▼ *Speaker Notes*

- Note the use of `&raw const` / `&raw mut` to get pointers to individual fields without creating an intermediate reference, which would be unsound.
- The example isn't included in the slides because it is very similar to the `safe-mmio` example which comes next. You can run it in QEMU with `make qemu` under `src/bare-metal/aps/examples` if you need to.

safe-mmio

The `safe-mmio` crate provides types to wrap registers that can be read or written safely.

	Can’t read	Read has no side-effects	Read has side-effects
Can’t write		`ReadPure`	`ReadOnly`
Can write	`WriteOnly`	`ReadPureWrite`	`ReadWrite`

```rust
use safe_mmio::fields::{ReadPure, ReadPureWrite, ReadWrite, WriteOnly};

#[repr(C, align(4))]
pub struct Registers {
    dr: ReadWrite<u16>,
    _reserved0: [u8; 2],
    rsr: ReadPure<ReceiveStatus>,
    _reserved1: [u8; 19],
    fr: ReadPure<Flags>,
    _reserved2: [u8; 6],
    ilpr: ReadPureWrite<u8>,
    _reserved3: [u8; 3],
    ibrd: ReadPureWrite<u16>,
    _reserved4: [u8; 2],
    fbrd: ReadPureWrite<u8>,
    _reserved5: [u8; 3],
    lcr_h: ReadPureWrite<u8>,
    _reserved6: [u8; 3],
    cr: ReadPureWrite<u16>,
    _reserved7: [u8; 3],
    ifls: ReadPureWrite<u8>,
    _reserved8: [u8; 3],
    imsc: ReadPureWrite<u16>,
    _reserved9: [u8; 2],
    ris: ReadPure<u16>,
    _reserved10: [u8; 2],
    mis: ReadPure<u16>,
    _reserved11: [u8; 2],
    icr: WriteOnly<u16>,
    _reserved12: [u8; 2],
    dmacr: ReadPureWrite<u8>,
    _reserved13: [u8; 3],
}
```

- Reading `dr` has a side effect: it pops a byte from the receive FIFO.
- Reading `rsr` (and other registers) has no side-effects. It is a ‘pure’ read.

▼ *Speaker Notes*

- There are a number of different crates providing safe abstractions around MMIO operations; we recommend the `safe-mmio` crate.
- The difference between `ReadPure` or `ReadOnly` (and likewise between `ReadPureWrite` and `ReadWrite`) is whether reading a register can have side-effects that change the state of the device, e.g., reading the data register pops a byte from the receive FIFO. `ReadPure` means that reads have no side-effects, they are purely reading data.

Driver

Now let's use the new `Registers` struct in our driver.

```rust
use safe_mmio::{UniqueMmioPointer, field, field_shared};

/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart<'a> {
    registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Uart<'a> {
    /// Constructs a new instance of the UART driver for a PL011 device wit
    /// given set of registers.
    pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
        Self { registers }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&mut self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // Write to the TX buffer.
        field!(self.registers, dr).write(byte.into());

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
    /// received.
    pub fn read_byte(&mut self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            let data = field!(self.registers, dr).read();
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        field_shared!(self.registers, fr).read()
    }
}
```

▼ *Speaker Notes*

for the given lifetime, so it can provide safe methods to read and write fields.

- Note that `Uart::new` is now safe; `UniqueMmioPointer::new` is unsafe instead.
- These MMIO accesses are generally a wrapper around `read_volatile` and `write_volatile`, though on aarch64 they are instead implemented in assembly to work around a bug where the compiler can emit instructions that prevent MMIO virtualization.
- The `field!` and `field_shared!` macros internally use `&raw mut` and `&raw const` to get pointers to individual fields without creating an intermediate reference, which would be unsound.
- `field!` needs a mutable reference to a `UniqueMmioPointer`, and returns a `UniqueMmioPointer` that allows reads with side effects and writes.
- `field_shared!` works with a shared reference to either a `UniqueMmioPointer` or a `SharedMmioPointer`. It returns a `SharedMmioPointer` that only allows pure reads.

Using It

Let's write a small program using our driver to write to the serial console, and echo incoming bytes.

```rust
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod pl011;

use crate::pl011::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::error;
use safe_mmio::UniqueMmioPointer;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<pl011::Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // nothing else accesses that address range.
    let mut uart = Uart::new(unsafe { UniqueMmioPointer::new(PL011_BASE_ADD

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    loop {
        if let Some(byte) = uart.read_byte() {
            uart.write_byte(byte);
            match byte {
                b'\r' => uart.write_byte(b'\n'),
                b'q' => break,
                _ => continue,
            }
        }
    }

    writeln!(uart, "\n\nBye!").unwrap();
    system_off::<Hvc>().unwrap();
}
```

▼ Speaker Notes

- Run the example in QEMU with `make qemu_safemmio` under `src/bare-metal/aps/examples`.

Logging

It would be nice to be able to use the logging macros from the `log` crate. We can do this by implementing the `Log` trait.

```rust
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart<'static>>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(
    uart: Uart<'static>,
    max_level: LevelFilter,
) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

▼ *Speaker Notes*

- The first unwrap in `log` will succeed because we initialize `LOGGER` before calling

Using it

We need to initialise the logger before we use it.

```rust
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info};
use safe_mmio::UniqueMmioPointer;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<pl011::Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})");

    assert_eq!(x1, 42);

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

▼ *Speaker Notes*

- Note that our panic handler can now log details of panics.
- Run the example in QEMU with `make qemu_logger` under `src/bare-`

Exceptions

AArch64 defines an exception vector table with 16 entries, for 4 types of exceptions (synchronous, IRQ, FIQ, SError) from 4 states (current EL with SP0, current EL with SPx, lower EL using AArch64, lower EL using AArch32). We implement this in assembly to save volatile registers to the stack before calling into Rust code:

```rust
use log::error;
use smccc::Hvc;
use smccc::psci::system_off;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    error!("irq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
```

```
}
```

▼ *Speaker Notes*

- EL is exception level; all our examples this afternoon run in EL1.
- For simplicity we aren’t distinguishing between SP0 and SPx for the current EL exceptions, or between AArch32 and AArch64 for the lower EL exceptions.
- For this example we just log the exception and power down, as we don’t expect any of them to actually happen.
- We can think of exception handlers and our main execution context more or less like different threads. `Send` and `Sync` will control what we can share between them, just like with threads. For example, if we want to share some value between exception handlers and the rest of the program, and it’s `Send` but not `Sync`, then we’ll need to wrap it in something like a `Mutex` and put it in a static.

aarch64-rt

The `aarch64-rt` crate provides the assembly entry point and exception vector that we implemented before. We just need to mark our main function with the `entry!` macro.

It also provides the `initial_pagetable!` macro to let us define an initial static pagetable in Rust, rather than in assembly code like we did before.

We can also use the UART driver from the `arm-pl011-uart` crate rather than writing our own.

```rust
#![no_main]
#![no_std]

mod exceptions;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::fmt::Write;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // nothing else accesses that address range.
    let mut uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADD

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    system_off::<Hvc>().unwrap();
    panic!("system_off returned");
}

}
```

▼ *Speaker Notes*

- Run the example in QEMU with `make qemu_rt` under `src/bare-metal/aps/examples`.

Other projects

- oreboot
  - “coreboot without the C”.
  - Supports x86, aarch64 and RISC-V.
  - Relies on LinuxBoot rather than having many drivers itself.
- Rust RaspberryPi OS tutorial
  - Initialization, UART driver, simple bootloader, JTAG, exception levels, exception handling, page tables.
  - Some caveats around cache maintenance and initialization in Rust, not necessarily a good example to copy for production code.
- `cargo-call-stack`
  - Static analysis to determine maximum stack usage.

▼ *Speaker Notes*

- The RaspberryPi OS tutorial runs Rust code before the MMU and caches are enabled. This will read and write memory (e.g. the stack). However, this has the problems mentioned at the beginning of this session regarding unaligned access and cache coherency.

Useful crates

We'll look at a few crates that solve some common problems in bare-metal programming.

zerocopy

The `zerocopy` crate (from Fuchsia) provides traits and macros for safely converting between byte sequences and other types.

```rust
use zerocopy::{Immutable, IntoBytes};

#[repr(u32)]
#[derive(Debug, Default, Immutable, IntoBytes)]
enum RequestType {
    #[default]
    In = 0,
    Out = 1,
    Flush = 4,
}

#[repr(C)]
#[derive(Debug, Default, Immutable, IntoBytes)]
struct VirtioBlockRequest {
    request_type: RequestType,
    reserved: u32,
    sector: u64,
}

fn main() {
    let request = VirtioBlockRequest {
        request_type: RequestType::Flush,
        sector: 42,
        ..Default::default()
    };

    assert_eq!(
        request.as_bytes(),
        &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
    );
}
```

This is not suitable for MMIO (as it doesn't use volatile reads and writes), but can be useful for working with structures shared with hardware e.g. by DMA, or sent over some external interface.

▼ *Speaker Notes*

- `FromBytes` can be implemented for types for which any byte pattern is valid, and so can safely be converted from an untrusted sequence of bytes.
- Attempting to derive `FromBytes` for these types would fail, because `RequestType` doesn't use all possible u32 values as discriminants, so not all byte patterns are valid.
- `zerocopy::byteorder` has types for byte-order aware numeric primitives.

