Welcome

“Async” is a concurrency model where multiple tasks are executed concurrently by executing each task until it would block, then switching to another task that is ready to make progress. The model allows running a larger number of tasks on a limited number of threads. This is because the per-task overhead is typically very low and operating systems provide primitives for efficiently identifying I/O that is able to proceed.

Rust’s asynchronous operation is based on “futures”, which represent work that may be completed in the future. Futures are “polled” until they signal that they are complete.

Futures are polled by an async runtime, and several different runtimes are available.

Comparisons

- Python has a similar model in its `asyncio`. However, its `Future` type is callback-based, and not polled. Async Python programs require a “loop”, similar to a runtime in Rust.
- JavaScript’s `Promise` is similar, but again callback-based. The language runtime implements the event loop, so many of the details of Promise resolution are hidden.

Schedule

Including 10 minute breaks, this session should take about 3 hours and 30 minutes. It contains:

Segment	Duration
Async Basics	40 minutes
Channels and Control Flow	20 minutes
Pitfalls	55 minutes
Exercises	1 hour and 10 minutes

Async Basics

This segment should take about 40 minutes. It contains:

Slide	Duration
async/await	10 minutes
Futures	4 minutes
State Machine	10 minutes
Runtimes	10 minutes
Tasks	10 minutes

async/await

At a high level, async Rust code looks very much like “normal” sequential code:

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

async fn count_to(count: i32) {
    for i in 0..count {
        println!("Count is: {i}!");
    }
}

async fn async_main(count: i32) {
    count_to(count).await;
}

fn main() {
    block_on(async_main(10));
}
```

▼ *Speaker Notes*

This slide should take about 6 minutes.

Key points:

- Note that this is a simplified example to show the syntax. There is no long running operation or any real concurrency in it!
- The “async” keyword is syntactic sugar. The compiler replaces the return type with a future.
- You cannot make `main` async, without additional instructions to the compiler on how to use the returned future.
- You need an executor to run async code. `block_on` blocks the current thread until the provided future has run to completion.
- `.await` asynchronously waits for the completion of another operation. Unlike `block_on`, `.await` doesn’t block the current thread.
- `.await` can only be used inside an `async` function (or block; these are introduced later).

Futures

`Future` is a trait, implemented by objects that represent an operation that may not be complete yet. A future can be polled, and `poll` returns a `Poll`.

```
use std::pin::Pin;
use std::task::Context;

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

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

An async function returns an `impl Future`. It's also possible (but uncommon) to implement `Future` for your own types. For example, the `JoinHandle` returned from `tokio::spawn` implements `Future` to allow joining to it.

The `.await` keyword, applied to a Future, causes the current async function to pause until that Future is ready, and then evaluates to its output.

▼ *Speaker Notes*

This slide should take about 4 minutes.

- The `Future` and `Poll` types are implemented exactly as shown; click the links to show the implementations in the docs.
- `Context` allows a Future to schedule itself to be polled again when an event such as a timeout occurs.
- `Pin` ensures that the Future isn't moved in memory, so that pointers into that future remain valid. This is required to allow references to remain valid after an `.await`. We will address `Pin` in the "Pitfalls" segment.

State Machine

Rust transforms an async function or block to a hidden type that implements `Future`, using a state machine to track the function's progress. The details of this transform are complex, but it helps to have a schematic understanding of what is happening. The following function

```
/// Sum two D10 rolls plus a modifier.
async fn two_d10(modifier: u32) -> u32 {
    let first_roll = roll_d10().await;
    let second_roll = roll_d10().await;
    first_roll + second_roll + modifier
}
```

is transformed to something like

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

/// Sum two D10 rolls plus a modifier.
fn two_d10(modifier: u32) -> TwoD10 {
    TwoD10::Init { modifier }
}

enum TwoD10 {
    // Function has not begun yet.
    Init { modifier: u32 },
    // Waiting for first `.await` to complete.
    FirstRoll { modifier: u32, fut: RollD10Future },
    // Waiting for second `.await` to complete.
    SecondRoll { modifier: u32, first_roll: u32, fut: RollD10Future },
}

impl Future for TwoD10 {
    type Output = u32;
    fn poll(mut self: Pin<&mut Self>, ctx: &mut Context) -> Poll<Self::Output> {
        loop {
            match *self {
                TwoD10::Init { modifier } => {
                    // Create future for first dice roll.
                    let fut = roll_d10();
                    *self = TwoD10::FirstRoll { modifier, fut };
                }
                TwoD10::FirstRoll { modifier, ref mut fut } => {
                    // Poll sub-future for first dice roll.
                    if let Poll::Ready(first_roll) = fut.poll(ctx) {
                        // Create future for second roll.
                        let fut = roll_d10();
                        *self = TwoD10::SecondRoll { modifier, first_roll, fut };
                    } else {
                        return Poll::Pending;
                    }
                }
                TwoD10::SecondRoll { modifier, first_roll, ref mut fut } => {
                    // Poll sub-future for second dice roll.
                    if let Poll::Ready(second_roll) = fut.poll(ctx) {
                        return Poll::Ready(first_roll + second_roll + modifier);
                    } else {
                        return Poll::Pending;
                    }
                }
            }
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

This example is illustrative, and isn't an accurate representation of the Rust compiler's transformation. The important things to notice here are:

- Calling an async function does nothing but construct and return a future.
- All local variables are stored in the function's future, using an enum to identify where execution is currently suspended.
- An `.await` in the async function is translated into an a new state containing all live variables and the awaited future. The `loop` then handles that updated state, polling the future until it returns `Poll::Ready`.
- Execution continues eagerly until a `Poll::Pending` occurs. In this simple example, every future is ready immediately.
- `main` contains a naïve executor, which just busy-loops until the future is ready. We will discuss real executors shortly.

More to Explore

Imagine the `Future` data structure for a deeply nested stack of async functions. Each function's `Future` contains the `Future` structures for the functions it calls. This can result in unexpectedly large compiler-generated `Future` types.

This also means that recursive async functions are challenging. Compare to the common error of building recursive type, such as

```
enum LinkedList<T> {
    Node { value: T, next: LinkedList<T> },
    Nil,
}
```

The fix for a recursive type is to add a layer of indirection, such as with `Box`. Similarly, a recursive async function must box the recursive future:

```
async fn count_to(n: u32) {
    if n > 0 {
        Box::pin(count_to(n - 1)).await;
        println!("{n}");
    }
}
```

Runtimes

A *runtime* provides support for performing operations asynchronously (a *reactor*) and is responsible for executing futures (an *executor*). Rust does not have a "built-in" runtime, but several options are available:

- Tokio: performant, with a well-developed ecosystem of functionality like Hyper for HTTP or Tonic for gRPC.
- smol: simple and lightweight

Several larger applications have their own runtimes. For example, Fuchsia already has one.

▼ *Speaker Notes*

This slide and its sub-slides should take about 10 minutes.

- Note that of the listed runtimes, only Tokio is supported in the Rust playground. The playground also does not permit any I/O, so most interesting async things can't run in the playground.
- Futures are "inert" in that they do not do anything (not even start an I/O operation) unless there is an executor polling them. This differs from JS Promises, for example, which will run to completion even if they are never used.

Tokio

Tokio provides:

- A multi-threaded runtime for executing asynchronous code.
- An asynchronous version of the standard library.
- A large ecosystem of libraries.

```
use tokio::time;

async fn count_to(count: i32) {
    for i in 0..count {
        println!("Count in task: {i}!");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

#[tokio::main]
async fn main() {
    tokio::spawn(count_to(10));

    for i in 0..5 {
        println!("Main task: {i}");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}
```

▼ *Speaker Notes*

- With the `tokio::main` macro we can now make `main` async.
- The `spawn` function creates a new, concurrent "task".
- Note: `spawn` takes a `Future`, you don't call `.await` on `count_to`.

Further exploration:

- Why does `count_to` not (usually) get to 10? This is an example of async cancellation. `tokio::spawn` returns a handle which can be awaited to wait until it finishes.
- Try `count_to(10).await` instead of spawning.
- Try awaiting the task returned from `tokio::spawn`.

