mutexes are allowed
- Can be used easily if low contention is expected
- If high contention is an issue:
    - Restructure the code to avoid the mutex
    - Shard the mutex
    - Message passing
    - Use an async mutex (comes at a higher cost)

# Async Mutex Example

```rust
use tokio::sync::Mutex; // note! This uses the Tokio mutex

// This compiles!
// (but restructuring the code would be better in this case)
async fn increment_and_do_stuff(mutex: &Mutex<i32>) {
    let mut lock = mutex.lock().await;
    *lock += 1;

    do_something_async().await;
} // lock goes out of scope here
```

# Bridging with Synchronous Code -- Option 1

```rust
// Snippet example from Tokio Redis project
impl BlockingSubscriber {
    pub fn get_subscribed(&self) -> &[String] {
        self.inner.get_subscribed()
    }

    pub fn next_message(&mut self) -> crate::Result<Option<Message>> {
        self.rt.block_on(self.inner.next_message())
    }

    pub fn subscribe(&mut self, channels: &[String]) -> crate::Result<()> {
        self.rt.block_on(self.inner.subscribe(channels))
    }
}
```

- Build a synchronous interface to async code
- Call `block_on` on futures synchronous code needs

# Bridging with Synchronous Code -- Option 2

```rust
fn main() {
    let runtime = Builder::new_multi_thread().worker_threads(1).enable_all().build().unwrap();

    let mut handles = Vec::with_capacity(10);
    for i in 0..10 {
        handles.push(runtime.spawn(my_bg_task(i)));
    }

    // Do something time-consuming while the background tasks execute.
    std::thread::sleep(Duration::from_millis(750));
    println!("Finished time-consuming task.");

    // Wait for all of them to complete.
    for handle in handles {
        // The `spawn` method returns a `JoinHandle`. A `JoinHandle` is
        // a future, so we can wait for it using `block_on`.
        runtime.block_on(handle).unwrap();
    }
}
```

- Spawning async jobs on the run time

# Bridging with Synchronous Code -- Option 3

```
pub fn new() -> TaskSpawner {
    let (send, mut recv) = mpsc::channel(16);
    let rt = Builder::new_current_thread().enable_all().build().unwrap();

    std::thread::spawn(move || {
        rt.block_on(async move {
            while let Some(task) = recv.recv().await {
                tokio::spawn(handle_task(task));
            }
        });
    });

    TaskSpawner {
        spawn: send,
    }
}
// Sync code that sends message to async running thread
pub fn spawn_task(&self, task: Task) {
    match self.spawn.blocking_send(task) {
        // <--- snip --->
    }
}
```

- Message passing from async to sync code and vice versa

## Takeaways

- `Async/Await` is a powerful tool
- There are lots of libraries to help manage asynchronous tasks
- Is not a drop-in replacement for standard parallelism
- Has slightly different rules and best practices compared to other concurrent models

# Next Lecture: Macros

- Thank you for coming!