- Run the example with `cargo run` under `src/bare-metal/useful-crates/zerocopy-example/`. (It won't run in the Playground because of the crate dependency.)

aarch64-paging

The `aarch64-paging` crate lets you create page tables according to the AArch64 Virtual Memory System Architecture.

```rust
use aarch64_paging::{
    idmap::IdMap,
    paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// Create a new page table with identity mapping.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// Map a 2 MiB region of memory as read-only.
idmap.map_range(
    &MemoryRegion::new(0x80200000, 0x80400000),
    Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// Set `TTBR0_EL1` to activate the page table.
idmap.activate();
```

▼ *Speaker Notes*

- This is used in Android for the Protected VM Firmware.
- There’s no easy way to run this example by itself, as it needs to run on real hardware or under QEMU.

buddy_system_allocator

`buddy_system_allocator` is a crate that implements a basic buddy system allocator. It can be used both to implement `GlobalAlloc` (using `LockedHeap`) so you can use the standard `alloc` crate (as we saw before), or for allocating other address space (using `FrameAllocator`). For example, we might want to allocate MMIO space for PCI BARs:

```rust
use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
    let mut allocator = FrameAllocator::<32>::new();
    allocator.add_frame(0x200_0000, 0x400_0000);

    let layout = Layout::from_size_align(0x100, 0x100).unwrap();
    let bar = allocator
        .alloc_aligned(layout)
        .expect("Failed to allocate 0x100 byte MMIO region");
    println!("Allocated 0x100 byte MMIO region at {:#x}", bar);
}
```

▼ *Speaker Notes*

- PCI BARs always have alignment equal to their size.
- Run the example with `cargo run` under `src/bare-metal/useful-crates/allocator-example/`. (It won't run in the Playground because of the crate dependency.)

tinyvec

Sometimes you want something that can be resized like a `Vec`, but without heap allocation. `tinyvec` provides this: a vector backed by an array or slice, which could be statically allocated or on the stack, that keeps track of how many elements are used and panics if you try to use more than are allocated.

```rust
use tinyvec::{ArrayVec, array_vec};

fn main() {
    let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
    println!("{numbers:?}");
    numbers.push(7);
    println!("{numbers:?}");
    numbers.remove(1);
    println!("{numbers:?}");
}
```

▼ *Speaker Notes*

- `tinyvec` requires that the element type implement `Default` for initialization.
- The Rust Playground includes `tinyvec`, so this example will run fine inline.

spin

`std::sync::Mutex` and the other synchronisation primitives from `std::sync` are not available in `core` or `alloc`. How can we manage synchronisation or interior mutability, such as for sharing state between different CPUs?

The `spin` crate provides spinlock-based equivalents of many of these primitives.

```rust
use spin::mutex::SpinMutex;

static COUNTER: SpinMutex<u32> = SpinMutex::new(0);

fn main() {
    dbg!(COUNTER.lock());
    *COUNTER.lock() += 2;
    dbg!(COUNTER.lock());
}
```

▼ *Speaker Notes*

- Be careful to avoid deadlock if you take locks in interrupt handlers.
- `spin` also has a ticket lock mutex implementation; equivalents of `RwLock`, `Barrier` and `Once` from `std::sync`; and `Lazy` for lazy initialization.
- The `once_cell` crate also has some useful types for late initialization with a slightly different approach to `spin::once::Once`.
- The Rust Playground includes `spin`, so this example will run fine inline.

Bare-Metal on Android

To build a bare-metal Rust binary in AOSP, you need to use a `rust_ffi_static` Soong rule to build your Rust code, then a `cc_binary` with a linker script to produce the binary itself, and then a `raw_binary` to convert the ELF to a raw binary ready to be run.

```
rust_ffi_static {
    name: "libvmbase_example",
    defaults: ["vmbase_ffi_defaults"],
    crate_name: "vmbase_example",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libvmbase",
    ],
}

cc_binary {
    name: "vmbase_example",
    defaults: ["vmbase_elf_defaults"],
    srcs: [
        "idmap.S",
    ],
    static_libs: [
        "libvmbase_example",
    ],
    linker_scripts: [
        "image.ld",
        ":vmbase_sections",
    ],
}

raw_binary {
    name: "vmbase_example_bin",
    stem: "vmbase_example.bin",
    src: ":vmbase_example",
    enabled: false,
    target: {
        android_arm64: {
            enabled: true,
        },
    },
}
```

vmbase

For VMs running under crosvm on aarch64, the vmbase library provides a linker script and useful defaults for the build rules, along with an entry point, UART console logging and more.

```
#![no_main]
#![no_std]

use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
    println!("Hello world");
}
```

▼ *Speaker Notes*

- The `main!` macro marks your main function, to be called from the `vmbase` entry point.
- The `vmbase` entry point handles console initialisation, and issues a PSCI_SYSTEM_OFF to shutdown the VM if your main function returns.

Exercises

We will write a driver for the PL031 real-time clock device.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

RTC driver

The QEMU aarch64 virt machine has a PL031 real-time clock at 0x9010000. For this exercise, you should write a driver for it.

1. Use it to print the current time to the serial console. You can use the `chrono` crate for date/time formatting.
2. Use the match register and raw interrupt status to busy-wait until a given time, e.g. 3 seconds in the future. (Call `core::hint::spin_loop` inside the loop.)
3. *Extension if you have time:* Enable and handle the interrupt generated by the RTC match. You can use the driver provided in the `arm-gic` crate to configure the Arm Generic Interrupt Controller.
   - Use the RTC interrupt, which is wired to the GIC as `IntId::spi(2)`.
   - Once the interrupt is enabled, you can put the core to sleep via `arm_gic::wfi()`, which will cause the core to sleep until it receives an interrupt.

Download the exercise template and look in the `rtc` directory for the following files.

src/main.rs:

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_gic::gicv3::{GicCpuInterface, GicV3};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: NonNull<Gicd> = NonNull::new(0x800_0000 as
_).unwrap();
const GICR_BASE_ADDRESS: NonNull<GicrSgi> = NonNull::new(0x80A_0000 as
_).unwrap();

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic = unsafe {
        GicV3::new(
            UniqueMmioPointer::new(GICD_BASE_ADDRESS),
            GICR_BASE_ADDRESS,
            1,
            false,
        )
    };
    gic.setup(0);

    // TODO: Create instance of RTC driver and print current time.

    // TODO: Wait for 3 seconds.

    system_off::<Hvc>().unwrap();
    panic!("system_off returned");
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

src/exceptions.rs (you should only need to change this for the 3rd part of the exercise):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_gic::gicv3::{GicCpuInterface, InterruptGroup};
use log::{error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    trace!("irq_current");
    let intid =
        GicCpuInterface::get_and_acknowledge_interrupt(InterruptGroup::Group1)
            .expect("No pending interrupt");
    info!("IRQ {intid:?}");
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
```

```
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

src/logger.rs (you shouldn't need to change this):

```rust
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_pl011_uart::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart<'static>>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(
    uart: Uart<'static>,
    max_level: LevelFilter,
) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

```toml
[workspace]

[package]
name = "rtc"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
aarch64-paging = { version = "0.10.0", default-features = false }
aarch64-rt = "0.2.2"
arm-gic = "0.7.1"
arm-pl011-uart = "0.4.0"
bitflags = "2.10.0"
chrono = { version = "0.4.42", default-features = false }
log = "0.4.28"
safe-mmio = "0.2.5"
smccc = "0.2.2"
spin = "0.10.0"
zerocopy = "0.8.27"
```

build.rs (you shouldn't need to change this):

```rust
// Copyright 2025 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

fn main() {
    println!("cargo:rustc-link-arg=-Timage.ld");
    println!("cargo:rustc-link-arg=-Tmemory.ld");
    println!("cargo:rerun-if-changed=memory.ld");
}
```

memory.ld (you shouldn't need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

MEMORY
{
        image : ORIGIN = 0x40080000, LENGTH = 2M
}
```

Makefile (you shouldn't need to change this):

```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

.PHONY: build qemu_minimal qemu qemu_logger

all: rtc.bin

build:
	cargo build

rtc.bin: build
	cargo objcopy -- -O binary $@

qemu: rtc.bin
	qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio
-display none -kernel $< -s

clean:
	cargo clean
```

.cargo/config.toml (you shouldn't need to change this):

```toml
[build]
target = "aarch64-unknown-none"
```

Run the code in QEMU with `make qemu`.

Bare Metal Rust Afternoon

RTC driver

(back to exercise)

main.rs:

```rust
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::{IntId, Trigger, irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_gic::gicv3::{GicCpuInterface, GicV3};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: NonNull<Gicd> = NonNull::new(0x800_0000 as
_).unwrap();
const GICR_BASE_ADDRESS: NonNull<GicrSgi> = NonNull::new(0x80A_0000 as
_).unwrap();

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
```

```
const PL031_BASE_ADDRESS: NonNull<pl031::Registers> =
    NonNull::new(0x901_0000 as _).unwrap();
/// The IRQ used by the PL031 RTC.
const PL031_IRQ: IntId = IntId::spi(2);

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS))
};
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic = unsafe {
        GicV3::new(
            UniqueMmioPointer::new(GICD_BASE_ADDRESS),
            GICR_BASE_ADDRESS,
            1,
            false,
        )
    };
    gic.setup(0);

    // SAFETY: `PL031_BASE_ADDRESS` is the base address of a PL031 device, and
    // nothing else accesses that address range.
    let mut rtc = unsafe {
Rtc::new(UniqueMmioPointer::new(PL031_BASE_ADDRESS)) };
    let timestamp = rtc.read();
    let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
    info!("RTC: {time}");

    GicCpuInterface::set_priority_mask(0xff);
    gic.set_interrupt_priority(PL031_IRQ, None, 0x80).unwrap();
    gic.set_trigger(PL031_IRQ, None, Trigger::Level).unwrap();
    irq_enable();
    gic.enable_interrupt(PL031_IRQ, None, true).unwrap();

    // Wait for 3 seconds, without interrupts.
    let target = timestamp + 3;
    rtc.set_match(target);
    info!("Waiting for {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.matched() {
        spin_loop();
    }
```

```
        );
        info!("Finished waiting");

        // Wait another 3 seconds for an interrupt.
        let target = timestamp + 6;
        info!("Waiting for {}", Utc.timestamp_opt(target.into(), 0).unwrap());
        rtc.set_match(target);
        rtc.clear_interrupt();
        rtc.enable_interrupt(true);
        trace!(
            "matched={}, interrupt_pending={}",
            rtc.matched(),
            rtc.interrupt_pending()
        );
        while !rtc.interrupt_pending() {
            wfi();
        }
        trace!(
            "matched={}, interrupt_pending={}",
            rtc.matched(),
            rtc.interrupt_pending()
        );
        info!("Finished waiting");

        system_off::<Hvc>().unwrap();
        panic!("system_off returned");
    }

    #[panic_handler]
    fn panic(info: &PanicInfo) -> ! {
        error!("{info}");
        system_off::<Hvc>().unwrap();
        loop {}
    }
```