Tasks

Rust has a task system, which is a form of lightweight threading.

A task has a single top-level future which the executor polls to make progress. That future may have one or more nested futures that its `poll` method polls, corresponding loosely to a call stack. Concurrency within a task is possible by polling multiple child futures, such as racing a timer and an I/O operation.

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:0").await?;
    println!("listening on port {}", listener.local_addr()?.port());

    loop {
        let (mut socket, addr) = listener.accept().await?;

        println!("connection from {addr:?}");

        tokio::spawn(async move {
            socket.write_all(b"Who are you?\n").await.expect("socket error");

            let mut buf = vec![0; 1024];
            let name_size = socket.read(&mut buf).await.expect("socket error");
            let name = std::str::from_utf8(&buf[..name_size]).unwrap().trim();
            let reply = format!("Thanks for dialing in, {name}!\n");
            socket.write_all(reply.as_bytes()).await.expect("socket error");
        });
    }
}
```

▼ *Speaker Notes*

This slide should take about 6 minutes.

Copy this example into your prepared `src/main.rs` and run it from there.

Try connecting to it with a TCP connection tool like nc or telnet.

- Ask students to visualize what the state of the example server would be with a few connected clients. What tasks exist? What are their Futures?
- This is the first time we've seen an `async` block. This is similar to a closure, but does not take any arguments. Its return value is a Future, similar to an `async fn`.
- Refactor the async block into a function, and improve the error handling using `?`.

Channels and Control Flow

This segment should take about 20 minutes. It contains:

Slide	Duration
Async Channels	10 minutes
Join	4 minutes
Select	5 minutes

Async Channels

Several crates have support for asynchronous channels. For instance `tokio`:

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

async fn ping_handler(mut input: mpsc::Receiver<()>) {
    let mut count: usize = 0;

    while let Some(_) = input.recv().await {
        count += 1;
        println!("Received {count} pings so far.");
    }

    println!("ping_handler complete");
}

#[tokio::main]
async fn main() {
    let (sender, receiver) = mpsc::channel(32);
    let ping_handler_task = tokio::spawn(ping_handler(receiver));
    for i in 0..10 {
        sender.send(()).await.expect("Failed to send ping.");
        println!("Sent {} pings so far.", i + 1);
    }

    drop(sender);
    ping_handler_task.await.expect("Something went wrong in ping handler task.");
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Change the channel size to `3` and see how it affects the execution.
- Overall, the interface is similar to the `sync` channels as seen in the morning class.
- Try removing the `std::mem::drop` call. What happens? Why?
- The Flume crate has channels that implement both `sync` and `async` `send` and `recv`. This can be convenient for complex applications with both IO and heavy CPU processing tasks.
- What makes working with `async` channels preferable is the ability to combine them with other `future`s to combine them and create complex control flow.

Join

A join operation waits until all of a set of futures are ready, and returns a collection of their results. This is similar to `Promise.all` in JavaScript or `asyncio.gather` in Python.

```
use anyhow::Result;
use futures::future;
use reqwest;
use std::collections::HashMap;

async fn size_of_page(url: &str) -> Result<usize> {
    let resp = reqwest::get(url).await?;
    Ok(resp.text().await?.len())
}

#[tokio::main]
async fn main() {
    let urls: [&str; 4] = [
        "https://google.com",
        "https://httpbin.org/ip",
        "https://play.rust-lang.org/",
        "BAD_URL",
    ];
    let futures_iter = urls.into_iter().map(size_of_page);
    let results = future::join_all(futures_iter).await;
    let page_sizes_dict: HashMap<&str, Result<usize>> =
        urls.into_iter().zip(results.into_iter()).collect();
    println!("{page_sizes_dict:?}");
}
```

▼ *Speaker Notes*

This slide should take about 4 minutes.

Copy this example into your prepared `src/main.rs` and run it from there.

- For multiple futures of disjoint types, you can use `std::future::join!` but you must know how many futures you will have at compile time. This is currently in the `futures` crate, soon to be stabilised in `std::future`.
- The risk of `join` is that one of the futures may never resolve, this would cause your program to stall.
- You can also combine `join_all` with `join!` for instance to join all requests to an http service as well as a database query. Try adding a `tokio::time::sleep` to the future, using `futures::join!`. This is not a timeout (that requires `select!`, explained in the next chapter), but demonstrates `join!`.

Select

A select operation waits until any of a set of futures is ready, and responds to that future's result. In JavaScript, this is similar to `Promise.race`. In Python, it compares to `asyncio.wait(task_set, return_when=asyncio.FIRST_COMPLETED)`.

Similar to a match statement, the body of `select!` has a number of arms, each of the form `pattern = future => statement`. When a `future` is ready, its return value is destructured by the `pattern`. The `statement` is then run with the resulting variables. The `statement` result becomes the result of the `select!` macro.

```
use tokio::sync::mpsc;
use tokio::time::{Duration, sleep};

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(32);
    let listener = tokio::spawn(async move {
        tokio::select! {
            Some(msg) = rx.recv() => println!("got: {msg}"),
            _ = sleep(Duration::from_millis(50)) => println!("timeout"),
        };
    });
    sleep(Duration::from_millis(10)).await;
    tx.send(String::from("Hello!")).await.expect("Failed to send greeting");

    listener.await.expect("Listener failed");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The `listener` async block here is a common form: wait for some async event, or for a timeout. Change the `sleep` to sleep longer to see it fail. Why does the `send` also fail in this situation?
- `select!` is also often used in a loop in "actor" architectures, where a task reacts to events in a loop. That has some pitfalls, which will be discussed in the next segment.

Pitfalls

Async / await provides convenient and efficient abstraction for concurrent asynchronous programming. However, the async/await model in Rust also comes with its share of pitfalls and footguns. We illustrate some of them in this chapter.

This segment should take about 55 minutes. It contains:

Slide	Duration
Blocking the Executor	10 minutes
Pin	20 minutes
Async Traits	5 minutes
Cancellation	20 minutes

Blocking the executor

Most async runtimes only allow IO tasks to run concurrently. This means that CPU blocking tasks will block the executor and prevent other tasks from being executed. An easy workaround is to use async equivalent methods where possible.

```
use futures::future::join_all;
use std::time::Instant;

async fn sleep_ms(start: &Instant, id: u64, duration_ms: u64) {
    std::thread::sleep(std::time::Duration::from_millis(duration_ms));
    println!(
        "future {id} slept for {duration_ms}ms, finished after {}ms",
        start.elapsed().as_millis()
    );
}

#[tokio::main(flavor = "current_thread")]
async fn main() {
    let start = Instant::now();
    let sleep_futures = (1..=10).map(|t| sleep_ms(&start, t, t * 10));
    join_all(sleep_futures).await;
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- Run the code and see that the sleeps happen consecutively rather than concurrently.
- The `"current_thread"` flavor puts all tasks on a single thread. This makes the effect more obvious, but the bug is still present in the multi-threaded flavor.
- Switch the `std::thread::sleep` to `tokio::time::sleep` and await its result.
- Another fix would be to `tokio::task::spawn_blocking` which spawns an actual thread and transforms its handle into a future without blocking the executor.
- You should not think of tasks as OS threads. They do not map 1 to 1 and most executors will allow many tasks to run on a single OS thread. This is particularly problematic when interacting with other libraries via FFI, where that library might depend on thread-local storage or map to specific OS threads (e.g., CUDA). Prefer `tokio::task::spawn_blocking` in such situations.
- Use sync mutexes with care. Holding a mutex over an `.await` may cause another task to block, and that task may be running on the same thread.

Pin

Recall an async function or block creates a type implementing `Future` and containing all of the local variables. Some of those variables can hold references (pointers) to other local variables. To ensure those remain valid, the future can never be moved to a different memory location.

To prevent moving the future type in memory, it can only be polled through a pinned pointer. `Pin` is a wrapper around a reference that disallows all operations that would move the instance it points to into a different memory location.

```
use tokio::sync::{mpsc, oneshot};
use tokio::task::spawn;
use tokio::time::{Duration, sleep};

// A work item. In this case, just sleep for the given time and respond
// with a message on the `respond_on` channel.
#[derive(Debug)]
struct Work {
    input: u32,
    respond_on: oneshot::Sender<u32>,
}

// A worker which listens for work on a queue and performs it.
async fn worker(mut work_queue: mpsc::Receiver<Work>) {
    let mut iterations = 0;
    loop {
        tokio::select! {
            Some(work) = work_queue.recv() => {
                sleep(Duration::from_millis(10)).await; // Pretend to work.
                work.respond_on
                    .send(work.input * 1000)
                    .expect("failed to send response");
                iterations += 1;
            }
            // TODO: report number of iterations every 100ms
        }
    }
}

// A requester which requests work and waits for it to complete.
async fn do_work(work_queue: &mpsc::Sender<Work>, input: u32) -> u32 {
    let (tx, rx) = oneshot::channel();
    work_queue
        .send(Work { input, respond_on: tx })
        .await
        .expect("failed to send on work queue");
    rx.await.expect("failed waiting for response")
}

#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::channel(10);
    spawn(worker(rx));
    for i in 0..100 {
        let resp = do_work(&tx, i).await;
        println!("work result for iteration {i}: {resp}");
    }
}
```

▼ *Speaker Notes*

This slide should take about 20 minutes.

- You may recognize this as an example of the actor pattern. Actors typically call `select!` in a loop.

- Naively add a `_ = sleep(Duration::from_millis(100)) => { println!(..) }` to the `select!`. This will never execute. Why?
- Instead, add a `timeout_fut` containing that future outside of the `loop`:

```
let timeout_fut = sleep(Duration::from_millis(100));
loop {
    select! {
        ..,
        _ = timeout_fut => { println!(..); },
    }
}
```

- This still doesn't work. Follow the compiler errors, adding `&mut` to the `timeout_fut` in the `select!` to work around the move, then using `Box::pin`:

```
let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
loop {
    select! {
        ..,
        _ = &mut timeout_fut => { println!(..); },
    }
}
```

- This compiles, but once the timeout expires it is `Poll::Ready` on every iteration (a fused future would help with this). Update to reset `timeout_fut` every time it expires:

```
let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
loop {
    select! {
        _ = &mut timeout_fut => {
            println!(..);
            timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
        },
    }
}
```

- Box allocates on the heap. In some cases, `std::pin::pin!` (only recently stabilized, with older code often using `tokio::pin!`) is also an option, but that is difficult to use for a future that is reassigned.
- Another alternative is to not use `pin` at all but spawn another task that will send to a `oneshot` channel every 100ms.
- Data that contains pointers to itself is called self-referential. Normally, the Rust borrow checker would prevent self-referential data from being moved, as the references cannot outlive the data they point to. However, the code transformation for async blocks and functions is not verified by the borrow checker.
- `Pin` is a wrapper around a reference. An object cannot be moved from its place using a pinned pointer. However, it can still be moved through an unpinned pointer.
- The `poll` method of the `Future` trait uses `Pin<&mut Self>` instead of `&mut Self` to refer to the instance. That's why it can only be called on a pinned pointer.

Async Traits

Async methods in traits were stabilized in the 1.75 release. This required support for using return-position `impl Trait` in traits, as the desugaring for `async fn` includes `-> impl Future<Output = ...>`.

However, even with the native support, there are some pitfalls around `async fn`:

- Return-position `impl Trait` captures all in-scope lifetimes (so some patterns of borrowing cannot be expressed).
- Async traits cannot be used with trait objects (`dyn Trait` support).

The async_trait crate provides a workaround for `dyn` support through a macro, with some caveats:

```
use async_trait::async_trait;
use std::time::Instant;
use tokio::time::{Duration, sleep};

#[async_trait]
trait Sleeper {
    async fn sleep(&self);
}

struct FixedSleeper {
    sleep_ms: u64,
}

#[async_trait]
impl Sleeper for FixedSleeper {
    async fn sleep(&self) {
        sleep(Duration::from_millis(self.sleep_ms)).await;
    }
}

async fn run_all_sleepers_multiple_times(
    sleepers: Vec<Box<dyn Sleeper>>,
    n_times: usize,
) {
    for _ in 0..n_times {
        println!("Running all sleepers...");
        for sleeper in &sleepers {
            let start = Instant::now();
            sleeper.sleep().await;
            println!("Slept for {} ms", start.elapsed().as_millis());
        }
    }
}

#[tokio::main]
async fn main() {
    let sleepers: Vec<Box<dyn Sleeper>> = vec![
        Box::new(FixedSleeper { sleep_ms: 50 }),
        Box::new(FixedSleeper { sleep_ms: 100 }),
    ];
    run_all_sleepers_multiple_times(sleepers, 5).await;
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `async_trait` is easy to use, but note that it's using heap allocations to achieve this. This heap allocation has performance overhead.

- The challenges in language support for `async trait` are too deep to describe in-depth in this class. See this blog post by Niko Matsakis if you are interested in digging deeper. See also these keywords:
    - RPIT: short for return-position `impl Trait`.
    - RPITIT: short for return-position `impl Trait` in trait (RPIT in trait).
- Try creating a new sleeper struct that will sleep for a random amount of time and adding it to the `Vec`.

Cancellation

Dropping a future implies it can never be polled again. This is called *cancellation* and it can occur at any `await` point. Care is needed to ensure the system works correctly even when futures are cancelled. For example, it shouldn't deadlock or lose data.

```
use std::io;
use std::time::Duration;
use tokio::io::{AsyncReadExt, AsyncWriteExt, DuplexStream};

struct LinesReader {
    stream: DuplexStream,
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream }
    }

    async fn next(&mut self) -> io::Result<Option<String>> {
        let mut bytes = Vec::new();
        let mut buf = [0];
        while self.stream.read(&mut buf[..]).await? != 0 {
            bytes.push(buf[0]);
            if buf[0] == b'\n' {
                break;
            }
        }
        if bytes.is_empty() {
            return Ok(None);
        }
        let s = String::from_utf8(bytes)
            .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "not UTF-8"))?
        Ok(Some(s))
    }
}

async fn slow_copy(source: String, mut dest: DuplexStream) -> io::Result<()> {
    for b in source.bytes() {
        dest.write_u8(b).await?;
        tokio::time::sleep(Duration::from_millis(10)).await
    }
    Ok(())
}

#[tokio::main]
async fn main() -> io::Result<()> {
    let (client, server) = tokio::io::duplex(5);
    let handle = tokio::spawn(slow_copy("hi\nthere\n".to_owned(), client));

    let mut lines = LinesReader::new(server);
    let mut interval = tokio::time::interval(Duration::from_millis(60));
    loop {
        tokio::select! {
            _ = interval.tick() => println!("tick!"),
            line = lines.next() => if let Some(l) = line? {
                print!("{}", l)
            } else {
                break
            },
        }
    }
    handle.await.unwrap()?;
    Ok(())
}
```

This slide should take about 18 minutes.

- The compiler doesn't help with cancellation-safety. You need to read API documentation and consider what state your `async fn` holds.
- Unlike `panic` and `?`, cancellation is part of normal control flow (vs error-handling).
- The example loses parts of the string.
  - Whenever the `tick()` branch finishes first, `next()` and its `buf` are dropped.
  - `LinesReader` can be made cancellation-safe by making `buf` part of the struct:

```
struct LinesReader {
    stream: DuplexStream,
    bytes: Vec<u8>,
    buf: [u8; 1],
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream, bytes: Vec::new(), buf: [0] }
    }
    async fn next(&mut self) -> io::Result<Option<String>> {
        // prefix buf and bytes with self.
        // ...
        let raw = std::mem::take(&mut self.bytes);
        let s = String::from_utf8(raw)
            .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "not
UTF-8"))?;
        // ...
    }
}
```

- `Interval::tick` is cancellation-safe because it keeps track of whether a tick has been 'delivered'.
- `AsyncReadExt::read` is cancellation-safe because it either returns or doesn't read data.
- `AsyncBufReadExt::read_line` is similar to the example and *isn't* cancellation-safe. See its documentation for details and alternatives.