pl031.rs:

```
#[repr(C, align(4))]
pub struct Registers {
    /// Data register
    dr: ReadPure<u32>,
    /// Match register
    mr: ReadPureWrite<u32>,
    /// Load register
    lr: ReadPureWrite<u32>,
    /// Control register
    cr: ReadPureWrite<u8>,
    _reserved0: [u8; 3],
    /// Interrupt Mask Set or Clear register
    imsc: ReadPureWrite<u8>,
    _reserved1: [u8; 3],
    /// Raw Interrupt Status
    ris: ReadPure<u8>,
    _reserved2: [u8; 3],
    /// Masked Interrupt Status
    mis: ReadPure<u8>,
    _reserved3: [u8; 3],
    /// Interrupt Clear Register
    icr: WriteOnly<u8>,
    _reserved4: [u8; 3],
}

/// Driver for a PL031 real-time clock.
#[derive(Debug)]
pub struct Rtc<'a> {
    registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Rtc<'a> {
    /// Constructs a new instance of the RTC driver for a PL031 device with
the
    /// given set of registers.
    pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
        Self { registers }
    }

    /// Reads the current RTC value.
    pub fn read(&self) -> u32 {
        field_shared!(self.registers, dr).read()
    }

    /// Writes a match value. When the RTC value matches this then an
interrupt
    /// will be generated (if it is enabled).
    pub fn set_match(&mut self, value: u32) {
        field!(self.registers, mr).write(value);
    }

    /// Returns whether the match register matches the RTC value, whether or
not
    /// the interrupt is enabled.
```

```
        }

        /// Returns whether there is currently an interrupt pending.
        ///
        /// This should be true if and only if `matched` returns true and the
        /// interrupt is masked.
        pub fn interrupt_pending(&self) -> bool {
            let mis = field_shared!(self.registers, mis).read();
            (mis & 0x01) != 0
        }

        /// Sets or clears the interrupt mask.
        ///
        /// When the mask is true the interrupt is enabled; when it is false the
        /// interrupt is disabled.
        pub fn enable_interrupt(&mut self, mask: bool) {
            let imsc = if mask { 0x01 } else { 0x00 };
            field!(self.registers, imsc).write(imsc);
        }

        /// Clears a pending interrupt, if any.
        pub fn clear_interrupt(&mut self) {
            field!(self.registers, icr).write(0x01);
        }
    }
```

Welcome to Concurrency in Rust

Rust has full support for concurrency using OS threads with mutexes and channels.

The Rust type system plays an important role in making many concurrency bugs compile time bugs. This is often referred to as *fearless concurrency* since you can rely on the compiler to ensure correctness at runtime.

Schedule

Including 10 minute breaks, this session should take about 3 hours and 20 minutes. It contains:

Segment	Duration
Threads	30 minutes
Channels	20 minutes
Send and Sync	15 minutes
Shared State	30 minutes
Exercises	1 hour and 10 minutes

▼ *Speaker Notes*

- Rust lets us access OS concurrency toolkit: threads, sync. primitives, etc.
- The type system gives us safety for concurrency without any special features.
- The same tools that help with “concurrent” access in a single thread (e.g., a called function that might mutate an argument or save references to it to read later) save us from multi-threading issues.

Threads

This segment should take about 30 minutes. It contains:

Slide	Duration
Plain Threads	15 minutes
Scoped Threads	15 minutes

Plain Threads

Rust threads work similarly to threads in other languages:

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

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("Count in thread: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

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

- Spawning new threads does not automatically delay program termination at the end of `main`.
- Thread panics are independent of each other.
  - Panics can carry a payload, which can be unpacked with `Any::downcast_ref`.

▼ *Speaker Notes*

This slide should take about 15 minutes.

- Run the example.
  - 5ms timing is loose enough that main and spawned threads stay mostly in lockstep.
  - Notice that the program ends before the spawned thread reaches 10!
  - This is because `main` ends the program and spawned threads do not make it persist.
    - Compare to `pthreads`/C++ `std::thread` / `boost::thread` if desired.
- How do we wait around for the spawned thread to complete?
- `thread::spawn` returns a `JoinHandle`. Look at the docs.
  - `JoinHandle` has a `.join()` method that blocks.
- Use `let handle = thread::spawn(...)` and later `handle.join()` to wait for the thread to finish and have the program count all the way to 10.

- Look at docs again:
  - `thread::spawn`'s closure returns `T`
  - `JoinHandle` `.join()` returns `thread::Result<T>`
- Use the `Result` return value from `handle.join()` to get access to the returned value.
- Ok, what about the other case?
  - Trigger a panic in the thread. Note that this doesn't panic `main`.
  - Access the panic payload. This is a good time to talk about `Any`.
- Now we can return values from threads! What about taking inputs?
  - Capture something by reference in the thread closure.
  - An error message indicates we must move it.
  - Move it in, see we can compute and then return a derived value.
- If we want to borrow?
  - Main kills child threads when it returns, but another function would just return and leave them running.
  - That would be stack use-after-return, which violates memory safety!
  - How do we avoid this? See next slide.

Scoped Threads

Normal threads cannot borrow from their environment:

```rust
use std::thread;

fn foo() {
    let s = String::from("Hello");
    thread::spawn(|| {
        dbg!(s.len());
    });
}

fn main() {
    foo();
}
```

However, you can use a scoped thread for this:

```rust
use std::thread;

fn foo() {
    let s = String::from("Hello");
    thread::scope(|scope| {
        scope.spawn(|| {
            dbg!(s.len());
        });
    });
}

fn main() {
    foo();
}
```

▼ *Speaker Notes*

This slide should take about 13 minutes.

- The reason for that is that when the `thread::scope` function completes, all the threads are guaranteed to be joined, so they can return borrowed data.
- Normal Rust borrowing rules apply: you can either borrow mutably by one thread, or immutably by any number of threads.

Channels

This segment should take about 20 minutes. It contains:

Slide	Duration
Senders and Receivers	10 minutes
Unbounded Channels	2 minutes
Bounded Channels	10 minutes

Senders and Receivers

Rust channels have two parts: a `Sender<T>` and a `Receiver<T>`. The two parts are connected via the channel, but you only see the end-points.

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

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

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("Received: {:?}", rx.recv());
    println!("Received: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Received: {:?}", rx.recv());
}
```

▼ *Speaker Notes*

This slide should take about 9 minutes.

- `mpsc` stands for Multi-Producer, Single-Consumer. `Sender` and `SyncSender` implement `Clone` (so you can make multiple producers) but `Receiver` does not.
- `send()` and `recv()` return `Result`. If they return `Err`, it means the counterpart `Sender` or `Receiver` is dropped and the channel is closed.

Unbounded Channels

You get an unbounded and asynchronous channel with `mpsc::channel()`:

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

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

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

- An unbounded channel will allocate as much space as is necessary to store pending messages. The `send()` method will not block the calling thread.
- A call to `send()` will abort with an error (that is why it returns `Result`) if the channel is closed. A channel is closed when the receiver is dropped.

Bounded Channels

With bounded (synchronous) channels, `send()` can block the current thread:

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

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Calling `send()` will block the current thread until there is space in the channel for the new message. The thread can be blocked indefinitely if there is nobody who reads from the channel.
- Like unbounded channels, a call to `send()` will abort with an error if the channel is closed.
- A bounded channel with a size of zero is called a “rendezvous channel”. Every send will block the current thread until another thread calls `recv()`.

Send and Sync

This segment should take about 15 minutes. It contains:

Slide	Duration
Marker Traits	2 minutes
Send	2 minutes
Sync	2 minutes
Examples	10 minutes

Marker Traits

How does Rust know to forbid shared access across threads? The answer is in two traits:

- `Send`: a type `T` is `Send` if it is safe to move a `T` across a thread boundary.
- `Sync`: a type `T` is `Sync` if it is safe to move a `&T` across a thread boundary.

`Send` and `Sync` are unsafe traits. The compiler will automatically derive them for your types as long as they only contain `Send` and `Sync` types. You can also implement them manually when you know it is valid.

▼ *Speaker Notes*

This slide should take about 2 minutes.

- One can think of these traits as markers that the type has certain thread-safety properties.
- They can be used in the generic constraints as normal traits.

Send

> A type `T` is `Send` if it is safe to move a `T` value to another thread.

The effect of moving ownership to another thread is that *destructors* will run in that thread. So the question is when you can allocate a value in one thread and deallocate it in another.

▼ *Speaker Notes*

This slide should take about 2 minutes.

As an example, a connection to the SQLite library must only be accessed from a single thread.

Sync

> A type `T` is `Sync` if it is safe to access a `T` value from multiple threads at the same time.

More precisely, the definition is:

> `T` is `Sync` if and only if `&T` is `Send`

▼ *Speaker Notes*

This slide should take about 2 minutes.

This statement is essentially a shorthand way of saying that if a type is thread-safe for shared use, it is also thread-safe to pass references of it across threads.

This is because if a type is Sync it means that it can be shared across multiple threads without the risk of data races or other synchronization issues, so it is safe to move it to another thread. A reference to the type is also safe to move to another thread, because the data it references can be accessed from any thread safely.

Examples

`Send + Sync`

Most types you come across are `Send + Sync`:

- `i8`, `f32`, `bool`, `char`, `&str`, …
- `(T1, T2)`, `[T; N]`, `&[T]`, `struct { x: T }`, …
- `String`, `Option<T>`, `Vec<T>`, `Box<T>`, …
- `Arc<T>`: Explicitly thread-safe via atomic reference count.
- `Mutex<T>`: Explicitly thread-safe via internal locking.
- `mpsc::Sender<T>`: As of 1.72.0.
- `AtomicBool`, `AtomicU8`, …: Uses special atomic instructions.

The generic types are typically `Send + Sync` when the type parameters are `Send + Sync`.

`Send + !Sync`

These types can be moved to other threads, but they're not thread-safe. Typically because of interior mutability:

- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

`!Send + Sync`

These types are safe to access (via shared references) from multiple threads, but they cannot be moved to another thread:

- `MutexGuard<T: Sync>`: Uses OS level primitives which must be deallocated on the thread which created them. However, an already-locked mutex can have its guarded variable read by any thread with which the guard is shared.

`!Send + !Sync`

These types are not thread-safe and cannot be moved to other threads:

- `Rc<T>`: each `Rc<T>` has a reference to an `RcBox<T>`, which contains a non-atomic reference count.
- `*const T`, `*mut T`: Rust assumes raw pointers may have special concurrency considerations.

Shared State

This segment should take about 30 minutes. It contains:

Slide	Duration
Arc	5 minutes
Mutex	15 minutes
Example	10 minutes

Arc

`Arc<T>` allows shared, read-only ownership via `Arc::clone`:

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

/// A struct that prints which thread drops it.
#[derive(Debug)]
struct WhereDropped(Vec<i32>);

impl Drop for WhereDropped {
    fn drop(&mut self) {
        println!("Dropped by {:?}", thread::current().id())
    }
}

fn main() {
    let v = Arc::new(WhereDropped(vec![10, 20, 30]));
    let mut handles = Vec::new();
    for i in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            // Sleep for 0-500ms.
            std::thread::sleep(std::time::Duration::from_millis(500 - i * 1
            let thread_id = thread::current().id();
            println!("{thread_id:?}: {v:?}");
        }));
    }

    // Now only the spawned threads will hold clones of `v`.
    drop(v);

    // When the last spawned thread finishes, it will drop `v`'s contents.
    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `Arc` stands for "Atomic Reference Counted", a thread safe version of `Rc` that uses atomic operations.
- `Arc<T>` implements `Clone` whether or not `T` does. It implements `Send` and `Sync` if and only if `T` implements them both.
- `Arc::clone()` has the cost of atomic operations that get executed, but after that the use of the `T` is free.
- Beware of reference cycles, `Arc` does not use a garbage collector to detect them.
    - `std::sync::Weak` can help

Mutex

`Mutex<T>` ensures mutual exclusion *and* allows mutable access to `T` behind a read-only interface (another form of interior mutability):