Exercises

This segment should take about 1 hour and 10 minutes. It contains:

Slide	Duration
Dining Philosophers	20 minutes
Broadcast Chat Application	30 minutes
Solutions	20 minutes

Dining Philosophers — Async

See dining philosophers for a description of the problem.

As before, you will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

struct Philosopher {
    name: String,
    // left_chopstick: ...
    // right_chopstick: ...
    // thoughts: ...
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        // Keep trying until we have both chopsticks
        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

// tokio scheduler doesn't deadlock with 5 philosophers, so have 2.
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia"];

#[tokio::main]
async fn main() {
    // Create chopsticks

    // Create philosophers

    // Make them think and eat

    // Output their thoughts
}
```

Since this time you are using Async Rust, you'll need a `tokio` dependency. You can use the following `Cargo.toml`:

```
[package]
name = "dining-philosophers-async-dine"
version = "0.1.0"
edition = "2024"

[dependencies]
tokio = { version = "1.26.0", features = ["sync", "time", "macros", "rt-multi-
thread"] }
```

Also note that this time you have to use the `Mutex` and the `mpsc` module from the `tokio` crate.

This slide should take about 20 minutes.

- Can you make your implementation single-threaded?

Broadcast Chat Application

In this exercise, we want to use our new knowledge to implement a broadcast chat application. We have a chat server that the clients connect to and publish their messages. The client reads user messages from the standard input, and sends them to the server. The chat server broadcasts each message that it receives to all the clients.

For this, we use a broadcast channel on the server, and `tokio_websockets` for the communication between the client and the server.

Create a new Cargo project and add the following dependencies:

Cargo.toml:

```
[package]
name = "chat-async"
version = "0.1.0"
edition = "2024"

[dependencies]
futures-util = { version = "0.3.31", features = ["sink"] }
http = "1.3.1"
tokio = { version = "1.47.0", features = ["full"] }
tokio-websockets = { version = "0.12.0", features = ["client", "fastrand", "server",
"sha1_smol"] }
```

The required APIs

You are going to need the following functions from `tokio` and `tokio_websockets`. Spend a few minutes to familiarize yourself with the API.

- StreamExt::next() implemented by `WebSocketStream`: for asynchronously reading messages from a Websocket Stream.
- SinkExt::send() implemented by `WebSocketStream`: for asynchronously sending messages on a Websocket Stream.
- Lines::next_line(): for asynchronously reading user messages from the standard input.
- Sender::subscribe(): for subscribing to a broadcast channel.

Two binaries

Normally in a Cargo project, you can have only one binary, and one `src/main.rs` file. In this project, we need two binaries. One for the client, and one for the server. You could potentially make them two separate Cargo projects, but we are going to put them in a single Cargo project with two binaries. For this to work, the client and the server code should go under `src/bin` (see the documentation).

Copy the following server and client code into `src/bin/server.rs` and `src/bin/client.rs`, respectively. Your task is to complete these files as described below.

src/bin/server.rs:

```rust
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{Sender, channel};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    // TODO: For a hint, see the description of the task below.

}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("listening on port 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Wrap the raw TCP stream into a websocket.
            let (_req, ws_stream) = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```rust
use futures_util::SinkExt;
use futures_util::stream::StreamExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();


    // TODO: For a hint, see the description of the task below.

}
```

Running the binaries

Run the server with:

```
cargo run --bin server
```

and the client with:

```
cargo run --bin client
```

Tasks

- Implement the `handle_connection` function in `src/bin/server.rs`.
  - Hint: Use `tokio::select!` for concurrently performing two tasks in a continuous loop. One task receives messages from the client and broadcasts them. The other sends messages received by the server to the client.
- Complete the main function in `src/bin/client.rs`.
  - Hint: As before, use `tokio::select!` in a continuous loop for concurrently performing two tasks: (1) reading user messages from standard input and sending them to the server, and (2) receiving messages from the server, and displaying them for the user.
- Optional: Once you are done, change the code to broadcast messages to all clients, but the sender of the message.

Solutions

Dining Philosophers — Async

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

struct Philosopher {
    name: String,
    left_chopstick: Arc<Mutex<Chopstick>>,
    right_chopstick: Arc<Mutex<Chopstick>>,
    thoughts: mpsc::Sender<String>,
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        // Keep trying until we have both chopsticks
        // Pick up chopsticks...
        let _left_chopstick = self.left_chopstick.lock().await;
        let _right_chopstick = self.right_chopstick.lock().await;

        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;

        // The locks are dropped here
    }
}

// tokio scheduler doesn't deadlock with 5 philosophers, so have 2.
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia"];

#[tokio::main]
async fn main() {
    // Create chopsticks
    let mut chopsticks = vec![];
    PHILOSOPHERS
        .iter()
        .for_each(|_| chopsticks.push(Arc::new(Mutex::new(Chopstick))));

    // Create philosophers
    let (philosophers, mut rx) = {
        let mut philosophers = vec![];
        let (tx, rx) = mpsc::channel(10);
        for (i, name) in PHILOSOPHERS.iter().enumerate() {
            let mut left_chopstick = Arc::clone(&chopsticks[i]);
            let mut right_chopstick =
                Arc::clone(&chopsticks[(i + 1) % PHILOSOPHERS.len()]);
            if i == PHILOSOPHERS.len() - 1 {
                std::mem::swap(&mut left_chopstick, &mut right_chopstick);
            }
            philosophers.push(Philosopher {
                name: name.to_string(),
                left_chopstick,
                right_chopstick,
                thoughts: tx.clone(),
```

```
        (philosophers, rx)
        // tx is dropped here, so we don't need to explicitly drop it later
    };

    // Make them think and eat
    for phil in philosophers {
        tokio::spawn(async move {
            for _ in 0..100 {
                phil.think().await;
                phil.eat().await;
            }
        });
    }

    // Output their thoughts
    while let Some(thought) = rx.recv().await {
        println!("Here is a thought: {thought}");
    }
}
```

Broadcast Chat Application

src/bin/server.rs:

```rust
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{Sender, channel};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    ws_stream
        .send(Message::text("Welcome to chat! Type a message".to_string()))
        .await?;
    let mut bcast_rx = bcast_tx.subscribe();

    // A continuous loop for concurrently performing two tasks: (1) receiving
    // messages from `ws_stream` and broadcasting them, and (2) receiving
    // messages on `bcast_rx` and sending them to the client.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("From client {addr:?} {text:?}");
                            bcast_tx.send(text.into())?;
                        }
                    }
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            msg = bcast_rx.recv() => {
                ws_stream.send(Message::text(msg?)).await?;
            }
        }
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("listening on port 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Wrap the raw TCP stream into a websocket.
            let (_req, ws_stream) = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```rust
use futures_util::SinkExt;
use futures_util::stream::StreamExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();

    // Continuous loop for concurrently sending and receiving messages.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("From server: {}", text);
                        }
                    },
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            res = stdin.next_line() => {
                match res {
                    Ok(None) => return Ok(()),
                    Ok(Some(line)) =>
ws_stream.send(Message::text(line.to_string())).await?,
                    Err(err) => return Err(err.into()),
                }
            }

        }
    }
}
```

Welcome to Idiomatic Rust

Rust Fundamentals introduced Rust syntax and core concepts. We now want to go one step further: how do you use Rust *effectively* in your projects? What does *idiomatic* Rust look like?

This course is opinionated: we will nudge you towards some patterns, and away from others. Nonetheless, we do recognize that some projects may have different needs. We always provide the necessary information to help you make informed decisions within the context and constraints of your own projects.

> ⚠ This course is under **active development**.
>
> The material may change frequently and there might be errors that have not yet been spotted. Nonetheless, we encourage you to browse through and provide early feedback!

Schedule

Including 10 minute breaks, this session should take about 25 minutes. It contains:

Segment	Duration
Leveraging the Type System	25 minutes

▼ *Speaker Notes*

The course will cover the topics listed below. Each topic may be covered in one or more slides, depending on its complexity and relevance.

Foundations of API design

- Golden rule: prioritize clarity and readability at the callsite. People will spend much more time reading the call sites than declarations of the functions being called.
- Make your API predictable
  - Follow naming conventions (case conventions, prefer vocabulary precedented in the standard library - e.g., methods should be called “push” not “push_back”, “is_empty” not “empty” etc.)
  - Know the vocabulary types and traits in the standard library, and use them in your APIs. If something feels like a basic type/algorithm, check in the standard library first.
  - Use well-established API design patterns that we will discuss later in this class (e.g., newtype, owned/view type pairs, error handling)
- Write meaningful and effective doc comments (e.g., don’t merely repeat the method name with spaces instead of underscores, don’t repeat the same information just to fill out every markdown tag, provide usage examples)

Leveraging the type system

- Short recap on enums, structs and type aliases
- Newtype pattern and encapsulation: parse, don’t validate
- Extension traits: avoid the newtype pattern when you want to provide additional behaviour
- RAII, scope guards and drop bombs: using `Drop` to clean up resources, trigger actions or

- “Token” types: force users to prove they’ve performed a specific action
- The typestate pattern: enforce correct state transitions at compile-time
- Using the borrow checker to enforce invariants that have nothing to do with memory ownership
    - OwnedFd/BorrowedFd in the standard library
    - Branded types

Don’t fight the borrow checker

- “Owned” types and “view” types: `&str` and `String`, `Path` and `PathBuf`, etc.
- Don’t hide ownership requirements: avoid hidden `.clone()`, learn to love `Cow`
- Split types along ownership boundaries
- Structure your ownership hierarchy like a tree
- Strategies to manage circular dependencies: reference counting, using indexes instead of references
- Interior mutability (Cell, RefCell)
- Working with lifetime parameters on user-defined data types

Polymorphism in Rust

- A quick refresher on traits and generic functions
- Rust has no inheritance: what are the implications?
    - Using enums for polymorphism
    - Using traits for polymorphism
    - Using composition
    - How do I pick the most appropriate pattern?
- Working with generics
    - Generic type parameter in a function or trait object as an argument?
    - Trait bounds don’t have to refer to the generic parameter
    - Type parameters in traits: should it be a generic parameter or an associated type?
- Macros: a valuable tool to DRY up code when traits are not enough (or too complex)

Error Handling

- What is the purpose of errors? Recovery vs. reporting.
- Result vs. Option
- Designing good errors:
    - Determine the error scope.
    - Capture additional context as the error flows upwards, crossing scope boundaries.
    - Leverage the `Error` trait to keep track of the full error chain.
    - Leverage `thiserror` to reduce boilerplate when defining error types.
    - `anyhow`
- Distinguish fatal errors from recoverable errors using `Result<Result<T, RecoverableError>, FatalError>`.

Leveraging the Type System

Rust's type system is *expressive*: you can use types and traits to build abstractions that make your code harder to misuse.

In some cases, you can go as far as enforcing correctness at *compile-time*, with no runtime overhead.

Types and traits can model concepts and constraints from your business domain. With careful design, you can improve the clarity and maintainability of the entire codebase.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Additional items speaker may mention:

- Rust's type system borrows a lot of ideas from functional programming languages.

  For example, Rust's enums are known as "algebraic data types" in languages like Haskell and OCaml. You can take inspiration from learning material geared towards functional languages when looking for guidance on how to design with types. "Domain Modeling Made Functional" is a great resource on the topic, with examples written in F#.

- Despite Rust's functional roots, not all functional design patterns can be easily translated to Rust.

  For example, you must have a solid grasp on a broad selection of advanced topics to design APIs that leverage higher-order functions and higher-kinded types in Rust.

  Evaluate, on a case-by-case basis, whether a more imperative approach may be easier to implement. Consider using in-place mutation, relying on Rust's borrow-checker and type system to control what can be mutated, and where.

- The same caution should be applied to object-oriented design patterns. Rust doesn't support inheritance, and object decomposition should take into account the constraints introduced by the borrow checker.

- Mention that type-level programming can be often used to create "zero-cost abstractions", although the label can be misleading: the impact on compile times and code complexity may be significant.

This segment should take about 25 minutes. It contains:

Slide	Duration
Leveraging the Type System	5 minutes
Newtype Pattern	20 minutes

Newtype Pattern

A *newtype* is a wrapper around an existing type, often a primitive:

```
/// A unique user identifier, implemented as a newtype around `u64`.
pub struct UserId(u64);
```

Unlike type aliases, newtypes aren't interchangeable with the wrapped type:

```
fn double(n: u64) -> u64 {
    n * 2
}

double(UserId(1)); // 🛠❌
```

The Rust compiler won't let you use methods or operators defined on the underlying type either:

```
assert_ne!(UserId(1), UserId(2)); // 🛠❌
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 20 minutes.

- Students should have encountered the newtype pattern in the "Fundamentals" course, when they learned about tuple structs.
- Run the example to show students the error message from the compiler.
- Modify the example to use a typealias instead of a newtype, such as `type MessageId = u64`. The modified example should compile, thus highlighting the differences between the two approaches.
- Stress that newtypes, out of the box, have no behaviour attached to them. You need to be intentional about which methods and operators you are willing to forward from the underlying type. In our `UserId` example, it is reasonable to allow comparisons between `UserId`s, but it wouldn't make sense to allow arithmetic operations like addition or subtraction.

Semantic Confusion

When a function takes multiple arguments of the same type, call sites are unclear:

```
pub fn login(username: &str, password: &str) -> Result<(), LoginError> {
    // [...]
}

// In another part of the codebase, we swap arguments by mistake.
// Bug (best case), security vulnerability (worst case)
login(password, username);
```

The newtype pattern can prevent this class of errors at compile time:

```
pub struct Username(String);
pub struct Password(String);

pub fn login(username: &Username, password: &Password) -> Result<(), LoginError> {
    // [...]
}

login(password, username); // 🛠❌
```

▼ *Speaker Notes*

- Run both examples to show students the successful compilation for the original example, and the compiler error returned by the modified example.
- Stress the *semantic* angle. The newtype pattern should be leveraged to use distinct types for distinct concepts, thus ruling out this class of errors entirely.
- Nonetheless, note that there are legitimate scenarios where a function may take multiple arguments of the same type. In those scenarios, if correctness is of paramount importance, consider using a struct with named fields as input:

  ```
  pub struct LoginArguments<'a> {
      pub username: &'a str,
      pub password: &'a str,
  }

  // No need to check the definition of the `login` function to spot the issue.
  login(LoginArguments {
      username: password,
      password: username,
  })
  ```

  Users are forced, at the callsite, to assign values to each field, thus increasing the likelihood of spotting bugs.

Parse, Don't Validate

The newtype pattern can be leveraged to enforce *invariants*.

```
pub struct Username(String);

impl Username {
    pub fn new(username: String) -> Result<Self, InvalidUsername> {
        if username.is_empty() {
            return Err(InvalidUsername::CannotBeEmpty)
        }
        if username.len() > 32 {
            return Err(InvalidUsername::TooLong { len: username.len() })
        }
        // Other validation checks...
        Ok(Self(username))
    }

    pub fn as_str(&self) -> &str {
        &self.0
    }
}
```

▼ *Speaker Notes*

- The newtype pattern, combined with Rust's module and visibility system, can be used to *guarantee* that instances of a given type satisfy a set of invariants.

  In the example above, the raw `String` stored inside the `Username` struct can't be accessed directly from other modules or crates, since it's not marked as `pub` or `pub(in ...)`. Consumers of the `Username` type are forced to use the `new` method to create instances. In turn, `new` performs validation, thus ensuring that all instances of `Username` satisfy those checks.

- The `as_str` method allows consumers to access the raw string representation (e.g., to store it in a database). However, consumers can't modify the underlying value since `&str`, the returned type, restricts them to read-only access.

- Type-level invariants have second-order benefits.

  The input is validated once, at the boundary, and the rest of the program can rely on the invariants being upheld. We can avoid redundant validation and "defensive programming" checks throughout the program, reducing noise and improving performance.

Is It Truly Encapsulated?

You must evaluate *the entire API surface* exposed by a newtype to determine if invariants are indeed bullet-proof. It is crucial to consider all possible interactions, including trait implementations, that may allow users to bypass validation checks.