```rust
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

    println!("v: {:?}", v.lock().unwrap());
}
```

Notice how we have a `impl<T: Send> Sync for Mutex<T>` blanket implementation.

▼ *Speaker Notes*

This slide should take about 14 minutes.

- `Mutex` in Rust looks like a collection with just one element — the protected data.
  - It is not possible to forget to acquire the mutex before accessing the protected data.
- You can get an `&mut T` from an `&Mutex<T>` by taking the lock. The `MutexGuard` ensures that the `&mut T` doesn't outlive the lock being held.
- `Mutex<T>` implements both `Send` and `Sync` if and only if `T` implements `Send` .
- A read-write lock counterpart: `RwLock` .
- Why does `lock()` return a `Result` ?
  - If the thread that held the `Mutex` panicked, the `Mutex` becomes "poisoned" to signal that the data it protected might be in an inconsistent state. Calling `lock()` on a poisoned mutex fails with a `PoisonError` . You can call `into_inner()` on the error to recover the data regardless.

Example

Let us see `Arc` and `Mutex` in action:

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

fn main() {
    let v = vec![10, 20, 30];
    let mut handles = Vec::new();
    for i in 0..5 {
        handles.push(thread::spawn(|| {
            v.push(10 * i);
            println!("v: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

Possible solution:

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

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));
    let mut handles = Vec::new();
    for i in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            let mut v = v.lock().unwrap();
            v.push(10 * i);
            println!("v: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

Notable parts:

- `v` is wrapped in both `Arc` and `Mutex`, because their concerns are orthogonal.
  - Wrapping a `Mutex` in an `Arc` is a common pattern to share mutable state between threads.
- `v: Arc<_>` needs to be cloned to make a new reference for each new spawned

- Blocks are introduced to narrow the scope of the `LockGuard` as much as possible.

Exercises

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

Slide	Duration
Dining Philosophers	20 minutes
Multi-threaded Link Checker	20 minutes
Solutions	30 minutes

Dining Philosophers

The dining philosophers problem is a classic problem in concurrency:

> Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a chopstick between each plate. The dish served is spaghetti which requires two chopsticks to eat. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right chopstick. Thus two chopsticks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both chopsticks.

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:

```rust
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
use std::time::Duration;

struct Chopstick;

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

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

    fn eat(&self) {
        // Pick up chopsticks...
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    // Create chopsticks

    // Create philosophers

    // Make each of them think and eat 100 times

    // Output their thoughts
}
```

You can use the following `Cargo.toml`:

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

▼ *Speaker Notes*

This slide should take about 20 minutes.

- The deadlock in the simplest solution is a general concurrency problem and highlights that Rust does not automatically prevent this sort of bug.

Multi-threaded Link Checker

Let us use our new knowledge to create a multi-threaded link checker. It should start at a webpage and check that links on the page are valid. It should recursively check other pages on the same domain and keep doing this until all pages have been validated.

For this, you will need an HTTP client such as `reqwest`. You will also need a way to find links, we can use `scraper`. Finally, we'll need some way of handling errors, we will use `thiserror`.

Create a new Cargo project and `reqwest` it as a dependency with:

```
cargo new link-checker
cd link-checker
cargo add --features blocking,rustls-tls reqwest
cargo add scraper
cargo add thiserror
```

> If `cargo add` fails with `error: no such subcommand`, then please edit the `Cargo.toml` file by hand. Add the dependencies listed below.

The `cargo add` calls will update the `Cargo.toml` file to look like this:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

You can now download the start page. Try with a small site such as `https://www.google.org/`.

Your `src/main.rs` file should look something like this:

```rust
use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>,
Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}:
{err}");
            }
        }
    }
    Ok(link_urls)
}
```

```
        let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
        match visit_page(&client, &crawl_command) {
            Ok(links) => println!("Links: {links:#?}"),
            Err(err) => println!("Could not extract links: {err:#}"),
        }
    }
```

Run the code in `src/main.rs` with

```
cargo run
```

Tasks

- Use threads to check the links in parallel: send the URLs to be checked to a channel and let a few threads check the URLs in parallel.
- Extend this to recursively extract links from all pages on the `www.google.org` domain. Put an upper limit of 100 pages or so so that you don't end up being blocked by the site.

▼ *Speaker Notes*

This slide should take about 20 minutes.

- This is a complex exercise and intended to give students an opportunity to work on a larger project than others. A success condition for this exercise is to get stuck on some "real" issue and work through it with the support of other students or the instructor.

Solutions

Dining Philosophers

```rust
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
use std::time::Duration;

struct Chopstick;

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

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

    fn eat(&self) {
        println!("{} is trying to eat", &self.name);
        let _left = self.left_chopstick.lock().unwrap();
        let _right = self.right_chopstick.lock().unwrap();

        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    let (tx, rx) = mpsc::sync_channel(10);

    let chopsticks = PHILOSOPHERS
        .iter()
        .map(|_| Arc::new(Mutex::new(Chopstick)))
        .collect::<Vec<_>>();

    for i in 0..chopsticks.len() {
        let tx = tx.clone();
        let mut left_chopstick = Arc::clone(&chopsticks[i]);
        let mut right_chopstick =
        // either of them.
        if i == chopsticks.len() - 1 {
            std::mem::swap(&mut left_chopstick, &mut right_chopstick);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
            thoughts: tx,
            left_chopstick,
            right_chopstick,
        };

        thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```

Link Checker

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

use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>,
Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}:
{err}");
```

```
    Ok(link_urls)
}

struct CrawlState {
    domain: String,
    visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
    fn new(start_url: &Url) -> CrawlState {
        let mut visited_pages = std::collections::HashSet::new();
        visited_pages.insert(start_url.as_str().to_string());
        CrawlState { domain: start_url.domain().unwrap().to_string(),
visited_pages }
    }

    /// Determine whether links within the given page should be extracted.
    fn should_extract_links(&self, url: &Url) -> bool {
        let Some(url_domain) = url.domain() else {
            return false;
        };
        url_domain == self.domain
    }

    /// Mark the given page as visited, returning false if it had already
    /// been visited.
    fn mark_visited(&mut self, url: &Url) -> bool {
        self.visited_pages.insert(url.as_str().to_string())
    }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
fn spawn_crawler_threads(
    command_receiver: mpsc::Receiver<CrawlCommand>,
    result_sender: mpsc::Sender<CrawlResult>,
    thread_count: u32,
) {
    // To multiplex the non-cloneable Receiver, wrap it in Arc<Mutex<_>>.
    let command_receiver = Arc::new(Mutex::new(command_receiver));

    for _ in 0..thread_count {
        let result_sender = result_sender.clone();
        let command_receiver = Arc::clone(&command_receiver);
        thread::spawn(move || {
            let client = Client::new();
            loop {
                let command_result = {
                    let receiver_guard = command_receiver.lock().unwrap();
                    receiver_guard.recv()
                };
                let Ok(crawl_command) = command_result else {
                    // The sender got dropped. No more commands coming in.
                    break;
                };
                result_sender.send(crawl_result).unwrap();
            }
        });
    }
}

fn control_crawl(
    start_url: Url,
    command_sender: mpsc::Sender<CrawlCommand>,
    result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
    let mut crawl_state = CrawlState::new(&start_url);
    let start_command = CrawlCommand { url: start_url, extract_links: true };
    command_sender.send(start_command).unwrap();
    let mut pending_urls = 1;

    let mut bad_urls = Vec::new();
    while pending_urls > 0 {
        let crawl_result = result_receiver.recv().unwrap();
        pending_urls -= 1;

        match crawl_result {
            Ok(link_urls) => {
                for url in link_urls {
                    if crawl_state.mark_visited(&url) {
                        let extract_links =
crawl_state.should_extract_links(&url);
                        let crawl_command = CrawlCommand { url, extract_links
};

                        command_sender.send(crawl_command).unwrap();
                        pending_urls += 1;
                    }
                }
            }
            Err((url, error)) => {
                bad_urls.push(url);
                println!("Got crawling error: {:#}", error);
                continue;
            }
        }
    }
    bad_urls
}

fn check_links(start_url: Url) -> Vec<Url> {
    let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
    let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
    spawn_crawler_threads(command_receiver, result_sender, 16);
    control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
    let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
    let bad_urls = check_links(start_url);
```

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:

```rust
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`.

```rust
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

```rust
/// 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

```rust
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::Outp
        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,
                    } 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 + modif
                    } else {
                        return Poll::Pending;
                    }
                }
            }
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- 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 indrection, 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.