```
pub struct Username(String);

impl Username {
    pub fn new(username: String) -> Result<Self, InvalidUsername> {
        // Validation checks...
        Ok(Self(username))
    }
}

impl std::ops::DerefMut for Username { // ❗
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}
```

▼ *Speaker Notes*

- `DerefMut` allows users to get a mutable reference to the wrapped value.

  The mutable reference can be used to modify the underlying data in ways that may violate the invariants enforced by `Username::new`!

- When auditing the API surface of a newtype, you can narrow down the review scope to methods and traits that provide mutable access to the underlying data.

- Remind students of privacy boundaries.

  In particular, functions and methods defined in the same module of the newtype can access its underlying data directly. If possible, move the newtype definition to its own separate module to reduce the scope of the audit.

Welcome to Unsafe Rust

> IMPORTANT: THIS MODULE IS IN AN EARLY STAGE OF DEVELOPMENT
>
> Please do not consider this module of Comprehensive Rust to be complete. With that in mind, your feedback, comments, and especially your concerns, are very welcome.
>
> To comment on this module's development, please use the GitHub issue tracker.

The `unsafe` keyword is easy to type, but hard to master. When used appropriately, it forms a useful and indeed essential part of the Rust programming language.

By the end of this deep dive, you'll know how to work with `unsafe` code, review others' changes that include the `unsafe` keyword, and produce your own.

What you'll learn:

- What the terms undefined behavior, soundness, and safety mean
- Why the `unsafe` keyword exists in the Rust language
- How to write your own code using `unsafe` safely
- How to review `unsafe` code

Links to other sections of the course

The `unsafe` keyword has treatment in:

- *Rust Fundamentals*, the main module of Comprehensive Rust, includes a session on Unsafe Rust in its last day.
- *Rust in Chromium* discusses how to interoperate with C++. Consult that material if you are looking into FFI.
- *Bare Metal Rust* uses unsafe heavily to interact with the underlying host, among other things.

Setting Up

Local Rust installation

You should have a Rust compiler installed that supports the 2024 edition of the language, which is any version of rustc higher than 1.84.

```
$ rustc --version
rustc 1.87
```

(Optional) Create a local instance of the course

```
$ git clone --depth=1 https://github.com/google/comprehensive-rust.git
Cloning into 'comprehensive-rust'...
...
$ cd comprehensive-rust
$ cargo install-tools
...
$ cargo serve # then open http://127.0.0.1:3000/ in a browser
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

Ask everyone to confirm that everyone is able to execute `rustc` with a version older that 1.87.

For those people who do not, tell them that we'll resolve that in the break.

Motivations

We know that writing code without the guarantees that Rust provides ...

> "Use-after-free (UAF), integer overflows, and out of bounds (OOB) reads/writes comprise 90% of vulnerabilities with OOB being the most common."
>
> –— **Jeff Vander Stoep and Chong Zang**, Google. "Queue the Hardening Enhancements"

... so why is `unsafe` part of the language?

This segment should take about 20 minutes. It contains:

Slide	Duration
Motivations	1 minute
Interoperability	5 minutes
Data Structures	5 minutes
Performance	5 minutes

▼ *Speaker Notes*

This slide should take about 1 minute.

The `unsafe` keyword exists because there is no compiler technology available today that makes it obsolete. Compilers cannot verify everything.

TODO: Refactor this content into multiple slides as this slide is intended as an introduction to the motivations only, rather than to be an elaborate discussion of the whole problem.

Interoperability

Language interoperability allows you to:

- Call functions written in other languages from Rust
- Write functions in Rust that are callable from other languages

However, this requires unsafe.

```
unsafe extern "C" {
    safe fn random() -> libc::c_long;
}

fn main() {
    let a = random() as i64;
    println!("{a:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

The Rust compiler can't enforce any safety guarantees for programs that it hasn't compiled, so it delegates that responsibility to you through the unsafe keyword.

The code example we're seeing shows how to call the random function provided by libc within Rust. libc is available to scripts in the Rust Playground.

This uses Rust's *foreign function interface*.

This isn't the only style of interoperability, however it is the method that's needed if you want to work between Rust and some other language in a zero cost way. Another important strategy is message passing.

Message passing avoids unsafe, but serialization, allocation, data transfer and parsing all take energy and time.

Answers to questions

- *Where does "random" come from?*
  libc is dynamically linked to Rust programs by default, allowing our code to rely on its symbols, including `random`, being available to our program.
- *What is the "safe" keyword?*
  It allows callers to call the function without needing to wrap that call in `unsafe`. The `safe` function qualifier was introduced in the 2024 edition of Rust and can only be used within `extern` blocks. It was introduced because `unsafe` became a mandatory qualifier for `extern` blocks in that edition.
- *What is the `std::ffi::c_long` type?*
  According to the C standard, an integer that's at least 32 bits wide. On today's systems, It's an `i32` on Windows and an `i64` on Linux.

Consideration: type safety

Modify the code example to remove the need for type casting later. Discuss the potential UB - long's width is defined by the target.

```
unsafe extern "C" {
    safe fn random() -> i64;
}

fn main() {
    let a = random();
    println!("{a:?}");
}
```

Changes from the original:

```
unsafe extern "C" {
-    safe fn random() -> libc::c_long;
+    safe fn random() -> i64;
}

fn main() {
-    let a = random() as i64;
+    let a = random();
     println!("{a:?}");
}
```

It's also possible to completely ignore the intended type and create undefined behavior in multiple ways. The code below produces output most of the time, but generally results in a stack overflow. It may also produce illegal `char` values. Although `char` is represented in 4 bytes (32 bits), not all bit patterns are permitted as a `char`.

Stress that the Rust compiler will trust that the wrapper is telling the truth.

```
unsafe extern "C" {
    safe fn random() -> [char; 2];
}

fn main() {
    let a = random();
    println!("{a:?}");
}
```

Changes from the original:

```
unsafe extern "C" {
-    safe fn random() -> libc::c_long;
+    safe fn random() -> [char; 2];
}

fn main() {
-    let a = random() as i64;
-    println!("{a}");
+    let a = random();
+    println!("{a:?}");
}
```

Attempting to print a [char; 2] from randomly generated input will often produce strange

```
thread 'main' panicked at library/std/src/io/stdio.rs:1165:9:
failed printing to stdout: Bad address (os error 14)
```

```
thread 'main' has overflowed its stack
fatal runtime error: stack overflow, aborting
```

Mention that type safety is generally not a large concern in practice. Tools that produce wrappers automatically, i.e. bindgen, are excellent at reading header files and producing values of the correct type.

Consideration: Ownership and lifetime management

While libc's `random` function doesn't use pointers, many do. This creates many more possibilities for unsoundness.

- both sides might attempt to free the memory (double free)
- both sides can attempt to write to the data

For example, some C libraries expose functions that write to static buffers that are re-used between calls.