```rust
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.

```rust
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

- 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`:

```rust
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 ta
}
```

▼ *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.

```rust
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.

```rust
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.

```rust
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.

```rust
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

- This serves as a summation of a few of the previous lessons, so take your time with it.
    - 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)));
              },
          }
      }
      ```

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:

```rust
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:

  - 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.

```rust
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 UT
        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
            },
}
```

Speaker Notes

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.

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:

```rust
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

```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.

▼ *Speaker Notes*

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 = "1.48.0", features = ["full"] }
tokio-websockets = { version = "0.13.0", 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

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:

```
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.

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))));
        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();
ServerBuilder::new().accept(socket).await?;

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

src/bin/client.rs:

```
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),
                    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 5 hours and 5 minutes. It contains:

Segment	Duration
Leveraging the Type System	5 hours and 5 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.)

    - 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 enforce invariants
- "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

- 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 5 hours and 5 minutes. It contains:

Slide	Duration
Leveraging the Type System	5 minutes
Newtype Pattern	20 minutes
Extension Traits	1 hour and 5 minutes
Typestate Pattern	30 minutes
Borrow checking invariants	1 hour and 30 minutes
Token Types	1 hour and 35 minutes

Newtype Pattern

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

```rust
/// 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:

```rust
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:

```rust
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:

```rust
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*.

```rust
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.

```rust
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.

Extension Traits

It may desirable to **extend** foreign types with new inherent methods. For example, allow your code to check if a string is a palindrome using method-calling syntax: `s.is_palindrome()`.

It might feel natural to reach out for an `impl` block:

```rust
// 🛠️❌
impl &'_ str {
    pub fn is_palindrome(&self) -> bool {
        self.chars().eq(self.chars().rev())
    }
}
```

The Rust compiler won't allow it, though. But you can use the **extension trait pattern** to work around this limitation.

▼ *Speaker Notes*

This slide and its sub-slides should take about 65 minutes.

- A Rust item (be it a trait or a type) is referred to as:
  - **foreign**, if it isn't defined in the current crate
  - **local**, if it is defined in the current crate The distinction has significant implications for coherence and orphan rules, as we'll get a chance to explore in this section of the course.
- Compile the example to show the compiler error that's emitted.

  Highlight how the compiler error message nudges you towards the extension trait pattern.
- Explain how many type-system restrictions in Rust aim to prevent *ambiguity*.

  What would happen if you were allowed to define new inherent methods on foreign types? Different crates in your dependency tree might end up defining different methods on the same foreign type with the same name.

  As soon as there is room for ambiguity, there must be a way to disambiguate. If disambiguation happens implicitly, it can lead to surprising or otherwise unexpected behavior. If disambiguation happens explicitly, it can increase the cognitive load on developers who are reading your code.

explicit disambiguation.

Rust has decided to avoid the issue altogether by forbidding the definition of new inherent methods on foreign types.

- Other languages (e.g, Kotlin, C#, Swift) allow adding methods to existing types, often called “extension methods.” This leads to different trade-offs in terms of potential ambiguities and the need for global reasoning.

Extending Foreign Types

An **extension trait** is a local trait definition whose primary purpose is to attach new methods to foreign types.

```rust
mod ext {
    pub trait StrExt {
        fn is_palindrome(&self) -> bool;
    }

    impl StrExt for &str {
        fn is_palindrome(&self) -> bool {
            self.chars().eq(self.chars().rev())
        }
    }
}

// Bring the extension trait into scope...
pub use ext::StrExt as _;
// ...then invoke its methods as if they were inherent methods
assert!("dad".is_palindrome());
assert!(!"grandma".is_palindrome());
```

▼ *Speaker Notes*

- The `Ext` suffix is conventionally attached to the name of extension traits.

  It communicates that the trait is primarily used for extension purposes, and it is therefore not intended to be implemented outside the crate that defines it.

  Refer to the “Extension Trait” RFC as the authoritative source for naming conventions.

- The extension trait implementation for a foreign type must be in the same crate as the trait itself, otherwise you’ll be blocked by Rust’s *orphan rule*.

- The extension trait must be in scope when its methods are invoked.

  Comment out the `use` statement in the example to show the compiler error that’s emitted if you try to invoke an extension method without having the corresponding extension trait in scope.

- The example above uses an *underscore import* (`use ext::StringExt as _`) to minimize the likelihood of a naming conflict with other imported traits.

  With an underscore import, the trait is considered to be in scope and you’re allowed directly accessible. This prevents you, for example, from using that trait in a `where` clause.

Since extension traits aren't meant to be used in `where` clauses, they are conventionally imported via an underscore import.

Method Resolution Conflicts

What happens when you have a name conflict between an inherent method and an extension method?

```rust
mod ext {
    pub trait CountOnesExt {
        fn count_ones(&self) -> u32;
    }

    impl CountOnesExt for i32 {
        fn count_ones(&self) -> u32 {
            let value = *self;
            (0..32).filter(|i| ((value >> i) & 1i32) == 1).count() as u32
        }
    }
}
fn main() {
    pub use ext::CountOnesExt;
    // Which `count_ones` method is invoked?
    // The one from `CountOnesExt`? Or the inherent one from `i32`?
    assert_eq!((-1i32).count_ones(), 32);
}
```

▼ *Speaker Notes*

- A foreign type may, in a newer version, add a new inherent method with the same name as our extension method.

  Ask: What will happen in the example above? Will there be a compiler error? Will one of the two methods be given higher priority? Which one?

  Add a `panic!("Extension trait");` in the body of `CountOnesExt::count_ones` to clarify which method is being invoked.

- To prevent users of the Rust language from having to manually specify which method to use in all cases, there is a priority ordering system for how methods get "picked" first:

  - Immutable (`&self`) first
    - Inherent (method defined in the type's `impl` block) before Trait (method added by a trait impl).
  - Mutable (`&mut self`) Second
    - Inherent before Trait. If every method with the same name has different mutability and was either defined in as an inherent method or trait method, with no overlap, this makes the job easy for the compiler.

instead of relying on this mechanism if you can.

Demonstrate: Change the signature and implementation of `CountOnesExt::count_ones` to `fn count_ones(&mut self) -> u32` and modify the invocation accordingly:

```
assert_eq!((&mut -1i32).count_ones(), 32);
```

`CountOnesExt::count_ones` is invoked, rather than the inherent method, since `&mut self` has a higher priority than `&self`, the one used by the inherent method.

If an immutable inherent method and a mutable trait method exist for the same type, we can specify which one to use at the call site by using `(&<value>).count_ones()` to get the immutable (higher priority) method or `(&mut <value>).count_ones()`

Point the students to the Rust reference for more information on method resolution.

- Avoid naming conflicts between extension trait methods and inherent methods. Rust's method resolution algorithm is complex and may surprise users of your code.

More to explore

- The interaction between the priority search used by Rust's method resolution algorithm and automatic `Deref`ing can be used to emulate specialization on the stable toolchain, primarily in the context of macro-generated code. Check out "Autoref Specialization" for the specific details.

Trait Method Conflicts

What happens when you have a name conflict between two different trait methods implemented for the same type?

```rust
mod ext {
    pub trait Ext1 {
        fn is_palindrome(&self) -> bool;
    }

    pub trait Ext2 {
        fn is_palindrome(&self) -> bool;
    }

    impl Ext1 for &str {
        fn is_palindrome(&self) -> bool {
            self.chars().eq(self.chars().rev())
        }
    }

    impl Ext2 for &str {
        fn is_palindrome(&self) -> bool {
            self.chars().eq(self.chars().rev())
        }
    }
}

pub use ext::{Ext1, Ext2};

// Which method is invoked?
// The one from `Ext1`? Or the one from `Ext2`?
fn main() {
    assert!("dad".is_palindrome());
}
```

▼ *Speaker Notes*

- The trait you are extending may, in a newer version, add a new trait method with the same name as your extension method. Or another extension trait for the same type may define a method with a name that conflicts with your own extension method.

  Ask: what will happen in the example above? Will there be a compiler error? Will one of the two methods be given higher priority? Which one?

- The compiler rejects the code because it cannot determine which method to invoke. Neither `Ext1` nor `Ext2` has a higher priority than the other.

  To resolve this conflict, you must specify which trait you want to use.

For methods with more complex signatures, you may need to use a more explicit fully-qualified syntax.

- Demonstrate: replace `"dad".is_palindrome()` with `<&str as Ext1>::is_palindrome(&"dad")` or `<&str as Ext2>::is_palindrome(&"dad")`.

Extending Other Traits

As with types, it may be desirable to **extend foreign traits**. In particular, to attach new methods to *all* implementors of a given trait.

```rust
mod ext {
    use std::fmt::Display;

    pub trait DisplayExt {
        fn quoted(&self) -> String;
    }

    impl<T: Display> DisplayExt for T {
        fn quoted(&self) -> String {
            format!("'{}'", self)
        }
    }
}

pub use ext::DisplayExt as _;

assert_eq!("dad".quoted(), "'dad'");
assert_eq!(4.quoted(), "'4'");
assert_eq!(true.quoted(), "'true'");
```

▼ *Speaker Notes*

- Highlight how we added new behavior to *multiple* types at once. `.quoted()` can be called on string slices, numbers, and booleans since they all implement the `Display` trait.

  This flavor of the extension trait pattern uses *blanket implementations*.

  A blanket implementation implements a trait for all types `T` that satisfy the trait bounds specified in the `impl` block. In this case, the only requirement is that `T` implements the `Display` trait.

- Draw the students' attention to the implementation of `DisplayExt::quoted`: we can't make any assumptions about `T` other than that it implements `Display`. All our logic must either use methods from `Display` or functions/macros that don't require other traits.

  For example, we can call `format!` with `T`, but can't call `.to_uppercase()` because it is not necessarily a `String`.

- Conventionally, the extension trait is named after the trait it extends, followed by the `Ext` suffix. In the example above, `DisplayExt`.
- There are entire crates that extend standard library traits with new functionality.
  - `itertools` crate provides the `Itertools` trait that extends `Iterator`. It adds many iterator adapters, such as `interleave` and `unique`. It provides new algorithmic building blocks for iterator pipelines built with method chaining.
  - `futures` crate provides the `FutureExt` trait, which extends the `Future` trait with new combinators and helper methods.

More To Explore

- Extension traits can be used by libraries to distinguish between stable and experimental methods.

  Stable methods are part of the trait definition.

  Experimental methods are provided via an extension trait defined in a different library, with a less restrictive stability policy. Some utility methods are then "promoted" to the core trait definition once they have been proven useful and their design has been refined.
- Extension traits can be used to split a dyn-incompatible trait in two:
  - A **dyn-compatible core**, restricted to the methods that satisfy dyn-compatibility requirements.
  - An **extension trait**, containing the remaining methods that are not dyn-compatible (e.g., methods with a generic parameter).
- Concrete types that implement the core trait will be able to invoke all methods, thanks to the blanket impl for the extension trait. Trait objects (`dyn CoreTrait`) will be able to invoke all methods on the core trait as well as those on the extension trait that don't require `Self: Sized`.

Should I Define An Extension Trait?

In what scenarios should you prefer an extension trait over a free function?

```
pub trait StrExt {
    fn is_palindrome(&self) -> bool;
}

impl StrExt for &str {
    fn is_palindrome(&self) -> bool {
        self.chars().eq(self.chars().rev())
    }
}

// vs

fn is_palindrome(s: &str) -> bool {
    s.chars().eq(s.chars().rev())
}
```

The main advantage of extension traits is **ease of discovery**.

▼ *Speaker Notes*

- Extension methods can be easier to discover than free functions. Language servers (e.g., `rust-analyzer`) will suggest them if you type `.` after an instance of the foreign type.
- However, a bespoke extension trait might be overkill for a single method. Both approaches require an additional import, and the familiar method syntax may not justify the boilerplate of a full trait definition.
- **Discoverability:** Extension methods are easier to discover than free functions. Language servers (e.g., `rust-analyzer`) will suggest them if you type `.` after an instance of the foreign type.
- **Method Chaining:** A major ergonomic win for extension traits is method chaining. This is the foundation of the `Iterator` trait, allowing for fluent calls like `data.iter().filter(...).map(...)`. Achieving this with free functions would be far more cumbersome (`map(filter(iter(data), ...), ...)`).
- **API Cohesion:** Extension traits help create a cohesive API. If you have several related functions for a foreign type (e.g., `is_palindrome`, `word_count`, `to_kebab_case`), grouping them in a single `StrExt` trait is often cleaner than having multiple free

- **Trade-offs:** Despite these advantages, a bespoke extension trait might be overkill for a single, simple function. Both approaches require an additional import, and the familiar method syntax may not justify the boilerplate of a full trait definition.

Typestate Pattern: Problem

How can we ensure that only valid operations are allowed on a value based on its current state?

```rust
use std::fmt::Write as _;

#[derive(Default)]
struct Serializer {
    output: String,
}

impl Serializer {
    fn serialize_struct_start(&mut self, name: &str) {
        let _ = writeln!(&mut self.output, "{name} {{");
    }

    fn serialize_struct_field(&mut self, key: &str, value: &str) {
        let _ = writeln!(&mut self.output, "  {key}={value};");
    }

    fn serialize_struct_end(&mut self) {
        self.output.push_str("}\n");
    }

    fn finish(self) -> String {
        self.output
    }
}

fn main() {
    let mut serializer = Serializer::default();
    serializer.serialize_struct_start("User");
    serializer.serialize_struct_field("id", "42");
    serializer.serialize_struct_field("name", "Alice");

    // serializer.serialize_struct_end(); // ← Oops! Forgotten

    println!("{}", serializer.finish());
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 30 minutes.

- This `Serializer` is meant to write a structured value.
- However, in this example we forgot to call `serialize_struct_end()` before `finish()`. As a result, the serialized output is incomplete or syntactically incorrect.

- One approach to fix this would be to track internal state manually, and return a `Result` from methods like `serialize_struct_field()` or `finish()` if the current state is invalid.
- But this has downsides:
  - It is easy to get wrong as an implementer. Rust's type system cannot help enforce the correctness of our state transitions.
  - It also adds unnecessary burden on the user, who must handle `Result` values for operations that are misused in source code rather than at runtime.
- A better solution is to model the valid state transitions directly in the type system.

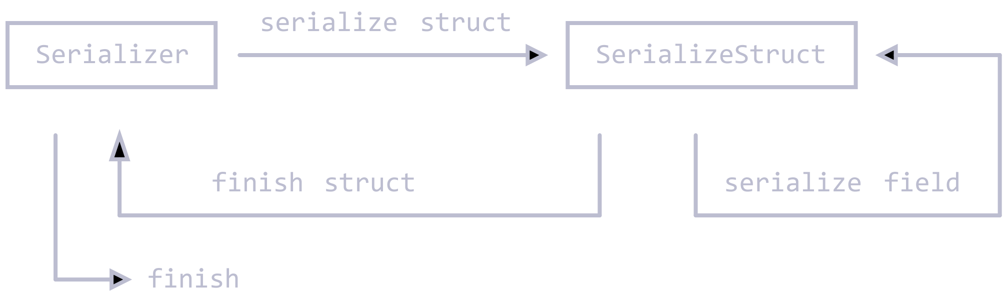
  In the next slide, we will apply the **typestate pattern** to enforce correct usage at compile time and make it impossible to call incompatible methods or forget to do a required action.

Typestate Pattern: Example

The typestate pattern encodes part of a value's runtime state into its type. This allows us to prevent invalid or inapplicable operations at compile time.

```rust
use std::fmt::Write as _;

#[derive(Default)]
struct Serializer {
    output: String,
}

struct SerializeStruct {
    serializer: Serializer,
}

impl Serializer {
    fn serialize_struct(mut self, name: &str) -> SerializeStruct {
        writeln!(&mut self.output, "{name} {{").unwrap();
        SerializeStruct { serializer: self }
    }

    fn finish(self) -> String {
        self.output
    }
}

impl SerializeStruct {
    fn serialize_field(mut self, key: &str, value: &str) -> Self {
        writeln!(&mut self.serializer.output, "  {key}={value};").unwrap();
        self
    }

    fn finish_struct(mut self) -> Serializer {
        self.serializer.output.push_str("}\n");
        self.serializer
    }
}

fn main() {
    let serializer = Serializer::default()
        .serialize_struct("User")
        .serialize_field("id", "42")
        .serialize_field("name", "Alice")
        .finish_struct();

    println!("{}", serializer.finish());
}
```

Serializer usage flowchart:



▼ *Speaker Notes*

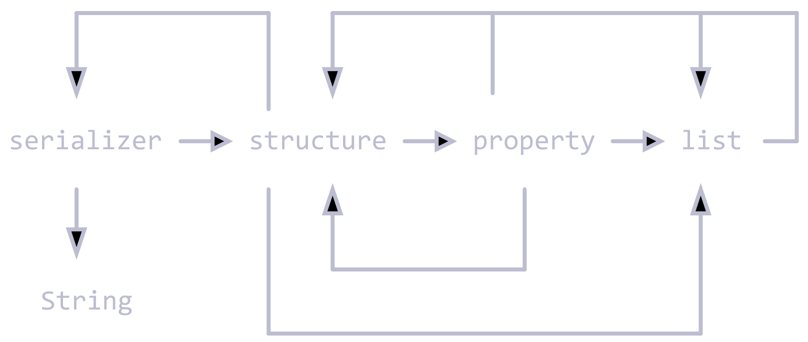
- This example is inspired by Serde's `Serializer` trait. Serde uses typestates internally to ensure serialization follows a valid structure. For more, see: https://serde.rs/impl-serializer.html
- The key idea behind typestate is that state transitions happen by consuming a value and producing a new one. At each step, only operations valid for that state are available.
- In this example:
  - We begin with a `Serializer`, which only allows us to start serializing a struct.
  - Once we call `.serialize_struct(...)`, ownership moves into a `SerializeStruct` value. From that point on, we can only call methods related to serializing struct fields.
  - The original `Serializer` is no longer accessible — preventing us from mixing modes (such as starting another *struct* mid-struct) or calling `finish()` too early.
  - Only after calling `.finish_struct()` do we receive the `Serializer` back. At that point, the output can be finalized or reused.
- If we forget to call `finish_struct()` and drop the `SerializeStruct` early, the `Serializer` is also dropped. This ensures incomplete output cannot leak into the system.
- By contrast, if we had implemented everything on `Serializer` directly — as seen on the previous slide, nothing would stop someone from skipping important steps or mixing serialization flows.

Beyond Simple Typestate

How do we manage increasingly complex configuration flows with many possible states and transitions, while still preventing incompatible operations?