```
use std::ffi::{CStr, c_char};
use std::time::{SystemTime, UNIX_EPOCH};

unsafe extern "C" {
    /// Create a formatted time based on time `t`, including trailing newline.
    /// Read `man 3 ctime` details.
    fn ctime(t: *const libc::time_t) -> *const c_char;
}

unsafe fn format_timestamp<'a>(t: u64) -> &'a str {
    let t = t as libc::time_t;

    unsafe {
        let fmt_ptr = ctime(&t);
        CStr::from_ptr(fmt_ptr).to_str().unwrap()
    }
}

fn main() {
    let now = SystemTime::now().duration_since(UNIX_EPOCH).unwrap();

    let now = now.as_secs();
    let now_fmt = unsafe { format_timestamp(now) };
    print!("now (1): {}", now_fmt);

    let future = now + 60;
    let future_fmt = unsafe { format_timestamp(future) };
    print!("future:  {}", future_fmt);

    print!("now (2): {}", now_fmt);
}
```

Aside: Lifetimes in the `format_timestamp()` function

Neither `'a`, nor `'static`, correctly describe the lifetime of the string that's returned. Rust treats it as an immutable reference, but subsequent calls to `ctime` will overwrite the static buffer that the string occupies.

Consideration: Representation mismatch

Different programming languages have made different design decisions and this can create impedance mismatches between different domains.

Consider string handling. C++ defines `std::string`, which has an incompatible memory layout with Rust's `String` type. `String` also requires text to be encoded as UTF-8, whereas `std::string` does not. In C, text is represented by a null-terminated sequence of bytes (`char*`).

```
fn main() {
    let c_repr = b"Hello, C\0";
    let rust_repr = (b"Hello, Rust", 11);

    let c: &str = unsafe {
        let ptr = c_repr.as_ptr() as *const i8;
        std::ffi::CStr::from_ptr(ptr).to_str().unwrap()
    };
    println!("{c}");

    let rust: &str = unsafe {
        let ptr = rust_repr.0.as_ptr();
        let bytes = std::slice::from_raw_parts(ptr, rust_repr.1);
        std::str::from_utf8_unchecked(bytes)
    };
    println!("{rust}");
}
```

Data Structures

Some families of data structures are impossible to create in safe Rust.

- graphs
- bit twiddling
- self-referential types
- intrusive data structures

▼ *Speaker Notes*

This slide should take about 5 minutes.

Graphs: General-purpose graphs cannot be created as they may need to represent cycles. Cycles are impossible for the type system to reason about.

Bit twiddling: Overloading bits with multiple meanings. Examples include using the NaN bits in `f64` for some other purpose or the higher-order bits of pointers on `x86_64` platforms. This is somewhat common when writing language interpreters to keep representations within the word size the target platform.

Self-referential types are too hard for the borrow checker to verify.

Intrusive data structures: store structural metadata (like pointers to other elements) inside the elements themselves, which requires careful handling of aliasing.

Performance

> TODO: Stub for now

It's easy to think of performance as the main reason for unsafe, but high performance code makes up the minority of unsafe blocks.

Foundations

Some fundamental concepts and terms.

This segment should take about 25 minutes. It contains:

Slide	Duration
What is unsafe?	10 minutes
When is unsafe used?	2 minutes
Data structures are safe	2 minutes
Actions might not be	2 minutes
Less powerful than it seems	10 minutes

What is “unsafety”?

Unsafe Rust is a superset of Safe Rust.

Let’s create a list of things that are enabled by the `unsafe` keyword.

▼ *Speaker Notes*

This slide should take about 6 minutes.

Definitions from authoritative docs:

From the unsafe keyword’s documentation:

> Code or interfaces whose memory safety cannot be verified by the type system.
>
> …
>
> Here are the abilities Unsafe Rust has in addition to Safe Rust:
>
> - Dereference raw pointers
> - Implement unsafe traits
> - Call unsafe functions
> - Mutate statics (including external ones)
> - Access fields of unions

From the reference

> The following language level features cannot be used in the safe subset of Rust:
>
> - Dereferencing a raw pointer.
> - Reading or writing a mutable or external static variable.
> - Accessing a field of a union, other than to assign to it.
> - Calling an unsafe function (including an intrinsic or foreign function).
> - Calling a safe function marked with a target_feature from a function that does not have a target_feature attribute enabling the same features (see attributes.codegen.target_feature.safety-restrictions).
> - Implementing an unsafe trait.
> - Declaring an extern block.
> - Applying an unsafe attribute to an item.

Group exercise

> You may have a group of learners who are not familiar with each other yet. This is a way for you to gather some data about their confidence levels and the psychological safety that they’re feeling.

Part 1: Informal definition

> Use this to gauge the confidence level of the group. If they are uncertain, then tailor the next section to be more directed.

Ask the class: **By raising your hand, indicate if you would feel comfortable defining unsafe?**

If anyone's feeling confident, allow them to try to explain.

Part 2: Evidence gathering

Ask the class to spend 3-5 minutes.

- Find a use of the unsafe keyword. What contract/invariant/pre-condition is being established or satisfied?
- Write down terms that need to be defined (unsafe, memory safety, soundness, undefined behavior)

Part 3: Write a working definition

Part 4: Remarks

Mention that we'll be reviewing our definition at the end of the day.

Note: Avoid detailed discussion about precise semantics of memory safety

It's possible that the group will slide into a discussion about the precise semantics of what memory safety actually is and how define pointer validity. This isn't a productive line of discussion. It can undermine confidence in less experienced learners.

Perhaps refer people who wish to discuss this to the discussion within the official documentation for pointer types (excerpt below) as a place for further research.

> Many functions in this module take raw pointers as arguments and read from or write to them. For this to be safe, these pointers must be *valid* for the given access.
>
> ...
>
> The precise rules for validity are not determined yet.

When is unsafe used?

The unsafe keyword indicates that the programmer is responsible for upholding Rust's safety guarantees.

The keyword has two roles:

- define pre-conditions that must be satisfied
- assert to the compiler (= promise) that those defined pre-conditions are satisfied

Further references

- The unsafe keyword chapter of the Rust Reference

▼ *Speaker Notes*

This slide should take about 2 minutes.

Places where pre-conditions can be defined (Role 1)

- unsafe functions (`unsafe fn foo() { ... }`). Example: `get_unchecked` method on slices, which requires callers to verify that the index is in-bounds.
- unsafe traits (`unsafe trait`). Examples: `Send` and `Sync` marker traits in the standard library.

Places where pre-conditions must be satisfied (Role 2)

- unsafe blocks (`unafe { ... }`)
- implementing unsafe traits (`unsafe impl`)
- access external items (`unsafe extern`)
- adding unsafe attributes o an item. Examples: `export_name` , `link_section` and `no_mangle` . Usage: `#[unsafe(no_mangle)]`

Data structures are safe ...

Data structures are inert. They cannot do any harm by themselves.

Safe Rust code can create raw pointers:

```
fn main() {
    let n: i64 = 12345;
    let safe = &raw const n;
    println!("{safe:p}");
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

Consider a raw pointer to an integer, i.e., the value `safe` is the raw pointer type `*const i64`. Raw pointers can be out-of-bounds, misaligned, or be null. But the unsafe keyword is not required when creating them.

… but actions on them might not be

```
fn main() {
    let n: i64 = 12345;
    let safe = &n as *const _;
    println!("{safe:p}");
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

Modify the example to de-reference `safe` without an `unsafe` block.

Less powerful than it seems

The `unsafe` keyword does not allow you to break Rust.

```
use std::mem::transmute;

let orig = b"RUST";
let n: i32 = unsafe { transmute(orig) };

println!("{n}")
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

Suggested outline

- Request that someone explains what `std::mem::transmute` does
- Discuss why it doesn't compile
- Fix the code

Expected compiler output

```
   Compiling playground v0.0.1 (/playground)
error[E0512]: cannot transmute between types of different sizes, or dependently-sized
types
 --> src/main.rs:5:27
  |
5 |     let n: i32 = unsafe { transmute(orig) };
  |                           ^^^^^^^^^
  |
  = note: source type: `&[u8; 4]` (64 bits)
  = note: target type: `i32` (32 bits)
```

Suggested change

```
- let n: i32 = unsafe { transmute(orig) };
+ let n: i64 = unsafe { transmute(orig) };
```

Notes on less familiar Rust

- the `b` prefix on a string literal marks it as byte slice (`&[u8]`) rather than a string slice (`&str`)

Thanks!

Thank you for taking Comprehensive Rust 🦀! We hope you enjoyed it and that it was useful.

We've had a lot of fun putting the course together. The course is not perfect, so if you spotted any mistakes or have ideas for improvements, please get in contact with us on GitHub. We would love to hear from you.