```rust
struct Serializer {/* [...] */}
struct SerializeStruct {/* [...] */}
struct SerializeStructProperty {/* [...] */}
struct SerializeList {/* [...] */}

impl Serializer {
    // TODO, implement:
    //
    // fn serialize_struct(self, name: &str) -> SerializeStruct
    // fn finish(self) -> String
}

impl SerializeStruct {
    // TODO, implement:
    //
    // fn serialize_property(mut self, name: &str) -> SerializeStructProperty

    // TODO,
    // How should we finish this struct? This depends on where it appears:
    // - At the root level: return `Serializer`
    // - As a property inside another struct: return `SerializeStruct`
    // - As a value inside a list: return `SerializeList`
    //
    // fn finish(self) -> ???
}

impl SerializeStructProperty {
    // TODO, implement:
    //
    // fn serialize_string(self, value: &str) -> SerializeStruct
    // fn serialize_struct(self, name: &str) -> SerializeStruct
    // fn serialize_list(self) -> SerializeList
    // fn finish(self) -> SerializeStruct
}

impl SerializeList {
    // TODO, implement:
    //
    // fn serialize_string(mut self, value: &str) -> Self
    // fn serialize_struct(mut self, value: &str) -> SerializeStruct
    // fn serialize_list(mut self) -> SerializeList

    // TODO:
    // Like `SerializeStruct::finish`, the return type depends on nesting.
```

Diagram of valid transitions:



▼ *Speaker Notes*

- Building on our previous serializer, we now want to support **nested structures** and **lists**.
- However, this introduces both **duplication** and **structural complexity**.
- Even more critically, we now hit a **type system limitation**: we cannot cleanly express what `finish()` should return without duplicating variants for every nesting context (e.g. root, struct, list).
- From the diagram of valid transitions, we can observe:
  - The transitions are recursive
  - The return types depend on *where* a substructure or list appears
  - Each context requires a return path to its parent
- With only concrete types, this becomes unmanageable. Our current approach leads to an explosion of types and manual wiring.
- In the next chapter, we'll see how **generics** let us model recursive flows with less boilerplate, while still enforcing valid operations at compile time.

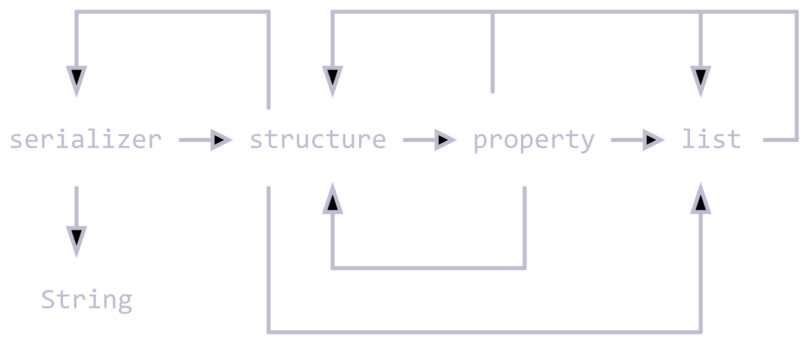
Typestate Pattern with Generics

By combining typestate modeling with generics, we can express a wider range of valid states and transitions without duplicating logic. This approach is especially useful when the number of states grows or when multiple states share behavior but differ in structure.

```rust
struct Serializer<S> {
    // [...]
    indent: usize,
    buffer: String,
    state: S,
}

struct Root;
struct Struct<S>(S);
struct Property<S>(S);
struct List<S>(S);
```

We now have all the tools needed to implement the methods for the `Serializer` and its state type definitions. This ensures that our API only permits valid transitions, as illustrated in the following diagram:

Diagram of valid transitions:



▼ *Speaker Notes*

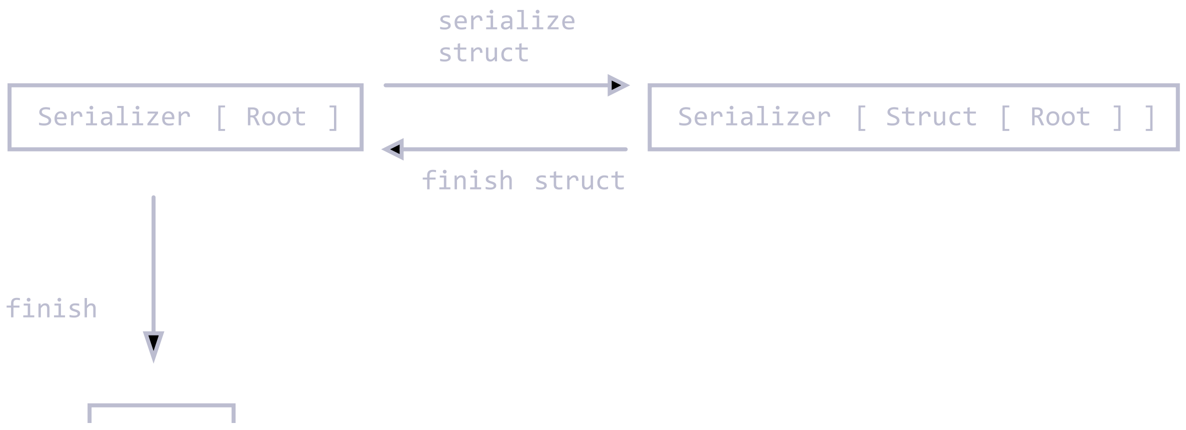
- By leveraging generics to track the parent context, we can construct arbitrarily nested serializers that enforce valid transitions between struct, list, and property states.
- This enables us to build a recursive structure while maintaining strict control over which methods are accessible in each state.
- Methods common to all states can be defined for any `S` in `Serializer<S>`.

correct API usage through the type system.

Serializer: implement Root

```rust
struct Serializer<S> {
    // [...]
    indent: usize,
    buffer: String,
    state: S,
}

struct Root;
struct Struct<S>(S);

impl Serializer<Root> {
    fn new() -> Self {
        // [...]
        Self { indent: 0, buffer: String::new(), state: Root }
    }

    fn serialize_struct(mut self, name: &str) -> Serializer<Struct<Root>> {
        // [...]
        writeln!(self.buffer, "{name} {{").unwrap();
        Serializer {
            indent: self.indent + 1,
            buffer: self.buffer,
            state: Struct(self.state),
        }
    }

    fn finish(self) -> String {
        // [...]
        self.buffer
    }
}
```

Referring back to our original diagram of valid transitions, we can visualize the beginning of our implementation as follows:



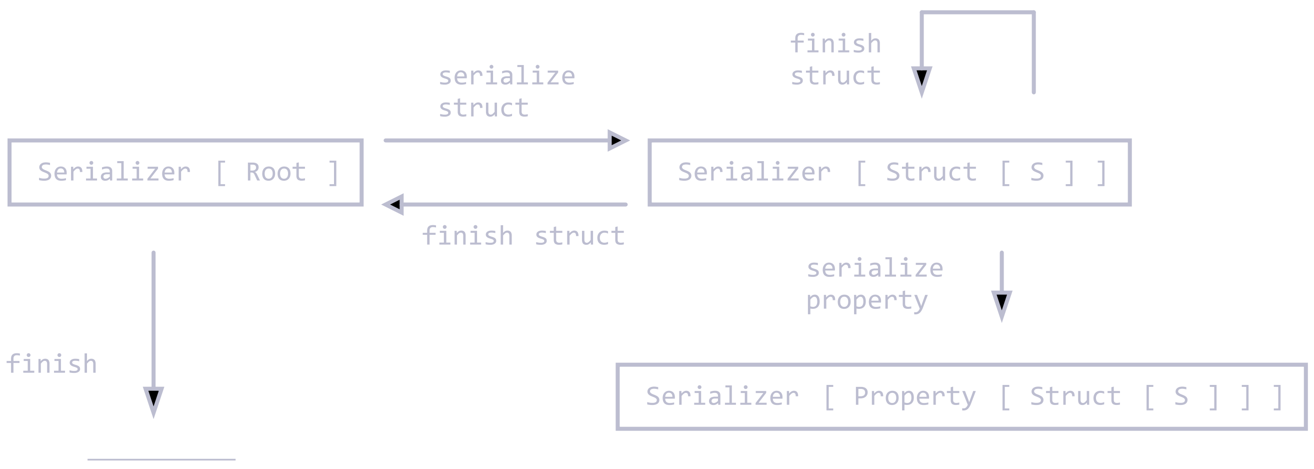
▼ *Speaker Notes*

- At the “root” of our `Serializer`, the only construct allowed is a `Struct`.
- The `Serializer` can only be finalized into a `String` from this root level.

Serializer: implement Struct

```rust
struct Serializer<S> {
    // [...]
    indent: usize,
    buffer: String,
    state: S,
}

struct Struct<S>(S);
struct Property<S>(S);

impl<S> Serializer<Struct<S>> {
    fn serialize_property(mut self, name: &str) ->
Serializer<Property<Struct<S>>> {
        // [...]
        write!(self.buffer, "{}{name}: ", " ".repeat(self.indent *
2)).unwrap();
        Serializer {
            indent: self.indent,
            buffer: self.buffer,
            state: Property(self.state),
        }
    }

    fn finish_struct(mut self) -> Serializer<S> {
        // [...]
        self.indent -= 1;
        writeln!(self.buffer, "{}}}", " ".repeat(self.indent * 2)).unwrap();
        Serializer { indent: self.indent, buffer: self.buffer, state:
self.state.0 }
    }
}
```

The diagram can now be expanded as follows:



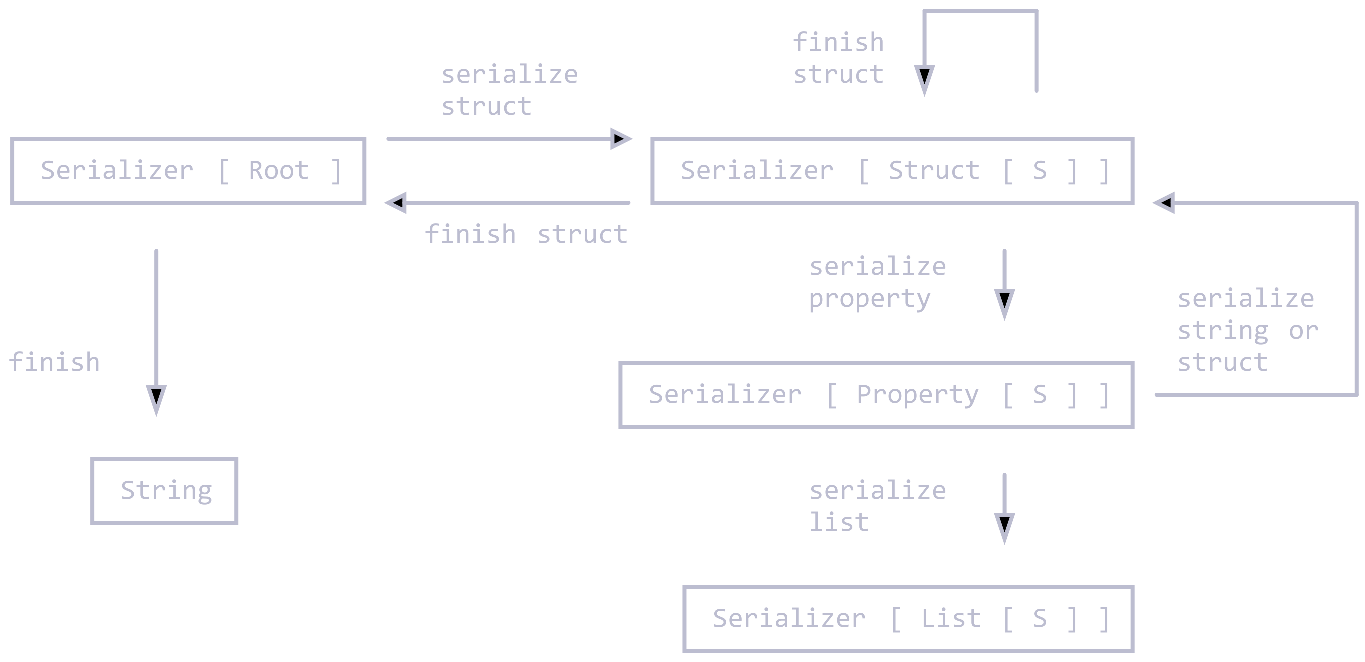
▼ *Speaker Notes*

- A `Struct` can only contain a `Property`;
- Finishing a `Struct` returns control back to its parent, which in our previous slide was assumed the `Root`, but in reality however it can be also something else such as `Struct` in case of nested “structs”.

Serializer: implement Property

```
struct Serializer<S> {
    // [...]
    indent: usize,
    buffer: String,
    state: S,
}

struct Struct<S>(S);
struct Property<S>(S);
struct List<S>(S);

impl<S> Serializer<Property<Struct<S>>> {
    fn serialize_struct(mut self, name: &str) -> Serializer<Struct<Struct<S>>>
{
        // [...]
        writeln!(self.buffer, "{name} {{").unwrap();
        Serializer {
            indent: self.indent + 1,
            buffer: self.buffer,
            state: Struct(self.state.0),
        }
    }

    fn serialize_list(mut self) -> Serializer<List<Struct<S>>> {
        // [...]
        writeln!(self.buffer, "[").unwrap();
        Serializer {
            indent: self.indent + 1,
            buffer: self.buffer,
            state: List(self.state.0),
        }
    }

    fn serialize_string(mut self, value: &str) -> Serializer<Struct<S>> {
        // [...]
        writeln!(self.buffer, "{value},").unwrap();
        Serializer { indent: self.indent, buffer: self.buffer, state:
self.state.0 }
    }
}
```

With the addition of the Property state methods, our diagram is now nearly complete:

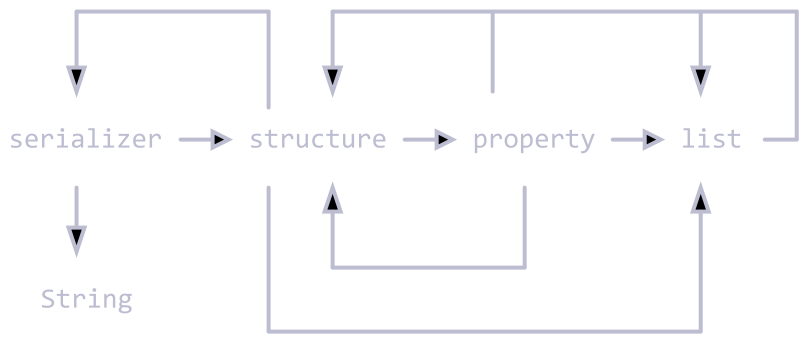

serialize struct
finish struct
Serializer [Root]
finish struct
Serializer [Struct [S]]
finish
serialize property
serialize string or struct
Serializer [Property [S]]
String
serialize list
Serializer [List [S]]

▼ *Speaker Notes*

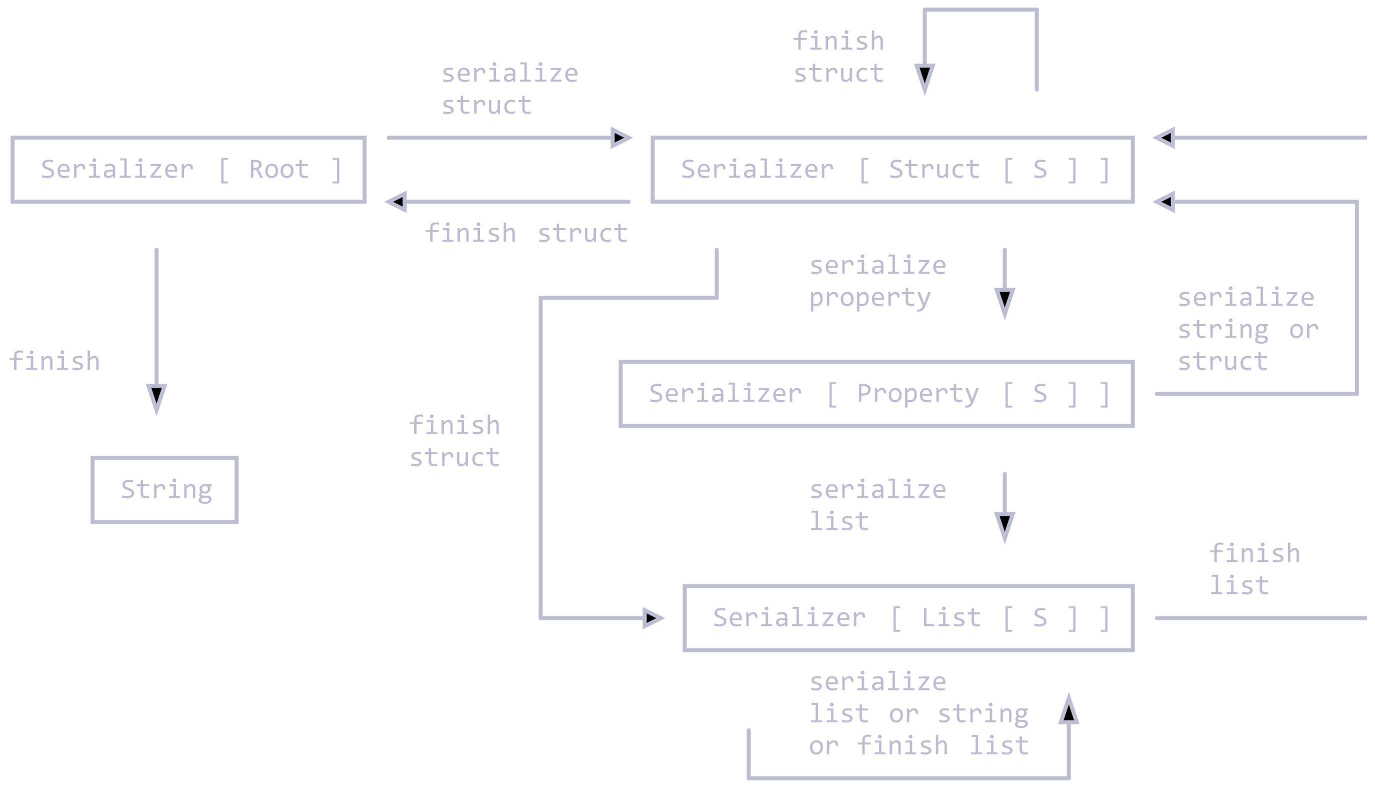
- A property can be defined as a `String`, `Struct<S>`, or `List<S>`, enabling the representation of nested structures.
- This concludes the step-by-step implementation. The full implementation, including support for `List<S>`, is shown in the next slide.

Serializer: complete implementation

Looking back at our original desired flow:



We can now see this reflected directly in the types of our serializer:


Serializer [Root]
serialize struct
finish struct
Serializer [Struct [S]]
finish struct
finish
String
finish struct
serialize property
Serializer [Property [S]]
serialize string or struct
serialize list
Serializer [List [S]]
finish list
serialize list or string or finish list

The code for the full implementation of the `Serializer` and all its states can be found in this Rust playground.

▼ *Speaker Notes*

- This pattern isn't a silver bullet. It still allows issues like:

  - Duplicate property names (which could be tracked in `Struct<S>` and handled via `Result`)

- If validation failures occur, we can also change method signatures to return a `Result`, allowing recovery:

```
struct PropertySerializeError<S> {
    kind: PropertyError,
    serializer: Serializer<Struct<S>>,
}

impl<S> Serializer<Struct<S>> {
    fn serialize_property(
        self,
        name: &str,
    ) -> Result<Serializer<Property<Struct<S>>>,
PropertySerializeError<S>> {
        /* ... */
    }
}
```

- While this API is powerful, it's not always ergonomic. Production serializers typically favor simpler APIs and reserve the typestate pattern for enforcing critical invariants.

- One excellent real-world example is `rustls::ClientConfig`, which uses typestate with generics to guide the user through safe and correct configuration steps.