▼ *Speaker Notes*

- Thank you for reading the speaker notes! We hope they have been useful. If you find pages without notes, please send us a PR and link it to issue #1083. We are also very grateful for fixes and improvements to the existing notes.

Glossary

The following is a glossary which aims to give a short definition of many Rust terms. For translations, this also serves to connect the term back to the English original.

allocate:
Dynamic memory allocation on the heap.
argument:
Information that is passed into a function or method.
associated type:
A type associated with a specific trait. Useful for defining the relationship between types.
Bare-metal Rust:
Low-level Rust development, often deployed to a system without an operating system. See Bare-metal Rust.
block:
See Blocks and *scope*.
borrow:
See Borrowing.
borrow checker:
The part of the Rust compiler which checks that all borrows are valid.
brace:
`{` and `}`. Also called *curly brace*, they delimit *blocks*.
build:
The process of converting source code into executable code or a usable program.
call:
To invoke or execute a function or method.
channel:
Used to safely pass messages between threads.
Comprehensive Rust 🦀:
The courses here are jointly called Comprehensive Rust 🦀.
concurrency:
The execution of multiple tasks or processes at the same time.
Concurrency in Rust:
See Concurrency in Rust.
constant:
A value that does not change during the execution of a program.
control flow:
The order in which the individual statements or instructions are executed in a program.
crash:
An unexpected and unhandled failure or termination of a program.
enumeration:
A data type that holds one of several named constants, possibly with an associated tuple or struct.
error:
An unexpected condition or result that deviates from the expected behavior.
error handling:
The process of managing and responding to errors that occur during program execution.
exercise:
A task or problem designed to practice and test programming skills.
function:
A reusable block of code that performs a specific task.
garbage collector:
A mechanism that automatically frees up memory occupied by objects that are no longer in

generics:
A feature that allows writing code with placeholders for types, enabling code reuse with different data types.

immutable:
Unable to be changed after creation.

integration test:
A type of test that verifies the interactions between different parts or components of a system.

keyword:
A reserved word in a programming language that has a specific meaning and cannot be used as an identifier.

library:
A collection of precompiled routines or code that can be used by programs.

macro:
Rust macros can be recognized by a `!` in the name. Macros are used when normal functions are not enough. A typical example is `format!`, which takes a variable number of arguments, which isn't supported by Rust functions.

`main` function:
Rust programs start executing with the `main` function.

match:
A control flow construct in Rust that allows for pattern matching on the value of an expression.

memory leak:
A situation where a program fails to release memory that is no longer needed, leading to a gradual increase in memory usage.

method:
A function associated with an object or a type in Rust.

module:
A namespace that contains definitions, such as functions, types, or traits, to organize code in Rust.

move:
The transfer of ownership of a value from one variable to another in Rust.

mutable:
A property in Rust that allows variables to be modified after they have been declared.

ownership:
The concept in Rust that defines which part of the code is responsible for managing the memory associated with a value.

panic:
An unrecoverable error condition in Rust that results in the termination of the program.

parameter:
A value that is passed into a function or method when it is called.

pattern:
A combination of values, literals, or structures that can be matched against an expression in Rust.

payload:
The data or information carried by a message, event, or data structure.

program:
A set of instructions that a computer can execute to perform a specific task or solve a particular problem.

programming language:
A formal system used to communicate instructions to a computer, such as Rust.

receiver:
The first parameter in a Rust method that represents the instance on which the method is called.

reference counting:
A memory management technique in which the number of references to an object is tracked,

return:
A keyword in Rust used to indicate the value to be returned from a function.
Rust:
A systems programming language that focuses on safety, performance, and concurrency.
Rust Fundamentals:
Days 1 to 4 of this course.
Rust in Android:
See Rust in Android.
Rust in Chromium:
See Rust in Chromium.
safe:
Refers to code that adheres to Rust's ownership and borrowing rules, preventing memory-related errors.
scope:
The region of a program where a variable is valid and can be used.
standard library:
A collection of modules providing essential functionality in Rust.
static:
A keyword in Rust used to define static variables or items with a `'static` lifetime.
string:
A data type storing textual data. See Strings for more.
struct:
A composite data type in Rust that groups together variables of different types under a single name.
test:
A Rust module containing functions that test the correctness of other functions.
thread:
A separate sequence of execution in a program, allowing concurrent execution.
thread safety:
The property of a program that ensures correct behavior in a multithreaded environment.
trait:
A collection of methods defined for an unknown type, providing a way to achieve polymorphism in Rust.
trait bound:
An abstraction where you can require types to implement some traits of your interest.
tuple:
A composite data type that contains variables of different types. Tuple fields have no names, and are accessed by their ordinal numbers.
type:
A classification that specifies which operations can be performed on values of a particular kind in Rust.
type inference:
The ability of the Rust compiler to deduce the type of a variable or expression.
undefined behavior:
Actions or conditions in Rust that have no specified result, often leading to unpredictable program behavior.
union:
A data type that can hold values of different types but only one at a time.
unit test:
Rust comes with built-in support for running small unit tests and larger integration tests. See Unit Tests.
unit type:
Type that holds no data, written as a tuple with no members.
unsafe:
The subset of Rust which allows you to trigger *undefined behavior*. See Unsafe Rust.

variable:

A memory location storing data. Variables are valid in a *scope*.

Other Rust Resources

The Rust community has created a wealth of high-quality and free resources online.

Official Documentation

The Rust project hosts many resources. These cover Rust in general:

The Rust Programming Language: the canonical free book about Rust. Covers the language in detail and includes a few projects for people to build.
Rust By Example: covers the Rust syntax via a series of examples which showcase different constructs. Sometimes includes small exercises where you are asked to expand on the code in the examples.
Rust Standard Library: full documentation of the standard library for Rust.
The Rust Reference: an incomplete book which describes the Rust grammar and memory model.
Rust API Guidelines: recommendations on how to design APIs.

More specialized guides hosted on the official Rust site:

The Rustonomicon: covers unsafe Rust, including working with raw pointers and interfacing with other languages (FFI).
Asynchronous Programming in Rust: covers the new asynchronous programming model which was introduced after the Rust Book was written.
The Embedded Rust Book: an introduction to using Rust on embedded devices without an operating system.

Unofficial Learning Material

A small selection of other guides and tutorial for Rust:

Learn Rust the Dangerous Way: covers Rust from the perspective of low-level C programmers.
Rust for Embedded C Programmers: covers Rust from the perspective of developers who write firmware in C.
Rust for professionals: covers the syntax of Rust using side-by-side comparisons with other languages such as C, C++, Java, JavaScript, and Python.
Rust on Exercism: 100+ exercises to help you learn Rust.
Ferrous Teaching Material: a series of small presentations covering both basic and advanced part of the Rust language. Other topics such as WebAssembly, and async/await are also covered.
Advanced testing for Rust applications: a self-paced workshop that goes beyond Rust's built- in testing framework. It covers `googletest`, snapshot testing, mocking as well as how to write your own custom test harness.
Beginner's Series to Rust and Take your first steps with Rust: two Rust guides aimed at new developers. The first is a set of 35 videos and the second is a set of 11 modules which covers Rust syntax and basic constructs.
Learn Rust With Entirely Too Many Linked Lists: in-depth exploration of Rust's memory management rules, through implementing a few different types of list structures.

Please see the Little Book of Rust Books for even more Rust books.

Credits

The material here builds on top of the many great sources of Rust documentation. See the page on other resources for a full list of useful resources.

The material of Comprehensive Rust is licensed under the terms of the Apache 2.0 license, please see `LICENSE` for details.

Rust by Example

Some examples and exercises have been copied and adapted from Rust by Example. Please see the `third_party/rust-by-example/` directory for details, including the license terms.

Rust on Exercism

Some exercises have been copied and adapted from Rust on Exercism. Please see the `third_party/rust-on-exercism/` directory for details, including the license terms.

CXX

The Interoperability with C++ section uses an image from CXX. Please see the `third_party/cxx/` directory for details, including the license terms.